

# Functional programming, Seminar No. 4

---

Danya Rogozin

Institute for Information Transmission Problems, RAS

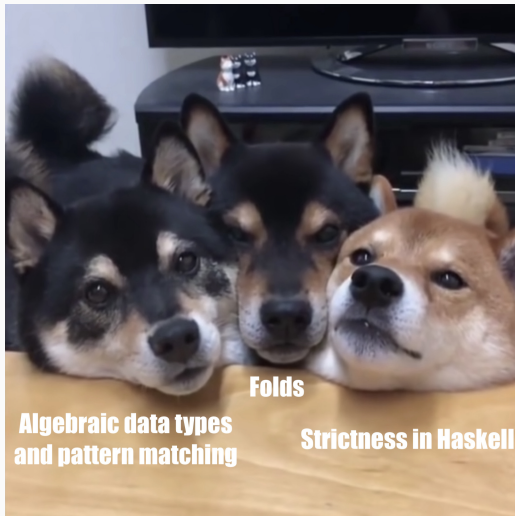
Serokell OÜ

Higher School of Economics

The Faculty of Computer Science

# Today

We will study



# **Algebraic data types and pattern matching**

---

# Pattern matching

Let us take a look at the following functions:

```
swap :: (a, b) -> (b, a)
```

```
swap (a, b) = (b, a)
```

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

```
length [] = 0
```

```
length (x : xs) = 1 + length xs
```

# Pattern matching

Let us take a look at the following functions:

```
swap :: (a, b) -> (b, a)
```

```
swap (a, b) = (b, a)
```

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

```
length [] = 0
```

```
length (x : xs) = 1 + length xs
```

- Expressions like `(a,b)`, `[]`, and `(x : xs)` are called *patterns*
- One needs to check that constructors `(,)` and `( : )` are relevant.
- Consider `swap (45, True)`. Variables `a` and `b` are binded with the values `45` and `True`.
- Consider `length [1,2,3]`. Variables `x` and `xs` are binded with the values `1` and `[2,3]`

# Algebraic data types. Sums

The simplest example of an algebraic data type is a data type defined with an enumeration of constructors that stores no values.

```
data Colour = Red | Blue | Green | Purple | Yellow
    deriving (Show, Eq)
```

```
isRGB :: Colour -> Bool
isRGB Red   = True
isRGB Blue  = True
isRGB Green = True
isRGB _     = False           -- Wild-card
```

# Algebraic data types. Products

- An example of a product data type:

```
data Point = Point Double Double
    deriving Show
```

```
> :type Point
```

```
Point :: Double -> Double -> Point
```

- An example of a function

```
taxCab :: Point -> Point -> Double
taxCab (Point x1 y1) (Point x2 y2) =
    abs (x1 - x2) + abs (y1 - y2)
```

# Polymorphic data types

- That point data type might be parametrised with a type parameter:

```
data Point a = Point a a
    deriving Show
```

- The type of the Point data constructor. The Point from the left is a type constructor that has its type (kind).

```
> :type Point
Point :: a -> a -> Point a
> :k Point
Point :: * -> *
```



# Polymorphic data types and type classes

- Suppose we have a function:

```
midPoint
```

```
  :: Fractional a => Point a -> Point a -> Point a
```

```
midPoint (Pt x1 y1) (Pt x2 y2) =
```

```
  Pt ((x1 + x2) / 2) ((y1 + y2) / 2)
```

- Playing with GHCi:

```
> :t midPoint (Pt 3 5) (Pt 6 4)
```

```
midPoint (Pt 3 5) (Pt 6 4) :: Fractional a => Point a
```

```
> midPoint (Pt 3 5) (Pt 6 4)
```

```
Pt 4.5 4.5
```

```
> :t it
```

```
it :: Fractional a => Point a
```

# Inductive data types

- The list is the first example of an inductive data type

```
data List a = Nil | Cons a (List a)
  deriving Show
```

- The data constructors are `Nil :: List a` and  
`Cons :: a -> List a -> List a`
- Pattern matching and recursion

```
concat :: List a -> List a -> List a
concat Nil ys = ys
concat (Cons x xs) ys = Cons x (xs `concat` ys)
```

# Standard lists

- The list data type is already in the standard library, but its approximate definition is the following one:

```
infixr 5 :  
data [] a = [] | a : ([] a)  
    deriving Show
```

- Syntax sugar:

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

- The example of a definition with built-in lists:

```
infixr 5 ++  
(++) :: [a] -> [a] -> [a]  
(++) []      ys = ys  
(++) (x:xs)  ys = x : xs ++ ys
```

## case ... of ... expressions

- case ... of ... expressions allows one to patternmatch everywhere

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x : xs) =
    case p x of
        True  -> x : filter p xs
        False -> filter p xs
```

- The pattern matching from the previous slide is a syntax sugar for the corresponding case ... of ... expression

# Semantic aspects of pattern matching

- Pattern matching is performed from up to down and from left to right after that.
- A pattern match is either
  - succeed
  - or failed
  - or diverged

- Here is an example:

`foo (1,4) = 7`

`foo (0,_) = 8`

- `(0, undefined)` fails in the first case and it succeeds in the second one
- `(undefined, 0)` diverges during a match
- What about `(1,7-3)`?

- Suppose we have the following function

```
dupHead :: [a] -> [a]
```

```
dupHead (x : xs) = x : x : xs
```

- One may rewrite this function as follows:

```
dupHead :: [a] -> [a]
```

```
dupHead s@(x : xs) = x : s
```

- Here, the name `s` is assigned to the whole pattern `x : xs`

# Irrefutable patterns

- Irrefutable patterns are wild-cards, variables, and lazy patterns
- An example of a lazy pattern:

```
> (***) f g (a,b) = (f a, g b)
> const 2 *** const 1 $ undefined
*** Exception: Prelude.undefined
> (***) f g ~(a,b) = (f a, g b)
> const 2 *** const 1 $ undefined
(2,1)
```

## `newtype` and type declarations

- The keyword `type` introduces type synonyms.

```
type String = [Char]
```

- In Haskell, the string data type `String` is merely a type synonym for the list of characters
- The keyword `newtype` defines a new type with the single constructor that packs a value of a given type

```
newtype Age = Age Int
```

- The same type `Age` defined with the accessor `runAge`

```
newtype Age = Age { runAge :: Int }  
-- where runAge :: Age -> Int
```



# Field labels

- Sometimes product data types are too cumbersome:

```
data Person = Person String String Int Float String
```

- As an alternative, one may define a data type with field labels

```
data Person =  
  Person { firstName :: String  
          , lastName  :: String  
          , age       :: Int  
          , height    :: Float  
          , phoneNumber :: String  
          }  
}
```

- Such a data type is a record with accessors such as  
 firstName :: Person -> String

# Field labels and type classes

- Let us recall the Eq type class once more

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

```
instance Eq Int where
  x == y = x `eqInt` y
```

```
eqFunction :: Eq a => a -> a -> Int
eqFunction x y =
  case x == y of
    True  -> 42
    False -> 0
```

- In fact, type classes are sugar for data types with field labels
- The constraint Eq a is an additional argument

# Field labels and type classes

- The previous listing a bit unsugared (very roughly):

```
data Eq a =  
  Eq { eq :: a -> a -> Bool  
      , neq :: a -> a -> Bool  
      }
```

```
intInstance :: Eq Int  
intInstance = Eq eqInt (\x y -> not $ x `eqInt` y)
```

```
eqFunction :: Eq a -> a -> a -> Int  
eqFunction eqInst x y =  
  case ((eq eqInst) x y) of  
    True -> 42  
    False -> 0
```

# Some standard algebraic data types

- The `Maybe a` data type allows one to define an optional value:

```
data Maybe a = Nothing | Just a
```

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

```
maybe b _ Nothing = b
```

```
maybe b f (Just x) = f x
```

- A simple example

```
safeHead :: [a] -> Maybe a
```

```
safeHead [] = Nothing
```

```
safeHead (x : _) = Just x
```

# Some standard algebraic data types

- The Either data type describes one or the other value

```
data Either e a = Left e | Right a
```

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

```
either f _ (Left x) = f x
```

```
either _ g (Right x) = g x
```

- An example:

```
safeTail :: [a] -> Either String [a]
```

```
safeTail [] = Left "I have no tail, mate"
```

```
safeTail (_ : xs) = Right xs
```

# Folds

---

# Folds and lists. Motivation

Suppose we have the following functions

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

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

```
product :: Num a => [a] -> a
```

```
product [] = 1
```

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

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

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

# The definition of a right fold

- The definition of a right fold is the following one

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

```
foldr _ ini [] = []
```

```
foldr f ini (x : xs) = f x (foldr f ini xs)
```

- An informal explanation:

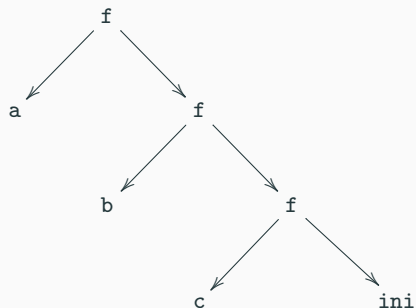
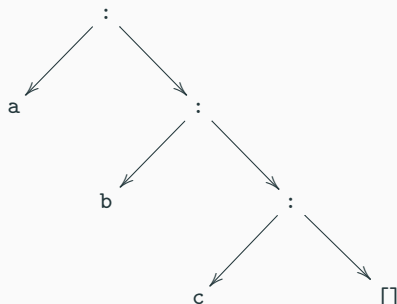
```
foldr f z [x1, x2, ..., xn] ==
```

```
x1 `f` (x2 `f` ... (xn `f` z) ...)
```



# The definition of a right fold

One may visualise that for some list  $[a, b, c]$ . The list from the left and its right fold from the right



## Functions `sum`, `product`, and `concat` with `foldr`

```
sum :: Num a => [a] -> a
sum = foldr (+) 0
```

```
product :: Num a => [a] -> a
product = foldr (*) 1
```

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

# The universal property of a right fold

## The universal property

Let  $g$  be a function defined by the following equations:

$$g [] = v$$

$$g (x : xs) = f x (g xs)$$

then one has  $\forall xs :: [a] \ (g xs \equiv \text{foldr } f \ v \ xs)$

- The universal property is inductively
- The meaning of this fact:  $\text{foldr } f \ v$  and  $g$  are interchangeable in this case

# The definition of a left fold

- In addition to the right fold, one also has the left one

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ ini [] = ini
foldl f ini (x : xs) = foldl f (f ini x) xs
```

- Informally:

```
foldl f ini [x1, x2, ..., xn]
  == (...((z `f` x1) `f` x2) `f` ...) `f` xn
```

# The definition of a left fold

- In addition to the right fold, one also has the left one

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ ini [] = ini
foldl f ini (x : xs) = foldl f (f ini x) xs
```

- Informally:

```
foldl f ini [x1, x2, ..., xn]
  == (...((z `f` x1) `f` x2) `f` ...) `f` xn
```

- The implementation of the left fold function might be optimised. Here we have a thunk
- `foldl` is the most optimal function, but we are not capable of processing infinite lists using the left fold function.

# Are foldr and foldl equivalent?

- Note that foldr and foldl are not equivalent to each other

```
> foldl (/) 64 [4,2,4]
```

```
2.0
```

```
> foldr (/) 64 [4,2,4]
```

```
0.125
```

```
> foldl (\x y -> 2*x + y) 4 [1,2,3]
```

```
43
```

```
> foldr (\x y -> 2*x + y) 4 [1,2,3]
```

```
16
```

- foldr and foldl are equivalent if the folding operation is commutative

# The right scan

- The right scan is the foldr that yields a list that contains all intermediate values

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ ini [] = [ini]
scanr f ini (x:xs) = f x q : qs
    where qs@(q:_) = scanr f ini xs
```

- foldr and scanr are connected with each other as follows

$$\text{head } (\text{scanr } f \ z \ xs) \equiv \text{foldr } f \ z \ xs$$

- The examples are

```
> scanr (:) [] [1,2,3]
[[1,2,3],[2,3],[3],[]]
> scanr (+) 0 [1..10]
[55,54,52,49,45,40,34,27,19,10,0]
> scanr (*) 1 [1..5]
[120,120,60,20,5,1]
```

# The left scan

- One also has a scan function for the `foldl` function:

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl f q ls = q : (case ls of
                      []    -> []
                      x:xs -> scanl f (f q x) xs)
```

- `foldl` and `scanl` are connected with each other as follows:

$$\text{last } (\text{scanl } f \ z \ xs) \equiv \text{foldl } f \ z \ xs$$

- The examples:

```
> scanl (++) "!" ["a","b","c"]
["!", "!a", "!ab", "!abc"]
> scanl (*) 1 [1..] !! 5
120
```



# Strictness in Haskell

---

- Any well-formed expression in Haskell has a type
- Prima facie, the `Bool` data type has two values: `False` and `True` according to its definition:

```
data Bool = False | True
```

- One may define an expression `dno :: Bool` which is defined recursively as `dno = not dno`
- `dno` is neither `False` nor `True`, but it's a Boolean value!
- This value is a bottom ( $\perp$ ). In Haskell,  $\perp$  is a value that has a type for all `a`. Such errors as undefined have this type.

# Strict functions

- Haskell is lazy. That's why `const 42 undefined == 42`
- Lazy functions are non-strict ones

# Strict functions

- Haskell is lazy. That's why `const 42 undefined == 42`
- Lazy functions are non-strict ones
- In contrast to lazy functions, strict functions satisfy this equation

$$f\ x_1\ x_2\ \dots\ \perp\ \dots\ x_n = \perp$$

- For this reason `constStrict 42 undefined = undefined`

## Strictness in Haskell. The `seq` function

- We've already had a look at the `seq` function.
- `seq` is a combinator that enforce computation.
- This combinator has a type  $a \rightarrow b \rightarrow b$ .
- It seems that the body of `seq` looks like `\x y -> y`, but `seq` satisfies the following equations:

$$\text{seq } \perp x = \perp$$

$$\text{seq } v x = x, v \neq \perp$$

- Such an enforcing breaks the lazy semantics of Haskell! But this enforcing is not so far-reaching. Data constructors and lambdas put a barrier for the  $\perp$  expansion:

```
> seq (4,undefined) 5
```

```
5
```

```
> seq (\x -> undefined) 5
```

```
5
```

```
> seq (id . undefined) 5
```

```
5
```

# Strictness in Haskell. The strict application

- One may implement the strict application using `seq`

```
infixr 0 $!
```

```
($!) :: (a -> b) -> a -> b
```

```
f $! x = x `seq` f x
```

- That is, this application behaves as usual if the second argument is not bottom.

# Strictness in Haskell. The strict application

- Let us recall the tail-recursive factorial. The second version is strict:

```
tailFactorial :: Integer -> Integer
```

```
tailFactorial n = helper 1 n
```

```
  where
```

```
    helper acc x =
```

```
      if x > 1
```

```
      then helper (acc * x) (x - 1)
```

```
      else acc
```

```
tailFactorialStrict :: Integer -> Integer
```

```
tailFactorialStrict n = helper 1 n
```

```
  where
```

```
    helper acc x =
```

```
      if x > 1
```

```
      then (helper $! (acc * x)) (x - 1)
```

```
      else acc
```

# The strict foldl

- The strict version of foldl

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f ini [] = ini
foldl' f ini (x:xs) = foldl' f arg xs
  where arg = (f ini) $! x
```



# Strictness in Haskell. Bang patterns

- A data type might contain strict values with the strictness flag `!`, e.g.

```
data Complex a = !a :+ !a
    deriving Show
infix 6 :+
```

```
im :: Complex a -> a
```

```
im (x :+ y) = y
```

```
> im (undefined :+ 5) *** Exception: Prelude.undefined
```

- The `BangPatterns` extension allows one to make pattern a strict one

```
> :set -XBangPatterns
```

```
> foo !x = True
```

```
> foo undefined
```

```
*** Exception: Prelude.undefined
```

Today we

- discussed the data type landscape and together with pattern matching
- studied folds
- realised how one can enforce lazy evaluation

Today we

- discussed the data type landscape and together with pattern matching
- studied folds
- realised how one can enforce lazy evaluation

On the next seminar, we will

- study such type classes as `Functor`, `Foldable`, and `Monoid`