# Functional programming, Seminar No. 4

Danya Rogozin
Lomonosov Moscow State University,
Serokell OÜ

Higher School of Economics
Faculty of Computer Science

# Intro

On the previous seminar, we

- introduced parametric polymorphism
- discussed type classes and their examples (`Show`, `Eq`, `Ord`, etc)

# Intro

On the previous seminar, we

- introduced parametric polymorphism
- discussed type classes and their examples (`Show`, `Eq`, `Ord`, etc)

Today, we

- study pattern matching and such type constructions as algebraic data types, new types, type synonyms, and records
- learn folds
- talk about lazy evalution enforcing more systematically

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

- Expressions like `(a,b)`, `[]`, and `(x : xs)` are often called patterns
- In such calls as `swap (45, True)` or `lenght [1,2,3]`, we deal with *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
lenght (x : xs) = 1 + length xs
```

- Expressions like `(a,b)`, `[]`, and `(x : xs)` are often called patterns
- In such calls as `swap (45, True)` or `lenght [1,2,3]`, we deal with *pattern matching*
- One needs to check that constructors `(,)` and `( : )` are relevant.
- In the call `swap (45, True)`, variables `a` and `b` are binded with the values `45` and `True`.
- In the call `lenght [1,2,3]`, variables `x` and `xs` are binded with the values `1` and `[2,3]`

# Algebraic data types. Enumerations

- 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)
```

- The example of pattern matching for this data type

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

# Algebraic data types. Products

- The example of a product data type:

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

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

- The example of a function

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

- The example in a GHCi session

```
> taxCab (Point 3.0 5.0) (Point 7.0 9.0)
8.0
```

# Polymorphic data types

- The point data type might parametrised with a type parameter:

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

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

- The type of the `Point` constructor

```
> :type Point
Point :: a -> a -> Point a
```

- `Point` is a type operator. One also has a type (kind) system for type operators:

```
> :k Point
Point :: * -> *
```

# Polymorphic data types and type classes

- Suppose we have a function:

```haskell
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:

```haskell
> :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
```

# Polymorphic data types and type classes

- Suppose we have a function:

```haskell
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:

```haskell
> :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
```

- The type of point is a polymorphic itself. But one needs to use ad hoc polymorphism (the `Fractional` context) to apply division.
- On the other hand, polymorphism here is ambiguous. The fractional type is `Double` by default. Haskell has a defaulting mechanism for numerical data types

# Inductive data types

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

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

- The data constructors are `Nil :: List a` and `Cons :: a -> List a -> List a`
- The processing of such data types: pattern matching and recursion

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

- The GHCi session:

```haskell
> x = Cons 'a' (Cons 'b' Nil)
> y = Cons 'c' (Cons 'd' Nil)
> concat x y
Cons 'a' (Cons 'b' (Cons 'c' (Cons 'd' Nil)))
```

# Standard lists

- The list data type is a default one, but its approximate definition is the following one:

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

- Some 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 perform pattern matching everywhere

```haskell
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

# Semantical aspects of pattern matching

- Pattern matching is performed from up to down and from left to right after that.
- Pattern matching 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's succeed in the second one
- `(undefined, 0)` is diverged automatically
- `(2,1)` is a diverged pattern
- What about `(1,7-3)`?

# As-patterns

- Suppose we have the following function

```haskell
dupHead :: [a] -> [a]
dupHead (x : xs) = x : x : xs
```

- One may rewrite this function as follows:

```haskell
dupHead :: [a] -> [a]
dupHead s@(x : xs) = x : s
```

- Here, the name `s` is assigned to the whole pattern `x : xs`
- In fact, such a construction is a syntax sugar for the following one. It is not so hard to ensure that both functions have the same behaviour

```haskell
dupHead :: [a] -> [a]
dupHead (x : xs) =
  let s = (x : xs) in x : s
```

# Irrefutable patterns

- Irrefutable patterns are wild-cards, variables, and lazy patterns
- The 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)
```

# The `newtype` and `type` declarations

- The keyword `type` allows one to introduce type synonyms. The example given

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

- In Haskell, the string data type type is merely a type synonym for the list of characters
- The keyword `newtype` defined a new type with the single constructor that packs an existing types

```haskell
newtype Age = Age Int
```

- The same type `Age` defined with the equipped function `runAge`

```haskell
newtype Age = Age { runAge :: Int }
```

- The type of `runAge`

```haskell
> :t runAge
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, e.g. `firstName :: Person -> String`
- In fact, this data type is a product data type with accessor function

# Field labels and type classes

- Let us recall the `Eq` type class once more

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

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

isZero :: Int -> Bool
isZero x = if x == y then True else False

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

- In fact, type classes are syntax sugar for records defined with field labels
- The constraint `Eq a` is an additional argument

# Field labels and type classes

- The previous listing has the following meaning (very roughly):

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

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

isZero :: Int -> Bool
isZero x = if (eq eqInstance) x 0 then True else False

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

# Some of 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
```

- The simple example given

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

```
\item The GHCi session:
> maybe (maxBound :: Int) (+ 176) (safeHead [])
9223372036854775807
> maybe (maxBound :: Int) (+ 176) (safeHead [1..1500])
177
```

# Some of 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
```

- The example given:

```
safeTail :: [a] -> Either String [a]
safeTail [] = Left "I have no tail, mate"
safeTail (_ : xs) = Right xs
```

- The GHCi example

```
> either id (map succ) (safeTail [])
"I have no tail, mate"
> either id (map succ) (safeTail "\USdmbqxos\USld+\USokd‘rd")
"encrypt me, please"
```

# Folds and lists. Motivation

- Suppose we have these functions

```
sum :: [Integer] -> Integer
sum [] = 0
sum (x : xs) = x + sum xs

product :: [Integer] -> Integer
product [] = 1
product (x : xs) = x * product xs

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

- It is clear that one has a common recursion pattern

# The definition of a right fold

- The definition of a right is the following one

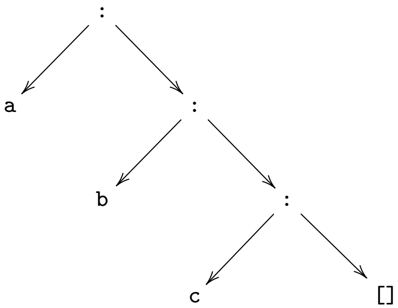
```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ ini [] = []
foldr f ini (x : xs) = f x (foldr f ini xs)
```

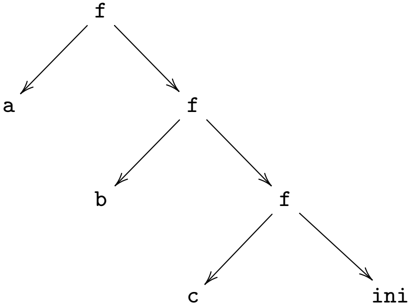
- Informally, this function behaves as follows:

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

# The definition of a right fold

One may visualise the story above for some list `[a,b,c]`. The list from the left and its right fold from the right:



# Functions sum, product, and concat via foldr

- Let us rewrite those functions with `foldr`

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

product :: [Integer] -> Integer
product = foldr (*) 1

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

- What about `foldr (:) []`?

# The universal property of a right fold

## The universal property

Let `f` 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` = `foldr f v xs`)

- The universal property is proved by induction on `xs`
- The converse implication is quite trivial
- The meaning of this fact: `foldr f v` and `g` are interchangeable in this case

## The definition of a left fold

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

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

- Informally:

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

# The definition of a left fold

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

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

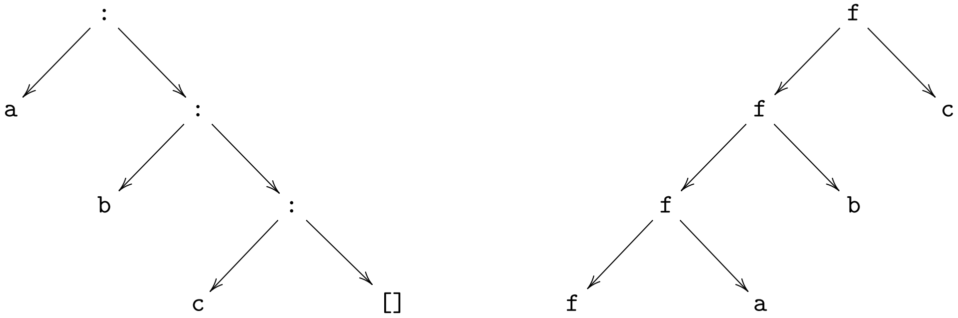
- Informally:

```
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 an increasing thunk
- We take a look at the strict version of `foldl`
- `foldl` is the most optimal function, but we are not capable of processing infinite lists using the left fold function.

# The definition of a left fold

One may visualise left fold in the same manner:



# The right scan

# The left scan

# Strictness in Haskell. The seq function

# Strictness in Haskell. The strict application

# Strictness in Haskell. Bang patterns