

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 evaluation enforcing more systematically

Pattern matching

Let us take a look at the following functions:

```
1  swap :: (a, b) -> (b, a)
2  swap (a, b) = (b, a)
3
4  length :: [a] -> Int
5  length [] = 0
6  length (x : xs) = 1 + length xs
```

Pattern matching

Let us take a look at the following functions:

```
1  swap :: (a, b) -> (b, a)
2  swap (a, b) = (b, a)
3
4  length :: [a] -> Int
5  length [] = 0
6  length (x : xs) = 1 + length xs
```

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

Pattern matching

Let us take a look at the following functions:

```
1  swap :: (a, b) -> (b, a)
2  swap (a, b) = (b, a)
3
4  length :: [a] -> Int
5  length [] = 0
6  length (x : xs) = 1 + length xs
```

- Expressions like (a, b) , $[]$, and $(x : xs)$ are often called patterns
- In such calls as `swap (45, True)` or `length [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 `length [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.

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

- The example of pattern matching for this data type

```
1 isRGB :: Colour -> Bool
2 isRGB Red = True
3 isRGB Blue = True
4 isRGB Green = True
5 isRGB _ = False      -- Wild-card
```

Algebraic data types. Products

- The example of a product data type:

```
1 data Point = Point Double Double
2   deriving Show
```

```
1 > :type Point
2 Point :: Double -> Double -> Point
```

- The example of a function

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

- The example in a GHCi session

```
1 > taxCab (Point 3.0 5.0) (Point 7.0 9.0)
2 8.0
```


Polymorphic data types

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

```
1  data Point a = Point a a
2  deriving Show
```

- The type of the Point constructor

```
1  > :type Point
2  Point :: a -> a -> Point a
```

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

```
1  > :k Point
2  Point :: * -> *
```

Polymorphic data types and type classes

- Suppose we have a function:

```
1  midPoint :: Fractional a => Point a -> Point a -> Point a
2  midPoint (Pt x1 y1) (Pt x2 y2) = Pt ((x1 + x2) / 2) ((y1 + y2) / 2)
```

- Playing with GHCi:

```
1  > :t midPoint (Pt 3 5) (Pt 6 4)
2  midPoint (Pt 3 5) (Pt 6 4) :: Fractional a => Point a
3  > midPoint (Pt 3 5) (Pt 6 4)
4  Pt 4.5 4.5
5  > :t it
6  it :: Fractional a => Point a
```

Polymorphic data types and type classes

- Suppose we have a function:

```
1  midPoint :: Fractional a => Point a -> Point a -> Point a
2  midPoint (Pt x1 y1) (Pt x2 y2) = Pt ((x1 + x2) / 2) ((y1 + y2) / 2)
```

- Playing with GHCi:

```
1  > :t midPoint (Pt 3 5) (Pt 6 4)
2  midPoint (Pt 3 5) (Pt 6 4) :: Fractional a => Point a
3  > midPoint (Pt 3 5) (Pt 6 4)
4  Pt 4.5 4.5
5  > :t it
6  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

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

- The data constructors are `Nil :: List a` and `Cons :: a -> List a -> List a`
- Here come pattern matching and recursion

```
1  concat :: List a -> List a -> List a
2  concat Nil ys = ys
3  concat (Cons x xs) ys = Cons x (xs 'concat' ys)
```

- The GHCi session:

```
1  > x = Cons 'a' (Cons 'b' Nil)
2  > y = Cons 'c' (Cons 'd' Nil)
3  > concat x y
4  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:

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

- Some syntax sugar

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

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

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

case ... of ... expressions

- case ... of ... expressions allows one to perform pattern matching everywhere

```
1  filter :: (a -> Bool) -> [a] -> [a]
2  filter p [] = []
3  filter p (x : xs) =
4      case p x of
5          True -> x : filter p xs
6          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:

1 `foo (1,4) = 7`

2 `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

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

- One may rewrite this function as follows:

```
1  dupHead :: [a] -> [a]
2  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

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


Irrefutable patterns

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

```
1 > (***) f g (a,b) = (f a, g b)
2 > const 2 *** const 1 $ undefined
3 *** Exception: Prelude.undefined
4 > (***) f g ~(a,b) = (f a, g b)
5 > const 2 *** const 1 $ undefined
6 (2,1)
```

The newtype and type declarations

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

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

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

```
1  newtype Age = Age Int
```

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

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

- The type of `runAge`

```
1  > :t runAge
2  runAge :: Age -> Int
```

Field labels

- Sometimes product data types are too cumbersome:

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

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

```
1    data Person =  
2        Person { firstName :: String  
3                , lastName  :: String  
4                , age       :: Int  
5                , height   :: Float  
6                , phoneNumber :: String  
7                }
```

- 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

```
1  class Eq a where
2      (==) :: a -> a -> Bool
3      (/=) :: a -> a -> Bool
4
5  instance Eq Int where
6      x == y = x `eqInt` y
7
8  isZero :: Int -> Bool
9  isZero x = if x == 0 then True else False
10
11 eqFunction :: Eq a => a -> a -> Int
12 eqFunction x y =
13     case x == y of
14         True -> 42
15         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):

```
1  data Eq a =
2      Eq { eq :: a -> a -> Bool
3          , neq :: a -> a -> Bool
4          }
5
6  intInstance :: Eq Int
7  intInstance = Eq eqlnt (\x y -> not $ x 'eqlnt' y)
8
9  isZero :: Int -> Bool
10 isZero x = if (eq eqlnt) x 0 then True else False
11
12 eqFunction :: Eq a -> a -> a -> Int
13 eqFunction eqlnt x y =
14     case ((eq eqlnt) x y) of
15         True -> 42
16         False -> 0
```

Some of standard algebraic data types

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

```
1  data Maybe a = Nothing | Just a
2
3  maybe :: b -> (a -> b) -> Maybe a -> b
4  maybe b _ Nothing = b
5  maybe b f (Just x) = f x
```

- The simple example given

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

```
1  \item The GHCi session:
2  > maybe (maxBound :: Int) (+ 176) (safeHead [])
3  9223372036854775807
4  > maybe (maxBound :: Int) (+ 176) (safeHead [1..1500])
5  177
```

Some of standard algebraic data types

- The `Either` data type describes one or the other value

```
1  data Either e a = Left e | Right a
2
3  either :: (a -> c) -> (b -> c) -> Either a b -> c
4  either f _ (Left x) = f x
5  either _ g (Right x) = g x
```

- The example given:

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

- The GHCi example

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

Folds and lists. Motivation

- Suppose we have these functions

```
1  sum :: [Integer] -> Integer
2  sum [] = 0
3  sum (x : xs) = x + sum xs
4
5  product :: [Integer] -> Integer
6  product [] = 1
7  product (x : xs) = x * product xs
8
9  concat :: [[a]] -> [a]
10 concat [] = []
11 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

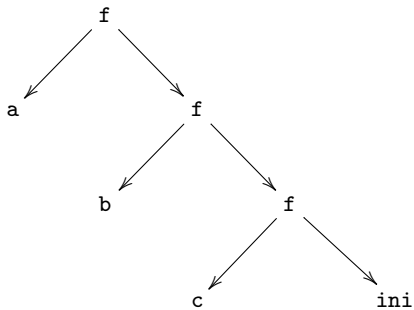
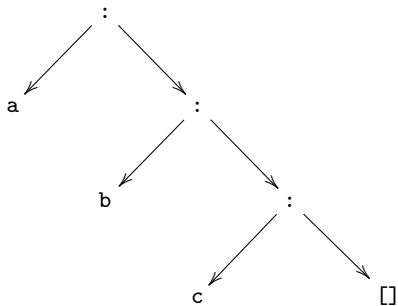
```
1  foldr :: (a -> b -> b) -> b -> [a] -> b
2  foldr _ ini [] = []
3  foldr f ini (x : xs) = f x (foldr f ini xs)
```

- Informally, this function behaves as follows:

```
1  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

```
1  sum :: [Integer] -> Integer
2  sum = foldr (+) 0
3
4  product :: [Integer] -> Integer
5  product = foldr (*) 1
6
7  concat :: [[a]] -> [a]
8  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:

- 1 $g [] = v$
- 2 $g (x : xs) = f x (g xs)$

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

- The universal property is proved by induction on xs
- The converse implication is quite trivial
- 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 a right fold, one also has a left one

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

- Informally:

```
1  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

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

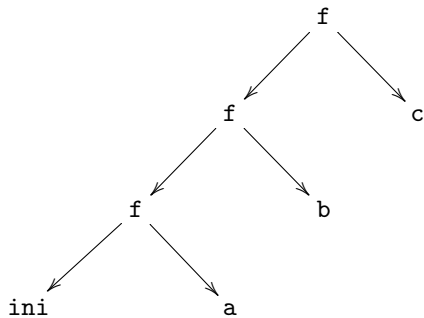
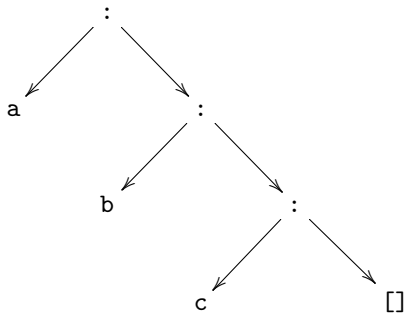
- Informally:

```
1  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:



Are foldr and foldl equivalent?

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

```
1  > foldl (/) 64 [4,2,4]
2  2.0
3  > foldr (/) 64 [4,2,4]
4  0.125
5  > foldl (\x y -> 2*x + y) 4 [1,2,3]
6  43
7  > foldr (\x y -> 2*x + y) 4 [1,2,3]
8  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

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

- foldr and scanr are connected with each other as follows

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

- The examples are

```
1  > scanr (++) "!" ["aa","bb","cc"]
2  ["aabbcc!", "bbcc!", "cc!", "!!"]
3  > scanr (:) [] [1,2,3]
4  [[1,2,3],[2,3],[3],[]]
5  > scanr (+) 0 [1..10]
6  [55,54,52,49,45,40,34,27,19,10,0]
7  > scanr (*) 1 [1..10]
8  [3628800,3628800,1814400,604800,151200,30240,5040,720,90,10,1]
```

The left scan

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

```
1  scanl :: (b -> a -> b) -> b -> [a] -> [b]
2  scanl f q ls = q : (case ls of
3                      []    -> []
4                      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:

```
1  > scanl (++) "!" ["a","b","c"]
2  ["!", "!a", "!ab", "!abc"]
3  > scanl (*) 1 [1..] !! 5
4  120
```

The presence of a bottom

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

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

- One may define an expression `dno :: Bool` which is recursively defined 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 `forall a. a`. Such errors as undefined have this type.

Strict function

- Haskell has the call-by name semantics. That's the reason according to which `const 42 undefined` yields 42
- Lazy functions are non-strict ones

Strict function

- Haskell has the call-by name semantics. That's the reason according to which `const 42 undefined` yields 42
- Lazy functions are non-strict ones
- In contrast to lazy functions, strict functions satisfy this equivation

$$f \perp = \perp$$

- That is, the strict `const` should yield `undefined` in the call `const 42 undefined`

Strictness in Haskell. The seq function

- We took a look at the seq function. Let us recall it.
- seq is a combinator that enforce a computation.
- This combinator has a type $a \rightarrow b \rightarrow b$.
- It seems that the body of seq looks like $\backslash x \ y \rightarrow y$, but seq satisfies the following equations:

$$\begin{aligned}\text{seq } \perp \ x &= \perp \\ \text{seq } \textit{value} \ x &= x\end{aligned}$$

- 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:

```
1  > seq (4,undefined) 5
2  5
3  > seq (\x -> undefined) 5
4  5
5  > seq (id . undefined) 5
6  5
```

Strictness in Haskell. The strict application

- One may implement the strict application using `seq`

```
1   infixr 0 $!  
2   ($!) :: (a -> b) -> a -> b  
3   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:

```
1      tailFactorial n = helper 1 n
2      where
3      helper acc x =
4          if x > 1
5          then helper (acc * x) (x - 1)
6          else acc
```

- The optimal version of the tail-recursive factorial is the following one:

```
1      tailFactorial n = helper 1 n
2      where
3      helper acc x =
4          if x > 1
5          then (helper $! (acc * x)) (x - 1)
6          else acc
```


The strict foldl

- A strict version of foldl

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

Strictness in Haskell. Bang patterns

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

```
1  data Complex a = !a :+ !a
2      deriving Show
3  infix 6 :+
4
5  im :: Complex a -> a
6  im (x :+ y) = y
```

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

- The BangPatterns allows one to make pattern a strict one

```
1  > foo !x = True
2  > foo undefined
3  *** Exception: Prelude.undefined
```