Functional programming, Seminar No. 3

Danya Rogozin
Lomonosov Moscow State University,
Serokell OÜ

Higher School of Economics
Faculty of Computer Science

Intro

On the previous seminar, we

- studied the basic Haskell syntax
- introduced the notion of a weak head normal form to describe the operatonal semantics of Haskell
- analysed the regrettable cicrumstances according to which Haskell doesn't have the Church-Rosser property as a system of typed lambda calculus

Intro

On the previous seminar, we

- studied the basic Haskell syntax
- introduced the notion of a weak head normal form to describe the operatonal semantics of Haskell
- analysed the regrettable cicrumstances according to which Haskell doesn't have the Church-Rosser property as a system of typed lambda calculus

Today we

- investigate the Haskell type system more deeply and overview the advantages of parametric polymorphism
- take a look at bounded polymorphism and discuss type classes

Motivation

Let us recall the example of a higher order function from the previous seminar:

```haskell
changeTwiceBy :: (Int -> Int) -> Int -> Int
changeTwiceBy operation value = operation (operation  value)
```

It is clear that one may implement the function for Boolean values and strings that have the same behaviour as the function above:

```haskell
changeTwiceByBool :: (Bool -> Bool) -> Bool -> Bool
changeTwiceByBool operation value = operation (operation  value)

changeTwiceByString :: (String -> String) -> String -> String
changeTwiceByString operation value = operation (operation  value)
```

Motivation

Let us recall the example of a higher order function from the previous seminar:

```
changeTwiceBy :: (Int -> Int) -> Int -> Int
changeTwiceBy operation value = operation (operation  value)
```

It is clear that one may implement the function for Boolean values and strings that have the same behaviour as the function above:

```
changeTwiceByBool :: (Bool -> Bool) -> Bool -> Bool
changeTwiceByBool operation value = operation (operation  value)

changeTwiceByString :: (String -> String) -> String -> String
changeTwiceByString operation value = operation (operation  value)
```

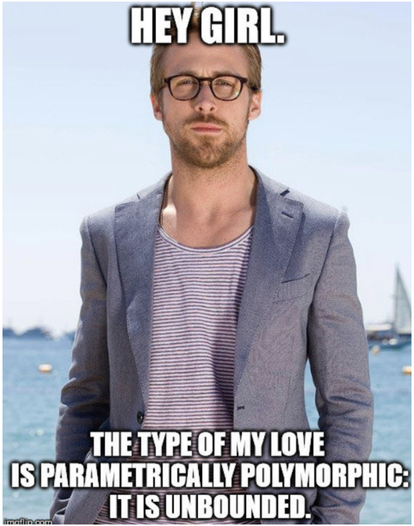
One needs to have a way to avoid such a boilerplate.

Parametric polymorphism

The key idea of parametric polymorphism that the same function might be called on distinct data types. Here are the first polymorphic examples:

```
id :: a -> a
id x = x

const :: a -> b -> a
const a b = a

fst :: (a, b) -> a
fst (a, b) = a

snd :: (a, b) -> b
snd = "guess what"

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

The meme time



The functions above in the GHCi session

```
Prelude> id 7
7
Prelude> id "string"
"string"
Prelude> const 7 "string"
7
Prelude> const "string" 7
"string"
Prelude> fst (7, 'k')
7
Prelude> snd (7, 'k')
'k'
Prelude> fst (swap (7, 'k'))
'k'
```

A brief clarification

- In such signatures as $a \to b \to a$, a, b are type variables that range over arbitrary data types. In fact, a, b are bounded by universal quantifier that hidden under the carpet.
- In a matter of fact, the functions from the previous slide have the following signatures:

```haskell
id :: forall a. a -> a
id x = x

const :: forall a b. a -> b -> a
const a b = a

fst :: forall a b. (a, b) -> a
fst (a, b) = a

swap :: forall a b. (a, b) -> (b, a)
swap (a, b) = (b, a)
```

Higher order functions and parametric polymorpism

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

flip :: (a -> b -> c) -> b -> a -> c
flip f b a = f a b

fix :: (a -> a) -> a
fix f = f (fix f)

curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p = f (fst p) (snd p)
```

The functions above in the GHCi session. The composition examples

```
incNegate :: Int -> Int
incNegate x = negate (x + 1)

incNegate x = negate $ x + 1

incNegate x = (negate . (+1)) x

incNegate x = negate . (+1) $ x

incNegate   = negate . (+1)
```

The functions above in the GHCi session. `curry` and `uncurry`

```
Prelude> uncurry (+) (3,4)
7
Prelude> curry fst 3 4
3
Prelude> curry snd 3 4
4
Prelude> curry id 3 4
(3,4)
Prelude> uncurry const (3,4)
3
Prelude> uncurry (flip const) (3,4)
4
```

The functions above in the GHCi session. The `flip` example

```haskell
show2 :: Int -> Int -> String
show2 x y = show x ++ " and " ++ show y

showSnd, showFst, showFst' :: Int -> String
showSnd = show2 1
showFst  = flip show2 2
showFst' = (`show2` 2)
```

The functions above in the GHCi session. The `flip` example

```
show2 :: Int -> Int -> String
show2 x y = show x ++ " and " ++ show y

showSnd, showFst, showFst' :: Int -> String
showSnd  = show2 1
showFst  = flip show2 2
showFst' = (`show2` 2)
```

```
Prelude> showSnd 10
"1 and 10"
Prelude> showFst 10
"10 and 2"
Prelude> showFst' 42
"42 and 2"
```

Bye-bye boilerplate!

All these functions

```haskell
changeTwiceBy :: (Int -> Int) -> Int -> Int
changeTwiceBy operation value = operation (operation value)

changeTwiceByBool :: (Bool -> Bool) -> Bool -> Bool
changeTwiceByBool operation value = operation (operation value)

changeTwiceByString :: (String -> String) -> String -> String
changeTwiceByString operation value = operation (operation value)
```

might be replaced to the following ones:

```haskell
applyTwice :: (a -> a) -> a -> a
applyTwice f a = f (f a)

applyTwice' :: (a -> a) -> a -> a
applyTwice' f a = f . f $ a

applyTwice'' :: (a -> a) -> a -> a
applyTwice'' f = f . f
```

HOF, polymorpism, and lists

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

filter  :: (a -> Bool) -> [a] -> [a]

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

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

HOF, polymorpism, and lists

```haskell
map     :: (a -> b)      -> [a] -> [b]

filter  :: (a -> Bool) -> [a] -> [a]

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

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

We discuss their implementations closely on the next seminar. Here we just take a look at their behaviour.

The composition examples + list functions

```
foo, bar :: [Int] -> Int
foo patak =
  length $ filter odd $
  map (div 2) $ filter even $ map (div 7) patak

bar        =
  length . filter odd .
  map (div 2) . filter even . map (div 7)

zip :: [a] -> [b] -> [(a, b)]
zip = zipWith (,)
```

The composition examples + list functions

```
stringsTransform :: [String] -> [String]
stringsTransform l = map (\s -> map toUpper s) (filter (\s -> length s == 5) l)

stringsTransform l = map (\s -> map toUpper s) $ filter (\s -> length s == 5) l

stringsTransform l = map (map toUpper) $ filter ((== 5) . length) l

stringsTransform = map (map toUpper) . filter ((== 5) . length)
```

Bounded polymorphism and type classes

The idea of bounded (ad hoc) polymorphism is that one has a general interface with instances for each concrete data type.

Bounded polymorphism and type classes

The idea of bounded (ad hoc) polymorphism is that one has a general interface with instances for each concrete data type.

```
Prelude> 9
9
Prelude> 9 :: Int
9
Prelude> 9 :: Integer
9
Prelude> 9 :: Float
9.0
Prelude> 9 :: Double
9.0
Prelude> 9 :: Rational
9 % 1
Prelude> 9 :: Char

<interactive>:7:1: error:
    • No instance for (Num Char) arising from the literal ‘9’
    • In the expression: 9 :: Char
      In an equation for ‘it’: it = 9 :: Char
```

Type classes. Motivation

- Let us take a look a the following function

```
elem :: a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

Type classes. Motivation

- Let us take a look a the following function

```
elem :: a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

- Is a type a arbitrary?

Type classes. Motivation

- Let us take a look a the following function

```
elem :: a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

- Is a type a arbitrary? Yes and no. a is an arbitrary type for which equality is defined.

Type classes. Motivation

As we observed, type variables in polymorphic function are bounded via universal quantifier. In ad hoc polymorphism, type variables are also bounded via $\forall$ but with the additional condition. Such a quantification is called bounded.

```
elem :: forall a. Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

The notion of a type class

- *A type class* is a collection of functions with type signatures with a common type parameter. The example given:

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

- A type class name introduce a constraint called *context*:

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

The notion of a type class

- *A type class* is a collection of functions with type signatures with a common type parameter. The example given:

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

- A type class name introduce a constraint called *context*:

```haskell
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

- The definition above without a context yields a type error:

```
<interactive>:6:20: error:
    • No instance for (Eq a) arising from a use of '=='
      Possible fix:
        add (Eq a) to the context of
          the type signature for:
            elem' :: forall a. a -> [a] -> Bool
    • In the first argument of '(||)', namely 'x == y'
      In the expression: x == y || elem x ys
      In an equation for 'elem'': elem' x (y : ys) = x == y || elem x ys
```

Instance declarations

A given data type a has the *instance* of a type class if every function of that class is implemented for a. The example:

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _       = False

  x /= y       = neg (x == y)
```

Polymorphism + instance declarations

- A type parameter in an instance declaration might be polymorphic itself:

```
instance Eq a => Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _        == _        = False
```

- Without the context `Eq a =>`, this definition yields type error since we don't know how to perform equality comparison of element from `a`

Some the Eq instances

- The Eq type class has the following instances (some of them)

```
instance Eq a => Eq [a]
instance Eq Ordering
instance Eq Int
instance Eq Float
instance Eq Double
instance Eq Char
instance Eq Bool
```

- See the standard library source code to take a look at the instances implementation.

The Show type class

- The Show type class allows one to represent a value as a string:

```haskell
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
```

- One needs to have a Show instance to display a value of a given type on a console.

Some of the Show instances

Here are some of the Show instances:

```
instance Show Integer
instance Show Int
instance Show Char
instance Show Bool
instance (Show a, Show b) => Show (a, b)
```

Ordering. Motivation

- Let us take a look at the following quicksort function:

```haskell
quicksort :: [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort small ++ (x : quicksort large)
  where
    small = [y | y <- xs, y <= x]
    large = [y | y <- xs, y > x]
```

Ordering. Motivation

- Let us take a look at the following quicksort function:

```
quicksort :: [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort small ++ (x : quicksort large)
  where
    small = [y | y <- xs, y <= x]
    large = [y | y <- xs, y > x]
```

- Here we have the same story as in the case of equality. A type a is an arbitrary type for which comparison is defined. The definition of quicksort as above is wrong. There exists a type element of which are incomparable, complex numbers, e.g.

The Ord type class

The full definition of `Ord` is the following one:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a

  compare x y = if x == y then EQ
                else if x <= y then LT
                else GT

  x <= y = case compare x y of { GT -> False; _ -> True }

  max x y = if x <= y then y else x

  {-# MINIMAL compare | (<=) #-}
```

Some of the Ord instances

- The `Ord` instances

```
instance Ord Int
instance Ord Float
instance Ord Double
instance Ord Char
instance Ord Bool
```

The Num type class

- Num is a type class with the general interface of usual arithmetical operations.

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
```

The Num type class

- Num is a type class with the general interface of usual arithmetical operations.

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
```

- The instances of Num form a collection of numerical data types in Haskell

The Num type class

- Num is a type class with the general interface of usual arithmetical operations.

```haskell
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
```

- The instances of Num form a collection of numerical data types in Haskell
- Note that we don't require the context `Ord a` since the set complex numbers is an instance of Num, but we don't have the instance `Ord Complex`, as you know.

Some of the Num instances

- The examples of `Num` instances are:

```
instance Num Integer
instance Num Int
instance Num Float
instance Num Double
```

The Enum and Bounded type classes

- The `Enum` is type class for type for which one may define an explicit enumeration.

```
class Enum a where
  succ, pred :: a -> a
  toEnum    :: Int -> a
  fromEnum :: a -> Int

  enumFrom     :: a -> [a]              -- [n..]
  enumFromThen :: a -> a -> [a]      -- [n,m..]
  enumFromTo :: a -> a -> [a]        -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,m..p]
  {-# MINIMAL toEnum, fromEnum #-}
```

- The `Bounded` type class is a type class for bounded types, i.e., types with minimal and maximal bounds

```
class Bounded a where
  minBound :: a
  maxBound :: a
  {-# MINIMAL minBound, maxBound #-}
```

Some of the `Enum` instances

- The instances are the following ones:

```
instance Enum Integer
instance Enum Int
instance Enum Char
instance Enum Bool
instance Enum Float
instance Enum Double
```

Some of the Bounded instances

- The examples of bounded data types

```
instance Bounded Int
instance Bounded Char
instance Bounded Bool
```

The `Fractional` type class

- The `Fractional` type class is a general interface for numerical division

```haskell
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
```

- It is clear that such a type should be a numerical one. Thus, we require the `Num a` restriction.
- The `Fractional` instances:

```haskell
instance Fractional Float
instance Fractional Double
```

Summary

On this seminar, we

- took a look at parametric polymorphism to see how to avoid boilerplate
- discussed type classes and ad hoc polymorphism
- studied such basic type classes as `Eq`, `Show`, etc

Summary

On this seminar, we

- took a look at parametric polymorphism to see how to avoid boilerplate
- discussed type classes and ad hoc polymorphism
- studied such basic type classes as `Eq`, `Show`, etc

On the next seminar, we

- delve into the variety of Haskell data types: algebraic data types, newtypes, type synonyms, etc
- feel the power of pattern matching
- discuss folds
- see how to enforce lazy evaluation in Haskell