

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 operational semantics of Haskell
- analysed the regrettable circumstances 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 operational semantics of Haskell
- analysed the regrettable circumstances 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:

```
1  changeTwiceBy :: (Int -> Int) -> Int -> Int
2  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:

```
1  changeTwiceByBool :: (Bool -> Bool) -> Bool -> Bool
2  changeTwiceByBool operation value = operation (operation value)
3
4  changeTwiceByString :: (String -> String) -> String -> String
5  changeTwiceByString operation value = operation (operation value)
```

Motivation

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

```
1  changeTwiceBy :: (Int -> Int) -> Int -> Int
2  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:

```
1  changeTwiceByBool :: (Bool -> Bool) -> Bool -> Bool
2  changeTwiceByBool operation value = operation (operation value)
3
4  changeTwiceByString :: (String -> String) -> String -> String
5  changeTwiceByString operation value = operation (operation value)
```

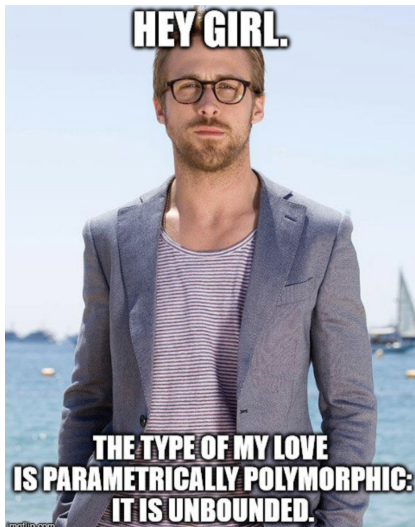
One needs to have a way to avoid such a boilerplate.

Parametric polymorphism

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

```
1  id :: a -> a
2  id x = x
3
4  const :: a -> b -> a
5  const a b = a
6
7  fst :: (a, b) -> a
8  fst (a, b) = a
9
10 snd :: (a, b) -> b
11 snd = "guess what"
12
13 swap :: (a, b) -> (b, a)
14 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'
```


Higher order functions and parametric polymorphism

```
1 infixr 9 .  
2 (.) :: (b -> c) -> (a -> b) -> a -> c  
3 f . g = \x -> f (g x)  
4  
5 flip :: (a -> b -> c) -> b -> a -> c  
6 flip f b a = f a b  
7  
8 fix :: (a -> a) -> a  
9 fix = error "this is your homework"  
10  
11 curry :: ((a, b) -> c) -> a -> b -> c  
12 curry f x y = f (x, y)  
13  
14 uncurry :: (a -> b -> c) -> ((a, b) -> c)  
15 uncurry f p = f (fst p) (snd p)
```

The functions above in the GHCi session. The composition examples

```
1 incNegate :: Int -> Int
2 incNegate x = negate (x + 1)
3
4 incNegate x = negate $ x + 1
5
6 incNegate x = (negate . (+1)) x
7
8 incNegate x = negate . (+1) $ x
9
10 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

```
1  show2 :: Int -> Int -> String
2  show2 x y = show x ++ " and " ++ show y
3
4  showSnd, showFst, showFst' :: Int -> String
5  showSnd = show2 1
6  showFst  = flip show2 2
7  showFst' = ('show2' 2)
```

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

```
1 show2 :: Int -> Int -> String
2 show2 x y = show x ++ " and " ++ show y
3
4 showSnd, showFst, showFst' :: Int -> String
5 showSnd = show2 1
6 showFst = flip show2 2
7 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

```
1  changeTwiceBy :: (Int -> Int) -> Int -> Int
2  changeTwiceBy operation value = operation (operation value)
3
4  changeTwiceByBool :: (Bool -> Bool) -> Bool -> Bool
5  changeTwiceByBool operation value = operation (operation value)
6
7  changeTwiceByString :: (String -> String) -> String -> String
8  changeTwiceByString operation value = operation (operation value)
```

might be replaced to the following ones:

```
1  applyTwice :: (a -> a) -> a -> a
2  applyTwice f a = f (f a)
3
4  applyTwice' :: (a -> a) -> a -> a
5  applyTwice' f a = f . f $ a
6
7  applyTwice'' :: (a -> a) -> a -> a
8  applyTwice'' f = f . f
```

HOF, polymorphism, and lists

```
1  map    :: (a -> b)    -> [a] -> [b]
2
3  filter  :: (a -> Bool) -> [a] -> [a]
4
5  zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
6
7  foldl   :: (b -> a -> b) -> b -> [a] -> b
8
9  foldr   :: (a -> b -> b) -> b -> [a] -> b
10
11 length :: [a] -> Int
```

HOF, polymorphism, and lists

```
1  map    :: (a -> b)    -> [a] -> [b]
2
3  filter  :: (a -> Bool) -> [a] -> [a]
4
5  zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
6
7  foldl   :: (b -> a -> b) -> b -> [a] -> b
8
9  foldr   :: (a -> b -> b) -> b -> [a] -> b
10
11 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

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

The composition examples + list functions

```
1 stringsTransform :: [String] -> [String]
2 stringsTransform l = map (\s -> map toUpper s) (filter (\s -> length s == 5) l)
3
4 stringsTransform l = map (\s -> map toUpper s) $ filter (\s -> length s == 5) l
5
6 stringsTransform l = map (map toUpper) $ filter ((== 5) . length) l
7
8 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 at the following function

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

Type classes. Motivation

- Let us take a look at the following function

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

- Is a type `a` arbitrary?

Type classes. Motivation

- Let us take a look at the following function

```
1  elem :: a -> [a] -> Bool
2  elem _ [] = False
3  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.

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:

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

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

```
1  elem :: Eq a => a -> [a] -> Bool
2  elem _ [] = False
3  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:

```
1  instance Eq Bool where
2      True == True = True
3      False == False = True
4      _ == _ = False
5
6  x /= y = neg (x == y)
```

Polymorphism + instance declarations

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

```
1  instance Eq a => Eq [a] where
2      []      == []      = True
3      (x : xs) == (y : ys) = x == y && xs == ys
4      _       == _       = 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)

```
1  instance Eq a => Eq [a]
2  instance Eq Word
3  instance Eq Ordering
4  instance Eq Int
5  instance Eq Float
6  instance Eq Double
7  instance Eq Char
8  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:

```
1  class Show a where
2      showsPrec :: Int -> a -> ShowS
3      show :: a -> String
4      showList :: [a] -> ShowS
5      {-# 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:

```
1  instance Show Integer
2  instance Show Int
3  instance Show Char
4  instance Show Bool
5  instance (Show a, Show b) => Show (a, b)
```

Ordering. Motivation

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

```
1  quicksort :: [a] -> [a]
2  quicksort [] = []
3  quicksort (x:xs) = quicksort small ++ (x : quicksort large)
4  where
5      small = [y | y <- xs, y <= x]
6      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:

```
1  class Eq a => Ord a where
2      compare :: a -> a -> Ordering
3      (<), (<=), (>), (>=) :: a -> a -> Bool
4      max, min :: a -> a -> a
5
6      compare x y = if x == y then EQ
7                    else if x <= y then LT
8                    else GT
9
10     x <= y = case compare x y of { GT -> False; _ -> True }
11
12     max x y = if x <= y then y else x
13
14     {-# MINIMAL compare | (<=) #-}
```

Some of the Ord instances

```
1  instance Ord Word
2  instance Ord Int
3  instance Ord Float
4  instance Ord Double
5  instance Ord Char
6  instance Ord Bool
```


The Num type class

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

```
1  class Num a where
2    (+), (-), (*) :: a -> a -> a
3    negate, abs, signum :: a -> a
4    fromInteger :: Integer -> a
5    {-# 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:

- 1 **instance Num Word**
- 2 **instance Num Integer**
- 3 **instance Num Int**
- 4 **instance Num Float**
- 5 **instance Num Double**

The Enum and Bounded type classes

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

```
1  class Enum a where
2    succ, pred :: a -> a
3    toEnum    :: Int -> a
4    fromEnum  :: a -> Int
5
6    enumFrom   :: a -> [a]           -- [n..]
7    enumFromThen :: a -> a -> [a]    -- [n,m..]
8    enumFromTo   :: a -> a -> [a]    -- [n..m]
9    enumFromThenTo :: a -> a -> a -> [a] -- [n,m..p]
10   {-# MINIMAL toEnum, fromEnum #-}
```

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

```
1  class Bounded a where
2    minBound :: a
3    maxBound :: a
4    {-# MINIMAL minBound, maxBound #-}
```

Some of the Enum instances

- 1 **instance Enum Word**
- 2 **instance Enum Integer**
- 3 **instance Enum Int**
- 4 **instance Enum Char**
- 5 **instance Enum Bool**
- 6 **instance Enum Float**
- 7 **instance Enum Double**

Some of the Bounded instances

- 1 **instance Bounded Word**
- 2 **instance Bounded Int**
- 3 **instance Bounded Char**
- 4 **instance Bounded Bool**

The Fractional type class

```
1 class Num a => Fractional a where
2   (/) :: a -> a -> a
3   recip :: a -> a
4   fromRational :: Rational -> a
5   {-# MINIMAL fromRational, (recip | (/)) #-}
```

Summary