

Functional programming, Seminar No. 3

Danya Rogozin

Institute for Information Transmission Problems, RAS

Serokell OÜ

Higher School of Economics

The Faculty of Computer Science

Intro

Today we study



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)
```

One may implement the function for Boolean values and strings that are the 'same' as the function above:

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

```
changeTwiceBy :: (String -> String) -> String -> String
changeTwiceBy 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)
```

One may implement the function for Boolean values and strings that are the 'same' as the function above:

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

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

Too much 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
```

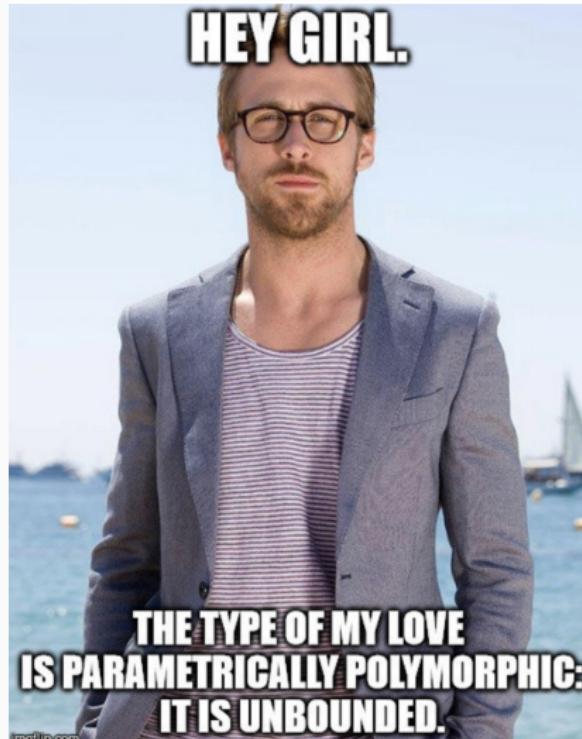
```
snd :: (a, b) -> b
```

```
snd = "guess what"
```

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

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

Meme time



memesfun.com

Example

GHCi session

```
> id 6
6
> id 6.0
6.0
> const True 6
True
> const 6 True
6
> fst ('5', 5)
'5'
> fst (5, '5')
5
```

A brief clarification

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

```
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 polymorphism

More examples

```
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)
```

Examples with composition

More 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)
```

curry **and** uncurry

More examples

```
> uncurry (*) (3,4)
12
> curry fst 3 4
3
> curry id 3 4
(3,4)
> uncurry const (3,4)
3
> uncurry (flip const) (3,4)
4
```

Examples with flip

More examples

```
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)
```

GHCI session

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

Bye-bye boilerplate!

All those functions such as the following one

```
changeTwiceBy with Int
```

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

might be replaced by one of the following ones:

```
applyTwice
```

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

HOF, polymorphism, and lists

Examples

```
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 discuss them briefly.

The composition examples + list functions

More examples

```
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

More examples

```
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.

More examples

```
> :t 9
9 :: Num p => p
> 9 :: Int
9
> 9 :: Double
9.0
> 9 :: Rational
9 % 1
> 9 :: Char
<interactive>:6:1: error:
* No instance for (Num Char) arising from the literal '9'
```

Type classes. Motivation

- Let us take a look at the following function

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

- a is an arbitrary type for which equality is defined as usual.

Type classes. Motivation

Type variables in polymorphic function are bounded with universal quantifier. In ad hoc polymorphism, type variables are also bounded with \forall but with the additional condition. This kind of 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. An 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
```

- The definition above without a context is not well-defined:

```
<interactive>:4:19: 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
```

Instance declarations

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

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 as well:

Example

```
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 of the Eq instances

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

Eq instances

```
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 have a look at the implementation.

The Show type class

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

Some Eq instances

```
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 show a value of a given type.

Some of the Show instances

Here are some of the Show instances:

Some 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:

Some Show instances

```
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:

Some Show instances

```
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 situation. The definition of quicksort as above is wrong. There exist types elements of which are incomparable, complex numbers, e.g.

The Ord type class

The full definition of Ord is the following one:

Ord

```
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 }

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

Ord instances

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 arithmetic operations.

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

```

Num instances

Some Num instances

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

The `Enum` type class

The `Enum` is type class for types for which we can define an explicit enumeration.

Some `Enum` instances

```
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 #-}
```

Enum instances

Some Num instances

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

The Fractional type class

- The Fractional type class is a general interface for division

Some Num instances

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

- We require the Num a restriction.
- The Fractional instances are Float and 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 evaluation in Haskell