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:

```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)
```

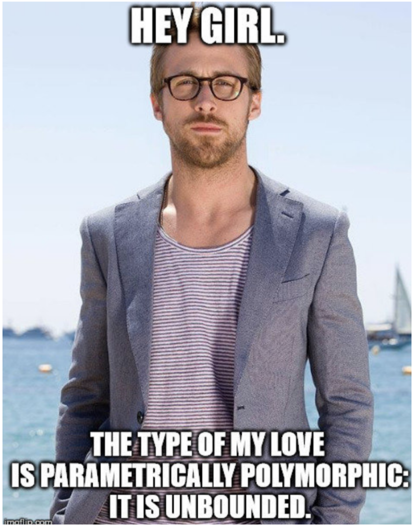
One needs to have a way to avoid such a boilerplate.

Parametric polymorphism

The key idea of parametric polymorphism that the same function might be called on distinct data types. Here are the initial 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'
```

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 = error "this is your homework"

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

```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)
```

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

Bye-bye boilerplate!

All these functions

```
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:

```
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

```
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

```haskell
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)
```

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)
```

Restricted strictness

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
```

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
```

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
```

The `Eq` type class generally

The `Eq` type class is a type class that allows one to

The Show type class

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

The Ord type class

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
```

The Num type class

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

The Enum and Bounded type classes

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

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

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

The Fractional type class

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

Summary