Functional programming, Seminar No. 2

Daniel Rogozin
Institute for Information Transmission Problems, RAS
Serokell OÜ

Higher School of Economics
The Faculty of Computer Science

Intro

On the previous seminar, we:

- discussed the general aspects of Haskell
- took a look at the Haskell ecosystem

Intro

On the previous seminar, we:

- discussed the general aspects of Haskell
- took a look at the Haskell ecosystem

Today, we:

- study the basic Haskell syntax
- study the list data type as one of the Haskell data structures
- realise why Haskell is a lazy language

Bindings

The equality sign in Haskell denotes binding:

Example

```haskell
fortyTwo = 42
coolString = "coolString"
```

Local binding with the `let`-keyword:

Example

```haskell
fortyTwo = let number = 43 in number - 1
```

Function definitions

The following functions are also defined as bindings:

Example

```haskell
add x y = x + y
userName name = "Username: " ++ name
id x = x
```

The same functions defined with lambda:

Example

```haskell
add = \x y -> x + y
userName = \name -> "Username: " ++ name
id = \x -> x
```

Function application

As in the lambda calculus, function application is left associative by default

Example

```
{-
foo x y z = f x y z = ((f x) y) z
-}
```

One may use the dollar infix operator to reduce the overuse of brackets. For example, the functions `function` and `function1` are equivalent:

Example

```
function f x y z = f ((x y) z)
function1 f x y z = f $ x y $ z
```

Prefix and infix notation

Every operator or function is prefix and infix in the following sense:

Example

```
> map (\x -> x * pi * 100) [1..3]
[314.1592653589793,628.3185307179587,942.4777960769379]
> (\x -> x * pi * 100) `map` [1..3]
[314.1592653589793,628.3185307179587,942.4777960769379]
```

One can declare an operator defining its priority and associativity explicitly. Here is an example:

Example

```
(^) :: (Num a, Integral b) => a -> b -> a
infixr 8 ^
```

Currying and partial application

Recall the function `add` once more. Here is an example of partial application:

Example

```
add x y = x + y
addFive = add 5
twentyEight = addFive 23
  -- 28
```

Partial application is well-defined since all many-argument functions in Haskell are curried by default.

Immutability and laziness

In Haskell, values are immutable. A small example:

Example

```
> list = [1,2,3,4]
> reverse list
[4,3,2,1]
> list
[1,2,3,4]
> 10 : list
[10,1,2,3,4]
> list
[1,2,3,4]
```

Recursion

The straighforward factorial and the tail-recursive one:

Example

```
factorial n
  = if n == 0 then 1 else n * factorial (n - 1)

tailFactorial n = helper 1 n
  where
  helper acc x =
    if x > 1
    then helper (acc * x) (x - 1)
    else acc
```

Guards

Let us take a look at the factorial implementation with guards:

Example

```
tailFactorial n = helper 1 n
  where
  helper acc x | x > 1 = helper (acc * x) (x - 1)
               | otherwise acc
```

Basic types

The basic types are:

- `Bool`
- `Int`
- `Integer`
- `Char`
- `()`
- If `a` and `b` are types, then `a -> b` is a type
- If `a` and `b` are types, then `(a,b)` is a type
- If `a` is a type, then `[a]` is a type

A type declaration has the following form:

```
term :: type
```

Datatypes and constructors

We take the list of basic data types and associate constructors with these types. A constructor is a term that allows one to obtain a value of a given type.

`Bool`	`True` and `False`
`Int`	Integers from -2^{29} to $2^{29}-1$
`Integer`	The set of integers
`Char`	Characters `'0'`, ..., `'9'`, `'a'`, ..., `'z'`, etc
`()`	`()` only
`a -> b`	$\lambda x \to m$
`(a,b)`	if `x :: a` and `y :: b`, then `(x, y) :: (a,b)`
`[a]`	the empty list `[]`
`[a]`	if `x :: a` and `xs :: [a]`, then `x : xs :: [a]`

Types in GHCi

Use the GHCi command `:t` to get a type of an expression:

Example

```
> :t 5
5 :: Num p => p
> :t not
not :: Bool -> Bool
> :t [0.5, 0.6, 0.7]
[0.5, 0.6, 0.7] :: Fractional a => [a]
> :t (\x -> "dratuti, " ++ x)
(\x -> "dratuti, " ++ x) :: [Char] -> [Char]
> :t 'x'
'x' :: Char
```

Function declaration with datatypes

Let us recall the examples:

Example

```haskell
add x y = x + y
userName name = "Username: " ++ name
```

One may annotate these functions with type signatures as follows:

Example

```haskell
add :: Int -> Int -> Int
add x y = x + y

userName :: String -> String
userName name = "Username: " ++ name
```

Lists

In Haskell, a list is a homogeneous collection of elements.

Example

```haskell
empty :: [Int]
empty = []

ten :: [Int]
ten = [10]

tenEleven :: [Int]
tenEleven = 11 : ten

tenElevenTwelve :: [Int]
tenElevenTwelve = 12 : tenEleven
-- 12 : (11 : [])
```

Lists. Ranges

Example

```
oneToFive :: [Int]
oneToFive = [1..5]

oneToSevenOdd :: [Int]
oneToSevenOdd = [1,3..7]

nat :: [Int]
nat = [0,1..]

evens :: [Int]
evens = [0,2,4..]
```

Lists. Heads and Tails

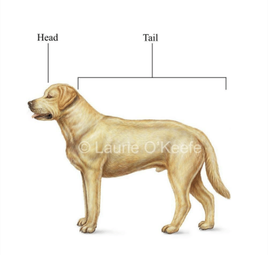
Example

```
> tail [1..3]
[2,3]
> head [1..3]
1
> head []
*** Exception: Prelude.head: empty list
> tail []
*** Exception: Prelude.tail: empty list
```

Lists. Heads and Tails

Example

```
> tail [1..3]
[2,3]
> head [1..3]
1
> head []
*** Exception: Prelude.head: empty list
> tail []
*** Exception: Prelude.tail: empty list
```



Other helpful list functions

Example

```
Prelude> drop 3 [1..7]
[4,5,6,7]
Prelude> take 4 ['a'..'h']
"abcd"
Prelude> replicate 3 "d"
["d","d","d"]
Prelude> replicate 3 'd'
"ddd"
Prelude> zip [1,2,3] "this is a word"
[(1,'t'),(2,'h'),(3,'i')]
Prelude> unzip [(1,'t'),(2,'h'),(3,'i')]
([1,2,3],"thi")
Prelude> ['a'..'h'] !! 3
'd'
```

List compeherension

Example

```
> take 4 [(i, j) | i <- [1..10], j <- [1..10], i == j*j]
[(1,1),(4,2),(9,3),(16,4)]
> [ i | i <- "a cool sentence", i < 'h']
"a c eece"
> [ i | i <- "a cool sentence", fromEnum i < 100 ]
"a c c"
```

Higher order functions

A function is a first-class object and one may pass any function as an argument:

Example

```
inc :: Int -> Int
inc x = x + 1

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

seven :: Int
seven = changeTwiceBy inc 5
```

Case-expressions

Case-expressions allows one to perform case analysis within a function body.

Example

```
getFont :: Int -> String
getFont n =
  case n of
    0 -> "PLAIN"
    1 -> "BOLD"
    2 -> "ITALIC"
    _ -> "UNKNOWN"
```

Reduction strategies

We recall a couple of definitions related to lambda calculus:

Reduction strategies

We recall a couple of definitions related to lambda calculus:

1. A term M is called *weakly normalisable* (WN), if there exists some halting reduction path that starts from M

Reduction strategies

We recall a couple of definitions related to lambda calculus:

1. A term M is called *weakly normalisable* (WN), if there exists some halting reduction path that starts from M
2. A term M is called *strongly normalisable* (SN), if any reduction path that starts from M terminates

Reduction strategies

We recall a couple of definitions related to lambda calculus:

1. A term M is called *weakly normalisable* (WN), if there exists some halting reduction path that starts from M
2. A term M is called *strongly normalisable* (SN), if any reduction path that starts from M terminates

It is clear, that SN implies WN, not vice versa. In other words, there exists a term with an infinite reduction path, but it has a finite one at the same time.

Reduction strategies

Let us consider the example of the following ridiculous term: $(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$. One may reduce this term in two ways:

Reduction strategies

Let us consider the example of the following ridiculous term: $(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$. One may reduce this term in two ways:

From the one hand:

$$
\begin{array}{l}
(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
(\lambda y.[x := (\lambda z.z)])((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
(\lambda y.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
(\lambda z.z)[y := (\lambda x.xx)(\lambda x.xx)] \to_\beta \\
\lambda z.z
\end{array}
$$

Reduction strategies

Let us consider the example of the following ridiculous term: $(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$. One may reduce this term in two ways:

From the one hand:

$$
\begin{aligned}
&(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \\
&(\lambda y.[x := (\lambda z.z)])((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \\
&(\lambda y.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \\
&(\lambda z.z)[y := (\lambda x.xx)(\lambda x.xx)] \rightarrow_\beta \\
&\lambda z.z
\end{aligned}
$$

From the other hand:

$$
\begin{aligned}
&(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \\
&(\lambda xy.x)(\lambda z.z)(xx)(x := [\lambda x.xx]) \rightarrow_\beta \\
&(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \ldots
\end{aligned}
$$

Reduction strategies

Let us consider the aforementioned example in some other aspects.

Reduction strategies

Let us consider the aforementioned example in some other aspects.

- In the first case, we have got a sensible result by several reduction steps. On the other hand, we have a loop in the second case.

Reduction strategies

Let us consider the aforementioned example in some other aspects.

- In the first case, we have got a sensible result by several reduction steps. On the other hand, we have a loop in the second case.
- Also, in the first case, we start our reduction from the leftmost innermost redex. But when we were trying to reduce the term $(\lambda x.xx)(\lambda x.xx)$, we have seen that something went wrong.

Reduction strategies

Let us consider the aforementioned example in some other aspects.

- In the first case, we have got a sensible result by several reduction steps. On the other hand, we have a loop in the second case.
- Also, in the first case, we start our reduction from the leftmost innermost redex. But when we were trying to reduce the term $(\lambda x.xx)(\lambda x.xx)$, we have seen that something went wrong.
- The moral is that the order of reduction matters.

Reduction strategies

Let us consider the aforementioned example in some other aspects.

- In the first case, we have got a sensible result by several reduction steps. On the other hand, we have a loop in the second case.
- Also, in the first case, we start our reduction from the leftmost innermost redex. But when we were trying to reduce the term $(\lambda x.xx)(\lambda x.xx)$, we have seen that something went wrong.
- The moral is that the order of reduction matters.

In fact, we need to distinguish possible ways of application reduction, so far as we have no other options in the remaining cases:

1. If x is a variable, then x is already in normal form
2. If a term has the form $\lambda x.M$, then we reduce M

Reduction strategies

Thus, one needs to analyse the possible ways of application reduction. We have the following alternatives:

Reduction strategies

Thus, one needs to analyse the possible ways of application reduction. We have the following alternatives:

1. $(\lambda x_1 \ldots x_n.M)N_1 \ldots N_n$: we firtsly reduce $(N_i)_{i\in\{1,\ldots,n\}}$
2. $(\lambda x_1 \ldots x_n.M)N_1 \ldots N_n$: reduce $(\lambda x.M)N_1$ and go further from left to right

Reduction strategies

Thus, one needs to analyse the possible ways of application reduction. We have the following alternatives:

1. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: we firtsly reduce $(N_i)_{i \in \{1,\dots,n\}}$
2. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: reduce $(\lambda x.M)N_1$ and go further from left to right

The first way is called *applicative order reduction*, the second one is normal order reduction. In the following sense, the second is better:

Thus, one needs to analyse the possible ways of application reduction. We have the following alternatives:

1. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: we firtsly reduce $(N_i)_{i \in \{1,\dots,n\}}$
2. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: reduce $(\lambda x.M)N_1$ and go further from left to right

The first way is called *applicative order reduction*, the second one is normal order reduction. In the following sense, the second is better:

Theorem
Let M be a term such that M has a normal form M', then M can be reduced to M' using normal order reduction.

Theoretical Flashback. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)

Theoretical Flashback. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)
- The most mainstream programming languages you know (Java, Python, Kotlin, etc) have the call-by-value semantics

Theoretical Flashback. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)
- The most mainstream programming languages you know (Java, Python, Kotlin, etc) have the call-by-value semantics
- The Haskell reduction has a call-by-need strategy which is quite close to call-by-name. Informally, such a stragety is called *lazy*. Laziness denotes that Haskell does not compute a value if it is not needed at the moment

Theoretical Flashback. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)
- The most mainstream programming languages you know (Java, Python, Kotlin, etc) have the call-by-value semantics
- The Haskell reduction has a call-by-need strategy which is quite close to call-by-name. Informally, such a stragety is called *lazy*. Laziness denotes that Haskell does not compute a value if it is not needed at the moment
- Call-by-name reduction reduces reducible terms to the bitter end, but it is not always optimal, unfortunately

Haskell reduction

Suppose we have the following trivial function:

Example

```
square :: Int -> Int
square x = x * x
```

Haskell reduction

Suppose we have the following trivial function:

Example

```
square :: Int -> Int
square x = x * x
```

If we call this function on $(1+2)$, then we would have the following story:

$$\text{square}\,(1+2) = (1+2)*(1+2) = 3*(1+2) = 3*3 = 9$$

Haskell reduction

Suppose we have the following trivial function:

Example

```
square :: Int -> Int
square x = x * x
```

If we call this function on $(1+2)$, then we would have the following story:

$\text{square}\,(1+2) = (1+2)*(1+2) = 3*(1+2) = 3*3 = 9$ We evalutate $(1+2)$ twice, even if we know that $1+2=3$ a priori.

Haskell reduction

Suppose we have the following trivial function:

Example

```
square :: Int -> Int
square x = x * x
```

If we call this function on $(1+2)$, then we would have the following story:

$\text{square}\,(1+2) = (1+2)*(1+2) = 3*(1+2) = 3*3 = 9$ We evalutate $(1+2)$ twice, even if we know that $1+2=3$ a priori. The question of performance is still relevant.

The notion of a weak head normal form

In Haskell, reduction evalutates a term to its weak head normal form, where the outermost must be either constructor or lambda. Here are examples: WHNFs from the left and non-WHNFs from the right

```haskell
78
```

`2 : [1,2]`	`1 + 665`
`'p' : ("ri" ++ "vet")`	`(\x -> x ++ "ab") "cd"`
`[1, 1 + 2, 1 + 3]`	`length [1..145]`
`("hel" ++ "lo", "world")`	`(\f g x -> f (g x)) id`
`\x -> (x + 2) + 2`	

Pure functions and side-effects

Pure functions and side-effects

- A function is called *pure* if it has no side effects

Pure functions and side-effects

- A function is called *pure* if it has no side effects
- It means that such a function behaves ’in the same way’ at every point. This principle is also called *referential transparency*

Pure functions and side-effects

- A function is called *pure* if it has no side effects
- It means that such a function behaves 'in the same way' at every point. This principle is also called *referential transparency*
- A side-effect function is a function that may yield different values for the same arguments. Mathematically, such a function is not function at all. This is rather a procedure.

Pure functions and side-effects

- A function is called *pure* if it has no side effects
- It means that such a function behaves 'in the same way' at every point. This principle is also called *referential transparency*
- A side-effect function is a function that may yield different values for the same arguments. Mathematically, such a function is not function at all. This is rather a procedure.
- Haskell functions are (mostly) pure ones, but Haskell is not confluent as a version of the lambda calculus

The failure of the Church-Rosser property

Let us consider the following quite simple example. In Haskell one has a function called `seq`. According to Hackage, "The value of `seq a b` is bottom if a is bottom, and otherwise equal to b." This function is a sort of instrument to introduce the restricted strictness to Haskell. The listing below demostrates the failure of the CRP:

```
seq :: a -> b -> b
seq _|_ _ = _|_
seq _ b   = b

bottom = undefined

seq bottom 14          == bottom
seq (bottom  . id) 14 == 14
```

Finally

On this seminar, we

- got acquinted with the basic Haskell syntax and basic data types
- discussed pure functions and the example of the confluence failure

Finally

On this seminar, we

- got acquinted with the basic Haskell syntax and basic data types
- discussed pure functions and the example of the confluence failure

On the next seminar, we will

- start to learn polymorphism and its advantages
- introduce typeclasses
- study the very first examples of typeclasses

Thank you!