Functional programming, Seminar No. 2

Danya Rogozin
Lomonosov Moscow State University,
Serokell OÜ

Higher School of Economics
Faculty of Computer Science

Bindings

The equality sign in Haskall denotes binding:

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

Local binding with the `let`-keyword:

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

Function definitions

Functions are also defined as bindings:

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

The same functions defined via lambda:

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

Function application

As in lambda calculus, function application is right associative by default

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

One may use the dollar infix operator in order to avoid brackets overuse. For example, the following functions have exactly the same behaviour:

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

Prefix and infix notation

Any operator or function might be called in prefix and infix:

```
Prelude> map (\x -> x * pi) [1..5]
[3.141592653589793,6.283185307179586,9.42477796076938,12.566370614359172,15.707963267948966]
Prelude> (\x -> x * pi) `map` [1..5]
[3.141592653589793,6.283185307179586,9.42477796076938,12.566370614359172,15.707963267948966]
Prelude> (+) 19 76
95
Prelude> 19 + 76
95
```

One may declare an operator defining its priority and associativity explicitly. Here's an example:

```
(&&) :: Bool -> Bool -> Bool
infixr 3 &&
```

Currying and partial application

Let us recall the function `add` once more:

```
  add x y = x + y
```

Here is an example of a partial application in the following GHCi session

```
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. The GHCi example:

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

Recursion

The straightforward recursion:

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

The factorial function implemented via so-called tail recursion:

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

Guards

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

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

Basic datatypes

The basic datatypes are:

- `Bool`: Boolean values
- `Int`: Bounded integer datatype
- `Integer`: Unbounded integer datatype
- `Char`: Unicode characters
- `()`: Unit value datatype
- 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

Let us recall the list of basic data types above and associative constructors with them. A constructor is a term that allows one to obtain a value of the desired type.

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

Types in GHCi

The command `:t` yields a type of a required expression:

```
Prelude> :t 5
5 :: Num p => p
Prelude> :t (||)
(||) :: Bool -> Bool -> Bool
Prelude> :t [0.5,0.6]
[0.5,0.6] :: Fractional a => [a]
Prelude> :t (\x -> x ++ "guten tag mein ")
(\x -> x ++ "guten tag mein ") :: [Char] -> [Char]
Prelude> :t '/'
'/' :: Char
```

Function declaration with datatypes

Let us recall the examples of function declarations:

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

One may annotate these functions with types as follows:

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

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

id :: Char -> Char
id x = x
```

Note that such calls as `userName 5` or `id ''hello stewart''` cause type errors.

Lists

Let's talk about lists a slightly closely. In Haskell, a list is a homogeneous collection of elements.

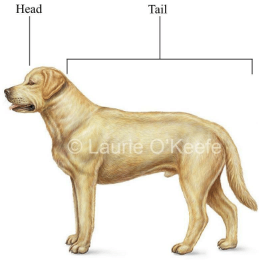
```
empty :: [Int]
empty = []

ten  :: [Int]
ten  = [10]

tenEleven  :: [Int]
tenEleven  = 11 : ten

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

Lists. Ranges

```
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

```
Prelude> tail [1..5]
[2,3,4,5]
Prelude> head [1..5]
1
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> tail []
*** Exception: Prelude.tail: empty list
```



Dogs according to Haskell

Other helpful list functions

```
Prelude> drop 3 [1..5]
[4,5]
Prelude> take 4 [1..10]
[1,2,3,4]
Prelude> replicate 3 "dratuti"
["dratuti","dratuti","dratuti"]
Prelude> zip [1,2,3] "abc"
[(1,'a'),(2,'b'),(3,'c')]
Prelude> unzip [(1,'a'),(2,'b'),(3,'c')]
([1,2,3],"abc")
Prelude> words "Anna Daniel \t\n\n LeonidYakubovich"
["Anna","Daniel","LeonidYakubovich"]
Prelude> [34..72] !! 7
41
```

List compeherension

```
Prelude> [ (i, j) | i <- [1..10], j <- [2..12], j - i < 4, j + i > 20 ]
[(9,12),(10,11),(10,12)]
Prelude> take 10 [ (i,j) | i <- [1..], j <- [1..i-1], gcd i j == 1 ]
[(2,1),(3,1),(3,2),(4,1),(4,3),(5,1),(5,2),(5,3),(5,4),(6,1)]
Prelude> [ c | c <- "the picture of dorian grey", c < 'o']
"he ice f dian ge"
Prelude> [ c | c <- "the picture of dorian grey", c < 'o', fromEnum c < 104 ]
"e ce f da ge"
```

Higher order functions

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

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

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

seven :: Int
seven = changeTwiceBy inc 5

three :: Int
three = changeTwiceBy dec 5
```

Case-expressions

Case-expressions allows one to perform case analysis within an observed function.

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

Theoretical Flashback. Reduction strategies

Here we recall some of relatable definition from 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, that has an infinite reduction path, but it has a finite reduction path.

Theoretical Flashback. 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:

Theoretical Flashback. 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:

$(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$
$(\lambda y.[x := (\lambda z.z)])((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$
$(\lambda y.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$
$(\lambda z.z)[y := (\lambda x.xx)(\lambda x.xx)] \rightarrow_\beta$
$\lambda z.z$

Theoretical Flashback. 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:

$(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$
$(\lambda y.[x := (\lambda z.z)])((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$
$(\lambda y.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$
$(\lambda z.z)[y := (\lambda x.xx)(\lambda x.xx)] \rightarrow_\beta$
$\lambda z.z$

From the other hand:

$(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$
$(\lambda xy.x)(\lambda z.z)(xx)(x := [\lambda x.xx]) \rightarrow_\beta$
$(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \dots$
spasiti pamagiti pajalusta ya tak bolshe ni magu (((((((9(99(

Theoretical Flashback. Reduction strategies

Let us consider the example above in some other aspects.

Theoretical Flashback. Reduction strategies

Let us consider the example above 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.

Theoretical Flashback. Reduction strategies

Let us consider the example above 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 started our reduction from the leftmost innermost redex. When we tried to start our reduction from the right redex $(\lambda x.xx)(\lambda x.xx)$, we have found ourselves in a spot of trouble. Something went wrong.

Theoretical Flashback. Reduction strategies

Let us consider the example above 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 started our reduction from the leftmost innermost redex. When we tried to start our reduction from the right redex $(\lambda x.xx)(\lambda x.xx)$, we have found ourselves in a spot of trouble. Something went wrong.
- Can we distinguish all possible ways of term reduction?

Theoretical Flashback. Reduction strategies

Let us consider the example above 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 started our reduction from the leftmost innermost redex. When we tried to start our reduction from the right redex $(\lambda x.xx)(\lambda x.xx)$, we have found ourselves in a spot of trouble. Something went wrong.
- Can we distinguish all possible ways of term reduction?

In a matter of fact, we need to distinguish all 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

Theoretical Flashback. Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

Theoretical Flashback. Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

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

Theoretical Flashback. Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

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

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

Theoretical Flashback. Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

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

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

Theorem

Let M be a term such that M has a normal form M', then M might be reduced to M' via normal order

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 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 call-by-value semantics
- The Haskell reduction has a call-by-value strategy. Informally, such a stragety is called *lazy*. Laziness denotes that Haskell doesn't compute a value if it's 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 call-by-value semantics
- The Haskell reduction has a call-by-value strategy. Informally, such a stragety is called *lazy*. Laziness denotes that Haskell doesn't compute a value if it's not needed at the moment
- Call-by-name reduction reduces reducible terms to the bitter end, but it's not always optimal, unfortunately

Haskell reduction

Suppose we have such a trivial function:

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

Haskell reduction

Suppose we have such a trivial function:

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

If we call this function on $(1+2)$, then we would have the following story:
square $(1+2) = (1+2) * (1+2) = 3 * (1+2) = 3 * 3 = 9$

Haskell reduction

Suppose we have such a trivial function:

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

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

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

If we call this function on $(1 + 2)$, then we would have the following story:
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. I sincerely hope that you have no doubts about it.

Haskell reduction

Suppose we have such a trivial function:

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

If we call this function on $(1+2)$, then we would have the following story:
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. I sincerely hope that you have no doubts about it. The question of optimality is still relevant.

The notion of a weak head normal form

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

```haskell
78

2 : [1,2]

'p' : ("ri" ++ "vet")

[1, 1 + 2, 1 + 3]

("hel" ++ "lo", "world")

\x -> (x + 2) + 2

\xs -> zip xs [1, 3+2]
```

```haskell
1 + 665

(\x -> x ++ "guten tag mein herr ") "Heinrich"

length [1..145]

(\f g x -> f (g x)) $ (\x y -> y)
```

Pure functions and side-effects

- A function is called *pure* if it yields the same value for the same argument each time
- In other words, a pure function is a function that satisfies Church-Rosser property

Pure functions and side-effects

- A function is called *pure* if it yields the same value for the same argument each time
- In other words, a pure function is a function that satisfies Church-Rosser property
- It means that, such a function has the same behaviour at every point. This principle is also called *referential transparency*

Pure functions and side-effects

- A function is called *pure* if it yields the same value for the same argument each time
- In other words, a pure function is a function that satisfies Church-Rosser property
- It means that, such a function has the same behaviour at every point. This principle is also called *referential transparency*
- A side-effect function is a function that may yield different value passing the same arguments. Mathematically, such a function is not function at all.

Pure functions and side-effects

- A function is called *pure* if it yields the same value for the same argument each time
- In other words, a pure function is a function that satisfies Church-Rosser property
- It means that, such a function has the same behaviour at every point. This principle is also called *referential transparency*
- A side-effect function is a function that may yield different value passing the same arguments. Mathematically, such a function is not function at all.
- Haskell functions are (mostly) pure ones, but Haskell isn't confluent as a version of lambda calculus

The failure of confluece

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." The listing below demostrates the failure of confluence:

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

dno = undefined

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

Finally

On this seminar, we

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

On the next seminar, we

- start to learn polymorphism and its advantages
- introduce typeclasses
- study the first examples of crucially important examples of type classes