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:

```
1  fortyTwo = 42
2  coolString = "coolString"
```

Local binding with the `let`-keyword:

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

Function definitions

Functions are also defined as bindings:

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

The same functions defined via lambda:

```
1  add = \x y -> x + y
2  userName = \name -> "Username: " ++ name
3  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
```

Currying and partial application

Let us recall the function add once more:

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

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

```
1 add x y = x + y
2 addFive = add 5
3 twentyEight = addFive 23
4   -- 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. For example, the following code doesn't terminate since it yields an infinite loop in contrast to Python or JS:

```
1  x = 5
2  x = x + x
```

The following program doesn't fail since our semantics is lazy:

```
1  seventyTwo = const 72 undefined
```

Recursion

The straightforward recursion:

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

The factorial function implemented via so-called tail recursion:

```
1  tailFactorial n = helper 1 n
2    where
3    helper acc x =
4      if x > 1
5      then then helper (acc * x) (x - 1)
6      else acc
```

Guards

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

```
1  tailFactorial  n = helper 1 n
2    where
3    helper acc x | x > 1 = helper (acc * x) (x - 1)
4                 | 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:

```
1    term :: type
```

Function declaration with datatypes

Let us recall the examples of function declarations:

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

One may annotate these functions with types as follows:

```
1  add :: Int -> Int -> Int
2  add x y = x + y
3
4  userName :: String -> String
5  userName name = "Username: " ++ name
6
7  id :: Char -> Char
8  id x = x
```

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

Another aspect of laziness: the notion of a weak head normal norm