Functional Programming

Vsevolod Nikulin

Okinawa Institute of Science and Technology

vsevolod.nikulin@oist.jp

November 26, 2020

Overview

1. Algebraic Data Types
2. Monads

Algebraic Data Types

Type Definitions

New types are defined with keyword **data**.

Example: The simplest type

```
data Singleton = Unit
```

Here **Singleton** is a *type constructor* and **Unit** is a *data constructor*. It is a legal term.

GHCi session

```
let x = Unit
:t x
```

Sum Type

We can define more then one data constructor for a new type.

Example: Simple sum type

$$\textbf{data Bool} = \textbf{True} \mid \textbf{False}$$
$$\textbf{data}\ \mathsf{Four} = \mathsf{Zero} \mid \mathsf{One} \mid \mathsf{Two} \mid \mathsf{Three}$$

Built-in type **Bool** is defined exactly in this way.

Product Type

Type for Cartesian product of types. Can be thought as a tuple of elements of given types.

Example: Product Types

$$\textbf{data}\ \mathsf{SingleInt}\ =\ \mathsf{SingleInt}\ \ \textbf{Int}$$
$$\textbf{data}\ \mathsf{PairInt}\ =\ \mathsf{PairInt}\ \ \textbf{Int}\ \textbf{Int}$$
$$\textbf{data}\ \ \mathsf{TripleInt}\ =\ \mathsf{TripleInt}\ \ \textbf{Int}\ \textbf{Int}\ \textbf{Int}$$

Now data constructor is a curried function from appropriate number of elements of appropriate type we want to define:

GHCi session

$$:\!t\ \ \mathsf{TripleInt}$$

Exponential Types

Remember, functions are first-class citizens. Therefore, they have types.

Example: Exponential Types

```haskell
data AlgebraInt = AlgebraInt (PairInt -> Int)
```

In general, data constructors are functions from any given type to the type you define.

Defining functions on custom types

There is only one way to define functions on custom types: pattern-matching

Example: Pattern-matching on custom types

```haskell
not True = False
not False = True

True && True = True
_ && _ = False

add :: PairInt -> Int
add (PairInt x y) = x + y

curry_algebra :: AlgebraInt -> Int -> Int -> Int
curry_algebra (AlgebraInt f) x y = f (PairInt x y)
```

Polymorphic types

We also can define families of types.

Example: Maybe

$$\textbf{data Maybe}\ \mathsf{a} = \textbf{Nothing} \mid \textbf{Just}\ \mathsf{a}$$
$$\textbf{data Either}\ \mathsf{a}\ \mathsf{b} = \textbf{Left}\ \mathsf{a} \mid \textbf{Right}\ \mathsf{b}$$

a and *b* are parameters.
Maybe and **Either** act similar to functions, but on types.
"Types of types" are called *kinds*

GHCi session

$$:\mathsf{k}\ \textbf{Int}$$
$$:\mathsf{k}\ \textbf{Maybe}$$
$$:\mathsf{k}\ \textbf{Either}$$
$$:\mathsf{k}\ \textbf{Either String Int}$$

Recursive types

We define types using other types. It is also possible to define a type in terms of itself!

Example: List

```haskell
data List a = Nil | Cons a (List a)
```

Example: Some Functions on Lists

```haskell
head' (Cons x _) = x
tail'  (Cons _ xs)  = xs
length' Nil  = 0
length' (Cons _ xs)  = 1 + (length' xs)
```

Exercises

Monads

Imperative programming

We can easily write functional programs in imperative languages.
Can we write imperative programs in Haskell? Yes we can!
First, recall operator "apply" and define operator "bind":

Definition of *bind*

```
($) :: (a -> b) -> a -> b
f $ x = f x

(>>=) :: a -> (a -> b) -> b
(>>=) = flip ($)
```

The following terms are identical

```
f $ x
x >>= f
```

Imperative programming is back!

Looks like imperative programs!

Python

```python
def f(x):
    y = 1
    z = y // x
    return z*z
```

Haskell

```haskell
f x = 1             >>= (\y ->
      (y `div` x) >>= (\z ->
      z*z))
```

There is a special syntactic sugar for these constructions. Will cover later!
Also, *let ... in ...* could be used for this purpose:

Haskell

```haskell
f x = let y = 1 in
      (y `div` x) >>= (\z ->
      z*z)
```

Maybe, revisited

Suppose we live in a universe where all functions instead of returning values of type *a* return values of type *Maybe a*. Lets define *bind* for such universe.

bind for Maybe

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(Just x) >>= f = f x
Nothing >>= f = Nothing
```

We also want to lift functions from our ordinary universe to Maybe universe. It is enough to just lift values of type *a* to type *Maybe a*. The function is called *return*.

return for Maybe

```haskell
return :: a -> Maybe a
return x = Just x
```

Maybe, example

Example: Safe division

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)
```

Then rewrite the imperative example with safe division:

Example: Usage of Maybe monad

```haskell
f x = let y = 1 in
      (y `safeDiv` x) >>= (\z ->
      return z*z)
```

Monad

If you have *bind* and *return* defined with the following signature

General signature for *bind* and *return*

```haskell
(>>=) :: m a -> (a -> m b) -> m b
return :: a -> m a
```

Where *m* is some polymorphic type of kind $* - > *$. Such type together with these functions is called *Monad*.
Then you can use special syntactic sugar called *do notation*.

Example: **do** notation

```haskell
f x = do
        let y = 1
        z <- y `safeDiv` x
        return z*z
```

Example: without **do** notation

```haskell
f x = let y = 1 in
      (y `safeDiv` x) >>= (\z ->
      return z*z)
```

Some Other Monads

Writer is a Monad for "things with description".

Writer Type

data Writer a = Writer a **String**

Reader is a Monad for "things which depend on shared environment".

Reader Type

data Reader e a = Reader (e −> a)

State is a Monad for "things which depend on some state and may modify that state".

State Type

data State s a = State (s −> (a, s))

The End