

Haskell: функторы, аппликативные функторы, монады, do-нотация

Пугачёв Константин (K.V.Pugachev@inp.nsk.su)

2018-11-20 17h

В предыдущей серии

- АТД: как `std::variant` + `std::tuple/struct`
- паттерн матчинг – умный разбор значения
- классы типов – как интерфейсы в ООП
- ленивые вычисления
- бесконечные структуры данных

Почему map только для списков?

map действует над списками:

$$\text{map}(f, [x_0, x_1, \dots, x_n]) \equiv [f(x_1), f(x_2), \dots, f(x_n)]$$

Но почему бы не использовать его для дерева?

$$\text{map}(f, \frac{a}{b \cdot c}) \equiv \frac{f(a)}{f(b) \cdot f(c)}$$

Для матрицы?

$$\text{map}(f, [[a, b], [c, d]]) \equiv [[f(a), f(b)], [f(c), f(d)]]$$

Для произвольного контейнера?

Аналогия с C++: `std::transform` – преобразователь чего-то итерируемого:

```
template< class InIt, class OutIt, class UnOp >
OutIt transform( InIt first1, InIt last1, OutIt d_first,
                 UnOp unary_op );
```

Функтор – что-то, над чем работает map

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

Тип с одним параметром с операцией преобразования.

++ +	++ +		++ +	++ +
1	4		2	5
++ +	++ +	fmap (+1)	++ +	++ +
	2	----->		3
++ +			++ +	
3			4	
++ +			++ +	

	fmap (const 2)	
(x)-(y)-(z)	----->	(2)-(2)-(2)

Примеры функторов

```
data List a      = Cons a (List a) | Nil
data Maybe a     = Just a         | Nothing
data Either a b  = Left a         | Right b
```

`f $ x = f x` -- *оператор применения (меньше скобок)*

```
fmap (+1) (Cons 1 (Cons 2 Nil)) -- (Cons 2 (Cons 3 Nil))
fmap (+1) Nil                  -- Nil
```

```
fmap (+1) $ Just 3             -- Just 4
fmap (+1) $ Nothing            -- Nothing
```

```
fmap (+1) $ Left "huge number" -- Left "huge number"
fmap (+1) $ Right 8             -- Right 9
```

List, Cons, Nil полностью эквивалентны [], :, [] из Haskell.

Примеры функторов

```
data List a      = Cons a (List a) | Nil
data Maybe a     = Just a          | Nothing
data Either a b  = Left a          | Right b
```

`f $ x = f x` *-- оператор применения (меньше скобок)*

```
instance Functor List where
  fmap f (Cons a as) = Cons (f a) $ fmap f as
  fmap _ Nil         = Nil
```

```
instance Functor Maybe where
  fmap f (Just a) = Just $ f a
  fmap _ Nothing  = Nothing
```

```
instance Functor (Either a) where
  fmap _ (Left e)  = Left e
  fmap f (Right v) = Right $ f v
```

Законы функторов

Определения:

```
id x = x                -- эквивалентное преобразование  
f . g = \x -> f (f x)  -- композиция
```

- 1 fmap с эквивалентным преобразованием не меняет функтор:

```
fmap id = id
```

- 2 fmap и композицию можно переставить:

```
fmap (f . g) == fmap f . fmap g
```

Если законы выполняются, получается преобразование, которое не меняет форму коллекции.

Зачем законы?

Законы эквивалентны контракту в ПП.

- Законы не проверяются компилятором – следит программист
- Если выполняются, всё работает, кучу всего можно вывести математически
- Если не выполняются, ломается то, что полагается на них

Пример

$$a * b = a + a * (b - 1) = a * b = b + b * (a - 1), b > 0$$

Можно не определять умножение, а вывести его из сложения.

Если сложение не удовлетворяет законам сложения (напр. $a + b = a - b$), то умножение сломается, даже если сложение выглядело вполне нормально.

Аппликативный функтор – когда функция тоже в коллекции

```
class Functor f => Applicative (f :: * -> *) where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

<pre> +--+ +--+ 1 4 +--+ +--+ 2 +--+ +--+ 3 +--+ </pre>	<pre> +---+---+ +1 *2 <*> +---+---+ -----> </pre>	<pre> +---+---+ +---+---+ 2 2 5 8 +---+---+ +---+---+ 3 4 +---+---+ +---+---+ 4 6 +---+---+ </pre>
---	--	--

<pre> [x]-[y] </pre>	<pre> [toUpper] <*> -----> </pre>	<pre> [X]-[Y] </pre>
----------------------	--	----------------------

<pre> [x]-[y] </pre>	<pre> [] <*> -----> </pre>	<pre> [] </pre>
----------------------	-------------------------------------	-----------------

pure – “поднятие” значения в функтор

pure оборачивает значение в аппликативный функтор тривиальным образом

```
pure :: Applicative a => t -> t a
```

```
pure 'x' :: Applicative a => a Char
```

```
pure 'x' :: Maybe Char      -- Just 'x'
```

```
pure 'x' :: [Char]          -- ['x']
```

```
pure 'x' :: Either Char     -- Right 'x'
```

```
Just toUpper <*> pure 'x'   -- pure 'x' :: Maybe Char
```

Заметим, что pure что-то – полиморфная константа, тип определяется явно или исходя из соседей по выражению.

Вычисления с эффектами

- Maybe – вычисление может завершиться неудачей
Just 1 – успех, результат 1
Nothing – неудача
- Either – вычисление может завершиться ошибкой
Left "всё плохо" – ошибка "всё плохо"
Right 1 – успех, результат 1
- [] – вычисление имеет произвольное количество результатов
[] – результатов нет
[1] – один результат
[1, 2, 3] – много результатов

Примеры аппликативных функторов

-- Ошибка портит всё:

```
Just (+1) <*> Just 3
Nothing    <*> Just 3
Just (+1) <*> Nothing
Nothing    <*> Nothing
```

```
-- Just 4
-- Nothing
-- Nothing
-- Nothing
```

-- Декартово произведение:

```
[(+0), (+10)] <*> [1, 2]
[]             <*> [1, 2]
[(+0), (+10)] <*> []
[]             <*> []
```

```
-- [1, 2, 11, 12]
-- []
-- []
-- []
```

-- Ошибка портит всё:

```
Right (+1) <*> Right 3
Left "ERR" <*> Right 3
Left "E1"  <*> Left "E2"
Right (+1) <*> Left "ERR"
```

```
-- Right 4
-- Left "ERR"
-- Left "E1"
-- Left "ERR"
```

Законы аппликативных функторов

- 1 pure и эквивалентное преобразование по сути ничего не делают:

```
pure id <*> v = v -- Identity
```

- 2 pure тривиальна, не мешает применению функций:

```
pure f <*> pure x = pure (f x) -- Homomorphism
```

- 3 Слева или справа – всё едино:

```
u <*> pure y = pure ($ y) <*> u -- Interchange
```

- 4 Апп.функтор не мешает композиции:

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

liftA* – “поднятие” функций в мир аппликативных функторов

liftA/liftA2/... поднимают унарную/бинарную/... функцию в мир аппликативных функторов

```
liftA  :: Applicative f => (a -> b) -> f a -> f b
liftA2 :: Applicative f => (a -> b -> c)
                        -> f a -> f b -> f c
```

...

```
liftA  (+1) [1, 2, 3]           -- [2, 3, 4]
liftA2 (+)  [1, 2] [10, 20]    -- [11, 21, 12, 22]
```

```
liftA  f x      = pure f <*> x
liftA2 f x y     = pure f <*> x <*> y
liftA3 f x y z   = pure f <*> x <*> y <*> z
```

`liftA2 (+) [1,2] [10,20] ≡ pure (+) <*> [1,2] <*> [10,20] ≡ [(+)] <*> [1,2] <*> [10,20] ≡ [(+1), (+2)] <*> [10, 20] ≡ [11, 21, 12, 22]`. Аналогично, `liftA*` применяет функцию к одному аргументу за проход.

Чистый мир и действия – этапы прозрения

- 1 Если использовать только чистые функции, настоящую программу написать невозможно
 - `print` – не чистая, узнать результат программы никак нельзя
 - программа может делать всё, что угодно, снаружи всё эквивалентно
- 2 Императивные алгоритмы станут чистыми, если добавить аргумент с контекстом
 - `random :: number -- либо константа, либо магия`
 - `random :: context -> (context, number) -- чистая`

Чистый мир и действия – этапы прозрения

- 3 Весь мир – аргумент функции, функция возвращает новый мир
 - `print :: (world, item) -> world -- чистая`
 - все функции – чистые
- 4 Давайте реализуем это через монады
 - внутри монады будем неявно передавать миры
- 5 Монады можно использовать ещё разными способами
 - удобная математическая абстракция

Монады: спецфункции, спецзначения

```
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b      -- bind
  return :: a -> m a                        -- return
```

<pre> +-+ +-+ 1 4 +-+ -+-+ 2 +-+ -+-+ 3 +-+</pre>	<pre> >>= \x -> +---+---+ +1 *2 +---+---+ -----></pre>	<pre> +-+ -+-+ +-+ -+-+ 2 2 5 8 +-+ -+-+ -+-+ -+-+ 3 4 +-+ -+-+ -+-+ 4 6 +-+ -+-+</pre>
---	--	---

<pre> [x]-[y] >>= \x -> [toUpper x] -----></pre>	<pre> [X]-[Y]</pre>
--	---------------------

<pre> [x]-[y] >>= \x -> [] -----></pre>	<pre> []</pre>
---	----------------

Монада – это

Определение: Монада – это моноид в категории эндофункторов.

Не понявшие математическую суть (чуть менее, чем все) придумывают и публикуют свои интерпретации:

- упаковка для значений, которая преобразуется под действием специальных правил

монада – контейнер, монадическое значение – инстанс контейнера

анalogии: вектор в математике, функтор в ФП

- компоновщик законсервированных действий

монада – принцип работы с эффектами, монадическое значение – законсервированное действие

анalogии: байты исполняемого кода, указатель на процедуру

Интерпретации неполны. Понимание монады как упаковки неприменимо, например, для IO, Reader; понимание монады как законсервированного действия не всегда уместно, например, для Maybe, [].

return – создание простейшего действия

return оборачивает значение в монаду тривиальным образом
return эквивалентна pure, только требует контекста монады.

```
return :: Monad a => t -> t a
```

```
return 'x' :: Monad a => a Char
```

```
return 'x' :: Maybe Char      -- Just 'x'
```

```
return 'x' :: [Char]         -- ['x']
```

```
return 'x' :: Either Char    -- Right 'x'
```

```
Just id >=> \_ -> return 'x'  -- return 'x' :: Maybe Char
```

Заметим, что Monad что-то – полиморфная константа, тип определяется явно или исходя из соседей по выражению.

Примеры монад

Maybe, Either – вычисления с возможностью ошибок.

[] – вычисления со множеством результатов

```
Just 3  >>= \x -> Just (2*x)      -- Just 6
Nothing >>= \x -> Just (2*x)      -- Nothing
Just x  >>= \_ -> Nothing         -- Nothing
Nothing >>= \_ -> Nothing         -- Nothing

[1,2,3] >>= \x -> [x, x+10]        -- [1,11,2,12,3,13]
[1,2,3] >>= \x -> []              -- []
[]      >>= \x -> [x, x+10]        -- []
[]      >>= \x -> []              -- []

Right 3 >>= \x -> Right (x+1)     -- Right 4
```

Монадные законы

Определение. Композиция монадических функций:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c  
(m >=> n) x = m x >>= \y -> n y
```

- 1 `return >=> g ≡ g` *-- Left identity*
- 2 `f >=> return ≡ f` *-- Right identity*
- 3 `(f >=> g) >=> h ≡ f >=> (g >=> h)` *-- Associativity*

liftM* – “поднятие” функций в мир монад

liftM/liftAM/... поднимают унарную/бинарную/... функцию в мир монад

liftM* эквивалентны liftA*, но в коде они реализуются по-разному, чтобы не требовать лишних контекстов

```
liftM    :: Monad m => (a -> b) -> m a -> m b
liftM2   :: Monad m => (a -> b -> c)
                                -> m a -> m b -> m c
...

liftM    (+1) [1, 2, 3]           -- [2, 3, 4]
liftM2   (+)  [1, 2] [10, 20]     -- [11, 21, 12, 22]

liftM f x = x >>= \x -> return (f x)
liftM2 f x y = x >>= \x -> y >>= \y -> return (f x y)
```

liftA2 (+) [1,2] [10,20] = [1, 2] >>= \x -> [10,20] >>= \y -> [x + y] = [11, 21, 12, 22]. liftA2 собирает аргументы функции, для каждого набора запускается функция.

Монада IO

Действия ввода-вывода “запрятаны” в монаду IO:

```
print      :: Show a => a -> IO ()  -- распечатать нечто
putStr     :: String -> IO ()      -- вывести строку
putStrLn   :: String -> IO ()      -- вывести строку + '\n'

readLn     :: Read a => IO a        -- считать и распарсить
getLine    :: IO String            -- считать строку

writeFile  :: FilePath -> String -> IO () -- писать в файл
...
```

Можно понимать как набор императивных функций:

```
void putStr (const char* str) {
    printf("%s", str);
}

void putStrLn (const char* str) {
    printf("%s\n", str);
}
```

Монада IO

В монадическом значении типа IO а лежит “законсервированное” (отложенное) действие, возвращающее а.

```
print    [1,2,3]  :: IO ()  -- распечатать список
putStr   "xxx"    :: IO ()  -- вывести строку
putStrLn "xxx"    :: IO ()  -- вывести строку + '\n'

putXXX = putStr "XXX" -- законсервированное действие
```

Монадические значения IO можно понимать как нуль-арные императивные функции, **не как результат** исполнения:

```
void print123 () {
    printf("[1, 2, 3]");
}

void putXXX () {
    printf("xxx", str);
}
```


bind – связывание

Набор действий – как набор функций, которые можно запустить:

```
>----->      >----->      >----->      >----->
| x := 1 |      | x := x+1 |      | x := 9 |      | result: x+3 |
>----->      >----->      >----->      >----->
```

Действия можно связать:

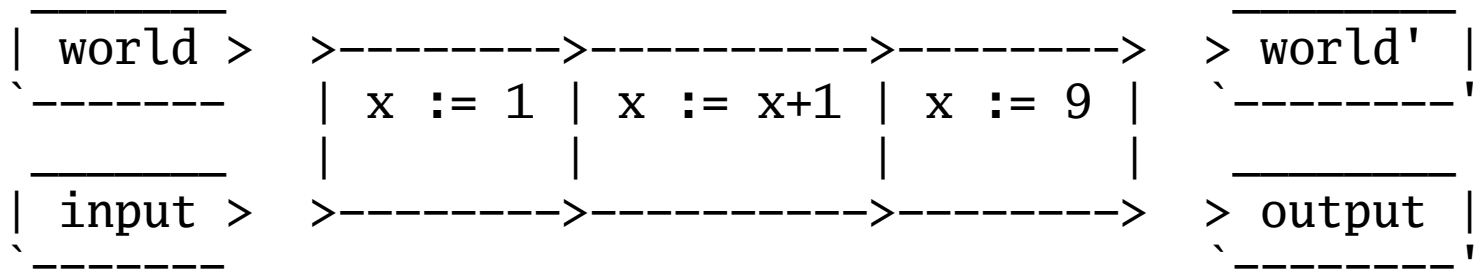
```
>----->----->----->----->
| x := 1 | x := x+1 | x := 9 | result: x+3 |
>----->----->----->----->
```

Аналог в C:

```
int proc() {
    x = 1;           // x - global
    x += 1;
    x = 9;
    return x+3;
}
```

bind – связывание

После связывания композитное действие можно запускать как программу:



Аналог в C:

```
int x;

int main() {
    proc();
}
```

bind

(>>=) – связывание м.значения с м.функцией:

```
[1,2,3] >>= \x -> [x,-x]      -- [1,-1,2,-2,3,-3]
```

```
getLine >>= \x -> putStr (x ++ x)  
-- дважды печатает введённое
```

В случае IO значение из монады достать никак нельзя, можно только использовать >>=.

(>>) – связывание м.значения с м.значением:

```
x >> y = x >>= \_ -> y      -- определение
```

```
[1,2,3] >> [10,20]          -- [10,20,10,20,10,20]
```

```
putStr "xxx" >> putStr "yyy" -- выводит xxxyyy
```

Эквивалентно последовательности действий (оператор запятая) в императивном программировании. Видно, что в случае `[1,2,3] >> [10,20]` эффекты `[1,2,3]` (утраивание результата) сохраняются, а значение игнорируется.

do-нотация – синтаксический сахар для использования монад

- 1 Одна “команда” – одна строка:

```
actions = monadicValue1 >> monadicValue2
```

```
actions = do  
  monadicValue1  
  monadicValue2
```

- 2 Абстракция + bind = “присваивание переменной”:

```
actions = monadicValue1 >>= \x -> monadicValue2
```

```
actions = do  
  x <- monadicValue1  
  monadicValue2
```

- 3 Именованное значение:

```
actions = let x = 3 in print x
```

```
actions = do  
  let x = 3 -- необязательно монадическое значение  
  print x
```

Пример программы

```
import Control.Monad (when)
```

```
readName :: IO String
```

```
readName = do  
    putStr "Enter name> "  
    getLine
```

```
readPassword :: String -> IO Bool
```

```
readPassword validPassword = do  
    putStr "Enter password> "  
    password <- getLine  
    let valid = password == validPassword  
    if valid  
    then putStrLn "Password is valid."  
    else do  
        putStrLn "Password is invalid."  
        putStrLn "Are you a hacker?"  
    return valid
```

```
main :: IO ()
```

```
main = do  
    name <- readName  
    when (name == "qwerty") $ putStrLn "Серьёзно?"  
    valid <- readPassword "12345"  
    if valid  
    then putStrLn ("Hello, " ++ name ++ "!!")  
    else main
```

-- рекурсия вместо цикла

Ещё функции как императивный сахар

-- map с монадической функцией

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

-- map с монадической функцией, учитываются только эффекты

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
```

-- forM с императивным синтаксисом

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
```

```
forM = flip mapM
```

```
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
```

```
forM_ = flip mapM_
```

-- включает действие, если выполнено условие

```
when :: Applicative f => Bool -> f () -> f ()
```

На самом деле, mapM объявлена как часть класса типов Traversable, работает не только со списками, но с любыми типами, относящимися к Functor и Foldable.

Одна do-нотация – одна монада

```
import Control.Monad (forM_)

triangles :: (Enum a, Num a, Eq a) => [(a,a,a)]
triangles = do
  z <- [1..20]           -- list
  y <- [1..z]            -- list
  x <- [1..y]            -- list
  -- print (x, y, z)     -- IO, error!
  if x^2 + y^2 == z^2
  then return (x, y, z) -- list
  else []               -- list

printNumbers :: IO ()
printNumbers = do
  print 1              -- IO
  print 2              -- IO
  -- x <- [1..10]       -- list, error!
  -- print x            -- IO

printNumbers' :: IO ()
printNumbers' = do
  print 1              -- IO
  print 2              -- IO
  forM_ [1..10] $ \x -> do
    print x            -- IO
```

Что дальше

- Разные монады (Reader, Writer, State и менее тривиальные)
Монады познаются на практике
- Монадные трансформеры (чтобы в одном монадическом выражении работало больше одной монады)
- Много математической теории (теория категорий, алгоритм Хиндли-Милнера, множество разных терминов, ...)

см. эволюция программиста на Haskell

Задачи

Знаний достаточно для реализации всех задач:

- Задачи 1, 2 о списках
- Задач 3, 4 о написании полезных функций и программ

Внимание, лекции, задания и сниппеты обновляются.