Про монады

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

15 ноября 2018 г.

Шаг 1. Функторы

```
Prelude> :i Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a        -> f b -> f a
  {-# MINIMAL fmap #-}   -- Defined in ‘GHC.’Base
```

...

Их же иногда аллегорично называют “контейнерами“.

Одна важная функция:

```
Prelude> :i <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
        -- Defined in ‘Data.’Functor
```

Законы функторов

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

Пример реализации `fmap`, которая **не согласуется** с законами

```
data PPP a = PPP a a

instance Functor PPP where
  fmap f (PPP a b) = PPP (f b) (f a)
```

Но обычно для всех наших типов возможно написать fmap.

Функторы в стандартной библиотеке

```
Prelude> :i Functor

...

instance Functor (Either a) –– Defined in 'Data.Either
instance Functor []    –– Defined in 'GHC.Base'
instance Functor Maybe –– Defined in 'GHC.'Base
instance Functor IO    –– Defined in 'GHC.'Base
instance Functor ((–>) r) –– Defined in ‘GHC.’Base
instance Functor ((,) a)  –– Defined in ‘GHC.’Base
```

Шаг 2. Аппликативные функторы

Документация тут. А вот ссылка как они появились.

```
Prelude> :i Applicative
class Functor f => Applicative (f :: * -> *) where
  pure  :: a -> f a                           -- 1
  (<*>) :: f (a -> b) -> f a -> f b           -- 2
  GHC.Base.liftA2 ::                          -- 3
    (a -> b -> c) -> f a -> f b -> f c

  {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Тут три функции, но нужны для полной реализации только две

Как переписать код на аппликативные функторы?

```
foo :: (a -> b -> c) -> a -> b -> c
foo f a b = f a b
```

$$\Downarrow$$

```
foo :: Applicative f =>
          f (a -> b -> c) -> f a -> f b -> f c
foo f a b = f <*> a <*> b
```

Законы аппликативов

- identity
  `pure id <*> v = v`
- composition
  `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- homomorphism
  `pure f <*> pure x = pure (f x)`
- interchange
  `u <*> pure y = pure (fmap y) <*> u`

Аппликативы в стандартной библиотеке

```
Prelude> :i Applicative

...

instance Applicative (Either e)
instance Applicative []
instance Applicative Maybe
instance Applicative IO
instance Applicative ((->) a)
instance Monoid a => Applicative ((,) a)
```

Шаг 3. Они самые

Документация.

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

  {-# MINIMAL (>>=) #-}

  fail :: String -> m a
```

Для минимальной реализации нужно иметь `(>>=)` и быть `Applicative`.

Законы монад

- Левый нейтральный:
  `return a >>= f` $\equiv$ `f a`
- Правый нейтральный:
  `m >>= return` $\equiv$ `m`
- Ассоциативность (почти):
  `(m >>= f) >>= g` $\equiv$ `m >>= (\x -> f x >>= g)`

Тут должен быть вслух сказан аллегоричная история про Форда

do-нотация (1/2)

```
thing1 >>= (\x -> func1 x >>= (\y -> thing2
       >>= (\_ -> func2 y >>= (\z -> return z))))
```

$$\Updownarrow$$

```
thing1  >>= \x ->
func1 x >>= \y ->
thing2  >>= \_ ->
func2 y >>= \z ->
return z
```

$$\Updownarrow$$

...

do-нотация (1/2)

```
thing1  >>= \x ->
func1 x >>= \y ->
thing2  >>= \_ ->
func2 y >>= \z ->
return z
```

$$\Updownarrow$$

```
do {
  x <- thing1 ;
  y <- func1 x ;
  thing2 ;
  z <- func2 y ;
  return z
  }
```

Лирическое отступление – list comprehensions

```
[ (x,y) | x <- [1..3], y <- [1..2] ]
```

$$\Updownarrow$$

```
do
  x <- [1..3]
  y <- [1..2]
  return (x,y)
```

$$\Updownarrow$$

```
[1..3] >>= \x -> [1..2] >>= \y -> return (x,y)
```

Пример: IO

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

(>>) l r  =  l >>= \_ -> r
```

```haskell
hello :: IO ()
hello = putChar 'H' >> putChar 'e' >> putChar 'l' >>
        putChar 'l' >> putChar 'o'

hi = do
      putChar 'H'
      putChar 'i'
      return ()
```

Пример: State (1/2)

```
data State s a = S { runState :: s -> (a, s) }

state :: (s -> (a,s)) -> State s a
state = S
```

Пример: State (1/2)

```
data State s a = S { runState :: s -> (a, s) }

state :: (s -> (a,s)) -> State s a
state = S

instance Monad (State s) where
  return x    = S (\s -> (x, s))
  st >>= f    = S (\s -> let (x, s') = runState st s
                         in runState (f x) s')
```

Пример: State (1/2)

```
data State s a = S { runState :: s -> (a, s) }

state :: (s -> (a,s)) -> State s a
state = S

instance Monad (State s) where
  return x    = S (\s -> (x, s))
  st >>= f    = S (\s -> let (x, s') = runState st s
                         in runState (f x) s')

evalState :: State s a -> s -> a
evalState s = fst . runState s

execState :: State s a -> s -> s
execState s  = snd . runState s
```

Пример: State (1/2)

Готовимся заглянуть внутрь:

```
get :: State s s
get = S (\s -> (s, s))
```

Модифицируем состояние:

```
put :: s -> State s ()
put s' = S (\s -> ((), s'))
```