

# Лекция 10: монады

Функциональное программирование на Haskell

---

Алексей Романов

6 марта 2023 г.

МИЭТ

# Монады

- Монады расширяют возможности аппликативных функторов так же, как те расширяют возможности функторов.
- А именно, там добавляется

```
class Applicative m => Monad m where
  (>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  mx >> my =
```

# Монады

- Монады расширяют возможности аппликативных функторов так же, как те расширяют возможности функторов.
- А именно, там добавляется

```
class Applicative m => Monad m where
  (>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  mx >> my = mx >=
```

# Монады

- Монады расширяют возможности аппликативных функторов так же, как те расширяют возможности функторов.
- А именно, там добавляется

```
class Applicative m => Monad m where
  (>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  mx >> my = mx >= \_ -> my
```

# Монады

- Монады расширяют возможности аппликативных функторов так же, как те расширяют возможности функторов.
- А именно, там добавляется

```
class Applicative m => Monad m where
    (>=>) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    mx >> my = mx >=> \_ -> my
```

- По историческим причинам есть ещё  
`return = pure` и `fail :: String -> m a`.

# Монады

- Монады расширяют возможности аппликативных функторов так же, как те расширяют возможности функторов.
- А именно, там добавляется

```
class Applicative m => Monad m where
    (>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    mx >> my = mx >= \_ -> my
```

- По историческим причинам есть ещё `return = pure` и `fail :: String -> m a`.
- Больше похоже на знакомые типы, если поменять аргументы местами:

```
fmap    :: (a -> b) -> f a -> f b
(<*>)   :: f (a -> b) -> f a -> f b
(<=<=)   :: (a -> m b) -> m a -> m b
```

## Что можно сделать с монадами

- В прошлой лекции:  
*Структура результата аппликативного вычисления зависит только от структуры аргументов, а не от значений внутри них*
- Монады снимают это ограничение. Теперь можно написать

```
ifM :: Monad m => m Bool -> m a -> m a -> m a
ifM mCond mThen mElse = mCond >=>
```

## Что можно сделать с монадами

- В прошлой лекции:  
*Структура результата аппликативного вычисления зависит только от структуры аргументов, а не от значений внутри них*
- Монады снимают это ограничение. Теперь можно написать

```
ifM :: Monad m => m Bool -> m a -> m a -> m a  
ifM mCond mThen mElse = mCond >=> \cond ->
```



## Что можно сделать с монадами

- В прошлой лекции:  
*Структура результата аппликативного вычисления зависит только от структуры аргументов, а не от значений внутри них*
- Монады снимают это ограничение. Теперь можно написать

```
ifM :: Monad m => m Bool -> m a -> m a -> m a
ifM mCond mThen mElse = mCond >=> \cond ->
    if cond then mThen else mElse
```

и проверить:

```
ghci> ifM [True] [1] [2,3]
[1]
ghci> ifM [False] [1] [2,3]
[2,3]
```

## Законы монад

- `pure x >>= f ≡ f x`
- `mx >>= pure ≡ mx`
- `(mx >>= f) >>= g ≡ mx >>= (\x -> f x >>= g)`
- Может быть, они не совсем интуитивны.

# Законы монад

- `pure x >=> f ≡ f x`
- `mx >=> pure ≡ mx`
- `(mx >=> f) >=> g ≡ mx >=> (\x -> f x >=> g)`
- Может быть, они не совсем интуитивны.
- С помощью монадической композиции  
`(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)`  
они записываются естественнее:
- `pure >=> f ≡ f`
- `f >=> pure ≡ f`
- `(f >=> g) >=> h ≡ f >=> (g >=> h)`

# Законы монад

- $\text{pure } x \gg= f \equiv f \ x$
- $\text{mx} \gg= \text{pure} \equiv \text{mx}$
- $(\text{mx} \gg= f) \gg= g \equiv \text{mx} \gg= (\backslash x \rightarrow f \ x \gg= g)$
- Может быть, они не совсем интуитивны.
- С помощью монадической композиции  
 $(\gg=) :: (a \rightarrow m \ b) \rightarrow (b \rightarrow m \ c) \rightarrow (a \rightarrow m \ c)$   
они записываются естественнее:
- $\text{pure} \gg= f \equiv f$
- $f \gg= \text{pure} \equiv f$
- $(f \gg= g) \gg= h \equiv f \gg= (g \gg= h)$
- Кроме этого, должно быть согласование с  $\langle * \rangle$ :  
 $\text{mf} \langle * \rangle \text{mx} \equiv$   
 $\text{mf} \gg= (\backslash f \rightarrow \text{mx} \gg= (\backslash x \rightarrow \text{pure } (f \ x)))$

# Примеры монад

- `Maybe` — монада. Определим:

```
instance Monad Maybe where  
  -- (>>=) ::
```

# Примеры монад

- `Maybe` — монада. Определим:

```
instance Monad Maybe where  
  -- (>>=) ::
```

# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
-- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
Nothing >>= _ =
```

# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
  Nothing >>= _ = Nothing
```

```
  Just x   >>= f =
```



# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
  -- (>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
  Nothing >= _ = Nothing
```

```
  Just x  >= f = f x
```

- Списки тоже монада:

```
-- (>=) ::
```

# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
  -- (>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
  Nothing >= _ = Nothing
```

```
  Just x  >= f = f x
```

- Списки тоже монада:

```
-- (>=) ::
```

# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
-- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
Nothing >>= _ = Nothing
```

```
Just x   >>= f = f x
```

- Списки тоже монада:

```
-- (>>=) :: [a] -> (a -> [b]) -> [b]
```

```
[]      >>= _ =
```

# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
  -- (>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
  Nothing >= _ = Nothing
```

```
  Just x   >= f = f x
```

- Списки тоже монада:

```
-- (>=) :: [a] -> (a -> [b]) -> [b]
```

```
[]      >= _ = []
```

```
(x:xs)  >= f =
```

# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
-- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
Nothing >>= _ = Nothing
```

```
Just x   >>= f = f x
```

- Списки тоже монада:

```
-- (>>=) :: [a] -> (a -> [b]) -> [b]
```

```
[]      >>= _ = []
```

```
(x:xs) >>= f = f x ++ (xs >>= f)
```

Или проще:

```
xs >>= f = [y |
```

# Примеры монад

- **Maybe** — монада. Определим:

```
instance Monad Maybe where
```

```
-- (>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
Nothing >= _ = Nothing
```

```
Just x   >= f = f x
```

- Списки тоже монада:

```
-- (>=) :: [a] -> (a -> [b]) -> [b]
```

```
[]      >= _ = []
```

```
(x:xs) >= f = f x ++ (xs >= f)
```

Или проще:

```
xs >= f = [y | x <- xs, y <- f x]
```

## Ещё примеры монад

- `instance Monad Identity`, где `newtype Identity a = Identity a`.
- `instance Monad (Either c)`.
- `instance Monad ((->) c)`: функции с фиксированным типом аргумента.
- `instance Monoid c => Monad ((,) c)`: пары с фиксированным типом первого элемента, если этот тип — моноид.

## ...и не монад

- Есть и примеры аппликативных функторов, для которых нельзя определить экземпляр монады:
- ```
newtype ConstInt a = ConstInt Int
fmap f (ConstInt x) = ConstInt x
pure _ = 0
ConstInt x <*> ConstInt y = ConstInt (x + y)
```
- Легко увидеть, что  $\text{pure } x \gg= f \equiv f \ x$  не может выполняться ни для какого определения  $\gg=$ : правая часть зависит от  $x$ , а левая нет.



## ... и не монад

- Есть и примеры аппликативных функторов, для которых нельзя определить экземпляр монады:
- ```
newtype ConstInt a = ConstInt Int
fmap f (ConstInt x) = ConstInt x
pure _ = 0
ConstInt x <*> ConstInt y = ConstInt (x + y)
```
- Легко увидеть, что `pure x >=> f ≡ f x` не может выполняться ни для какого определения `>=>`: правая часть зависит от `x`, а левая нет.
- Ещё пример — `ZipList` (если не допускать только бесконечные списки, как в домашнем задании, или активное использование  $\perp$ ).

## do-нотация

- Для монад есть специальный синтаксис, которым часто удобнее пользоваться.
- Скажем, у нас есть цепочка операций  
`action1 >=> (\x1 -> action2 x1 >=>  
 (\x2 -> action3 x1 x2 >> action4 x1))`
- Сначала перепишем так:  
`action1 >=> \x1 ->  
 action2 x1 >=> \x2 ->  
 action3 x1 x2 >>  
 action4 x1`
- В **do**-блоке строки вида `action1 >=> \x1 ->` превращаются в `x1 <- action1`, а `>>` пропадает:  
`do x1 <- action1  
 x2 <- action2 x1  
 action3 x1 x2  
 action4 x1`

# Законы монад в **do**-нотации

- Законы монад также можно записать через **do**:

## Законы монад в **do**-нотации

- Законы монад также можно записать через **do**:
- $\text{do } y \leftarrow \text{pure } x \quad \equiv \quad f \ x$   
     $\quad \quad \quad f \ y$

## Законы монад в **do**-нотации

- Законы монад также можно записать через **do**:
- **do**  $y \leftarrow \text{pure } x$   $\equiv$   $f \ x$   
     $f \ y$
- **do**  $x \leftarrow m x$   $\equiv$   $m x$   
     $\text{pure } x$

## Законы монад в **do**-нотации

- Законы монад также можно записать через **do**:
- $\text{do } y \leftarrow \text{pure } x \quad \equiv \quad f \ x$   
     $f \ y$
- $\text{do } x \leftarrow mx \quad \equiv \quad mx$   
     $\text{pure } x$
- $\text{do } x \leftarrow mx \quad \equiv \quad \text{do } y \leftarrow \text{do } x \leftarrow mx$   
     $y \leftarrow f \ x \quad \equiv \quad f \ x$   
     $g \ y \quad \equiv \quad g \ y$

## Законы монад в **do**-нотации

- Законы монад также можно записать через **do**:
- $\text{do } y \leftarrow \text{pure } x \quad \equiv \quad f \ x$   
     $\quad \quad \quad f \ y$
- $\text{do } x \leftarrow mx \quad \equiv \quad mx$   
     $\quad \quad \quad \text{pure } x$
- $\text{do } x \leftarrow mx \quad \equiv \quad \text{do } y \leftarrow \text{do } x \leftarrow mx$   
     $\quad \quad \quad y \leftarrow f \ x \quad \equiv \quad \quad \quad f \ x$   
     $\quad \quad \quad g \ y \quad \equiv \quad \quad \quad g \ y$
- $mf \leftarrow^* mx \quad \equiv \quad \text{do } f \leftarrow mf$   
     $\quad \quad \quad \quad \quad \quad x \leftarrow mx$   
     $\quad \quad \quad \quad \quad \quad \text{pure } (f \ x)$

## Общая форма **do**-нотации

- Каждая строка **do**-блока имеет вид  
**образец** `<-` `м_выражение`,  
**let** **образец** `=` `выражение` или просто  
`м_выражение`.
- Первые два вида не могут быть в конце.
- `м_выражение` должно иметь тип `m` а для какой-то монады `m` и типа `a`.
- `m` одна для всех строк, `a` могут различаться.
- **образец** в строке с `<-` имеет тип `a`.
- **образец** может не быть обязательным для всех значений типа `a`, как в `Just x <- pure Nothing`.  
В этом случае и нужна функция `fail`.
- Но это не рекомендуется использовать.



## Функции над произвольными монадами

- Кроме уже виденных `=<<` и `>=>`, в `Prelude` и `Control.Monad` есть ещё функции, которые работают для любых монад (или аппликативов).

## Функции над произвольными монадами

- Кроме уже виденных `=<<` и `>=>`, в `Prelude` и `Control.Monad` есть ещё функции, которые работают для любых монад (или аппликативов).
- `join :: Monad m => m (m a) -> m a`. Эту функцию можно было бы взять как базовую и выразить `>=>` через неё.

## Функции над произвольными монадами

- Кроме уже виденных `=<<` и `>=>`, в `Prelude` и `Control.Monad` есть ещё функции, которые работают для любых монад (или аппликативов).
- `join :: Monad m => m (m a) -> m a`. Эту функцию можно было бы взять как базовую и выразить `>=>` через неё.
- `sequence :: Monad m => [m a] -> m [a]`. На самом деле, в библиотеке более общий вариант.

## Функции над произвольными монадами

- Кроме уже виденных `=<<` и `>=>`, в `Prelude` и `Control.Monad` есть ещё функции, которые работают для любых монад (или аппликативов).
- `join :: Monad m => m (m a) -> m a`. Эту функцию можно было бы взять как базовую и выразить `>=>` через неё.
- `sequence :: Monad m => [m a] -> m [a]`. На самом деле, в библиотеке более общий вариант.
- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`

## Функции над произвольными монадами

- Кроме уже виденных `=<<` и `>=>`, в `Prelude` и `Control.Monad` есть ещё функции, которые работают для любых монад (или аппликативов).
- `join :: Monad m => m (m a) -> m a`. Эту функцию можно было бы взять как базовую и выразить `>=>` через неё.
- `sequence :: Monad m => [m a] -> m [a]`. На самом деле, в библиотеке более общий вариант.
- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
- `zipWithM :: Applicative m => (a -> b -> m c) -> [a] -> [b] -> m [c]`

## Функции над произвольными монадами

- Кроме уже виденных `=<<` и `>=>`, в `Prelude` и `Control.Monad` есть ещё функции, которые работают для любых монад (или аппликативов).
- `join :: Monad m => m (m a) -> m a`. Эту функцию можно было бы взять как базовую и выразить `>=>` через неё.
- `sequence :: Monad m => [m a] -> m [a]`. На самом деле, в библиотеке более общий вариант.
- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
- `zipWithM :: Applicative m => (a -> b -> m c) -> [a] -> [b] -> m [c]`
- Подставьте конкретные монады (например, `Maybe` и `[]`) и подумайте, что функции будут делать для них.

## Монада **State**

- Рассмотрим ещё один, более сложный пример:
- `newtype State s a = State { runState :: s -> (a, s) }`

Это «вычисления с состоянием», которые могут выдать результат, зависящий от состояния и изменить это состояние.

- Как сделать их аппликативным функтором?

```
instance Functor (State s) where  
    fmap f (State gx) = State (\s1 ->
```

## Монада **State**

- Рассмотрим ещё один, более сложный пример:
- `newtype State s a = State { runState :: s -> (a, s) }`

Это «вычисления с состоянием», которые могут выдать результат, зависящий от состояния и изменить это состояние.

- Как сделать их аппликативным функтором?

```
instance Functor (State s) where
  fmap f (State gx) = State (\s1 ->
    let (x, s2) = gx s1
    in
```



# Монада **State**

- Рассмотрим ещё один, более сложный пример:
- `newtype State s a = State { runState :: s -> (a, s) }`

Это «вычисления с состоянием», которые могут выдать результат, зависящий от состояния и изменить это состояние.

- Как сделать их аппликативным функтором?

```
instance Functor (State s) where
    fmap f (State gx) = State (\s1 ->
        let (x, s2) = gx s1
        in (f x, s2))
```

```
instance Applicative (State s) where
    pure x = State (\s1 ->
```

## Монада **State**

- Рассмотрим ещё один, более сложный пример:
- `newtype State s a = State { runState :: s -> (a, s) }`

Это «вычисления с состоянием», которые могут выдать результат, зависящий от состояния и изменить это состояние.

- Как сделать их аппликативным функтором?

```
instance Functor (State s) where
    fmap f (State gx) = State (\s1 ->
        let (x, s2) = gx s1
        in (f x, s2))
```

```
instance Applicative (State s) where
    pure x = State (\s1 -> (x, s1))
    (State gf) <*> (State gx) = State (\s1 ->
        let (f, s2) = gf s1
```

# Монада **State**

- Рассмотрим ещё один, более сложный пример:
- `newtype State s a = State { runState :: s -> (a, s) }`

Это «вычисления с состоянием», которые могут выдать результат, зависящий от состояния и изменить это состояние.

- Как сделать их аппликативным функтором?

```
instance Functor (State s) where
    fmap f (State gx) = State (\s1 ->
        let (x, s2) = gx s1
        in (f x, s2))
```

```
instance Applicative (State s) where
    pure x = State (\s1 -> (x, s1))
    (State gf) <*> (State gx) = State (\s1 ->
        let (f, s2) = gf s1
        (x, s3) = gx s2
        in
```

# Монада **State**

- Рассмотрим ещё один, более сложный пример:
- `newtype State s a = State { runState :: s -> (a, s) }`

Это «вычисления с состоянием», которые могут выдать результат, зависящий от состояния и изменить это состояние.

- Как сделать их аппликативным функтором?

```
instance Functor (State s) where
    fmap f (State gx) = State (\s1 ->
        let (x, s2) = gx s1
        in (f x, s2))
```

```
instance Applicative (State s) where
    pure x = State (\s1 -> (x, s1))
    (State gf) <*> (State gx) = State (\s1 ->
        let (f, s2) = gf s1
        (x, s3) = gx s2
        in (f x, s3))
```

# Монада **State**

- А теперь монадой:

```
instance Monad (State s) where
  State gx >=> f = State (\s1 ->
    let (x, s2) = gx s1
```

# Монада **State**

- А теперь монадой:

```
instance Monad (State s) where
  State gx >=> f = State (\s1 ->
    let (x, s2) = gx s1
    State gy = f x
    in
```

# Монада State

- А теперь монадой:

```
instance Monad (State s) where
  State gx >=> f = State (\s1 ->
    let (x, s2) = gx s1
    State gy = f x
    in gy s2)
```

# Монада **State**

- А теперь монадой:

```
instance Monad (State s) where
  State gx >=> f = State (\s1 ->
    let (x, s2) = gx s1
    State gy = f x
    in gy s2)
```

- Определим вспомогательные функции:

```
get :: State s s
get = State
```



# Монада **State**

- А теперь монадой:

```
instance Monad (State s) where
  State gx >=> f = State (\s1 ->
    let (x, s2) = gx s1
    State gy = f x
    in gy s2)
```

- Определим вспомогательные функции:

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

put :: s -> State s ()
put x = State
```

# Монада State

- А теперь монадой:

```
instance Monad (State s) where
  State gx >=> f = State (\s1 ->
    let (x, s2) = gx s1
    State gy = f x
    in gy s2)
```

- Определим вспомогательные функции:

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

```
put :: s -> State s ()
put x = State (\_ -> ((), x))
```

```
modify :: (s -> s) -> State s ()
modify f = do x <- get
             put (f x)
```

- [Functors, Applicatives, and Monads](#)
- [Typeclassopedia](#) (ещё раз)
- [What I Wish I Knew When Learning Haskell: Monads](#)
- [All About Monads](#)
- И две более сложные монады:
  - [The Mother of all Monads](#)
  - [Mindfuck: The Reverse State Monad](#)
  - [The Curious Time-Traveling Reverse State Monad](#)