Монады и do-синтаксис

Михаил Беляев

28 октября 2015

КЛАССЫ ТИПОВ: МОНАДА

Монада — это параметризованный тип `m`, удовлетворяющий требованиям:

- Он является функтором (см. предыдущую лекцию)
- Можно «завернуть» значение в монаду:
  `return :: a -> m a`
- Можно «расплющить» монаду от монады:
  `flatten :: m (m a) -> m a`
- Операция `bind` или `(>>=)`:
  `x >>= f = flatten (fmap f x)`
- Работают законы:

```
return x >>= f  === f x
m >>= return    === m
(m >>= f) >>= g === m >>= (\x -> f x >>= g)
```

МОНАДА: ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
```

ПРОБЛЕМА 1: COALESCING

Есть несколько функций, которые зависят друг от друга и используют результаты друг друга, но каждая из них может выдать ошибку.

```
x :: Maybe Int
y :: Maybe Int

xplusy =
    case x of
        Nothing -> Nothing
        Just x' ->
            case y of
                Nothing -> Nothing
                Just y' -> Just (x' + y')
```

Решение проблемы 1 с помощью монады maybe

```haskell
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return x = Just x
    fail message = Nothing

x :: Maybe Int
y :: Maybe Int
xplusy =
    x >>= \ x' ->
      y >>= \ y' ->
        return (x' + y')
```

Проблема 2: декартово произведение

Есть несколько списков, нужно скомбинировать их элементы «каждый с каждым»

```
x :: [Int]
y :: [Int]

xplusy = flatten (map (\x' -> map (+x') y) x)
```

Решение проблемы 2 с помощью монады list

```haskell
instance Monad [] where
    lst >>= f = flatten (map f lst)
    return x = [x]
    fail message = []

x :: [Int]
y :: [Int]
xplusy =
    x >>= \ x' ->
      y >>= \ y' ->
        return (x' + y')
```

композиция с помощью bind

```
xplusy =
    x >>= \ x' ->
      y >>= \ y' ->
        return (x' + y')

f :: a -> m a
xplusytimez =
    x >>= \ x' ->
      y >>= \ y' ->
        (f y) >>= \ z' ->
          return $ x' + y' * z'
```

DO-СИНТАКСИС

```haskell
xplusytimez =
    x >>= \ x' ->
      y >>= \ y' ->
        (f y) >>= \ z' ->
           return $ x' + y' * z'

xplusytimez =
    do
       x' <- x
       y' <- y
       z' <- f y
       return $ x' + y' * z'
```

do-синтаксис

- Вычисления со *скрытым контекстом*
- Примеры такого контекста: декартово произведение, вычисления с обработкой ошибок
- Контекст не является чем-то фиксированным, он просто должен отвечать законам монады

Примеры монад: монада identity

Монада с пустым контекстом

```
data Identity a = Identity a
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
instance Monad Identity where
    return x = Identity x
    (Identity x) >>= f = f x
    fail message = error message
```

Примеры монад: монада identity

Вычисления в монаде `Identity` — это просто обычные вычисления

```
xplusy x y =
    do
       x' <- x
       y' <- y
       return $ x' + y'
xplusy (Identity 2) (Identity 3) === (Identity 5)
```

Примеры монад: монада maybe

Контекст: вычисления с возможностью неудачи

```
data Maybe a = Just a | Nothing
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
instance Monad Maybe where
    return x = Just x
    (Just x) >>= f = f x
    Nothing  >>= f = Nothing
    fail message = Nothing
```

Примеры монад: монада maybe

Вычисления в монаде Maybe — это обычные вычисления, но любое действие может «не получиться»

```
xplusy x y =
    do
        x' <- x
        y' <- y
        return $ x' + y'
xplusy (Just 2) (Just 3) === (Just 5)
xplusy (Just 42) (Nothing) === Nothing
```

Расширение монады maybe: монада error

Контекст: вычисления с возможностью неудачи

```
data Error a = Success a | Error String
instance Functor Error where
    fmap f (Success x) = Success (f x)
    fmap f (Error s) = Error s
instance Monad Error where
    return x = Success x
    (Success x) >>= f = f x
    (Error s)  >>= f = (Error s)
    fail message = Error message
```

Примеры монад: монада `list`

Контекст (по умолчанию): декартово произведение списков

```
instance Functor [] where
    fmap = map
instance Monad [] where
    return x = [x]
    (>>=) = flatMap
    fail message = []
```

Примеры монад: монада List

Вычисления в монаде List — это обычные вычисления, но любое действие комбинируется в аргументах «каждый с каждым»

```
xplusy x y =
    do
       x' <- x
       y' <- y
       return $ x' + y'
xplusy [0..2] [4,5] === [4,5,6,5,6,7]
xplusy [0..10000] [] === []
```

LIST COMPREHENSIONS

```haskell
-- все сочетания чисел от 0 до 5
allComb = do
            x <- [0..5]
            y <- [0..5]
            return (x,y)
allComb = [(x,y) | x <- [0..5], y <- [0..5]]
```

Примеры монад: монады, не имеющие никакого побочного смысла

- Монада `Reader`
- Монада `Writer`
- Монада `State`

монада reader

Контекст: вычисления с контекстом, который можно читать

```
data Reader env a = Reader (env -> a)
runReader (Reader f) env = f env
ask :: Reader env env
ask = Reader (x -> x)
instance Monad (Reader env) where
    return x = Reader (\_ -> x)
    (Reader f) >>= g = Reader $ \x -> runReader (g (
```

монада reader

Контекст: вычисления с контекстом

```
xplusenv x = do
               x' <- x
               env <- ask
               return $ x' + env
fortyTwoPlusEnv = xplusenv (return 42)
runReader fortyTwoPlusEnv 3 === 45
runReader fortyTwoPlusEnv 8 === 50
```

монада writer

Контекст: вычисления с контекстом, который можно писать

```
data Writer env a = Writer (a, env)
runWriter (Writer x) = x
tell env = Writer ((), env)

instance (Monoid w) => Monad (Writer w) where
    return a                = Writer (a,mempty)
    (Writer (a,w)) >>= f = let (a',w') = runWriter
                           Writer (a',w `mappend` w
```

монада writer

Контекст: вычисления с контекстом, который можно писать

```
logFirst x y = do
                 x' <- x
                 y' <- y
                 if (x' > y') then tell [x']
                 return $ x + y
foo = logFirst (return 5) (return 4)
fee = logFirst (return 13) (return 12)
bar = logFirst foo fee
runWriter foo === (9, [5])
runWriter fee === (25, [13])
runWriter bar === (34, [5,13])
```

монада state

Контекст: вычисления с контекстом, который можно и читать, и писать

```
data State env a = State (env -> (a, env))
runState (State f) = f
put newState = State $ \ _ -> ((), newState)
get = State $ \ current -> (current, current)
instance Monad (State s) where
    return a          = State $ \s -> (a,s)
    (State x) >>= f = State $ \s -> let (v,s') = x
                                    runState (f v) s'
```

Промежуточные итоги

- Абстракция монады позволяет на *чистых функциях* реализовать псевдо-изменяемое состояние с сохранением всех следующих особенностей:
  - Порядок выполнения операций
  - Независимость результата от точки в коде, в которой операция происходит

Промежуточные итоги

- Почему это по прежнему функционально-чистый подход?
  - В рамках монадных операций «создать» контекст невозможно, для этого всегда нужны внешние функции
  - Сам контекст при этом изолирован в рамках набора монадных операций, для извлечения результатов так же всегда нужны внешние функции

монада `io`

Представим себе, что *весь окружающий программу мир* — это состояние, которое можно поместить в монаду `State`.

Только магической функции `runState` нет.

Получим монаду `IO`.

Монада io

Disclaimer: это не настоящий код!

```
data IO a = ...
instance Monad IO where
    return a = ... -> (world, a)
    io >>= f = (world, x) -> ... -> (getWorld f x,
```

Монада io

Как войти в вычисления в монаде `IO`?

Через функцию `main :: [String] -> IO ()`

Как достать значение из `IO x`?

Никак

Монады-трансформеры

Что делать, если вам нужны вычисления с множеством контекстов?

- Возможность ошибки + состояние
  - Парсеры
  - Разбор бинарных файлов
- Состояние + IO
- ...

МОНАДЫ-ТРАСФОРМЕРЫ

- Берём монаду и реализуем все операции так, чтобы они прозрачно передавались монаде «внутри»
- Примеры:
  - `ErrorT m a`
  - `StateT m s a`
  - `ReaderT m env a`
  - `WriterT m env a`

```
type GetCtx a = State (Input, Position) a
data Get a = ErrorT GetCtx a
```