Курс: Функциональное программирование

Лекция 9. Использование монад

Денис Николаевич Москвин

25.11.2011

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

План лекции

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

План лекции

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

Проблема ввода-вывода

В чистых языках, где значение функции зависит только от её параметров, ввод-вывод представляет собой проблему. Функция

```
getCharFromConsole :: Char
```

всегда должна возвращать одно и то же!

Решение:

```
getCharFromConsole :: World -> (Char, World)
```

При этом доступ к значениям типа `World` должен быть ограничен.

Тип IO

Значение типа IO — это вычисление, которое при выполнении осуществляет некоторый ввод-вывод.

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

data RealWorld :: *
```

Тип, про который в документации сказано «deeply magical» :)

Единственный способ выполнить действие ввода-вывода — связать его с функцией `main` программы.

Монада `IO`

```
instance  Monad IO  where
  return :: a -> IO a
  (>>=) :: IO a -> (a -> IO b) -> IO b
```

Реализация не важна! Важны гарантии, которые она должна давать:

- Побочный эффект каждого действия происходит один раз.

- Побочные эффекты действий происходят в заданном порядке.

Основные функции консольного ввода-вывода

Ввод:

```
getChar          :: IO Char
getLine          :: IO String
getContents      :: IO String
```

Вывод:

```
putChar          :: Char -> IO ()
putStr           :: String -> IO ()
putStrLn         :: String -> IO ()
print            :: Show a => a -> IO ()
```

Пример ввода-вывода

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
```

Какой тип имеет `main`?

Пример (2)

Как имея `getChar` сделать `getLine`?

```
getLine' :: IO String
getLine' = do
    c <- getChar
    if c == '\n' then
        return []
    else do
        cs <- getLine'
        return (c:cs)
```

В `if...then...else` конструкции повторный вызов `do` необходим!

Пример (3)

```
sequence_  :: Monad m => [m a] -> m ()
sequence_  =  foldr (>>) (return ())

putStr'  :: String -> IO ()
putStr'  =  sequence_ . map putChar

mapM_    :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f  =  sequence_ . map f

putStr''  :: String -> IO ()
putStr''  =  mapM_ putChar
```

Есть более «полновесные» аналоги

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

План лекции

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

Монада `Reader` (Environment)

Вычисление, допускающее чтение значений из разделяемого окружения.

```
instance Monad ((->) r) where
  return x = \_ -> x
  m >>= k  = \e -> k (m e) e
```

`return :: a -> (r -> a)` — просто игнорирует окружение;

`(>>=) :: (r -> a) -> (a -> (r -> b)) -> (r -> b)` — передаёт полученное окружение в оба вычисления.

Для большей общности вводят тип

```
newtype Reader r a = Reader { runReader :: (r -> a) }
```

Монада `Reader` (2)

На самом деле тип `Reader r a` определён по-другому, нам интересен публичный интерфейс его сборки и разборки:

```
reader :: (r -> a) -> Reader r a
runReader :: Reader e a -> r -> a

instance Monad (Reader r) where
  return x  =  reader $ \e -> x
  mv >>= k  =  reader $ \e -> let v = runReader mv e
                              in runReader (k v) e

simpleReader :: Reader Int String
simpleReader =
  reader (\e -> "Environment is " ++ show e)

*Fp09> runReader simpleReader 42
"Environment is 42"
```

Монада `Reader` (3)

Функция `ask :: Reader r r` возвращает окружение

```
*Fp09> runReader ask "Hello!"
"Hello!"

type User = String
type Password = String
type UsersTable = [(User,Password)]
pwds :: UsersTable
pwds = [("Bill","123456"),("Ann","qwerty"),("John","2sRqf78P")]

firstUser :: Reader UsersTable String
firstUser = do
  e <- ask
  return $ fst (head e)

*Fp09> runReader firstUser pwds
"Bill"
```

Монада `Reader` (4)

Функция `asks :: (r -> a) -> Reader r a` возвращает результат выполнения функции над окружением

```
getPwdLen :: User -> Reader UsersTable Int
getPwdLen person = do
  mbPwd <- asks $ lookup person
  let mbLen = fmap length mbPwd
  let len = fromMaybe (-1) mbLen
  return len

*Fp09> runReader (getPwdLen "Ann") pwds
6
*Fp09> runReader (getPwdLen "Ann") []
-1
```

Монада `Reader` (5)

Функция `local :: (r -> r) -> Reader r a -> Reader r a` позволяет локально модифицировать окружение

```
usersCount :: Reader UsersTable Int
usersCount = asks length

localTest :: Reader UsersTable (Int,Int)
localTest = do
  count1 <- usersCount
  count2 <- local (("Mike","1"):) usersCount
  return (count1, count2)

*Fp09> runReader localTest pwds
(3,4)
```

Класс типов MonadReader

В действительности `ask` и `local` определены в классе типов

```
class (Monad m) => MonadReader r m | m -> r where
  ask   :: m r
  local :: (r -> r) -> m a -> m a

asks :: (MonadReader r m) => (r -> a) -> m a
asks f = do
  r <- ask
  return (f r)
```

Простейший `Reader` — частично применённая стрелка:

```
instance MonadReader r ((->) r) where
  ask       = id
  local f m = m . f
```

План лекции

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

Монада `Writer`

Вычисление, допускающее запись в лог.

```
newtype Writer w a = Writer {runWriter :: (a, w)}

writer :: (a, w) -> Writer w a
runWriter :: Writer w a -> (a, w)

instance (Monoid w) => Monad (Writer w) where
  return x   =  writer (x, mempty)
  mv >>= k   =  let (x,u) = runWriter mv
                    (y,v) = runWriter $ k x
                in writer (y, u `mappend` v)
```

Монада `Writer`: примеры

Простейшие примеры:

```
*Fp09> runWriter (return 3 :: Writer String Int)
(3,"")
*Fp09> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
*Fp09> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

Монада `Writer`: расширенный пример (0)

Опишем базу данных для интернет-магазина:

```
type Vegetable = String

type Price = Double
type Qty = Double
type Cost = Double

type PriceList = [(Vegetable,Price)]

prices :: PriceList
prices = [("Potato",13),("Tomato",55),("Apple",48)]
```

Монада `Writer`: расширенный пример (1)

Функция `tell :: Monoid w => w -> Writer w ()` позволяет задать вывод

```
addVegetable :: Vegetable -> Qty -> Writer (Sum Cost) (Vegetable, Price)
addVegetable veg qty = do
  let pr = fromMaybe 0 $ lookup veg prices
  let cost = qty * pr
  tell $ Sum cost
  return (veg, pr)

*Fp09> runWriter $ addVegetable "Apple" 100
(("Apple",48.0),Sum {getSum = 4800.0})
```

Монада `Writer`: расширенный пример (2)

Суммарная стоимость копится «за кадром»

```
myCart0 = do
  x1 <- addVegetable "Potato" 3.5
  x2 <- addVegetable "Tomato" 1.0
  x3 <- addVegetable "AGRH!!" 1.6
  return [x1,x2,x3]

*Fp09> runWriter myCart0
([("Potato",13.0,45.5),("Tomato",55.0,55.0),("AGRH!!",0.0,0.0)],
Sum {getSum = 100.5})
```

Монада `Writer`: расширенный пример (3)

Если хотим знать промежуточные стоимости, используем

```
listen :: Monoid w => Writer w a -> Writer w (a, w)
```

```
myCart1 = do
  x1 <- listen $ addVegetable "Potato" 3.5
  x2 <- listen $ addVegetable "Tomato" 1.0
  x3 <- listen $ addVegetable "AGRH!!" 1.6
  return [x1,x2,x3]
```

```
*Fp09> runWriter myCart1
([(("Potato",13.0),Sum {getSum = 45.5}),(("Tomato",55.0),
Sum {getSum = 55.0}),(("AGRH!!",0.0),Sum {getSum = 0.0})],
Sum {getSum = 100.5})
```

Монада `Writer`: расширенный пример (4)

Для модификации лога используем

```
censor :: Monoid w => (w -> w) -> Writer a -> Writer a

myCart0' = censor (discount 10.0) myCart0

discount proc (Sum x) = Sum $ if x < 100 then x
                              else x * (100.0 - proc) / 100.0

*Fp09> runWriter myCart0
([("Potato",13.0),("Tomato",55.0),("AGRH!!",0.0)],
Sum {getSum = 100.5})
*Fp09> runWriter myCart0'
([("Potato",13.0),("Tomato",55.0),("AGRH!!",0.0)],
Sum {getSum = 90.45})
```

Класс типов MonadWriter

В действительности `tell` и `listen` определены в классе типов

```
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
    tell   :: w -> m ()
    listen :: m a -> m (a, w)
    pass   :: m (a, w -> w) -> m a

listens :: (MonadWriter w m) => (w -> b) -> m a -> m (a, b)
listens f m = do
    ~(a, w) <- listen m
    return (a, f w)

censor :: (MonadWriter w m) => (w -> w) -> m a -> m a
censor f m = pass $ do
    a <- m
    return (a, f)
```

План лекции

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

Монада State

Вычисление, которое содержит изменяемое состояние.

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

state :: (s -> (a, s)) -> State s a
runState :: State s a -> s -> (a, s)

instance Monad (State s) where
  return x   =  state $ \st -> (x, st)
  mv >>= k   =  state $ \st -> let (x, st') = runState mv st
                                   mv' = k x
                               in runState mv' st'

*Fp09> runState (return 3 :: State String Int) "Hi from State!"
(3,"Hi from State!")
```

Класс типов MonadState

Специальные функции для работы с состоянием

```
class (Monad m) => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()

modify :: (MonadState s m) => (s -> s) -> m ()
modify f = do  s <- get
               put (f s)

gets :: (MonadState s m) => (s -> a) -> m a
gets f = do  s <- get
             return (f s)

instance MonadState (State s) s where
    get   = state $ \s -> (s,s)
    put s = state $ \_ -> ((),s)
```

Монада `State`: пример

```
tick :: State Int Int
tick = do  n <- get
           put (n+1)
           return n

*Fp09> runState tick 3
(3,4)
*Fp09> evalState tick 3
3
*Fp09> execState tick 3
4

succ' :: Int -> Int
succ' n = execState tick n

plus :: Int -> Int -> Int
plus n x = execState (sequence $ replicate n tick) x
```