Функциональное программирование
Лекция 10. Монады

Денис Николаевич Москвин

ВШЭ СПб, СПбШФМиКН,
бакалавриат ПМиИ

20.11.2020

План лекции

План лекции

Стрелка Клейсли

Хотим расширить чистые функции (`a -> b`) до вычислений с «эффектом», которые

- иногда могут завершиться неудачей: `a -> Maybe b`
- могут возвращать много результатов: `a -> [b]`
- иногда могут завершиться ошибкой: `a -> (Either s) b`
- могут делать записи в лог: `a -> (s,b)`
- могут читать из внешнего окружения: `a -> ((->) e) b`
- работают с мутабельным состоянием: `a -> (State s) b`
- делают ввод/вывод (файлы, консоль): `a -> IO b`
- не делают ничего: `a -> Identity b`

Обобщая, получим *стрелку Клейсли*: `a -> m b`

Стрелка Клейсли

Хотим расширить чистые функции (`a -> b`) до вычислений с «эффектом», которые

- иногда могут завершиться неудачей: `a -> Maybe b`
- могут возвращать много результатов: `a -> [b]`
- иногда могут завершиться ошибкой: `a -> (Either s) b`
- могут делать записи в лог: `a -> (s,b)`
- могут читать из внешнего окружения: `a -> ((->) e) b`
- работают с мутабельным состоянием: `a -> (State s) b`
- делают ввод/вывод (файлы, консоль): `a -> IO b`
- не делают ничего: `a -> Identity b`

Обобщая, получим *стрелку Клейсли*: `a -> m b`
Стрелка Клейсли обеспечивает зависимость *эффекта* от *значения*. Например, `\n -> replicate n 'A'`.

Понятие монады

Какими должны быть требования к оператору над типами `m` в стрелке Клейсли `a -> m b`?

- Должен иметься универсальный интерфейс для упаковки значения в контейнер `m`.
- Должен иметься универсальный интерфейс для композиции вычислений с эффектом (стрелок Клейсли):

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Используя только интерфейс функтора не реализовать:

```
k1 <=< k2 = \x -> k1 <$> (k2 x) -- увы :: m (m c)
```

- Нет универсального интерфейса для извлечения значения из контейнера `m`.
  (Эффект в общем случае нельзя отбросить.)

Если бы миром правили теоретики, ...

... то класс типов `Monad` был бы определён так

```
class Pointed m => Monad m where
  join :: m (m a) -> m a
```

В нашем бренном мире, однако

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b -- произносят bind
  (>>) :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2
  fail :: String -> m a
  fail s = error s
infixl 1 >>, >>=
```

В современном GHC (>=8.6) `fail` изгнан в дочерний класс типов `MonadFail`.

Функция return :: a -> m a

`return :: a -> m a` определяет тривиальную стрелку Клейсли. Функция `pure :: a -> f a` из `Applicative` — полный её аналог.
Позволяет превратить `f :: a -> b` в стрелку Клейсли:

```
toKleisli :: Monad m => (a -> b) -> (a -> m b)
toKleisli f = \x -> return (f x)
```

```
GHCi> :type toKleisli cos
toKleisli cos :: (Monad m, Floating b) => b -> m b
GHCi> (toKleisli cos 0) :: Maybe Double
Just 1.0
GHCi> (toKleisli cos 0) :: [Double]
[1.0]
*Fp09> (toKleisli cos 0) :: IO Double
1.0
```

Функция (>>=) :: m a -> (a -> m b) -> m b

На что похож «связыватель» (>>=)?

```
($) :: (a -> b) -> a -> b

(&) :: a -> (a -> b) -> b
(&) = flip ($)
infixl 1 &
```

```
GHCi> (+1) $ (*3) $ (+2) $ 5
22
GHCi> 5 & (+2) & (*3) & (+1)
22
```

Конвейер вычислений развернут в обратную сторону.

Функция (>>=) :: m a -> (a -> m b) -> m b (2)

Имеется обратный «связыватель» (=<<) = flip (>>=), похожий на знакомые операции

```
($)     ::                          (a -> b) ->   a ->   b
(<$>)   :: Functor f =>             (a -> b) -> f a -> f b
(<*>)   :: Applicative f =>  f (a -> b) -> f a -> f b
(=<<)   :: Monad m =>          (a -> m b) -> m a -> m b
```

Прямой «связыватель» (>>=) :: m a -> (a -> m b) -> m b похож на их «флипы» (NB: для <*> не «флип»!)

```
(&)     ::                            a ->   (a -> b) ->   b
(<&>)   :: Functor f =>             f a ->   (a -> b) -> f b
(<**>)  :: Applicative f =>  f a -> f (a -> b) -> f b
(>>=)   :: Monad m =>          m a -> (a -> m b) -> m b
```

Функция fail

Вызов `fail` в современном GHC (>=8.6) ссылается на функцию из класса типов `Control.Monad.Fail.MonadFail`:

```
class Monad m => MonadFail m where
    fail :: String -> m a
```

При реализации представителей `Monad` функцию `fail` в 8.6 допустимо реализовывать, но уже бессмысленно.

План лекции

Монада Identity

Напишем представителя Monad для простейшего типа Identity, представляющего собой простую упаковку для другого типа:

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
  return :: a -> Identity a
  return  =  Identity

  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  Identity x >>= k  =  k x
  --m >>= k  =  k (runIdentity m)
```

В стрелку Клейсли k передаётся «распакованное» значение.

«Использование» монады Identity

Зададим нетривиальную стрелку Клейсли

```
wrap'n'succ :: Integer -> Identity Integer
wrap'n'succ x = Identity (succ x)
```

```
GHCi> runIdentity $ wrap'n'succ 3
4
GHCi> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ
5
GHCi> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ >>=
wrap'n'succ
6
```

Видно, что >>= работает как &.

Законы класса типов Monad

Для любого представителя `Monad` должны выполняться законы

```
return a >>= k      ≡  k a

m >>= return        ≡  m

(m >>= k) >>= k'    ≡  m >>= (\x -> k x >>= k')
```

Первые два закона выражают тривиальную природу `return`

```
GHCi> runIdentity $ wrap'n'succ 3
4
GHCi> runIdentity $ return 3 >>= wrap'n'succ
4
GHCi> runIdentity $ wrap'n'succ 3 >>= return
4
```

Третий закон Monad

Третий закон класса типов `Monad`

```
(m >>= k) >>= k'   ≡   m >>= (\x -> k x >>= k')

 m >>= k >>= k'    ≡   m >>= \x -> k x >>= k'
```

задаёт некоторое подобие ассоциативности

```
GHCi> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ >>= w
rap'n'succ
6
GHCi> runIdentity $ wrap'n'succ 3 >>= (\x -> wrap'n'suc
c x >>= wrap'n'succ)
6
```

Третий закон Monad (2)

Прицепим return (можно в силу второго закона), и применим третий закон ко всем связываниям (>>=)

```
goWrap0 = wrap'n'succ 3 >>=
          wrap'n'succ >>=
          wrap'n'succ >>=
          return
goWrap1 = wrap'n'succ 3 >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>=  \z ->
          return z))
```

```
GHCi> runIdentity goWrap0
6
GHCi> runIdentity goWrap1
6
```

Третий закон Monad (3)

```
goWrap1 = wrap'n'succ 3 >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>=  \z ->
          return z))

goWrap2 = wrap'n'succ 3 >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>=  \z ->
          return (x,y,z)))
```

```
GHCi> runIdentity goWrap1
6
GHCi> runIdentity goWrap2
(4,5,6)
```

Ой, мы изобрели императивное программирование!

Третий закон Monad (4)

Можем использовать `let`-связывание для обычных выражений:

```
goWrap3 = let i = 3 in
          wrap'n'succ i >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>=  \z ->
          return (i,x,y,z)))
```

```
GHCi> runIdentity goWrap3
(3,4,5,6)
```

Третий закон Monad (5)

Если результат не интересен, можно его игнорировать, используя усеченный связыватель >>:

```
goWrap4 = let i = 3 in
          wrap'n'succ i >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>
          return (i,x,y)))
```

```
GHCi> runIdentity goWrap4
(3,4,5)
```

do-нотация

- Для удобства «императивного программирования» внутри монады вводят специальную нотацию.

Правила трансляции в Haskell Kernel для do-нотации

```
do {e}                    ≡  e
do {e; stmts}             ≡  e >> do {stmts}
do {p <- e; stmts}        ≡  e >>= \p -> do {stmts}
do {let v = exp; stmts}   ≡  let v = exp in do {stmts}
```

Здесь все `e :: m a`.

- Третье правило в действительности сложнее:
  если сопоставление с образцом `p` неудачно, то вызывается `fail` (см. Haskell Report 2010, 3.14 и MonadFail proposal (MFP)).
- Обычно используют правило отступа, а не фигурные скобки и точку с запятой.

do-нотация: пример

```
goWrap4 =     let i = 3 in -- выравнивание для красоты
              wrap'n'succ i >>= (\x ->
              wrap'n'succ x >>= (\y ->
              wrap'n'succ y >>
              return (i,x,y)))
goWrap5 = do let i = 3     -- выравнивание обязательно
              x <- wrap'n'succ i
              y <- wrap'n'succ x
              wrap'n'succ y
              return (i,x,y)
```

```
GHCi> runIdentity goWrap4
(3,4,5)
GHCi> runIdentity goWrap5
(3,4,5)
```

План лекции

Монада Maybe

Простейшая монада, обеспечивающая эффект отсутствующего значения (ошибки).

```
instance  Monad Maybe  where
  return :: a -> Maybe a
  return  =  Just

  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (Just x) >>= k  =  k x
  Nothing  >>= _  =  Nothing

  (>>) :: Maybe a -> Maybe b -> Maybe b
  (Just _) >>  m  =  m
  Nothing  >>  _  =  Nothing

  fail :: String -> Maybe a
  fail _          =  Nothing
```

Монада Maybe: пример (1)

```
type Name = String
type ParentsTable = [(Name, Name)]

fathers, mothers :: ParentsTable
fathers =
  [("Bill","John"),("Ann","John"),("John","Piter")]
mothers =
  [("Bill","Jane"),("Ann","Jane"),("John","Alice")
  ,("Jane","Dorothy"),("Alice","Mary")]

getM, getF :: Name -> Maybe Name
getM person = lookup person mothers
getF person = lookup person fathers
```

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Монада Maybe: пример (2)

Ищем прабабушку Билла по материнской линии отца

```
GHCi> getF "Bill" >>= getM >>= getM
Just "Mary"
GHCi> do { f <- getF "Bill"; m <- getM f; getM m }
Just "Mary"
```

- Первая форма удобна только когда результат предыдущего действия должен передаваться непосредственно в следующее.
- В остальных случаях предпочтительна do-нотация.

Монада Maybe: пример (3)

```
granmas person = do
  m   <- getM person
  gmm <- getM m
  f   <- getF person
  gmf <- getM f
  return (gmm, gmf)
```

```
GHCi> granmas "Ann"
Just ("Dorothy","Alice")
GHCi> granmas "John"
Nothing
```

Хотя одна бабушка у Джона есть, но, как только результат одного действия стал `Nothing`, все дальнейшие действия игнорируются.

План лекции

Список как монада

Монада списка представляет недетерминированное вычисление (с нулём или большим числом возможных результатов).

```
instance Monad [] where
  (>>=) :: a -> [a]
  return x  =  [x]

  (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= k  =  concat (map k xs)

  fail :: String -> [a]
  fail _    =  []
```

Здесь `map k xs :: [[b]]` «уплощается» `concat`'ом.

```
GHCi> "abc" >>= replicate 4
"aaaabbbbcccc"
```

Список как монада: пример

Следующие три списка — это одно и то же:

```
list1 = [ (x,y) | x <- [1,2,3], y <- [1,2], x /= y ]

list2 = do
  x <- [1,2,3]
  y <- [1,2]
  True <- return (x /= y)
  return (x,y)

list3 =
  [1,2,3]          >>= (\x ->
  [1,2]            >>= (\y ->
  return (x /= y) >>= (\r ->
  case r of True -> return (x,y)
            _    -> fail "Will be ignored :)")))
```

Отличие Monad от Applicative

В монадах результат предыдущего вычисления может влиять на «структуру» последующих:

```
GHCi> do {a <- [1..3]; b <- [a..3]; return (a,b)}
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Для аппликативных функторов такое невозможно

```
GHCi> (,) <$> [1..3] <*> [3..3]
[(1,3),(2,3),(3,3)]
GHCi> (,) <$> [1..3] <*> [2..3]
[(1,2),(1,3),(2,2),(2,3),(3,2),(3,3)]
GHCi> (,) <$> [1..3] <*> [1..3]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```

План лекции

Класс типов MonadFail

Класс типов `MonadFail` предназначен для обработки неудачного сопоставления с образцом слева от `<-` в `do`-нотации.

```
class Monad m => MonadFail m where
  fail :: String -> m a

instance MonadFail Maybe where
  fail _ = Nothing
```

```
GHCi> do {3 <- Just 5; return 'Z'}
Nothing
GHCi> do {3 <- Identity 5; return 'Z'}
error: Could not deduce (MonadFail Identity) arising
      from a do statement with the failable pattern `3'
```

Сообщение об ошибке выдает тайпчекер, код не пройдет компиляцию.

Класс типов MonadFail: трансляция в Kernel

do-нотация транслируется в Haskell Kernel по-разному, в зависимости от того является ли образец «failable» или нет:

```
GHCi> :t do {x <- return 5; return 'Z'}
do {x <- return 5; return 'Z'} :: Monad m => m Char

GHCi> :t do {3 <- return 5; return 'Z'}
do {3 <- return 5; return 'Z'} :: MonadFail m => m Char

GHCi> :t do {~3 <- return 5; return 'Z'}
do {~3 <- return 5; return 'Z'} :: Monad m => m Char
```

Неопровержимые образцы не являются «failable».

Класс типов MonadFail: трансляция в Kernel (2)

`data` с одним конструктором и `newtype` не «failable» сами по себе, но могут оказаться «failable» при вложении образцов.

```
GHCi> :t do {(s,x) <- return ("Answer",42); return 'Z'}
do {(s,x) <- return ("Answer",42); return 'Z'}
    :: Monad m => m Char

GHCi> :t do {(s,42) <- return ("Answer",42); return 'Z'}
do {(s,42) <- return ("Answer",42); return 'Z'}
    :: MonadFail m => m Char

GHCi> :t do {Left x <- return (Left 42); return 'Z'}
do {Left x <- return (Left 42); return 'Z'}
    :: MonadFail m => m Char
```