Функциональное программирование
Лекция 9. Монады

Денис Николаевич Москвин

СПбАУ РАН, CSC

15.04.2015

План лекции

1. Класс типов `Monad`
2. Монада `Maybe`
3. Список как монада

План лекции

Стрелка Клейсли

Хотим расширить чистые функции (тип `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`

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

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)
```

- Нет универсального интерфейса для извлечения значение 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 Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b  -- произносят bind

  (>>) :: m a -> m b -> m b
  m >> k   =   m >>= \_ -> k
  fail :: String -> m a
  fail s   =   error s
infixl 1 >>, >>=
```

Функция `return :: a -> m a`

`return :: a -> m a` определяет тривиальную стрелку Клейсли.

`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)
```

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

euro :: a -> (a -> b) -> b
euro = flip ($)
```

GHCi

```
*Fp09> (+1) $ (*3) $ (+2) $ 5
22
*Fp09> 5 `euro` (+2) `euro` (*3) `euro` (+1)
22
```

Конвеер вычислений развернулся в другую сторону!

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

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

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

Прямой «связыватель» `(>>=) :: m a -> (a -> m b) -> m b` похож на их «флипы»

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

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 x           =  Identity x
  Identity x >>= k   =  k x
```

В стрелку Клейсли `k` передаётся «распакованное» значение.

```
return :: a -> Identity a
(>>=)  :: Identity a -> (a -> Identity b) -> Identity b
```

«Использование» монады `Identity`

Зададим нетривиальную стрелку Клейсли

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

GHCi

```
*Fp09> runIdentity $ wrap'n'succ 3
4
*Fp09> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ
5
*Fp09> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ >>= wrap
'n'succ
6
```

Видно, что `(>>=)` работает как `euro`.

Законы класса типов Monad

Для любого представителя Monad должны выполняться

Законы класса типов Monad

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

Первые два закона выражают тривиальную природу return

GHCi

```
*Fp09> runIdentity $ wrap'n'succ 3
4
*Fp09> runIdentity $ return 3 >>= wrap'n'succ
4
*Fp09> runIdentity $ wrap'n'succ 3 >>= return
4
```

Третий закон Monad

Третий закон класса типов Monad

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

задаёт некоторое подобие ассоциативности

GHCi

```
*Fp09> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ >>= wrap
'n'succ
6
*Fp09> runIdentity $ wrap'n'succ 3 >>= (\x -> wrap'n'succ 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))
```

```
*Fp09> runIdentity goWrap0
6
*Fp09> 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)))
```

```
*Fp09> runIdentity goWrap1
6
*Fp09> 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

```
*Fp09> runIdentity goWrap3
(3,4,5,6)
```

Третий закон Monad (5)

Если результат не интересен, можно его игнорировать:

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

GHCi

```
*Fp09> runIdentity goWrap4
(3,4,5)
```

do-нотация

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

Правила трансляции для do-нотации

```
do { e1 ; e2 }          ≡  e1 >> e2
do { p <- e1; e2 }      ≡  e1 >>= \p -> e2
do { let v = e1; e2 }   ≡  let v = e1 in do e2
```

- Второе правило в действительности сложнее:
  если сопоставление с образцом `p` неудачно, то вызывается `fail`.
- Обычно используют правило отступа, а не фигурные скобки и точку с запятой.

do-нотация: пример

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

```
*Fp09> runIdentity goWrap4
(3,4,5)
*Fp09> runIdentity goWrap5
(3,4,5)
```

План лекции

Монада `Maybe`

Простейшая монада, обеспечивающая эффект ошибки (исключения). Любая ошибка представляется как значение `Nothing`.

```
instance  Monad Maybe  where
  return           =  Just

  (Just x) >>= k   =  k x
  Nothing  >>= _   =  Nothing

  (Just _) >>  k   =  k
  Nothing  >>  _   =  Nothing

  fail _           =  Nothing
```

Монада Maybe: пример (1)

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

fathers, mothers :: DataBase
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

```
*Fp09> getF "Bill" >>= getM >>= getM
Just "Mary"
*Fp09> 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

```
*Fp09> granmas "Ann"
Just ("Dorothy","Alice")
*Fp09> 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 вычисление с нулём или большим числом возможных результатов.

```
instance Monad [] where
  return x  =  [x]
  xs >>= k  =  concat (map k xs)
  fail _    =  []
```

Связывание (>>=) отображает стрелку `k :: a -> [b]` на список `xs :: [a]` и выполняет конкатенацию получившегося списка списков типа `[[b]]`.

GHCi

```
*Fp09> "abc" >>= replicate 3
"aaabbbccc"
```

Список как монада: пример

Следующие три списка — это одно и то же:

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

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

list3 =
  [1,2,3]       >>= (\x ->
  [1,2,3]       >>= (\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:

```
*Fp09> 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

```
*Fp09> (,) <$> [1..3] <*> [3..3]
[(1,3),(2,3),(3,3)]
*Fp09> (,) <$> [1..3] <*> [2..3]
[(1,2),(1,3),(2,2),(2,3),(3,2),(3,3)]
*Fp09> (,) <$> [1..3] <*> [1..3]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```