Функциональное программирование
Лекция 10. Монады

Денис Николаевич Москвин

СПбГУ, факультет МКН,
бакалавриат «Современное программирование», 2 курс

13.11.2025

План лекции

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

План лекции

Стрелка Клейсли

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

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

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

Стрелка Клейсли

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

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

Как описать интерфейс сцепления двух эффектов в один?

- Можно задать сигнатуру для композиции стрелок Клейсли

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

- Первый шаг реализации по умолчанию очевиден: применить первый аргумент к третьему

```
k1 >=> k2 = \x -> _ (k1 x) k2
```

Здесь дырка имеет тип `m a -> (a -> m b) -> m b`.

- Если `m` — функтор, то можно иначе

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

Тут поможет переходник типа `m (m b) -> m b`.

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
```

Однако вычислительно удобнее так

```
infixl 1 >>, >>=
class Applicative m => Monad m where
  {-# MINIMAL (>>=) #-}
  (>>=) :: m a -> (a -> m b) -> m b -- произносят bind

  (>>) :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2

  return :: a -> m a
  return = pure
```

В современном GHC (>=8.6) fail изгнан в MonadFail.

Функция return :: a -> m a

Функция return определяет безэффектную стрелку Клейсли.
Функция pure из Applicative — полный её аналог.

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

```
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]
GHCi> toKleisli cos 0 :: IO Double
1.0
```

Операторы связывания: синтаксис использования

Помимо `>>=` имеется `(=<<) = flip (>>=)`; они похожи на `&` и `$` соответственно

```
(>>=) :: m a -> (a -> m b) -> m b
(&)   ::   a -> (a ->   b) ->   b

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

Конвейер вычислений может быть развернут в любую сторону

```
GHCi> 5 & (+2) & (*3)
21
GHCi> Just 5 >>= toKleisli (+2) >>= toKleisli (*3)
Just 21
GHCi> (*3) $ (+2) $ 5
21
GHCi> toKleisli (*3) =<< toKleisli (+2) =<< Just 5
Just 21
```

Бинарные операторы монадической иерархии

Обратный байндер =<< похож 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 байндер >>= похож на их «флипы»

```
(&)     ::                            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
```

Напомним, однако, что оператор <**> не «флип» для <*>!

План лекции

Монада Maybe

Простейшая монада, обеспечивающая эффект отсутствующего значения (ошибки).

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

  return :: a -> Maybe a
  return  =  Just
```

Монада Maybe: стрелки Клейсли

```
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

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

Ищем прабабушку по материнской линии отца

```
GHCi> getF "Bill"
Just "John"
GHCi> getF "Bill" >>= getM
Just "Alice"
GHCi> getF "Bill" >>= getM >>= getM
Just "Mary"

GHCi> getF "John" >>= getM >>= getM
Nothing
```

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> getF "Bill"
Just "John"
GHCi> return "Bill" >>= getF
Just "John"
GHCi> getF "Bill" >>= return
Just "John"
```

Третий закон Monad

Третий закон Monad задаёт некоторое подобие ассоциативности

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

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

Третий закон выполняется для Maybe

```
GHCi> getF "Bill" >>= getM >>= getM
Just "Mary"
GHCi> getF "Bill" >>= \x -> getM x >>= getM
Just "Mary"

ghci> getF "John" >>= getM >>= getM
Nothing
ghci> getF "John" >>= \x -> getM x >>= getM
Nothing
```

Третий закон для цепочки вычислений

Прицепим `return` (можно в силу второго закона), применим 2 раза третий закон и сделаем η-экспансию

```
go0 p = getF p >>=
        getM >>=
        getM >>=
        return
go1 p = getF p >>= (\x ->
        getM x >>= (\y ->
        getM y >>= (\z ->
        return z)))
```

```
GHCi> go0 "Bill"
Just "Mary"
GHCi> go1 "Bill"
Just "Mary"
```

Доступ к промежуточным значениям

```
go1 p = getF p >>= \x ->
        getM x >>= \y ->
        getM y >>= \z ->
        return z
go2 p = getF p >>= \x ->
        getM x >>= \y ->
        getM y >>= \z ->
        return (x,y,z)
```

Именование промежуточных значений дает к ним доступ.

```
GHCi> go1 "Bill"
Just "Mary"
GHCi> go2 "Bill"
Just ("John","Alice","Mary")
```

Ой, мы изобрели императивное программирование!

Преимущества перед let-версией

«Императивной» гибкости можно достичь через let-выражения, но лишь безэффектным образом.

```
go2 p  = getF p >>= \x ->
         getM x >>= \y ->
         getM y >>= \z ->
         return (x,y,z)
go2' p = let Just x = getF p in
         let Just y = getM x in
         let Just z = getM y in
         Just (x,y,z)
```

```
GHCi> go2 "Bill"
Just ("John","Alice","Mary")
GHCi> go2' "Bill"
Just ("John","Alice","Mary")
```

Дополнительные техники: именуем лишь необходимое

Если значение не интересно, можно его проигнорировать, используя усеченный связыватель >>

```
go3 :: Name -> Maybe (Name,Name)
go3 p  = getF p >>= \x ->
         getM x >>= \y ->
         getM y >>
         return (x,y)
```

```
GHCi> go3 "Bill"
Just ("John","Alice")
```

Игнорируется только значение, но не эффект. Если бы у Билла не было прабабушки Мэри, мы получили бы Nothing.

Дополнительные техники: микс чистого и монадического

Можем использовать `let`-связывание для обычных выражений

```
go4 p = getF p >>= \x ->
        getM x >>= \y ->
        let y' = map toUpper y in
        getM y >>
        return (x,y')
```

```
GHCi> go4 "Bill"
Just ("John","ALICE")
```

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

```
go4 p = getF p >>= \x ->   -- выравнивание для красоты
        getM x >>= \y ->
        let y' = map toUpper y in
        getM y >>
        return (x,y')
go5 p = do x <- getF p      -- выравнивание обязательно
           y <- getM x
           let y' = map toUpper y
           getM y
           return (x,y')
```

```
GHCi> go4 "Bill"
Just ("John","ALICE")
GHCi> go5 "Bill"
Just ("John","ALICE")
```

«Мутабельные» переменные

Вложенные области видимости позволяют переиспользовать имя, создавая видимость изменяемой переменной

```
go5 p = do x <- getF p
           y <- getM x
           let y' = map toUpper y
           getM y
           return (x,y')
go6 p = do x <- getF p
           x <- getM x
           let y' = map toUpper x
           getM x
           return (x,y')
```

```
GHCi> go5 "Bill"
Just ("John","ALICE")
GHCi> go6 "Bill"
Just ("Alice","ALICE")
```

Монада Maybe: пример do-нотации

```
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

Монада Identity

Напишем представителя класса типов `Monad` для типа `Identity`, представляющего собой простую упаковку для другого типа

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

instance Monad Identity where
  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  Identity x >>= k   =   k x
  --m >>= k   =   k (runIdentity m)

  return :: a -> Identity a
  return   =   Identity
```

Эта монада не имеет эффектов.

Связывание в do-нотации и образцы

Связывание в do-нотации может происходить не только с переменной, но и с произвольным образцом

```
GHCi> do {n <- getF "Bill"; return (map toUpper n)}
Just "JOHN"
GHCi> do {'J':xs <- getF "Bill"; return (map toUpper xs)}
Just "OHN"
```

Что произойдет при неудачном сопоставлении?

```
GHCi> do {'Z':xs <- getF "Bill"; return (map toUpper xs)}
```

Связывание в do-нотации и образцы

Связывание в do-нотации может происходить не только с переменной, но и с произвольным образцом

```
GHCi> do {n <- getF "Bill"; return (map toUpper n)}
Just "JOHN"
GHCi> do {'J':xs <- getF "Bill"; return (map toUpper xs)}
Just "OHN"
```

Что произойдет при неудачном сопоставлении?

```
GHCi> do {'Z':xs <- getF "Bill"; return (map toUpper xs)}
```

Прямая трансляция в лямбды, дала бы аварийное завершение

```
GHCi> getF "Bill" >>= \('Z':xs) -> return (map toUpper xs)
*** Exception: Non-exhaustive patterns in lambda
```

Однако в действительности при трансляции do-нотации в Kernel/Core в подобном случае вызывается метод `fail`.

Класс типов MonadFail

Исторически `fail` был методом `Monad` с реализацией по умолчанию `fail = error`. Это приводило к потенциальной нетотальности обобщенного монадического кода. В GHC начиная с 8.6 `fail` перенесен в дочерний класс

```
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
```

Строковой параметр fail

В GHCi расширенный меанизм дефолтинга при необходимости трактует произвольную монаду как IO

```
GHCi> :t fail "qqq"
fail "qqq" :: MonadFail m => m a
GHCi> fail "qqq"
*** Exception: user error (qqq)
```

Когда при неудачном сопоставлении fail вызывается системой, в строковой параметр передается информация о типе и месте ошибки

```
GHCi> do {True <- return False; return 42}
*** Exception: user error (Pattern match failure in do
    expression at <interactive>:4:5-8)
```

Закон класса типов MonadFail

Закон, связывающий классы типов `Monad` и `MonadFail`

`fail s` — это левый ноль для `(>>=)`

```
fail s >>= k  ≡  fail s
```

Для `Maybe` он, конечно же, выполняется

```
GHCi> :t fail "Oh!" >>= granmas
fail "Oh!" >>= granmas :: Maybe (Name, Name)
GHCi> fail "Oh!" >>= granmas
Nothing
GHCi> fail "Oh!" :: Maybe (Name, Name)
Nothing
```

План лекции

Список как монада

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

```
instance Monad [] where
  (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= k  =  concat (map k xs)
  return :: a -> [a]
  return x  =  [x]

instance MonadFail [] where
  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 :)")))
```

Законы монад на языке join-fmap-return

```
join . return      ≡  id              -- (1)
join . fmap return ≡  id              -- (2)
join . fmap join   ≡  join . join     -- (3)
```

Каков тип выражения `join . return`?

Каков тип выражения `join . fmap return`?

Законы монад на языке join-fmap-return

```
join . return      ≡  id              -- (1)
join . fmap return ≡  id              -- (2)
join . fmap join   ≡  join . join     -- (3)
```

Каков тип выражения `join . return`?

Каков тип выражения `join . fmap return`?

```
GHCi> return [1,2,3] :: [[Int]]
[[1,2,3]]
GHCi> fmap return [1,2,3] :: [[Int]]
[[1],[2],[3]]
GHCi> join (return [1,2,3] :: [[Int]])
[1,2,3]
GHCi> join (fmap return [1,2,3] :: [[Int]])
[1,2,3]
```

Третий закон монад на языке join-fmap-return

```
join . fmap join    ≡   join . join   -- (3)
```

Каков тип выражения `join . join`?

Каков тип выражения `join . fmap join`?

Третий закон монад на языке join-fmap-return

```
join . fmap join    ≡   join . join   -- (3)
```

Каков тип выражения `join . join`?
Каков тип выражения `join . fmap join`?

```
GHCi> join [[[1],[2,3]],[[4],[5,6]]]
[[1],[2,3],[4],[5,6]]
GHCi> fmap join [[[1],[2,3]],[[4],[5,6]]]
[[1,2,3],[4,5,6]]
GHCi> join (join [[[1],[2,3]],[[4],[5,6]]])
[1,2,3,4,5,6]
GHCi> join (fmap join [[[1],[2,3]],[[4],[5,6]]])
[1,2,3,4,5,6]
```