Функциональное программирование
Лекция 11. Трансформеры монад

Денис Николаевич Москвин

СПбАУ РАН, CSC

29.04.2015

План лекции

1. Моноиды, `Alternative`, `MonadPlus`
2. Мультипараметрические классы типов
3. Монады с обработкой ошибок
4. Трансформеры монад

План лекции

1. Моноиды, `Alternative`, `MonadPlus`
2. Мультипараметрические классы типов
3. Монады с обработкой ошибок
4. Трансформеры монад

Вспомним класс типов `Monoid`

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Некоторые аппликативные функторы и монады являются, помимо всего прочего, моноидами (списки, `Maybe`). Например,

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m         =  m
  m       `mappend` Nothing   =  m
  Just m1 `mappend` Just m2   =  Just (m1 `mappend` m2)
```

Другой представитель `Monoid` для `Maybe`

Полезны и другие способы сделать `Maybe` моноидом, например

```
instance Monoid (Maybe a) where
  mempty =  Nothing
  Nothing    `mappend` m  =  m
  m@(Just _) `mappend` _  =  m
```

Поскольку для нельзя объявить двух представителей для одного типа, в стандартной библиотеке используется упаковка

```
newtype First a = First { getFirst :: Maybe a }
```

В отличие от предыдущей реализации, параметризующий `Maybe` тип a совершенно не важен.

Класс Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  -- One or more.
  some :: f a -> f [a]
  some v = some_v where
    many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v
  -- Zero or more.
  many :: f a -> f [a]
  many v = many_v  where
    many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v

infixl 3 <|>
```

Представитель `Alternative` для `Maybe`

```
instance Alternative Maybe where
  empty  =  Nothing

  Nothing    <|> m  =  m
  m@(Just _) <|> _  =  m
```

Представитель `Alternative` для `Maybe` ведёт себя, как упаковка `First`, возвращая первый не-`Nothing` в цепочке альтернатив:

Сессия GHCi

```
*Fp11> Nothing <|> (Just 3) <|> (Just 5) <|> Nothing
Just 3
```

Интерфейс `Alternative` для парсера `Parsec`

Сессия GHCi

```
> import Text.Parsec
> let parser = string "ABC" <|> string "DEF"
> parseTest parser "ABC123"
"ABC"
> parseTest parser "DEF123"
"DEF"
> parseTest parser "GHI123"
parse error at (line 1, column 1):
unexpected "G"
expecting "ABC" or "DEF"
```

Класс MonadPlus

```
class Monad m => MonadPlus m where
   mzero :: m a
   mplus :: m a -> m a -> m a
```

```
instance MonadPlus [] where
   mzero = []
   mplus = (++)

instance MonadPlus Maybe where
   mzero = Nothing
   Nothing `mplus` ys  = ys
   xs      `mplus` _   = xs
```

Эти представители имеют функциональность, аналогичную `Alternative`.

Законы `MonadPlus`

Помимо законов моноидальной структуры требуют выполнения

Left Zero – Левый ноль

```
mzero >>= k ≡ mzero
```

и по крайней мере одного из двух

Left Distribution – Левая дистрибутивность

```
(a `mplus` b) >>= k ≡ (a >>= k) `mplus` (b >>= k)
```

Left Catch law

```
(return a) `mplus` b ≡ return a
```

Выполнения аналогичных законов требуют для `Alternative`.

Использование MonadPlus

```
guard          :: MonadPlus m  =>  Bool -> m ()
guard True     =  return ()
guard False    =  mzero
```

```
pythags = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)
```

Сессия GHCi

```
*Fp11> take 5 pythags
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17)]
```

Использование MonadPlus (2)

```
msum  :: MonadPlus m => [m a] -> m a
msum  =  foldr mplus mzero
```

```
mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
mfilter p ma = do
  a <- ma
  if p a
  then return a
  else mzero
```

План лекции

Мотивирующий пример

Рассмотрим линейную алгебру в $\mathbb{Z}^2$

```
data Vector = Vector Int Int
data Matrix = Matrix Vector Vector
```

Хотим реализовать умножение так, чтобы можно было

```
(***) :: Matrix -> Matrix -> Matrix
(***) :: Matrix -> Vector -> Vector
(***) :: Matrix -> Int -> Matrix
(***) :: Int -> Matrix -> Matrix
...
```

Обычная сигнатура умножения из `Num` слишком бедна для этого.

Мультипараметрические классы типов

```
class Mult a b c where
  (***) :: a -> b -> c

instance Mult Matrix Matrix Matrix where
  {- ... -}
instance Mult Matrix Vector Vector where
  {- ... -}
instance Mult Matrix Int Matrix where
  {- ... -}
instance Mult Int Matrix Matrix where
  {- ... -}
...
```

Мультипараметрические классы типов являются расширением стандарта и требуют прагмы `{-# LANGUAGE MultiParamTypeClasses #-}`.

Проблемы с использованием

К сожалению, такое решение слишком полиморфно:

Сессия GHCi

```
*Fp11> let a = Matrix (Vector 1 2) (Vector 3 4)
*Fp11> let i = Matrix (Vector 1 0) (Vector 0 1)
*Fp11> a *** i
    No instance for (Mult Matrix Matrix c) arising from
     a use of ‘***’
    The type variable ‘c’ is ambiguous
*Fp11> (a *** i) :: Matrix
Matrix (Vector 1 2) (Vector 3 4)
```

Типовая переменная `c` в действительности не является свободной, но системе вывода типов нужна соответствующая подсказка.

Функциональные зависимости

Можно задать «функциональную зависимость», указав, что тип `c` уникальным образом определяется типами `a` и `b`

```
class Mult a b c | a b -> c where
  (***) :: a -> b -> c
```

Теперь все работает

Сессия GHCi

```
*Fp11> let a = Matrix (Vector 1 2) (Vector 3 4)
*Fp11> let i = Matrix (Vector 1 0) (Vector 0 1)
*Fp11> a *** i
Matrix (Vector 1 2) (Vector 3 4)
```

Нужна прагма `{-# LANGUAGE FunctionalDependencies #-}`.

План лекции

Класс `Error` (Deprecated: Use Except instead)

Простое решение для обработки ошибок — использовать монаду `Either String`. Однако удобнее обобщить. Пользовательский класс ошибок (обобщаем `String`):

```
class Error e where
    noMsg :: e
    strMsg :: String -> e
```

Например,

```
data DivByError = ErrZero | Other String
                  deriving (Eq, Read, Show)

instance Error DivByError where
  strMsg s = Other s
  noMsg    = Other "Unknown DivByError"
```

Класс MonadError

```
class (Monad m) => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

Самый главный представитель

```
instance MonadError e (Either e) where
  throwError = Left
  Left  l `catchError` handler = handler l
  a       `catchError` _       = a
```

Использование

```
do { action1; action2; action3 } `catchError` handler
```

Пример использования

```
(/?) :: Double -> Double -> Either DivByError Double
x /? 0 = throwError ErrZero
x /? y = return $ x / y

example0 :: Double -> Double -> Either DivByError String
example0 x y = action  `catchError` handler where
  action = do
    q <- x /? y
    return $ show q
  handler = \err -> return $ show err
```

Сессия GHCi

```
*Fp11> example0 5 2
Right "2.5"
*Fp11> example0 5 0
Right "ErrZero"
```

Более полиморфный пример

Деление, совместимое с произвольным механизмом обработки ошибок. (Нужна прагма `{-# LANGUAGE FlexibleContexts #-}`.)

```
(?/?) :: (MonadError DivByError m)
             => Double -> Double -> m Double
x ?/? 0 = throwError ErrZero
x ?/? y = return $ x / y
```

Теперь могли бы использовать не только `Either`, но не будем.

```
example1 :: Double -> Double -> Either DivByError String
example1 x y = action  `catchError` handler where
  action = do
    q <- x ?/? y
    return $ show q
  handler = \err -> return $ show err
```

Для чего полезен контекст `Error`?

```
instance (Error e) => MonadPlus (Either e) where
    mzero             = Left noMsg
    Left _ `mplus` n = n
    m      `mplus` _ = m

example2 :: Double -> Double -> Either DivByError String
example2 x y = action `catchError` handler where
  action = do
    q <- x ?/? y
    guard $ y>=0
    return $ show q
  handler = \err -> return $ show err
```

Сессия GHCi

```
*Fp11> example1 7 (-5)
Right "Other \"Unknown DivByError\""
```

План лекции

Трансформеры монад: знакомство

```
stInteger :: State Integer Integer
stInteger = do modify (+1)
               a <- get
               return a
stString :: State String String
stString = do modify (++"1")
              b <- get
              return b
```

```
*Fp11> evalState stInteger 0
1
*Fp11> evalState stString "0"
"01"
```

Что делать если хотим в одном монадическом вычислении работать с обоими состояниями?

Monad transformers are like onions

```
stComb :: StateT Integer
                 (StateT String Identity)
                 (Integer, String)
stComb = do modify (+1)
            lift $ modify (++"1")
            a <- get
            b <- lift $ get
            return (a,b)
```

```
*Fp11> runIdentity $ evalStateT (evalStateT stComb 0) "0"
(1,"01")
```

В качестве основы помимо `Identity` используют также `IO` со специализированной `liftIO`.

Трансформеры монад

Трансформер монад — конструктор типа, который принимает монаду в качестве параметра и возвращает монаду как результат.

Требования:

1. Поскольку у монады кайнд `m : * -> *`, у трансформера должен быть кайнд `t: (* -> *) -> * -> *`
2. Для любой монады `m`, аппликация `t m` должна быть монадой, то есть её `return` и `(>>=)` должны удовлетворять законам монад.
3. Нужен `lift :: m a -> t m a`, «поднимающий» значение из трансформируемой монады в трансформированную.

В библиотеке `transformers` функция `lift` всегда вызывается вручную, в `mtl` — только для неоднозначных ситуаций.

Рецепт приготовления трансформера для `MyMonad` (1)

1. У трансформера должен быть кайнд

`t: (* -> *) -> * -> *`

Определяем наш конкретный трансформер `MyMonadT` для монады `MyMonad`

```
newtype MyMonadT m a
   =  MyMonadT { runMyMonadT :: m (MyMonad a) }
```

Такое определение согласовано с механизмом вызова

```
comp :: (MyMonadT Identity) a

runIdentity (runMyMonadT comp) :: a
```

(«Сцепляющая» вычисления конструкция может быть более сложной чем `m (MyMonad a)` и зависит от конкретной семантики эффектов `MyMonad`.)

Рецепт приготовления трансформера для `MyMonad` (2)

2. Для любой монады `m`, аппликация `t m` должна быть монадой

Делаем аппликацию нашего трансформера к монаде (`MyMonadT m`) представителем `Monad`

```
instance (Monad m) => Monad (MyMonadT m) where
  return x    = ...
  mx (>>=) k  = ...
```

Рецепт приготовления трансформера для `MyMonad` (3)

3. Операция `lift :: m a -> t m a` из `class MonadTrans`

Поднимаем значение из трансформируемой монады в трансформированную

```
class MonadTrans t where
  lift :: (Monad m) => m a -> t m a
```

```
instance MonadTrans MyMonadT where
  lift mx = ...
```

Законы для класса типов `MonadTrans`

Функция `lift` для любого представителя `MonadTrans` должна удовлетворять следующим законам

Right Zero – Правый ноль

```
lift . return  ≡  return
```

Left Distribution – Левая дистрибутивность

```
lift (m >>= k) ≡ lift m >>= (lift . k)
```

Трансформер для `Maybe` — шаги (1) и (3)

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

MaybeT :: m (Maybe a) -> MaybeT m a
runMaybeT :: MaybeT m a -> m (Maybe a)

instance MonadTrans MaybeT where
  lift :: m a -> MaybeT m a
  lift = MaybeT . liftM Just
```

Пояснение работы `lift` при поднятии `get` из `State`:

```
*Fp11> :t get
get :: MonadState s m => m s
*Fp11> :t lift get
lift get :: (MonadState a m, MonadTrans t) => t m a
```

(Здесь `m` можно читать как `State s`.)

Трансформер для `Maybe` — шаг (2)

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Monad m) => Monad (MaybeT m) where
  fail :: String -> MaybeT m a
  fail _  = MaybeT $ return Nothing

  return :: a -> MaybeT m a
  return  = lift . return

  (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
  x >>= f = MaybeT $ do    -- inner monad do
    v <- runMaybeT x
    case v of
      Nothing -> return Nothing
      Just y  -> runMaybeT (f y)
```

Пример использования MaybeT

```
mbSt :: MaybeT (StateT Integer Identity) Integer
mbSt = do
  lift $ modify (+1)
  a <- lift get
  True <- return $ a >= 3
  return a
```

```
*Fp11> runIdentity $ evalStateT (runMaybeT mbSt) 0
Nothing
*Fp11> runIdentity $ evalStateT (runMaybeT mbSt) 2
Just 3
```

Если хотим `guard $ a >= 3` нужно сделать `MaybeT m` представителем `MonadPlus`

Пример использования MaybeT (2)

```
instance (Monad m) =>  MonadPlus (MaybeT m) where
  mzero       = MaybeT $ return Nothing
  x `mplus` y = MaybeT $ do
    v <- runMaybeT x
    case v of
      Nothing -> runMaybeT y
      Just _  -> return v

mbSt' :: MaybeT (State Integer) Integer
mbSt' = do lift $ modify (+1)
           a <- lift get
           guard $ a >= 3            -- !!
           return a
```

```
*Fp11> runIdentity $ evalStateT (runMaybeT mbSt') 2
Just 3
```

Пример использования MaybeT (3)

Для любой пары монад можно избавиться от подъёма стандартных операций вложенной монады.
Например, для монады State стандартный интерфейс упакован (в библиотеке mtl) в класс типов с фундепсами

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

Мы можем реализовать для MaybeT

```
instance (MonadState s m) => MonadState s (MaybeT m) where
  get = lift get
  put = lift . put
```

Пример использования MaybeT (3)

Теперь можно не поднимать явно стандартные операции State

```
mbSt'' :: MaybeT (State Integer) Integer
mbSt'' = do
  modify (+1)                -- без lift
  a <- get                   -- без lift
  guard $ a >= 3
  return a
```

```
*Fp11> runIdentity $ evalStateT (runMaybeT mbSt'') 2
Just 3
```

Таблица стандартных трансформеров

Монада	Трансформер	Исходный тип	Тип трансформера
`Error`	`ErrorT`	`Either e a`	`m (Either e a)`
`State`	`StateT`	`s -> (a,s)`	`s -> m (a,s)`
`Reader`	`ReaderT`	`r -> a`	`r -> m a`
`Writer`	`WriterT`	`(a,w)`	`m (a,w)`
`Cont`	`ContT`	`(a -> r) -> r`	`(a -> m r) -> m r`

Они определены в библиотеке `mtl`. Более того, первый столбец определён через второй:

```
type Writer w = WriterT w Identity
type Reader r = ReaderT r Identity
type State  s = StateT  s Identity

...
```

Что во что вкладывать?

- Если нам нужна функциональность `Error` и `State`, то есть наша монада должна быть представителем `MonadError` и `MonadState`.
- Должны ли мы применять трансформер `StateT` к монаде `Error` или трансформер `ErrorT` к монаде `State`?
- Решение зависит от того, какой в точности семантики мы ожидаем от комбинированной монады.

Что во что вкладывать?

- Применение `StateT` к монаде `Error` даёт функцию трансформирования типа `s -> Either e (a, s)`.
- Применение `ErrorT` к монаде `State` даёт функцию трансформирования типа `s -> (Either e a, s)`.
- Порядок зависит от той роли, которую ошибка играет в вычислениях.
- Если ошибка обозначает, что состояние не может быть вычислено, то нам следует применять `StateT` к `Error`.
- Если ошибка обозначает, что значениене не может быть вычислено, но состояние при этом не «портится», то нам следует применять `ErrorT` к `State`.