Функциональное программирование
Лекция 12. Трансформеры монад

Денис Николаевич Москвин

СПбАУ РАН, CSC

27.04.2017

План лекции

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

План лекции

Класс Alternative

Alternative — это Applicative с дополнительной моноидальной операцией.

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

```
instance Alternative Maybe where
  empty  =  Nothing
  Nothing    <|> m  =  m
  m@(Just _) <|> _  =  m
```

Сессия GHCi

```
GHCi> Nothing <|> (Just 3) <|> (Just 5) <|> Nothing
Just 3
```

Класс MonadPlus

```
class (Alternative m, Monad m) => MonadPlus m where
   mzero :: m a
   mzero = empty
   mplus :: m a -> m a -> m a
   mplus = (<|>)
```

Минимальное полное определение: ничего не делать.

```
instance MonadPlus Maybe

instance Alternative [] where
    empty = []
    (<|>) = (++)

instance MonadPlus []
```

Законы MonadPlus

Помимо «унаследованных» законов требуют выполнения

Left and Right Zero

```
mzero >>= k  ≡  mzero
v >> mzero   ≡  mzero
```

и по крайней мере одного из двух

Left Distribution

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

Left Catch law

```
return a `mplus` b  ≡  return a
```

Использование MonadPlus

```
-- Haskell 2010 :: MonadPlus m   => Bool -> m ()
guard           :: Alternative f => Bool -> f ()
guard True      =  pure ()
guard False     =  empty
```

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

Сессия GHCi

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

Использование MonadPlus (2)

```
msum :: (Foldable t, MonadPlus m) => t (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

Монада Except

Вычисление, допускающее возбуждение и перехват исключений.

```
newtype Except e a = Except {runExcept :: Either e a}

except :: Either e a -> Except e a
except = Except
```

```
instance Monad (Except e) where
  return a = Except $ Right a
  m >>= k = case runExcept m of
              Left e  -> Except $ Left e
              Right x -> k x
```

Стандартный интерфейс

```
throwE :: e -> Except e a
throwE = except . Left

catchE :: Except e a -> (e -> Except e' a) -> Except e' a
m `catchE` h = case runExcept m of
                 Left  l -> h l
                 Right r -> Except $ Right r
```

Использование

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

В библиотеке mtl по историческим причинам `throwError` и `catchError`.

Пример использования

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

(/?) :: Double -> Double -> Except DivByError Double
x /? 0 = throwE ErrZero
x /? y = return $ x / y

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

```
GHCi> runExcept $ example0 5 2
Right "2.5"
GHCi> runExcept $ example0 5 0
Right "ErrZero"
```

Расширение функциональности

Если хотим использовать функциональность `MonadPlus`, то есть `guard`, `msum`, `mfilter`, нужно сделать `Except e` представителем:

```
instance Monoid e => MonadPlus (Except e) where
    mzero = Except $ Left mempty
    Except x `mplus` Except y = Except $
      case x of
           Left e  -> either (Left . mappend e) Right y
           r       -> r
```

Семантика:

- `mzero` — ошибка по умолчанию для `guard`, задается `mempty`;
- `mplus` — накапливает ошибки слева направо, но если происходит удачная попытка, то возвращает удачу.

Расширение функциональности, пример

```
instance Monoid DivByError where
  mempty = Other ""
  Other s1 `mappend` Other s2 = Other $ s1 ++ s2
  Other s1 `mappend` ErrZero  = Other $ s1 ++ "zero;"
  ErrZero  `mappend` Other s2 = Other $ "zero;" ++ s2
  ErrZero  `mappend` ErrZero  = Other $ "zero;zero"
```

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

Расширение функциональности, пример (2)

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

```
GHCi> runExcept $ example2 5 0
Right "ErrZero"
GHCi> runExcept $ example2 5 (-2)
Right "Other \"\""
GHCi> runExcept $ msum [5/?0, 7/?0, 2/?0]
Left (Other "zero;zero;zero;")
GHCi> runExcept $ msum [5/?0, 7/?0, 2/?4]
Right 0.5
```

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

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

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

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

Использование в mtl

Стандартные интерфейсы стандартных монад упакованы в mtl в мультипараметрические классы типов, что позволяет наделять любую монаду соответствующим интерфейсом.

```
class Monad m => MonadReader r m | m -> r where
    ask   :: m r
    local :: (r -> r) -> m a -> m a
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
    tell   :: w -> m ()
    listen :: m a -> m (a, w)
class Monad m => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

Простейшие представители

«Неупакованные» контейнеры `Either a b` и `(->) a b` являются представителями `MonadError a (Either a)` и `MonadReader a ((->) a)`, поэтому допустимо писать:

```
GHCi> import Control.Monad.Except
GHCi> Left 5 `catchError` (\e -> Right (e^2))
Right 25
GHCi> Right 5 `catchError` (\e -> Right (e^2))
Right 5
GHCi> import Control.Monad.Reader
GHCi> do {x<-(*2); y<-ask; return (x+y)} $ 5
15
```

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
```

```
GHCi> evalState stInteger 0
1
GHCi> 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)
```

```
GHCi> 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`:

```
GHCi> :t get
get :: MonadState s m => m s
GHCi> :t lift get
lift get :: (MonadState s m, MonadTrans t) => t m s
```

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

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
  mx >>= k = MaybeT $ do    -- inner monad do
    v <- runMaybeT mx
    case v of
      Nothing -> return Nothing
      Just y  -> runMaybeT (k y)
```

Пример использования MaybeT

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

```
GHCi> runIdentity $ evalStateT (runMaybeT mbSt) 0
Nothing
GHCi> 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
--instance (Monad m) => Alternative (MaybeT m) where...--GHC>=7.

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

```
GHCi> 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
```

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

Таблица стандартных трансформеров

Монада	Трансформер	Исходный тип	Тип трансформера
`Except`	`ExceptT`	`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 `Except` и `State`, то есть наша монада должна быть представителем `MonadError` и `MonadState`.
- Должны ли мы применять трансформер `StateT` к монаде `Except` 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 `Except` даёт функцию трансформирования типа `s -> Either e (a, s)`.
- Применение `ExceptT` 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 `Except`.
- Если ошибка обозначает, что значениене не может быть вычислено, но состояние при этом не «портится», то нам следует применять `ExceptT` к `State`.