Функциональное программирование
Лекция 11. Ещё о монадах

Денис Николаевич Москвин

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

25.05.2012

План лекции

1. Снова моноиды
2. Монады с обработкой ошибок
3. Трансформеры монад

План лекции

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
```

Парсеры регулярных выражений: **regex-applicative**

`RE s a` — тип регулярного выражения, распознающего символы типа `s` и возвращающего результат типа `a`

```
psym :: (s -> Bool) -> RE s s
match :: RE s a -> [s] -> Maybe a
```

Сессия GHCi

```
*Fp11> match (sym 'a' <|> sym 'b') "a"
Just 'a'
*Fp11> match (many $ psym isAlpha) "abc"
Just "abc"
*Fp11> match (many $ psym isAlpha) "123"
Nothing
*Fp11> match (many $ psym isAlpha) ""
Just ""
*Fp11> match (some $ psym isAlpha) ""
Nothing
```

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`.

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

Класс `Error`

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 = DivBy0 | OtherDivByError String
                  deriving (Eq, Read, Show)

instance Error DivByError where
    strMsg s = OtherDivByError s
```

Класс MonadError

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

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

Использование

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

Пример

```
myDiv :: (MonadError DivByError m)
             => Double -> Double -> m Double
myDiv x 0 = throwError DivBy0
myDiv x y = return $ x / y

example :: Double -> Double -> Either DivByError String
example x y = action `catchError` handler
  where action = do { q <- myDiv x y; return $ show q }
        handler = \err -> return $ show err
```

Сессия GHCi

```
*Fp11> example 5 2
Right "2.5"
*Fp11> example 5 0
Right "DivBy0"
```

План лекции

1. Снова моноиды
2. Монады с обработкой ошибок
3. Трансформеры монад

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.

Рецепт приготовления трансформера для `MyMonad`

1. У трансформера должен быть кайнд

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

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

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

Рецепт приготовления трансформера для `MyMonad`

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

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

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

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

Трансформер для Maybe

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

instance (Monad m) => Monad (MaybeT m) where
    fail _ = MaybeT (return Nothing)
    return = lift . return
    x >>= f = MaybeT $ do
        v <- runMaybeT x
        case v of
            Nothing -> return Nothing
            Just y  -> runMaybeT (f y)

instance MonadTrans MaybeT where
    lift = MaybeT . liftM Just
```

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`.