Функциональное программирование
Лекция 12. Трансформеры монад

Денис Николаевич Москвин

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

27.11.2025

План лекции

1. Мультипараметрические классы типов
2. Трансформеры монад
3. Трансформер `MaybeT`

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 (GHC2021)
{-# LANGUAGE MultiParamTypeClasses #-}.

Проблемы с использованием

К сожалению, такое решение слишком полиморфно:

```
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

Имеется возможность 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

```
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 #-}*.
(Не входит в GHC2021.)

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
```

Простейший представитель MonadReader

Неупакованная в `Reader` частично примененная стрелка `(->)` объявлена представителем `MonadReader`

```
class Monad m => MonadReader r m | m -> r where
  ask   :: m r
  local :: (r -> r) -> m a -> m a

instance MonadReader r ((->) r) where
  ask       = id
  local f m = m . f
```

Поэтому допустимо писать

```
GHCi> import Control.Monad.Reader
GHCi> do {x <- (*2); y <- ask; return (x+y)} $ 5
15
```

Простейший представитель MonadError

Тип Either объявлен представителем MonadError

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

instance MonadError e (Either e) where
  throwError              = Left
  Left  l `catchError` h = h l
  Right r `catchError` _ = Right r
```

```
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
```

План лекции

Пример с прошлой лекции

```
type PriceList  =  [(Vegetable,Price)]
prices  :: PriceList
prices  =  [("Potato",13),("Tomato",55),("Apple",48)]

addVegetable :: Vegetable -> Qty
                -> Writer (Sum Cost) (Vegetable, Price)
addVegetable veg qty = do
  let pr = fromMaybe 0 $ lookup veg prices
  let cost = qty * pr
  tell $ Sum cost
  return (veg,pr)
```

```
GHCi> runWriter $ addVegetable "Apple" 100
(("Apple",48.0),Sum {getSum = 4800.0})
```

Функция addVegetable ужасна: конкретная таблица prices зашита в нее намертво.

Трансформеры монад: синтаксис

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

Поскольку у монады кайнд `m : * -> *`, и требуется, чтобы `t m : * -> *` у самого трансформера должен быть кайнд `t : (* -> *) -> * -> *`.
Мы можем делать цепочки из трансформеров произвольной длины

```
t1 (t2 (t3 (...)))
```

А как терминировать такую цепочку?

Трансформеры монад: синтаксис

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

Поскольку у монады кайнд `m : * -> *`, и требуется, чтобы `t m : * -> *` у самого трансформера должен быть кайнд `t : (* -> *) -> * -> *`.
Мы можем делать цепочки из трансформеров произвольной длины

```
t1 (t2 (t3 (...)))
```

А как терминировать такую цепочку?

```
t1 (t2 (t3 Identity)) :: * -> *
t1 (t2 (t3 Identity)) a :: *
```

Трансформеры монад: знакомство

```
addVegetable :: Vegetable -> Qty
                -> WriterT (Sum Cost)
                           (ReaderT PriceList Identity)
                           (Vegetable, Price)
addVegetable veg qty = do
  priceList <- lift ask
  let pr = fromMaybe 0 $ lookup veg priceList
  let cost = qty * pr
  tell $ Sum cost
  return (veg, pr)
```

```
GHCi> runIdentity $ runReaderT (runWriterT $ addVegetab
le "Apple" 100) prices
(("Apple",48.0),Sum {getSum = 4800.0})
```

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

Трансформеры монад: семантика

Требования:

1. Тип данных трансформера должен иметь кайнд `(* -> *) -> * -> *`.
2. Нужен `lift :: m a -> t m a`, «поднимающий» значение из трансформируемой монады в трансформированную.
3. Для любой монады `m`, аппликация `t m` должна быть монадой, то есть её `return` и `>>=` должны удовлетворять законам монад.
4. Если `m` выставляет интерфейс `MonadFail`, `t m` тоже должна его выставлять.

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

Рецепт приготовления трансформера для MyMonad (1)

1. Тип данных трансформера должен иметь кайнд `(* -> *) -> * -> *`.

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

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

Такое определение согласовано с механизмом вызова

```
computation :: MyMonadT Identity a

runIdentity $ runMyMonadT computation :: MyMonad a
```

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

Рецепт приготовления трансформера для MyMonad (2)

2. Функция `lift :: m a -> t m a` определена как метод класса типов `MonadTrans`.

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

Поднимаем значение из трансформируемой монады в трансформированную, реализуя представителя

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

Рецепт приготовления трансформера для MyMonad (3)

3. 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 (4)

4. Функцию `fail` нужно реализовать обязательно.
   - Если монада заточена под обработку ошибок — реализовать содержательный обработчик:

```
instance Monad m => MonadFail (MyMonadT m) where
  fail s = ...
```

   - Если нет — переадресовать обработку ошибок внутренней монаде, в том случае, когда последняя это умеет:

```
instance MonadFail m => MonadFail (MyMonadT m) where
  fail = lift . fail
```

NB. Если GHC < 8.6 `fail` нужно реализовывать в представителе класса типов `Monad`.

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)
```

Первый закон, прочитанный справа налево, — эталонная реализация для `return` в трансформере.

Таблица стандартных трансформеров

В библиотеках `mtl`/`transformers` определены трансформеры

Монада	Исходный тип	Трансформ	Тип трансформера
`Except`	`Either e a`	`ExceptT`	`m (Either e a)`
`Writer`	`(a,w)`	`WriterT`	`m (a,w)`
`Reader`	`r -> a`	`ReaderT`	`r -> m a`
`State`	`s -> (a,s)`	`StateT`	`s -> m (a,s)`
`Cont`	`(a -> r) -> r`	`ContT`	`(a -> m r) -> m r`
`Select`	`(a -> r) -> a`	`SelectT`	`(a -> m r) -> m a`

Более того, первый столбец определён через третий

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

Что во что вкладывать?

- Пусть нам нужна функциональность `Except` и `State`.
- Должны ли мы применять трансформер `StateT` к монаде `Except` или трансформер `ExceptT` к монаде `State`?
- Решение зависит от того, какой в точности семантики мы ожидаем от комбинированной монады.

Некоммутативность трансформеров

- 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 и *состояние* и *значение* не могут быть вычислены, то нам следует применять `StateT` к `Except`.
- Если ошибка обозначает, что только *значение* не может быть вычислено, но *состояние* при этом не «портится», то нам следует применять `ExceptT` к `State`.

План лекции

1. Мультипараметрические классы типов
2. Трансформеры монад
3. Трансформер `MaybeT`

Трансформер для Maybe — шаги (1) и (2)

```haskell
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 . fmap Just
```

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

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

Контекст `Monad m` опущен для компактности.

В `transformers` нужно `m` заменить на `State s`.

Трансформер для Maybe — шаг (3)

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

instance Monad m => Monad (MaybeT m) where
  return :: a -> MaybeT m a
  return  = MaybeT . fmap Just . 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)
```

Трансформер для Maybe — шаг (4)

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

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

Если GHC $<$ 8.6, то метод `fail` нужно реализовывать в представителе класса типов `Monad`.

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 `Alternative` (раньше `MonadPlus`).

Трансформер MaybeT как Alternative

```haskell
instance Monad m => Alternative (MaybeT m) where
  empty   = MaybeT $ return Nothing
  x <|> 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
```

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

Стандартные интерфейсы других монад для MaybeT

В mtl для любого трансформера можно избавиться от подъёма стандартных операций вложенной монады.
Например, для монады State стандартный интерфейс упакован 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 его «протаскивание» через MaybeT

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

Требуются FlexibleInstances, UndecidableInstances.

«Делифтинг» для MaybeT

Теперь стандартные операции State можно не поднимать явно

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

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

Стандартный интерфейс для MaybeT

Наделив Maybe стандартным интерфейсом в mtl-стиле, нужно реализовать его для Maybe и MaybeT

```
class Monad m => MonadMaybe m where
  throwNothing :: m a
instance MonadMaybe Maybe where
  throwNothing = Nothing
instance Monad m => MonadMaybe (MaybeT m) where
  throwNothing = MaybeT $ return Nothing
```

```
stMb :: StateT Integer Maybe Integer
stMb = do
  a <- get
  if a < 3 then lift throwNothing else return a
```

```
GHCi> evalStateT stMb 0
Nothing
```

«Делифтинг» для остальных монад mtl

Чтобы убрать `lift` у метода `MonadMaybe`, нужно научить все стандартные монады протаскивать его через себя

```
instance MonadMaybe m => MonadMaybe (ReaderT r m) where
  throwNothing = lift throwNothing
instance (MonadMaybe m, Monoid w) =>
                          MonadMaybe (WriterT w m) where
  throwNothing = lift throwNothing
instance MonadMaybe m =>  MonadMaybe (StateT s m) where
  throwNothing = lift throwNothing
```

```
stMb' :: StateT Integer Maybe Integer
stMb' = do
  a <- get
  if a < 3 then throwNothing  -- можно без lift
  else return a
```