Курс: Функциональное программирование
Практика 12. Трансформеры монад

Трансформеры монад

Сделаем на основе однопараметрического типа данных `String -> a` (фактически это `Reader` с окружением строкового типа) трансформер монад `StrRdrT :: (* -> *) -> * -> *` с одноименным конструктором данных и меткой поля `runStrRdrT`

```
newtype StrRdrT m a = StrRdrT { runStrRdrT :: String -> m a }
```

```
GHCi> :t StrRdrT
StrRdrT :: (String -> m a) -> StrRdrT m a
GHCi> :t runStrRdrT
runStrRdrT :: StrRdrT m a -> String -> m a
```

▶ (Stepik, 1 балл) Реализуйте для произвольной монады `m` представителя класса типов `Monad` для `StrRdrT m :: * -> *`.

```
instance Monad m => Monad (StrRdrT m) where
  return :: a -> StrRdrT m a
  return = undefined

  (>>=) :: StrRdrT m a -> (a -> StrRdrT m b) -> StrRdrT m b
  (>>=) = undefined
```

Поскольку `StrRdr` не подразумевает специфического умения обрабатывать ошибки, семантика `MonadFail` должна протаскиваться из внутренней монады. Реализуйте соответствующего представителя

```
instance MonadFail m => MonadFail (StrRdrT m)  where
  fail :: String -> StrRdrT m a
  fail  = undefined
```

Для проверки используйте функции

```
srtTst :: StrRdrT Identity Int
srtTst = do
  x <- StrRdrT $ Identity <$> length
  y <- return 10
  return $ x + y

failTst :: StrRdrT [] Integer
failTst = do
  'z' <- StrRdrT id
  return 42
```

которые при правильной реализации монады должны вести себя так

```
GHCi> runStrRdrT srtTst "ABCDE"
Identity 15
GHCi> runIdentity (runStrRdrT srtTst "ABCDE")
15
GHCi> runStrRdrT failTst "zanzibar"
[42,42]
GHCi> runStrRdrT failTst "zzz..."
[42,42,42]
GHCi> runStrRdrT failTst "ABCD"
[]
```

► (Stepik, 1 балл) Напишите функции `askStrRdr` и `asksStrRdr` обеспечивающую трансформер `StrRdrT` стандартным интерфейсом обращения к окружению:

```
askStrRdr :: Monad m => StrRdrT m String
askStrRdr = undefined

asksStrRdr :: Monad m => (String -> a) -> StrRdrT m a
asksStrRdr = undefined
```

Введите для удобства упаковку для `StrRdrT Identity` и напишите функцию запускающую вычисления в этой монаде

```
type StrRdr = StrRdrT Identity

runStrRdr :: StrRdr a -> String -> a
runStrRdr sr = undefined
```

Тест

```
srtTst' :: StrRdr (String,Int)
srtTst'  = do
  env <- askStrRdr
  len <- asksStrRdr length
  return (env,len)
```

должен дать такой результат

```
GHCi> runStrRdr srtTst' "ABCD"
("ABCD",4)
```

А тест

```
stSrTst :: StateT Int StrRdr Int
stSrTst = do
  a <- get
  n <- lift $ asksStrRdr length
  modify (+n)
  return a
```

такой

```
GHCi> runStrRdr (runStateT stSrTst 33) "ABCD"
(33,37)
```

▶ (Stepik, 1 балл) В последнем примере функция

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

позволяла поднять вычисление из внутренней монады (в примере это был `StrRdr`) во внешний трансформер (`StateT Int`). Это возможно, поскольку для трансформера `StateT s` реализован представитель класса

типов `MonadTrans`. Сделайте трансформер `StrRdrT` представителем класса `MonadTrans`, так чтобы можно было поднимать вычисления из произвольной внутренней монады в наш трансформер:

```
instance MonadTrans StrRdrT where
  lift :: Monad m => m a -> StrRdrT m a
  lift = undefined
```

Сценарий использования:

```
srStTst :: StrRdrT (State Int) (Int,Int)
srStTst = do
  lift $ state $ \s -> ((),s+40)
  m <- lift get
  n <- asksStrRdr length
  lift $ put $ m + n
  return (m,n)
```

Проверка:

```
GHCi> runState (runStrRdrT srStTst "ABCD") 2
((42,4),46)
GHCi> runState (runStrRdrT srStTst "ABCDE") 2
((42,5),47)
```

▶ (Stepik, 1 балл) Избавьтесь от необходимости ручного подъема операций вложенной монады `State`, сделав трансформер `StrRdrT`, примененный к монаде с интерфейсом `MonadState`, представителем класса типов `MonadState`:

```
instance MonadState s m => MonadState s (StrRdrT m) where
  get :: MonadState s m => StrRdrT m s
  get = undefined
  put :: MonadState s m => s -> StrRdrT m ()
  put = undefined
  state :: MonadState s m => (s -> (a, s)) -> StrRdrT m a
  state = undefined

-- instance MonadWriter w (StrRdrT m), etc...
```

Сценарий использования:

```
srStTst' :: StrRdrT (State Int) (Int,Int)
srStTst' = do
  state $ \s -> ((),s + 40)    -- no lift!
  m <- get                     -- no lift!
  n <- asksStrRdr length
  put $ m + n                  -- no lift!
  return (m,n)
```

Проверка:

```
GHCi> runState (runStrRdrT srStTst' "ABCDE") 2
((42,5),47)
```

▶ (Stepik, 1 балл) Чтобы избавится от необходимости ручного подъема операций `askStrRdr` и `asksStrRdr`, обеспечивающих стандартный интерфейс вложенного трансформера `StrRdrT`, можно поступить по аналогии с другими трансформерами библиотеки `mtl`. А именно, разработать класс типов `MonadStrRdr`, выставляющий этот стандартный интерфейс[1] для нашего ридера:

```
class Monad m => MonadStrRdr m where
  askSR :: m String
  asksSR :: (String -> a) -> m a
  strRdr :: (String -> a) -> m a
```

Этот интерфейс, во-первых, должен выставлять сам трансформер `StrRdrT`, обернутый вокруг произвольной монады:

```
instance Monad m => MonadStrRdr (StrRdrT m) where
  askSR :: StrRdrT m String
```

[1] Мы переименовываем функцию `askStrRdr` в `askSR` (и анологично `asks`-версию), поскольку хотим держать всю реализацию в одном файле исходного кода. При следовании принятой в библиотеках `transformers`/`mtl` идеологии они имели бы одно и то же имя, но были бы определены в разных модулях. При работе с `transformers` мы импортировали бы свободную функцию с квалифицированным именем `Control.Monad.Trans.StrRdr.askStrRdr`, а при использовании `mtl` работали бы с методом `Control.Monad.StrRdr.askStrRdr` класса типов `MonadStrRdr`.

```
  askSR = undefined
  asksSR :: (String -> a) -> StrRdrT m a
  asksSR = undefined
  strRdr :: (String -> a) -> StrRdrT m a
  strRdr = undefined
```

Реализуйте этого представителя, для проверки используйте

```
srStTst'' :: StrRdrT (State Int) (Int,Int,Int,String)
srStTst'' = do
  m <- get
  n <- asksStrRdr length -- use asksStrRdr
  k <- strRdr length     -- use strRdr
  put $ m + n + k
  e <- askStrRdr         -- use askStrRdr
  return (m,n,k,e)
```

Результат должен быть таким:

```
GHCi> runState (runStrRdrT srStTst'' "ABCDE") 2
((2,5,5,"ABCDE"),12)
```

Во-вторых, интерфейс `MonadStrRdr` должен выставлять любой стандартный трансформер, обернутый вокруг произвольной монады, выставляющей этот интерфейс:

```
instance MonadStrRdr m => MonadStrRdr (StateT s m) where
  askSR :: StateT s m String
  askSR = undefined
  asksSR :: (String -> a) -> StateT s m a
  asksSR = undefined
  strRdr :: (String -> a) -> StateT s m a
  strRdr = undefined

-- WriterT w, etc...
```

Реализуйте этого представителя, для проверки используйте

```
stSrTst' :: StateT Int StrRdr (Int,Int,Int,String)
stSrTst' = do
  a <- get
  n <- asksSR length    -- no lift!
  k <- strRdr length    -- no lift!
  e <- askSR            -- no lift!
  modify (+n)
  return (a,n,k,e)
```

Результат должен быть таким:

```
GHCi> runStrRdr (runStateT stSrTst' 33) "ABCD"
((33,4,4,"ABCD"),37)
```

▶ Если трансформер требует операций ввода-вывода, то в качестве его основы используют не `Identity`, а `IO`. Для подъема операций ввода-вывода во внешние трансформеры используют специальный класс типов из модуля `Control.Monad.IO.Class`

```
class Monad m => MonadIO m where
  liftIO :: IO a -> m a

instance MonadIO IO where
  liftIO :: IO a -> IO a
  liftIO = id
```

Сделайте трансформер `StrRdrT` представителем этого класса типов, если внутренняя монада `m` выставляет этот интерфейс:

```
instance MonadIO m => MonadIO (StrRdrT m) where
  liftIO :: MonadIO m => IO a -> StrRdrT m a
  liftIO = undefined
```

Для проверки используйте

```
testSRIO :: StrRdrT IO String
testSRIO = do
  x <- liftIO getLine
```

```
  e <- askSR
  return $ x ++ e
```

Сессия ввода-вывода должна выглядеть как-то так:

```
GHCi> runStrRdrT testSRIO " rules!"
Simon Peyton Jones
"Simon Peyton Jones rules!"
GHCi>
```

Домашнее задание

▶ Сделайте на основе типа данных[2]

```
data Logged a = Logged String a deriving (Eq,Show)
```

трансформер монад `LoggT :: (* -> *) -> * -> *` с одноименным конструктором данных и меткой поля `runLoggT`:

```
newtype LoggT ...
```

Реализуйте для произвольной монады `m` представителя класса типов `Monad` для `LoggT m :: * -> *`

```
instance Monad m => Monad (LoggT m) where
  return x = undefined
  m >>= k  = undefined
```

а также класса типов `MonadFail`:

```
instance MonadFail m => MonadFail (LoggT m) where
  fail msg = undefined
```

Для проверки используйте функции

```
logTst :: LoggT Identity Integer
logTst = do
  x <- LoggT $ Identity $ Logged "AAA" 30
  y <- return 10
  z <- LoggT $ Identity $ Logged "BBB" 2
  return $ x + y + z

failTst :: [Integer] -> LoggT [] Integer
failTst xs = do
  5 <- LoggT $ fmap (Logged "") xs
  LoggT [Logged "A" ()]
  return 42
```

которые при правильной реализации монады должны вести себя так

[2] Семантика монады для него обсуждалась на прошлом занятии.

```
GHCi> runIdentity (runLoggT logTst)
Logged "BBBAAA" 42
GHCi> runLoggT $ failTst [5,5]
[Logged "A" 42,Logged "A" 42]
GHCi> runLoggT $ failTst [5,6]
[Logged "A" 42]
GHCi> runLoggT $ failTst [7,6]
[]
```

(1 балл)

▶ Напишите функцию `write2log` обеспечивающую трансформер `LoggT` стандартным логгирующим интерфейсом:

```
write2log :: Monad m => String -> LoggT m ()
write2log = undefined
```

Эта функция позволяет пользователю осуществлять запись в лог в процессе вычисления в монаде `LoggT m` для любой монады `m`. Введите для удобства упаковку для `LoggT Identity` и напишите функцию запускающую вычисления в этой монаде

```
type Logg = LoggT Identity

runLogg :: Logg a -> Logged a
runLogg = undefined
```

Тест

```
logTst' :: Logg Integer
logTst' = do
  write2log "AAA"
  write2log "BBB"
  return 42
```

должен дать такой результат

```
GHCi> runLogg logTst'
Logged "BBBAAA" 42
```

А тест

```
stLog :: StateT Integer Logg Integer
stLog = do
  modify (+1)
  a <- get
  lift $ write2log $ show $ a * 10
  put 42
  return $ a * 100
```

такой

```
GHCi> runLogg $ runStateT stLog 2
Logged "30" (300,42)
```

(1 балл)

▶ В последнем примере функция

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

позволяла поднять вычисление из внутренней монады (в примере это был `Logg`) во внешний трансформер (`StateT Integer`). Это возможно, поскольку для трансформера `StateT s` реализован представитель класса типов `MonadTrans`. Сделайте трансформер `LoggT` представителем класса `MonadTrans`, так чтобы можно было поднимать вычисления из произвольной внутренней монады в наш трансформер:

```
instance MonadTrans LoggT where
  lift = undefined

logSt :: LoggT (State Integer) Integer
logSt = do
  lift $ modify (+1)
  a <- lift get
  write2log $ show $ a * 10
  lift $ put 42
  return $ a * 100
```

Проверка:

```
GHCi> runState (runLoggT logSt) 2
(Logged "30" 300,42)
```

(1 балл)

▶ Избавьтесь от необходимости ручного подъема операций вложенной монады `State` , сделав трансформер `LoggT`, примененный к монаде с интерфейсом `MonadState`, представителем этого (`MonadState`) класса типов:

```
instance MonadState s m => MonadState s (LoggT m) where
  get   = undefined
  put   = undefined
  state = undefined

logSt' :: LoggT (State Integer) Integer
logSt' = do
  modify (+1)                    -- no lift!
  a <- get                       -- no lift!
  write2log $ show $ a * 10
  put 42                         -- no lift!
  return $ a * 100
```

Проверка:

```
GHCi> runState (runLoggT logSt') 2
(Logged "30" 300,42)
```

(1 балл)

▶ Избавьтесь от необходимости ручного подъема операций вложенной монады `Reader`, сделав монаду `LoggT m` представителем класса типов `MonadReader`:

```
instance MonadReader r m => MonadReader r (LoggT m) where
  ask    = undefined
  local  = undefined
  reader = undefined
```

Для упрощения реализации функции `local` имеет смысл использовать вспомогательную функцию, поднимающую стрелку между двумя «внутренними представлениями» трансформера `LoggT` в стрелку между двумя `LoggT`:

```
mapLoggT :: (m (Logged a) -> n (Logged b)) -> LoggT m a -> LoggT n b
mapLoggT f = undefined
```

Тест:

```
logRdr :: LoggT (Reader [(Int,String)]) ()
logRdr = do
  x <- asks $ lookup 2                        -- no lift!
  write2log $ maybe "NO DATA" id x
  y <- local ((3,"Jim"):) $ asks $ lookup 3 -- no lift!
  write2log $ maybe "NO DATA" id y
```

Ожидаемый результат:

```
GHCi> runReader (runLoggT logRdr) [(1,"John"),(2,"Jane")]
Logged "JimJane" ()
```

(1 балл)

▶ Чтобы избавится от необходимости ручного подъема операции `write2log`, обеспечивающей стандартный интерфейс вложенного трансформера `LoggT`, можно поступить по аналогии с другими трансформерами библиотеки `mtl`. А именно, разработать класс типов `MonadLogg`, выставляющий этот стандартный интерфейс[3] для нашего логгера:

```
class Monad m => MonadLogg m where
  w2log :: String -> m ()
  logg :: Logged a -> m a
```

[3]Мы переименовываем функцию `write2log` в `w2log`, поскольку хотим держать всю реализацию в одном файле исходного кода. При следовании принятой в библиотеках `transformers`/`mtl` идеологии они имели бы одно и то же имя, но были бы определены в разных модулях. При работе с `transformers` мы импортировали бы свободную функцию с квалифицированным именем `Control.Monad.Trans.Logg.write2log`, а при использовании `mtl` работали бы с методом класса типов `Control.Monad.Logg.write2log`.

Этот интерфейс, во-первых, должен выставлять сам трансформер `LoggT`, обернутый вокруг произвольной монады:

```
instance Monad m => MonadLogg (LoggT m) where
  w2log s = undefined
  logg = undefined
```

Реализуйте этого представителя, для проверки используйте

```
logSt'' :: LoggT (State Integer) Integer
logSt'' = do
  x <- logg $ Logged "BEGIN" 1
  modify (+x)
  a <- get
  w2log $ show $ a * 10
  put 42
  w2log "END"
  return $ a * 100
```

Результат должен быть таким:

```
GHCi> runState (runLoggT logSt'') 2
(Logged "END30BEGIN" 300,42)
```

Во-вторых, интерфейс `MonadLogg` должен выставлять любой стандартный трансформер, обернутый вокруг вокруг произвольной монады, выставляющей этот интерфейс:

```
instance MonadLogg m => MonadLogg (StateT s m) where
  w2log = undefined
  logg = undefined

instance MonadLogg m => MonadLogg (ReaderT r m) where
  w2log = undefined
  logg = undefined

-- etc...
```

Реализуйте двух этих представителей, для проверки используйте

```
rdrStLog :: ReaderT Integer (StateT Integer Logg) Integer
rdrStLog = do
  x <- logg $ Logged "BEGIN" 1
  y <- ask
  modify (+ (x+y))
  a <- get
  w2log $ show $ a * 10
  put 42
  w2log "END"
  return $ a * 100
```

Результат должен быть таким:

```
GHCi> runLogg $ runStateT (runReaderT rdrStLog 4) 2
Logged "END70BEGIN" (700,42)
```

(1 балл)

▶ Если трансформер требует операций ввода-вывода, то в качестве его основы используют не `Identity`, а `IO`. Для подъема операций ввода-вывода во внешние трансформеры используют специальный класс типов из модуля `Control.Monad.IO.Class`

```
class Monad m => MonadIO m where
    liftIO :: IO a -> m a

instance MonadIO IO where
    liftIO = id
```

Сделайте трансформер `LoggT` представителем этого класса типов, если внутренняя монада `m` выставляет этот интерфейс:

```
instance MonadIO m => MonadIO (LoggT m) where
  liftIO = undefined
```

Для проверки используйте

```
logIO :: LoggT IO ()
logIO = do
```

```
  x <- liftIO getLine
  w2log x
```

Сессия ввода-вывода должна выглядеть как-то так:

```
GHCi> runLoggT logIO
Simon Peyton Jones
Logged "Simon Peyton Jones" ()
```

(* баллов)

▶ Тип `Logged` является «внутренним представлением» для `LoggT`, так же как тип `Either e` для `ExceptT e` или частично примененная пара для `WriterT w`. Сам по себе тип `Logged` является монадой (так же как `Either e` или пара). Вы реализовывали эту монаду на прошлой практике

```
instance Monad Logged where
  return  = undefined
  m >>= k = undefined
```

Сделайте теперь монаду `Logged` представителем класса типов `MonadLogg`

```
instance MonadLogg Logged where
  w2log  = undefined
  logg   = undefined
```

Для проверки используйте

```
loggedTst :: Logged Integer
loggedTst = do
  w2log "AAA"
  logg $ Logged "BBB" 42
```

Результат должен быть таким:

```
GHCi> loggedTst
Logged "BBBAAA" 42
```

(* баллов)