Декларативное программирование

Весна 2023, семинар №6

Завьялов А.А.

14 марта 2023 г.

Кафедра систем информатики ФИТ НГУ

Монадные трансформеры

Монады и вычислительные эффекты

- Частичность (`Maybe`, `Either`)
- Недетерминированность
  - Нестабильность (`Reader`)
  - Множественность (`[]`)
- Побочный эффект (`Writer`)
- Вычисление с состоянием (`State`)
- Ввод-вывод (`IO`)

Монады и вычислительные эффекты

- Частичность (`Maybe`, `Either`)
- Недетерминированность
  - Нестабильность (`Reader`)
  - Множественность (`[]`)
- Побочный эффект (`Writer`)
- Вычисление с состоянием (`State`)
- Ввод-вывод (`IO`)

Как использовать несколько монад вместе?

Композиция монад. Наивный подход

Пример. Нужно прочитать из нескольких *файлов*, файлы могут отсутствовать:

```
getContent :: FileName -> IO (Maybe Text)

readFromFiles :: IO (Maybe a)
readFromFiles = do
  u <- getContent "file1"
  case u of
    Nothing -> return Nothing
    Just v -> do
      w <- getContent "file2"
        case w of
          Nothing -> return Nothing
          Just x -> ...
```

Комбинируем Maybe и IO

Сделаем тип-обёртку MaybeIO, который будет вести себя и как IO, и как Maybe:

```
newtype MaybeIO a = MaybeIO { runMaybeIO :: IO (Maybe a) }

instance Monad MaybeIO where
  return x = MaybeIO (return (Just x))
  MaybeIO action >>= f = MaybeIO $ do
    resutlt <- action
      case result of
        Nothing -> return Nothing
        Just x -> runMaybeIO (f x)
```

Пример. Нужно прочитать из нескольких *файлов*, файлы могут отсутствовать:

```
getContent :: FileName -> IO (Maybe Text)

readFromFiles :: IO (Maybe a)
readFromFiles = runMaybeIO $ do
  u <- MaybeIO $ getContent "file1"
  v <- MaybeIO $ getContent "file2"
  ...
```

Пример. Нужно прочитать из нескольких *файлов*, файлы могут отсутствовать:

```
getContent :: FileName -> IO (Maybe Text)

readFromFiles :: IO (Maybe a)
readFromFiles = runMaybeIO $ do
  u <- MaybeIO $ getContent "file1"
  v <- MaybeIO $ getContent "file2"
  ...
```

Команды IO в MaybeIO

Хотим исполнять произвольные IO-команды:

```
getContent :: FileName -> IO (Maybe Text)

readFromFiles :: IO (Maybe a)
readFromFiles = runMaybeIO $ do
  u <- MaybeIO $ getContent "file1"
  putStrLn "Файл прочитан"
  v <- MaybeIO $ getContent "file2"
  ...
```

Но `putStrLn "Файл прочитан" :: IO ()`, а нужен `MaybeIO ()`

Команды IO в MaybeIO

Напишем функцию-адаптер:

```
getContent :: FileName -> IO (Maybe Text)

liftIO2MaybeIO :: IO a -> MaybeIO a
liftIO2MaybeIO action = MaybeIO $ do
  result <- action
  return (Just result)

readFromFiles :: IO (Maybe a)
readFromFiles = runMaybeIO $ do
  u <- MaybeIO $ getContent "file1"
  liftIO2MaybeIO $ putStrLn "Файл прочитан"
  ...
```

Обобщение MaybeIO

Обобщим MaybeIO, чтобы вместо IO была произвольная монада:

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

instance Monad m => Monad (MaybeT m) where
  return x = MaybeT (return (Just x))

  MaybeT action >>= f = MaybeT $ do
    result <- action
    case result of
      Nothning -> return Nothing
      Just x -> runMaybeT (f x)
```

Обобщение `liftIO2MaybeIO`

Адаптер тоже обобщим:

```
liftToMaybeT :: Functor m => m a -> MaybeT m a
liftToMaybeT = MaybeT . fmap Just
```

Класс типов MonadTrans

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

instance MonadTrans MaybeT where
  lift = liftToMaybeT -- MaybeT . fmap Just
```

Законы:

- `lift . return == return`
- `lift (m >>= f) == lift m >>= (lift . f)`

Монадные трансформеры

Монада	Трансформер	Тип монады	Тип трансформера
Maybe	MaybeT	`Maybe a`	`m (Maybe a)`
Either	EitherT	`Either a b`	`m (Either a b)`
Writer	WriterT	`(a, w)`	`m (a, w)`
State	StateT	`s -> (a, s)`	`s -> m (a, s)`

Пакеты

- `transformers` – `MonadTrans` и реализации базовых трансформеров
- `mtl` – неявный лифтинг и другие расширения

Неявный лифтинг

```haskell
foo :: Int -> StateT [Int] (Reader Int) Int
-- или
foo :: Int -> ReaderT Int (State [Int]) Int

foo i = do
  baseCounter <- ask -- если StateT, то почему не lift ask?
  let newCounter = baseCounter + i
  put [baseCounter, newCounter] -- если ReaderT, то почему не lift $ put ...
  return newCounter
```

Компилируется!

Неявный лифтинг

```haskell
foo :: Int -> StateT [Int] (Reader Int) Int
-- или
foo :: Int -> ReaderT Int (State [Int]) Int

foo i = do
  baseCounter <- ask -- если StateT, то почему не lift ask?
  let newCounter = baseCounter + i
  put [baseCounter, newCounter] -- если ReaderT, то почему не lift $ put ...
  return newCounter
```

Компилируется! Работает...

Неявный лифтинг

```haskell
foo :: Int -> StateT [Int] (Reader Int) Int
-- или
foo :: Int -> ReaderT Int (State [Int]) Int

foo i = do
  baseCounter <- ask -- если StateT, то почему не lift ask?
  let newCounter = baseCounter + i
  put [baseCounter, newCounter] -- если ReaderT, то почему не lift $ put ...
  return newCounter
```

Компилируется! Работает... **Но как?**

Секрет неявного лифтинга – специальные классы типов

В пакете **mtl** есть классы вроде:

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

instance MonadReader r m => MonadReader r (StateT s m) where
    ask    = lift ask
    local  = mapStateT . local
    reader = lift . reader
```

- `MonadWriter`, `MonadState` и другие
- Не нужно писать цепочки `lift . lift . lift ...`

Класс типов MonadIO

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

instance MonadIO IO where
  liftIO = id

instance MonadIO m => MonadIO (StateT s m) where
  liftIO = lift . liftIO

instance MonadIO m => MonadIO (ReaderT r m) where
  liftIO = lift . liftIO
```

Класс типов MonadIO

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

instance MonadIO IO where
  liftIO = id

instance MonadIO m => MonadIO (StateT s m) where
  liftIO = lift . liftIO

instance MonadIO m => MonadIO (ReaderT r m) where
  liftIO = lift . liftIO
```

Проблема с трансформерами

Пакет `transformers`

- Хотим скомпоновать более двух монад $\Rightarrow$ `lift . lift . lift ...`

Пакет `mtl`

- Решает проблему `transformers`
- Плохо расширяется
  - BoxT $\Rightarrow$

[1] все трансформеры, реализованные в `transformers`, `mtl` и проекте

Проблема с трансформерами

Пакет `transformers`

- Хотим скомпоновать более двух монад $\Rightarrow$ `lift . lift . lift ...`

Пакет `mtl`

- Решает проблему `transformers`
- Плохо расширяется
  - BoxT $\Rightarrow$ MonadBox $\Rightarrow$

[1] все трансформеры, реализованные в `transformers`, `mtl` и проекте

Проблема с трансформерами

Пакет `transformers`

- Хотим скомпоновать более двух монад $\Rightarrow$ `lift . lift . lift ...`

Пакет `mtl`

- Решает проблему `transformers`
- Плохо расширяется
  - BoxT $\Rightarrow$ MonadBox $\Rightarrow$
  - $\Rightarrow$ `instance MonadBox r m => MonadBox r (TransT e m)`
    $\forall TransT \in DefinedTransformers$[1]

[1] все трансформеры, реализованные в `transformers`, `mtl` и проекте

Читаем

- https://en.wikibooks.org/wiki/Haskell/Monad_transformers
- https://github.com/sdiehl/wiwinwlh/blob/master/tutorial.md#monad-transformers

Q&A