

Если вы видите что-то необычное, просто сообщите мне.

# Преобразователи Монад

В нескольких прошлых частях серии, мы изучили множество новых монад. В 3 части мы увидели как часто вещи как `Maybe` и `IO` могут быть монадами. Затем в 4 и 5 частях мы изучили `Reader`, `Writer` и `State` монады. С этими монадами на поясе, вы возможно думаете как можно их объединять. Ответ, как мы обнаружи в этой части, это преобразователи монад.

С пониманием монад, вы открываете больше Haskell возможностей. Но вам всё ещё нужны идеи библиотек Haskell, который позволят вам их испытать.

## Пример Мотивации

Ранее, мы уже видели как монада `maybe` помогает избежать треугольника судьбы шаблонов кода. Без них, нам нужно проверять каждую функцию на успех. Однако, примеры на которые мы смотрим, где всё является чистым кодом предполагает следующее:

```
main1 :: IO ()
main1 = do
  maybeUserName <- readUserName
  case maybeUserName of
    Nothing -> print "Invalid user name!"
    Just (uName) -> do
      maybeEmail <- readEmail
      case maybeEmail of
        Nothing -> print "Invalid email!"
        Just (email) -> do
          maybePassword <- readPassword
          Case maybePassword of
            Nothing -> print "Invalid Password"
            Just password -> login uName email password
```

```
readUserName :: IO (Maybe String)
readUserName = do
  putStrLn "Please enter your username!"
  str <- getLine
  if length str > 5
    then return $ Just str
    else return Nothing

readEmail :: IO (Maybe String)
readEmail = do
  putStrLn "Please enter your email!"
  str <- getLine
  if '@' `elem` str && '.' `elem` str
    then return $ Just str
    else return Nothing

readPassword :: IO (Maybe String)
readPassword = do
  putStrLn "Please enter your Password!"
  str <- getLine
  if length str < 8 || null (filter isUpper str) || null (filter isLower str)
    then return Nothing
    else return $ Just str

login :: String -> String -> String -> IO ()
...
```

В этом примере, все наши потенциальные проблемы кода идут из `IO` монады. Как мы можем использовать `Maybe` монаду когда мы уже внутри другой монады?

# Преобразователи Монад

К счастью, мы можем получить желаемое поведение используя преобразователи монад для объединения. В этом примере, мы обернем `IO` действие внутри преобразованной монады `MaybeT`.

Преобразователи Монад это оберточный тип. В общем параметризуемый другим монадическим типом. Затем вы можете запустить действие из внутренней монады, в то время пока добавляете ваше собственное поведение для действия объединения в новую монаду. Общий преобразователь добавляет `T` в конец существующей монады. Ниже представлено определение `MaybeT`:

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

```
instance (Monad m) => Monad (MaybeT m) where
```

```
  return = lift . return
```

```
  x >>= f = MaybeT $ do
```

```
    v <- runMaybeT x
```

```
    case v of
```

```
      Nothing -> return Nothing
```

```
      Just y  -> runMaybeT (f y)
```

`MaybeT` сам по себе это newtype. Он содержит обертку над значением `Maybe`. Если тип `m` это `monad`, мы можем так же сделать монаду из `MaybeT`.

Представим наш пример. Мы хотим использовать `MaybeT` для оборачивания `IO` монады, чтобы запустить `IO` действия. Это значит, что наша новая монада `MaybeT IO`. Наши три вспомогательные функции все возвращают строки, поэтому каждая из них получает тип `MaybeT IO String`. Для преобразования старого `IO` кода в `MaybeT` монаду, всё, что нужно - обернуть `IO` действие в `MaybeT` конструктор.

```
readUserName' :: MaybeT IO String
```

```
readUserName' = MaybeT $ do
```

```
  putStrLn "Please enter your Username!"
```

```
  str <- getLine
```

```
  if length str > 5
```

```
    then return $ Just str
```

```
    else return Nothing
```

```
readEmail' :: MaybeT IO String
```

```
readEmail' = MaybeT $ do
```

```
  putStrLn "Please enter your Email!"
```

```
  str <- getLine
```

```
  if length str > 5
```

```
then return $ Just str
else return Nothing
```

```
readPassword' :: MaybeT IO String
readPassword' = MaybeT $ do
  putStrLn "Please enter your Password!"
  str <- getLine
  if length str < 8 || null (filter isUpper str) || null (filter isLower str)
  then return Nothing
  else return $ Just str
```

Теперь мы можем обернуть все три этих вызова в одно монадическое действие, и сделать простое сравнение для получения результата. Мы воспользуемся `runMaybeT` функцией для развертывания значения `Maybe` из `MaybeT`:

```
main2 :: IO ()
main2 = do
  maybeCreds <- runMaybeT $ do
    usr <- readUserName
    email <- readEmail
    pass <- readPassword
    return (usr, email, pass)
  case maybeCreds of
    Nothing -> print "Couldn't login!"
    Just (u, e, p) -> login u e p
```

И этот новый код будет иметь правильное простое поведение для `Maybe` монады. Если какая-то функция `read` упадет, наш код сразу же вернет `Nothing`.

## Добавление уровней.

Вот и мы дождались долгожданной части о преобразователях монад. Так как наш новосозданный тип сам по себе монада, мы можем обернуть её внутри другого преобразователя. Почти все распространенные монады имеют преобразователь типа, `MaybeT` в том числе, это преобразователь для обычной `Maybe` монады.

Для быстрого примера, предположим, у нас есть `Env` тип содержащий пользовательскую информацию. Мы можем обернуть это окружение в `Reader`. Однако, мы хотим всё ещё иметь доступ к `IO` функциональности, поэтому мы воспользуемся `Reader` преобразователем. Затем обернем результат с помощью `MaybeT`.

```
type Env = (Maybe String, Maybe String, Maybe String)

readUserName" :: MaybeT (ReaderT Env IO) String
readUserName" = MaybeT $ do
  (maybeOldUser, _, _) <- ask
  case maybeOldUser of
    Just str -> return $ Just str
    Nothing -> do
      -- lift allows normal IO functions from inside ReaderT Env IO!
      lift $ putStrLn "Please enter your Username!"
      input <- lift getLine
      if length input > 5
        then return (Just input)
        else return Nothing
```

Заметим, что у нас нужно использовать `lift` для запуска `IO` функции `getLine`. В преобразователе монады, `lift` функция позволяет нам запустить действия нижележащей монады. Это поведение захватывается классом `MonadTrans`:

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

Использование `lift` в `ReaderT Env IO` действии позволяет `IO` функцию. Использование типа шаблона из класса, мы можем заменить `Reader Env` на `t` и `IO` на `m`.

Внутри `MaybeT (ReaderT Env IO)` функции, вызываемой `lift` позволяет вам запустить функцию `Reader`. Нам не нужно то что выше, так как набор кода лежит в `Reader` действии в обертке `MaybeT` конструктора.

Чтобы понять идею лифтинга, подумайте о уровне вашей монады как о стеке. Когда вы имеете `ReaderT Env IO` действие, представьте, что `Reader Env` монада сверху `IO` монады. `IO` действие лежит на нижнем уровне. Поэтому, чтобы запустить всё это дело с верхнего слоя, вам нужно сначала подняться. Если ваш стек имеет больше чем 2 слоя, вы можете

подниматься несколько раз. Вызывая дважды `MaybeT (ReaderT Env IO)` монаду позволит вам вызывать `IO` функцию.

Не удобно каждый раз знать сколько раз тебе нужно вызывать функцию `lift` для получения текущего уровня. Отсюда вспомогательная функция часто используется для этого.

Вдобавок, после преобразования монады, можно запустить несколько уровней, типы могут становится сложнее. Поэтому обычно используют библиотеку `synonyms`.

```
type TripleMonad a = MaybeT (ReaderT Env IO) a

performReader :: ReaderT Env IO a -> TripleMonad a
performReader = lift

performIO :: IO a -> TripleMonad a
performIO = lift . lift
```

# Типоклассы

В качестве похожей идеи, есть `typeclass` который позволяет нам сделать определенные предположения о стеке монады. Для примера, вас часто не волнует, что именно в стеке, но вам нужен `IO` где-то внутри. В этом и заключается цель использования `MonadIO` типокласса.

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

We can use this behavior to get a function to print even when we don't know its exact monad:

```
debugFunc :: (MonadIO m) => String -> m ()
debugFunc input = liftIO $ putStrLn ("Successfully produced input: " ++ input)
```

Даже не смотря на то, что функция явно не находится в `MaybeT IO`, мы можем написать нашу версию `main` функции чтобы использовать её.

```
main3 :: IO ()
main3 = do
  maybeCreds <- runMaybeT $ do
```

```
usr <- readUserName'
debugFunc usr
email <- readEmail'
debugFunc email
pass <- readPassword'
debugFunc pass
return (usr, email, pass)
case maybeCreds of
  Nothing -> print "Couldn't login!"
  Just (u, e, p) -> login u e p
```

“Вы не можете, в общем, обернуть другую монаду с помощью `IO` монады используя преобразователь. Однако, можно сделать другое монаадическое значение чтобы вернуть тип `IO` действия.

```
func :: IO (Maybe String)
-- This type makes sense

func2 :: IO_T (ReaderT Env (Maybe)) string
-- This does not exist
```

# Выводы

Теперь, вы знаете, как объединять ваши монады, вы почти завершили понимание ключевых идей! Вы, возможно, хотите попробовать начать писать достаточно сложный код. Но, чтобы научиться владеть монадами, вам нужно знать как делать свою собственную монаду, и для этого вам нужно понять последнюю идею. Это идея типа `laws`. Каждая структура, которую мы прошли в этой части лекций, связана с `laws`. И чтобы ваши примеры имели смысл, они должны следовать `laws` (т.е. закону). Проверьте 7 главу, чтобы понять, понимаете ли вы что происходит.

Revision #6

Created 11 March 2022 05:43:59 by gasick

Updated 6 October 2022 05:34:54 by gasick