Взаимодействие с внешним миром на языке Haskell

Григорий Волков

2022

ИСП РАН

Задание, никак не связанное с темой занятия

Вспомним тип Maybe:

```haskell
data Maybe a = Nothing | Just a
```

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

Как работать с несколькими значениями? Давайте напишем такую функцию:

```haskell
sumOfHeads :: Num a => [a] -> [a] -> Maybe a
sumOfHeads xs ys =
```

Задание, никак не связанное с темой занятия

Вспомним тип Maybe:

```haskell
data Maybe a = Nothing | Just a
```

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

Как работать с несколькими значениями? Давайте напишем такую функцию:

```haskell
sumOfHeads :: Num a => [a] -> [a] -> Maybe a
sumOfHeads xs ys =

  case (safeHead xs, safeHead ys) of
    (Just xh, Just yh) -> Just (xh + yh)
    _ -> Nothing
```

Не очень красиво. Три раза Just? :/

Рефакторинг

Иногда чтобы сделать лучше, надо сначала сделать хуже:

```
sumOfHeads :: Num a => [a] -> [a] -> Maybe a
sumOfHeads xs ys =
  case safeHead xs of
    Nothing -> Nothing
    Just xh -> case safeHead ys of
      Nothing -> Nothing
      Just yh -> Just (xh + yh)
```

Так виднее, что мы два раза делаем операцию «достать значение из `Maybe` и продолжить операции если там `Just`, иначе вернуть `Nothing`». Выделяем:

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Рефакторинг

Иногда чтобы сделать лучше, надо сначала сделать хуже:

```
sumOfHeads :: Num a => [a] -> [a] -> Maybe a
sumOfHeads xs ys =
  case safeHead xs of
    Nothing -> Nothing
    Just xh -> case safeHead ys of
      Nothing -> Nothing
      Just yh -> Just (xh + yh)
```

Так виднее, что мы два раза делаем операцию «достать значение из Maybe и продолжить операции если там Just, иначе вернуть Nothing». Выделяем:

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b

andThen Nothing _ = Nothing
andThen (Just a) f = f a

sumOfHeads xs ys = safeHead xs `andThen` (\xh ->
                   safeHead ys `andThen` (\yh -> Just $ xh + yh))
```

Другие типы

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just a) f = f a
```

Нетрудно догадаться, что то же самое полезно для `Either`:

```
andThen :: Either e a -> (a -> Either e b) -> Either e b
andThen (Left l) _ = Left l
andThen (Right r) f = f r
```

А можем написать для списков?

```
andThen :: [a] -> (a -> [b]) -> [b]
```

Другие типы

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just a) f = f a
```

Нетрудно догадаться, что то же самое полезно для `Either`:

```
andThen :: Either e a -> (a -> Either e b) -> Either e b
andThen (Left l) _ = Left l
andThen (Right r) f = f r
```

А можем написать для списков?

```
andThen :: [a] -> (a -> [b]) -> [b]

andThen [] _ = []
andThen (x : xs) f = f x ++ andThen xs f
```

Что она делает?

Другие типы

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just a) f = f a
```

Нетрудно догадаться, что то же самое полезно для Either:

```
andThen :: Either e a -> (a -> Either e b) -> Either e b
andThen (Left l) _ = Left l
andThen (Right r) f = f r
```

А можем написать для списков?

```
andThen :: [a] -> (a -> [b]) -> [b]
```

```
andThen [] _ = []
andThen (x : xs) f = f x ++ andThen xs f
```

Что она делает?

```
Prelude> [1..5] `andThen` \x -> [x, x * 2, x * 3]
[1,2,3,2,4,6,3,6,9,4,8,12,5,10,15]
```

Подробнее о списках

```
Prelude> [1..5] `andThen` \x -> [x, x * 2, x * 3]
[1,2,3,2,4,6,3,6,9,4,8,12,5,10,15]
```

Такое уже есть в стандартной библиотеке: `concatMap`!

```
Prelude> concatMap (\x -> [x, x * 2, x * 3]) [1..5]
[1,2,3,2,4,6,3,6,9,4,8,12,5,10,15]
```

Подробнее о списках

```
Prelude> [1..5] `andThen` \x -> [x, x * 2, x * 3]
[1,2,3,2,4,6,3,6,9,4,8,12,5,10,15]
```

Такое уже есть в стандартной библиотеке: concatMap!

```
Prelude> concatMap (\x -> [x, x * 2, x * 3]) [1..5]
[1,2,3,2,4,6,3,6,9,4,8,12,5,10,15]
```

А помните подсчёт с прошлого занятия?

```
Prelude> length [ (a,b,c) | a <- [False, True]
                          , b <- [False, True], c <- [False, True] ]
8
```

Тот магический синтаксис на самом деле сделал именно это:

```
Prelude> [True, False] `andThen` \a ->
           [True, False] `andThen` \b ->
           [True, False] `andThen` \c -> [(a, b, c)]
[(True,True,True),(True,True,False),(True,False,True)
,(True,False,False),(False,True,True),(False,True,False)
,(False,False,True),(False,False,False)]
```

Обобщение

```
andThen ::    Maybe a  -> (a ->    Maybe b ) ->    Maybe b
andThen :: Either e a  -> (a -> Either e b ) -> Either e b
andThen ::          [a] -> (a ->          [b]) ->          [b]
```

Когда у нас есть такое, напрашивается класс типов.

Обобщение

```
andThen ::    Maybe a  -> (a ->    Maybe b ) ->    Maybe b
andThen :: Either e a  -> (a -> Either e b ) -> Either e b
andThen ::          [a] -> (a ->          [b]) ->          [b]
```

Когда у нас есть такое, напрашивается класс типов.

На самом деле это «те самые» монады!

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b -- читается как «bind»
```

Во всех предыдущих примерах andThen можно заменить на >>=.

Экземпляры класса Monad

```haskell
instance Monad Maybe where
  return = Just

  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= _  =  Nothing
  Just x  >>= f  =  f x
```

Экземпляры класса Monad

```haskell
instance Monad Maybe where
  return = Just

  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= _  =  Nothing
  Just x  >>= f  =  f x
```

Для списка: как мы уже заметили, есть встроенная функция...

```haskell
instance Monad [] where
  return x = [x]
  l >>= f  = concatMap f l
```

Законы класса Monad

На прошлом занятии мы узнали, что у класса `Functor` есть законы:

- `fmap id` эквивалентно `id`
- `fmap (f . g)` эквивалентно `fmap f . fmap g`

Эти формулы должны выполняться с каждым экземпляром `Functor`.

Законы класса Monad

На прошлом занятии мы узнали, что у класса `Functor` есть законы:

- `fmap id` эквивалентно `id`
- `fmap (f . g)` эквивалентно `fmap f . fmap g`

Эти формулы должны выполняться с каждым экземпляром `Functor`.

У `Monad` тоже есть законы:

- `return a >>= f` эквивалентно `f a`
- `m >>= return` эквивалентно `m`
- `(m >>= f) >>= g` эквивалентно `m >>= (\x -> f x >>= g)`

Ещё один класс типов

Monad — часть следующей иерархии классов:

```
class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  return = pure

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

Ещё один класс типов

Monad — часть следующей иерархии классов:

```
class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  return = pure

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

Интерфейс *аппликативных функторов* менее мощный, чем интерфейс монад:

`m1 <*> m2` эквивалентно

```
m1 >>= (\x1 -> m2 >>= (\x2 -> return (x1 x2)))
```

Он был придуман в частности для парсеров. Но об этом, наверное, потом :)

Ввод-вывод

Допустим, была бы функция `getLine :: String`. Что пошло бы не так?

Ввод-вывод

Допустим, была бы функция `getLine :: String`. Что пошло бы не так?

- нарушение чистоты (в других языках с этим живут...)
- **большие** проблемы с ленивостью! Например: `tail [getLine, getLine]`

Что можно сделать?

Ввод-вывод

Допустим, была бы функция `getLine :: String`. Что пошло бы не так?

- нарушение чистоты (в других языках с этим живут...)
- **большие** проблемы с ленивостью! Например: `tail [getLine, getLine]`

Что можно сделать? Заключить всё взаимодействие в специальный тип команд ввода-вывода:

```
data IO a = <magic>
```

Такую команду можно назвать `main` в исходном коде исполняемой программы:

```
main :: IO ()
main = putStrLn "hello"
```

Ввод-вывод

Допустим, была бы функция `getLine :: String`. Что пошло бы не так?

- нарушение чистоты (в других языках с этим живут...)
- **большие** проблемы с ленивостью! Например: `tail [getLine, getLine]`

Что можно сделать? Заключить всё взаимодействие в специальный тип команд ввода-вывода:

```
data IO a = <magic>
```

Такую команду можно назвать `main` в исходном коде исполняемой программы:

```
main :: IO ()
main = putStrLn "hello"
```

Или напрямую выполнить в консоли GHCi:

```
Prelude> putStrLn "hello world"
```

А чтобы исполнить что-то большее, чем одну команду?

Композиция IO

А мы не зря изучали монады! Какой смысл приобретает оператор >>= при его использовании на командах ввода-вывода?

```
(>>=)  ::  m a -> (a ->  m b) ->  m b
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

Композиция IO

А мы не зря изучали монады! Какой смысл приобретает оператор >>= при его использовании на командах ввода-вывода?

```
(>>=)  ::  m a -> (a ->  m b) ->  m b
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

Рассмотрим пример:

```
Prelude> getLine >>= \l -> putStrLn ("you said: " ++ l)
hi
you said: hi
```

Композиция IO

А мы не зря изучали монады! Какой смысл приобретает оператор >>= при его использовании на командах ввода-вывода?

```
(>>=)  ::  m a -> (a ->  m b) ->  m b
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

Рассмотрим пример:

```
Prelude> getLine >>= \l -> putStrLn ("you said: " ++ l)
hi
you said: hi
```

С помощью >>= можно создавать последовательности из команд ввода-вывода и функций. (Всё выражение в примере представляет собой одну объединённую команду, которую «привел в действие» GHCi!)

Композиция IO

Если на каком-то шаге не нужно ничего обрабатывать функцией, можно использовать:

```
  (>>) :: m a -> m b -> m b
```

```
Prelude> putStr "say: " >> getLine >>= \l ->
           putStrLn ("you said: " ++ l)
say: hello
you said: hello
```

Композиция IO

Если на каком-то шаге не нужно ничего обрабатывать функцией, можно использовать:

```
(>>) :: m a -> m b -> m b
```

```
Prelude> putStr "say: " >> getLine >>= \l ->
           putStrLn ("you said: " ++ l)
say: hello
you said: hello
```

Операторы из других классов типов тоже бывает удобно использовать:

```
Prelude> ("you said: " ++) <$> getLine >>= putStrLn
yo
you said: yo
```

Композиция, удобнее

Чтобы строить цепочки взаимодействий с внешним миром было удобнее, можно использовать do-нотацию: вместо

```
main :: IO ()
main = putStr "say: " >> getLine >>= \l ->
         putStrLn ("you said: " ++ l)
```

можно написать:

```
main :: IO ()
main = do
  putStr "say: "
  l <- getLine
  putStrLn $ "you said: " ++ l
```

do-нотация в общем

Строка в do-блоке эквивалентна применению >>=:

```
do res <- action1                    action1 >>= \res ->
   action2 res                         action2 res
```

do-нотация в общем

Строка в do-блоке эквивалентна применению >>=:

```
do res <- action1                    action1 >>= \res ->
   action2 res                         action2 res
```

Если у строки нет левой части (стрелки <-), её результат игнорируется (подобно оператору >>) или, если эта строка последняя, «возвращается» (остаётся последней в последовательности).

```
do action1                           action1 >>
   action2                             action2
```

do-нотация в общем

Строка в do-блоке эквивалентна применению >>=:

```
do res <- action1                    action1 >>= \res ->
   action2 res                         action2 res
```

Если у строки нет левой части (стрелки <-), её результат игнорируется (подобно оператору >>) или, если эта строка последняя, «возвращается» (остаётся последней в последовательности).

```
do action1                           action1 >>
   action2                             action2
```

Но ещё строки do-блока могут быть let без in:

```
do res <- action1                    action1 >>= \res ->
   let x = reverse res                 let x = reverse res
   action2 x                           in action2 res
```

do-нотация не привязана к IO

Вспомним нашу замечательную функцию safeHead:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x

sumOfHeads :: Num a => [a] -> [a] -> Maybe a
sumOfHeads xs ys = safeHead xs `andThen` (\xh ->
                     safeHead ys `andThen` (\yh -> Just $ xh + yh))

sumOfHeads xs ys = safeHead xs >>= \xh ->
                     safeHead ys >>= \yh -> Just (xh + yh)
```

Перепишем:

```
sumOfHeads xs ys = do
```

do-нотация не привязана к IO

Вспомним нашу замечательную функцию safeHead:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x

sumOfHeads :: Num a => [a] -> [a] -> Maybe a
sumOfHeads xs ys = safeHead xs `andThen` (\xh ->
                     safeHead ys `andThen` (\yh -> Just $ xh + yh))


sumOfHeads xs ys = safeHead xs >>= \xh ->
                     safeHead ys >>= \yh -> Just (xh + yh)
```

Перепишем:

```
sumOfHeads xs ys = do

  xh <- safeHead xs
  yh <- safeHead ys
  Just $ xh + yh -- или return/pure $ Just $ xh + yh
```

do ещё на одном типе

Был такой пример:

```
[True, False] `andThen` \a ->
  [True, False] `andThen` \b ->
  [True, False] `andThen` \c -> [(a, b, c)]
```

Как его переписать в do-нотацию?

do ещё на одном типе

Был такой пример:

```
[True, False] `andThen` \a ->
  [True, False] `andThen` \b ->
  [True, False] `andThen` \c -> [(a, b, c)]
```

Как его переписать в do-нотацию?

```
do a <- [True, False]
   b <- [True, False]
   c <- [True, False]
   pure (a, b, c)
```

do ещё на одном типе

Был такой пример:

```
[True, False] `andThen` \a ->
  [True, False] `andThen` \b ->
  [True, False] `andThen` \c -> [(a, b, c)]
```

Как его переписать в do-нотацию?

```
do a <- [True, False]
   b <- [True, False]
   c <- [True, False]
   pure (a, b, c)
```

Секрет того синтаксиса раскрыт!

```
Prelude> length [ (a,b,c) | a <- [False, True]
                          , b <- [False, True], c <- [False, True] ]
8
```

Стрелка влево там не случайно.

Монадические полезности

Допустим, у нас есть список IO команд: `[getLine, reverse <$> getLine, getLine]`. Как превратить его в команду, выдающую список результатов?

```
runAll :: [IO a] -> IO [a]
```

Монадические полезности

Допустим, у нас есть список IO команд: `[getLine, reverse <$> getLine, getLine]`. Как превратить его в команду, выдающую список результатов?

```
runAll :: [IO a] -> IO [a]

runAll [] = pure []
runAll (x : xs) = do
  hd <- x
  tl <- runAll xs
  pure $ hd : tl
```

Монадические полезности

Допустим, у нас есть список IO команд: `[getLine, reverse <$> getLine, getLine]`. Как превратить его в команду, выдающую список результатов?

```
runAll :: [IO a] -> IO [a]
```

```
runAll [] = pure []
runAll (x : xs) = do
  hd <- x
  tl <- runAll xs
  pure $ hd : tl
```

Как всегда, это есть в стандартной библиотеке:

```
Prelude> :t sequence
sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)
```

Монадические полезности 2 (import Control.Monad)

Кто хотел `if` без `else`? :)

```
when :: Applicative f => Bool -> f () -> f ()
```

Монадические полезности 2 (import Control.Monad)

Кто хотел `if` без `else`? :)

```
when :: Applicative f => Bool -> f () -> f ()
```

Можно использовать перевёрнутый >>=:

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
```

Монадические полезности 2 (import Control.Monad)

Кто хотел if без else? :)

```
when :: Applicative f => Bool -> f () -> f ()
```

Можно использовать перевёрнутый >>=:

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
```

Монадические аналоги обычных функций обработки коллекций:

```
filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]

userFilter :: Show a => [a] -> IO [a]
userFilter xs = filterM (\x -> do
  putStr $ "Include " ++ show x ++ "? y/n: "
  s <- getLine
  pure $ s == "y") xs
```

Переменные переменные

```haskell
import Data.IORef
```

В IO можно даже создавать настоящие переменные:

```haskell
main = do
  r <- newIORef 0
  writeIORef r 1
  modifyIORef r (+ 1)
  readIORef r >>= print -- print - совместно show и putStrLn
```

Однако не стоит их использовать для всего того, для чего они используются в императивных языках.

Переменные переменные

```haskell
import Data.IORef
```

В IO можно даже создавать настоящие переменные:

```haskell
main = do
  r <- newIORef 0
  writeIORef r 1
  modifyIORef r (+ 1)
  readIORef r >>= print -- print - совместно show и putStrLn
```

Однако не стоит их использовать для всего того, для чего они используются в императивных языках.

А как стоит организовывать программы?

“Functional Core, Imperative Shell”

```haskell
import Data.Ord (Down(..))          import Data.List (sortBy, nub)
import Control.Monad (forever)
import System.Directory (doesFileExist)

longestWords :: String -> [(Int, String)]
longestWords = take 4
  . sortBy (\(l1, _) (l2, _) -> compare (Down l1) (Down l2))
  . map (\w -> (length w, w)) . nub . words

main = forever $ do
  putStr "File path: "
  path <- getLine
  exists <- doesFileExist path
  if exists
    then do
      content <- readFile path
      putStrLn $ "Longest words: " ++ show (longestWords content)
    else putStrLn "File not found"
```

Спасибо за внимание!