Haskell-практикум. Разработка монадических интерфейсов. Тестирование. ДЗ.

Григорий Волков

2022

ИСП РАН

План занятия

Сегодня нам нужно успеть:

- изучить некоторые полезные штуки
- научиться работать с инструментом сборки `stack`

План занятия

Сегодня нам нужно успеть:

- изучить некоторые полезные штуки
- научиться работать с инструментом сборки `stack`
- ознакомиться со структурой домашнего задания
- прорешать его пробный вариант

План занятия

Сегодня нам нужно успеть:

- изучить некоторые полезные штуки
- научиться работать с инструментом сборки `stack`
- ознакомиться со структурой домашнего задания
- прорешать его пробный вариант

Поехали!

Работа с состоянием

Как в чисто функциональном языке работать с состоянием? Довольно логичное решение: брать состояние на вход, возвращать (возможно модифицированное) состояние на выход вместе с основным результатом вычисления.

Допустим, так выглядят операции, использующие стек в качестве состояния:

```
type Stack = [Int]

pop :: Stack -> (Int, Stack)
pop (x : xs) = (x, xs)

push :: Int -> Stack -> ((), Stack)
push a xs = ((), a : xs)
```

Работа с состоянием

Как в чисто функциональном языке работать с состоянием? Довольно логичное решение: брать состояние на вход, возвращать (возможно модифицированное) состояние на выход вместе с основным результатом вычисления.

Допустим, так выглядят операции, использующие стек в качестве состояния:

```
type Stack = [Int]

pop :: Stack -> (Int, Stack)
pop (x : xs) = (x, xs)

push :: Int -> Stack -> ((), Stack)
push a xs = ((), a : xs)
```

Тогда какой-нибудь последовательный алгоритм будет выглядеть так:

```
stkAlg :: Stack -> (Int, Stack)
stkAlg stack = let ((), stack') = push 123 stack
                       (a,  stack'') = pop stack'
                   in pop stack'' -- Так себе удовольствие…
```

Монады спешат на помощь

```
stkAlg :: Stack -> (Int, Stack)
stkAlg stack = let ((), stack') = push 123 stack
                   (a,  stack'') = pop stack'
               in pop stack''
```

можно превратить в

```
stkAlg = do
  push 123
  a <- pop
  pop
```

за счёт выноса всего этого шаблона «передай дальше» в >>=:

```
newtype State s a = State { runState :: s -> (a, s) }


instance Monad (State s) where
  return x = State $ \s -> (x, s)
  (State h) >>= f = State $ \s -> let (a, newState) = h s
                                      (State g) = f a
                                  in  g newState
```

State monad

Поскольку `State` — обёртка над функциями, наши операции должны быть соответственно завёрнуты:

```
pop :: Stack -> (Int, Stack)
pop (x : xs) = (x, xs)
```

```
pop :: State Stack Int
pop = State $ \(x : xs) -> (x,xs)
```

```
push :: Int->Stack->((), Stack)
push a xs = ((), a : xs)
```

```
push :: Int -> State Stack Int
push a = State $ \xs -> ((),a:xs)
```

Получается, >>= композирует такие функции, занимаясь передачей состояния вперёд.

State monad

Поскольку `State` — обёртка над функциями, наши операции должны быть соответственно завёрнуты:

```
pop :: Stack -> (Int, Stack)
pop (x : xs) = (x, xs)
```

```
pop :: State Stack Int
pop = State $ \(x : xs) -> (x,xs)
```

```
push :: Int->Stack->((), Stack)
push a xs = ((), a : xs)
```

```
push :: Int -> State Stack Int
push a = State $ \xs -> ((),a:xs)
```

Получается, `>>=` композирует такие функции, занимаясь передачей состояния вперёд.

Готовый тип `State` реализован в библиотеке `mtl`.

Полиморфизм плюс монады

Нетрудно догадаться, что объявление класса типов с монадическими операциями — хорошая идея:

```
class Console m where
  readln :: m String              writeln :: String -> m ()
```

Полиморфизм плюс монады

Нетрудно догадаться, что объявление класса типов с монадическими операциями — хорошая идея:

```
class Console m where
  readln :: m String              writeln :: String -> m ()
```

Такой интерфейс можно реализовать как для реального мира, так и для модели:

```
instance Console IO where
  readln = getLine

  writeln = putStrLn
```

```
instance Console (State [String]) where
  readln = get >>= \(x : xs) ->
    put xs >> pure x
  writeln = pure () -- TODO
```

Полиморфизм плюс монады

Нетрудно догадаться, что объявление класса типов с монадическими операциями — хорошая идея:

```
class Console m where
  readln :: m String              writeln :: String -> m ()
```

Такой интерфейс можно реализовать как для реального мира, так и для модели:

```
instance Console IO where
  readln = getLine
  writeln = putStrLn
```

```
instance Console (State [String]) where
  readln = get >>= \(x : xs) ->
    put xs >> pure x
  writeln = pure () -- TODO
```

И программировать относительно интерфейса, чтобы можно было запускать с любой реализацией:

```
echo :: (Console m, Monad m) => m ()
echo = do
  writeln "say something:"
  txt <- readln
  writeln $ "you said: " ++ txt
```

Stack

Инструмент сборки Stack

- автоматически устанавливает компилятор GHC
- устанавливает зависимости проекта
- собирает проект, запускает тесты и т.д.

Установка: https://haskellstack.org

Структура проекта

Командой `stack new homework` создаётся проект `homework` в соответствующей директории, на основе шаблона по умолчанию.

В созданной директории много всего:

- `package.yaml` содержит метаданные, определяет компоненты (пакеты) проекта и их зависимости
- в директории `src` расположены модули библиотеки — основного компонента проекта
- в `app` расположены модули исполняемого файла
  - в `package.yaml` прописано, что главный из них `Main.hs`
  - запуск: `stack run`
- в `test` расположены модули набора тестов
  - в `package.yaml` прописано, что главный из них `Spec.hs`
  - запуск: `stack test`

Важно: проблемы при работе с IDE

При возникновении рассинхронизации (например, поменяли тип в библиотеке, а в тестах IDE показывает ошибки, которые были бы со старым типом) нужно

- собрать проект
  - `stack build`, оно же уже включено в `stack test`
- перезапустить LSP сервер
  - VSCode: Command pallette $\rightarrow$ Haskell: Restart Haskell LSP Server

Если не помогло, перед этими действиями ещё сделать `stack clean`.

ДЗ!

Домашнее задание: общая структура

- В библиотеке
  - Реализовать функцию обработки данных в соответствии с вариантом задания
  - Определить класс типов, содержащий функции ввода-вывода для консоли
  - Реализовать интерактивную команду, запускающую вычисление из консоли (с прочитанными из неё входными данными, с выводом результата)

Домашнее задание: общая структура

- В библиотеке
  - Реализовать функцию обработки данных в соответствии с вариантом задания
  - Определить класс типов, содержащий функции ввода-вывода для консоли
  - Реализовать интерактивную команду, запускающую вычисление из консоли (с прочитанными из неё входными данными, с выводом результата)
- В исполняемом файле
  - Определить экземпляр класса типов консоли для `IO`
  - Вызвать интерактивную команду из `main`

Домашнее задание: общая структура

- В библиотеке
  - Реализовать функцию обработки данных в соответствии с вариантом задания
  - Определить класс типов, содержащий функции ввода-вывода для консоли
  - Реализовать интерактивную команду, запускающую вычисление из консоли (с прочитанными из неё входными данными, с выводом результата)
- В исполняемом файле
  - Определить экземпляр класса типов консоли для `IO`
  - Вызвать интерактивную команду из `main`
- В наборе тестов
  - Определить экземпляр класса типов консоли для типа вроде `State ([String], [String])` (из библиотеки `mtl`)
  - Используя библиотеку `hspec` построить тесты для интерактивной команды
  - Также можно построить тесты и для чистой функции тоже

Домашнее задание: общая структура

- В библиотеке
  - Реализовать функцию обработки данных в соответствии с вариантом задания
  - Определить класс типов, содержащий функции ввода-вывода для консоли
  - Реализовать интерактивную команду, запускающую вычисление из консоли (с прочитанными из неё входными данными, с выводом результата)
- В исполняемом файле
  - Определить экземпляр класса типов консоли для `IO`
  - Вызвать интерактивную команду из `main`
- В наборе тестов
  - Определить экземпляр класса типов консоли для типа вроде `State ([String], [String])` (из библиотеки `mtl`)
  - Используя библиотеку `hspec` построить тесты для интерактивной команды
  - Также можно построить тесты и для чистой функции тоже

Оценивание: за код, работающий только с «хорошими» входными данными ставится 8. Чтобы получить 10, нужно показать аккуратную обработку некорректного ввода (с тестами на это).

Защита: на оставшихся занятиях после лекции показываем, обсуждаем.

Домашнее задание: нулевой вариант

На вход подаётся строка, в которой закодировано несколько строк, перед началом каждой из которых записана её длина. Нужно вывести разобранный список строк (или сообщение об ошибке, если формат входных данных некорректный).

Тестовый пример: “5hello2hi” $\rightarrow$ “[”hello”,”hi”]”

LIVE DEMO

Результат: Lib (1/2)

```haskell
module Lib (module Lib) where

import Data.Char (isDigit)

readLV :: [Char] -> Maybe [[Char]]
readLV [] = Just []
readLV s | null (takeWhile isDigit s) = Nothing
readLV s = readLV (drop n s') >>= \xs -> Just (take n s' : xs)
  where n = read $ takeWhile isDigit s
        s' = dropWhile isDigit s
```

Результат: Lib (2/2)

```haskell
class Console m where
  readln :: m String
  writeln :: String -> m ()

consoleLV :: (Console m, Monad m) => m ()
consoleLV = do
  writeln "message:"
  msg <- readln
  case readLV msg of
    Nothing -> writeln "could not parse"
    Just x -> writeln $ show x
```

Результат: Main

```
module Main where

import Lib

instance Console IO where
  readln = getLine -- NOT readLn!
  writeln = putStrLn

main :: IO ()
main = consoleLV
```

Результат: Spec (1/2)

```haskell
{-# LANGUAGE FlexibleInstances #-}
import Test.Hspec
import Control.Monad.State
import Lib

data TestConsole = TC { linesToRead :: [String]
                      , linesWritten :: [String] } -- NOTE: reversed
                 deriving (Show, Eq)

instance Console (State TestConsole) where
  writeln l = modify (\(TC lr lw) -> TC lr (l : lw))
  readln = do
    TC lr lw <- get
    case lr of
      hd : tl -> do
        put $ TC tl lw
        pure hd
      _ -> pure "_no more input_"
```

Результат: Spec (2/2)

```haskell
runTest :: State TestConsole a -> [String] -> [String]
runTest f ls = reverse $ linesWritten $ execState f (TC ls [])

main :: IO ()
main = hspec $ do
  describe "consoleLV" $ do
    it "works" $ runTest consoleLV ["5hello2hi"]
      `shouldBe` ["message:", "[\"hello\",\"hi\"]"]
    it "handles fails" $ runTest consoleLV ["nope"]
      `shouldBe` ["message:", "could not parse"]
    it "handles fails after number" $ runTest consoleLV ["2nope"]
      `shouldBe` ["message:", "could not parse"]
```

Спасибо за внимание!