CS314. Функциональное программирование

Лекция 7. Монады

В. Н. Брагилевский

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

6 октября 2017 г.

Вычисление в контексте

```
func :: (...) => ... -> f a
```

- `f` определяет контекст, в котором вычисляется значение типа `a`
- `f :: * -> *`
- `func par1 ... parN :: f a` — это значение в контексте
- `f` может быть конкретным (конструктор типа):
  - `Maybe`
  - `Either a`
  - `[]`
  - `IO`
  - `(->) r`
- `f` может быть обобщённым (класс типов):
  - Functor
  - Applicative
  - ???

Класс Functor

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

- `fmap` — преобразование значения в контексте без изменения контекста
- `(<$)` — замена значения в контексте без изменения контекста (можно реализовать через `fmap`)

```
ghci> fmap (+1) (5 <$ Just 3)
Just 6
ghci> fmap (+1) (5 <$ [1..3])
[6,6,6]
```

Класс Applicative

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

- `pure` помещает значение в контекст
- `(<*>)` применяет функцию в контексте к значению в контексте
- `Applicative` позволяет обрабатывать несколько значений в контексте многопараметрической функцией (комбинировать результаты вычислений с учётом контекста)

```
ghci> pure (+) <*> [1,2,3] <*> [4,5,6]
[5,6,7,6,7,8,7,8,9]
ghci> pure (+) <*> [1,2,3] <*> []
[]
ghci> pure (+) <*> Nothing <*> Just 3
Nothing
```

Аппликативный функтор и его ограничения

- Можно:
  - Преобразовать значение в контексте без изменения контекста (`fmap`)
  - Поместить значение в контекст (`pure`)
  - Скомбинировать несколько вычислений в одно (`<*>`)
- Нельзя: извлечь результат одного вычисления и воспользоваться им во втором, например
  - прочитать строку с клавиатуры и вывести её
  - добавить каждый элемент первого списка в начало второго

Содержание

Проблема и её решение: класс типов Monad

- Как ввести строку (`getLine`), а затем её напечатать (`putStr`)?
- Наблюдение: необходимо скомбинировать два действия, причём второе зависит от результата первого.

```
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
```

- Операция `>>=` называется монадическим связыванием (bind)
- `(>>)` игнорирует результат первого вычисления
- `return` — это то же самое, что `pure`

Примеры

```
ghci> getLine >>= putStrLn
abc
abc
ghci> getLine >>= putStr >> putStrLn "!"
hello
hello!
```

```
ghci> Just 7 >>= (\x -> Just $ x+1)
Just 8
ghci> Nothing >>= (\x -> Just $ x+1)
Nothing
ghci> [1,2] >>= (\x -> [x-1, x+1])
[0,2,1,3]
```

Монады и функторы

- Любая монада является функтором (`fmap = <$> = liftM`).
- Любая монада является аппликативным функтором (`pure = return`, `<*> = ap`).
- Монады позволяют структурировать вычисления, причём каждый следующий шаг может зависеть от предыдущего.

```
myAction :: IO String
myAction = (++) `liftM` getLine `ap` getLine
--          (++)   <$>   getLine <*>  getLine

main = myAction >>= putStrLn . ("Результат: " ++)
```

Синтаксис монадических операций: do-блоки

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

```
do                          expr >>= (\pat -> do
  pat <- expr                                   morelines)
  morelines
```

- Например:

```
do                          action >>= process
  x <- action
  process x
```

```
do                          action
  x <- action
  return x
```

Содержание

Монада IO

- Область применения: ввод-вывод.
- Эффект: наличие побочных эффектов во время вычислений.

Пример

Вычислить количество строк в файле, имя которого задано в параметрах командной строки.

Решение

```
main = do
  fname <- head `liftM` getArgs
  content <- readFile fname
  print $ length $ lines content
```

Решение без do-блока

```
main = head <$> getArgs >>= readFile >>= print.length.lines
```

Приоритеты (:info)

```
infixl 4 <$>
infixl 1 >>=
infixr 9 .
```

Решение с явными приоритетами

```
main = ((head <$> getArgs) >>= readFile)
                           >>= (print.(length.lines))
```

Проверка условий в монаде

```
import System.Environment

main = do
  args <- getArgs
  if length args > 0
    then do
           content <- readFile (head args)
           print $ length $ lines content
    else return ()
```

- Конструкция if/then/else
- Вложенные do-блоки

Проверка условий в монаде: функция when

```
import System.Environment
import Control.Monad

main = do
  args <- getArgs
  when (length args > 0) $
            readFile (head args) >>= print.length.lines
```

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

Монады: выхода нет

```
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
```

- Поместить в монаду значение можно, а извлечь нельзя.
- Для конкретных монад можно написать соответствующую функцию (`fromJust`).

Содержание

Maybe: экземпляр класса типов Monad

```
instance Monad Maybe where
    return x = Just x

    Nothing >>= f = Nothing
    Just x  >>= f = f x
```

- Область применения: вычисления с возможной неудачей.
- Эффект: не нужно проверять предыдущую операцию на неудачу.

```
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

Задача о канатоходце

Условие

Канатоходец передвигается по канату с помощью длинного шеста. На левый и правый концы шеста могут садиться птицы. Если в какой-то момент разница в количестве птиц на концах шеста оказывается больше трёх, канатоходец падает.

- Вычисление — это учёт количества птиц на двух концах шеста.
- Ошибка возникает при отсутствии баланса в количестве птиц.

Пример

+2

-34

Примеры вычислений

Вычисление 1

- Начальное значение: `(0,0)`
- +1 слева
- +4 справа
- +2 слева
- -3 справа
- Результат: `(3,1)`

Вычисление 2

- Начальное значение: `(0,0)`
- +1 слева
- +10 справа
- -8 справа
- Результат: падение

Типы данных и функции

```haskell
type Birds = Int
type Pole = (Birds, Birds)

balanceLimit = 3

checkBalance :: Pole -> Bool
checkBalance (l, r) = abs (l - r) <= balanceLimit
```

Типы данных и функции: учёт падений

```
setPole :: Pole -> Maybe Pole
setPole p
    | checkBalance p = Just p
    | otherwise = Nothing

landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left, right) = setPole (left + n, right)

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left, right) = setPole (left, right + n)
```

Примеры использования

```
ghci> landLeft 2 (0, 0)
Just (2,0)
ghci> landLeft 10 (0, 3)
Nothing
```

```
ghci> landRight 1 (0, 0) >>= landLeft 2
Just (2,1)
ghci> Nothing >>= landLeft 2
Nothing
ghci> setPole (0, 0) >>= landRight 2 >>= landLeft 3
Just (3,2)
ghci> setPole (0, 0) >>= landLeft 10 >>= landRight 4
Nothing
```

Примеры вычислений

Вычисление 1

- Начальное значение: (0,0)
- +1 слева
- +4 справа
- +2 слева
- -3 справа
- Результат: (3,1)

```
ex1 = setPole (0,0)
      >>= landLeft 1
      >>= landRight 4
      >>= landLeft 2
      >>= landRight (-3)
```

```
ghci> ex1
Just (3,1)
```

Вычисление 2

- Начальное значение: (0,0)
- +1 слева
- +10 справа
- -8 справа
- Результат: падение

```
ex2 = setPole (0,0)
      >>= landLeft 1
      >>= landRight 10
      >>= landRight (-8)
```

```
ghci> ex2
Nothing
```

Как это всё работает

```
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
```

```
setPole :: Pole -> Maybe Pole
landLeft :: Birds -> Pole -> Maybe Pole
landRight :: Birds -> Pole -> Maybe Pole
```

Как учитывается возможность падения?

- `setPole` при нарушении баланса помещает в монаду `Nothing`
- Определение `(>>=)` для `Maybe` гарантирует, что `Nothing` останется до конца вычислений.

Недостатки do-блоков

Решение с монадическим связыванием

```
ex1 = setPole (0,0)
      >>= landLeft 1
      >>= landRight 4
      >>= landLeft 2
      >>= landRight (-3)
```

Решение с использованием do-блока

```
ex1' = do
  p1 <- landLeft 1 (0,0)
  p2 <- landRight 4 p1
  p3 <- landLeft 2 p2
  landRight (-3) p3
```

Решение без использования монадических операций

```
ex1'' =
  case landLeft 1 (0,0) of
     Nothing -> Nothing
     Just p1 -> case landRight 4 p1 of
            Nothing -> Nothing
            Just p2 -> case landLeft 2 p2 of
                   Nothing -> Nothing
                   Just p3 -> landRight (-3) p3
```

```
ghci> ex1
Just (3,1)
ghci> ex1'
Just (3,1)
ghci> ex1''
Just (3,1)
```

Задача о канатоходце: случай неизвестного числа ходов

Команда и её обработка

```
data Side = SLeft | SRight
data Command = Cmd Side Birds

processCmd :: Pole -> Command -> Maybe Pole
processCmd p (Cmd SLeft n) = landLeft n p
processCmd p (Cmd SRight n) = landRight n p
```

Свёртка списка команд

```
ex1''' = foldM processCmd (0,0)
                [Cmd SLeft 1, Cmd SRight 4,
                 Cmd SLeft 2, Cmd SRight (-3)]
```

```
foldM :: (Foldable t, Monad m) =>
         (b -> a -> m b) -> b -> t a -> m b
```

```
ghci> ex1'''
Just (3,1)
```

Задача о поиске

Условие

Имеются три функции, позволяющие определять номер мобильного телефона по имени человека, название мобильного оператора по номеру телефона, адрес мобильного оператора по его названию. Все функции могут возвращать Nothing при неудачном поиске. Написать функцию, определяющую адрес мобильного оператора по имени человека.

Решение задачи: типы

```
type PersonName = String
type CompanyName = String
type Phone = String
type Address = String

mobilePhone :: PersonName -> Maybe Phone
mobileOper :: Phone -> Maybe CompanyName
address :: CompanyName -> Maybe Address

addrByName :: PersonName -> Maybe Address
```

Решение задачи

Версия с do

```
addrByName :: PersonName -> Maybe Address
addrByName name = do
    phone <- mobilePhone name
    operator <- mobileOper phone
    address operator
```

Версия с >>=

```
addrByName :: PersonName -> Maybe Address
addrByName name = mobilePhone name >>= mobileOper >>= address
```

Вариация задачи

Задача

Написать функцию, определяющую номер телефона и название мобильного оператора по имени человека.

Решение с do

```
infoByName :: PersonName -> Maybe (Phone, Address)
infoByName name = do
    phone <- mobilePhone name
    operator <- mobileOper phone
    return (phone, operator)
```

- Можно ли эту задачу решить средствами аппликативных функторов?
- Как решать с помощью >>=?

Решение с помощью >>=

```
infoByName name =
  mobilePhone name >>=
    \phone -> mobileOper phone >>= \op -> return (phone, op)
```

Ещё одна задача о поиске

В ассоциативном списке с ключами “title”, “author” и “publisher” хранится информация о некоторой книге. Сформировать из этих данных значение следующего типа:

```
type Title = String
type Author = String
type Publisher = String
data Book = Book Title Author Publisher
```

Следует учесть, что список может не содержать всех необходимых полей, в этом случае результатом должно быть Nothing.

Решение с do

```
makeBook :: [(String, String)] -> Maybe Book
makeBook alist = do
  title <- lookup "title" alist
  author <- lookup "author" alist
  publisher <- lookup "publisher" alist
  return (Book title author publisher)
```

- Можно ли эту задачу решить средствами аппликативных функторов?

Три коротких решения

```
makeBook alist = Book <$> lookup "title" alist
                      <*> lookup "author" alist
                      <*> lookup "publisher" alist
```

```
makeBook alist = Book `liftM` lookup "title" alist
                        `ap` lookup "author" alist
                        `ap` lookup "publisher" alist
```

```
makeBook alist = Book `liftM3` (lookup "title" alist)
                               (lookup "author" alist)
                               (lookup "publisher" alist)
```