CS314. Функциональное программирование

Лекция 12. Монады

В. Н. Брагилевский

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

22 октября 2016 г.

Вычислительные контексты и их использование

Примеры контекстов (* -> *)

- `Maybe`
- `Either a`
- `[]`
- `IO`
- `(->) r`

Использование контекстов

- Functor
- Applicative
- ???

Содержание

Содержание

Проблема

- Как ввести строку, а затем вывести её на консоль?
- 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)

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.

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

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

Монады и моноиды

Классы Alternative и MonadPlus

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

class Alternative m, Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Функция guard (Control.Monad)

```
guard :: Alternative f => Bool -> f ()
guard True = pure ()
guard False = empty
```

Примеры использования функции guard

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
ghci> guard (5 > 2) >> return "good" :: [String]
["good"]
ghci> guard (1 > 2) >> return "good" :: [String]
[]
```

Синтаксис для монадических операций: do-блоки

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

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

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

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

- Блоки do можно использовать с любой монадой!

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 ()
```

- Чем функция when отличается от guard?

```
guard :: Alternative f => Bool -> f ()
```

Содержание

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x  >>= f = f x

instance MonadPlus Maybe where
    mzero = Nothing
    Nothing `mplus` ys = ys
    xs `mplus` ys = xs
```

- 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.

Типы данных и функции: без учёта падения

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

```haskell
landLeft :: Birds -> Pole -> Pole
landLeft n (left, right) = (left + n, right)

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

Типы данных и функции: с учётом падения

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

```haskell
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left, right)
    | abs ((left + n) - right) <= 3 = Just (left + n, right)
    | otherwise = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left, right)
    | abs (left - (right + n)) <= 3 = Just (left, right + n)
    | otherwise = Nothing
```

Пример 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> return (0, 0) >>= landRight 2 >>= landLeft 2
                                        >>= landRight 2
Just (2,4)
ghci> return (0, 0) >>= landLeft 1 >>= landRight 4
                      >>= landLeft (-1) >>= landRight (-2)
Nothing
```

Принудительное падение: банан на канате

```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

```
ghci> return (0, 0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
ghci> return (0, 0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

Пример

Решение в монаде Maybe

Решение с явным использованием монадического связывания

```
example :: Pole -> Maybe Pole
example p = landLeft 1 p >>= landRight 4 >>= landLeft 2
                                         >>= landRight (-3)
```

Решение с использованием do-блока

```
example :: Pole -> Maybe Pole
example p = do
   first <- landLeft 1 p
   second <- landRight 4 first
   third <- landLeft 2 second
   landRight (-3) third
```

Решение без использования монадических операций

```
example :: Pole -> Maybe Pole
example p = case landLeft 1 p of
     Nothing -> Nothing
     Just pole1 -> case landRight 4 pole1 of
            Nothing -> Nothing
            Just pole2 -> case landLeft 2 pole2 of
                   Nothing -> Nothing
                   Just pole3 -> landLeft (-3) pole3
```

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)
```

Содержание

Список как монада

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)

instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

- Недетерминированные вычисления — функция (возвращающая список) применяется к каждому элементу исходного списка.

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Три нотации для списков

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

```
ghci> [(n,ch) | n <- [1,2], ch <- ['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Три нотации: guard

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```

```haskell
withSevensOnly :: [Int]
withSevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    return x
```

```
ghci> [x | x <- [1..50], '7' `elem` show x]
[7,17,27,37,47]
```

Задача о перемещении коня

Условие

Определить, может ли конь перейти из одной позиции в другую за заданное число ходов?

```
type KnightPos = (Int, Int)

moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
    (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
               ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
    return (c',r')
```

Упрощение задачи

Задача о коне

Определить, может ли конь перейти из одной позиции в другую за три хода?

```
in3 start = do
   first <- moveKnight start
   second <- moveKnight first
   moveKnight second
```

```
in3 start = moveKnight start >>= moveKnight >>= moveKnight
```

- Здесь гнездо из трёх циклов!

Решение задачи о коне

Упрощённая задача

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

Исходная задача

```
inMany :: Int -> KnightPos -> [KnightPos]
inMany n st = foldr (<=<) return (replicate n moveKnight) st

canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn n start end = end `elem` inMany n start
```

- Здесь гнездо из n циклов!

Как понимать сложное выражение?

```
inMany :: Int -> KnightPos -> [KnightPos]
inMany n st = foldr (<=<) return (replicate n moveKnight) st
```

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
return :: a -> m a
replicate :: Int -> a -> [a]
moveKnight :: KnightPos -> [KnightPos]
```