CS314. Функциональное программирование

Лекция 13. Класс Traversable и специальные монады

В. Н. Брагилевский

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

29 октября 2016 г.

Содержание

Содержание

Класс Traversable (Data.Traversable)

```
class (Functor t, Foldable t) => Traversable t where
  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequence :: Monad m => t (m a) -> m (t a)
  ...
```

Простые примеры

```
ghci> sequenceA [Just 1, Just 2, Just 3]
Just [1,2,3]
ghci> sequenceA [Just 1, Nothing, Just 3]
Nothing
ghci> sequenceA [print 1, print 2]
1
2
[(),()]
```

Класс Traversable (Data.Traversable)

```
class (Functor t, Foldable t) => Traversable t where
  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequence :: Monad m => t (m a) -> m (t a)
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
```

```
traverse f = sequenceA . fmap f
mapM = traverse
```

Простые примеры

```
ghci> let f = \a -> if odd a then Just a else Nothing
ghci> traverse f [1..10]
Nothing
ghci> traverse f [1,3..10]
Just [1,3,5,7,9]
```

Вспомогательные функции

```
for :: (Traversable t, Applicative f) =>
                t a -> (a -> f b) -> f (t b)
forM :: (Monad m, Traversable t) =>
                t a -> (a -> m b) -> m (t b)
sequence_ :: (Monad m, Foldable t) => t (m a) -> m ()
mapM_ :: (Monad m, Foldable t) => (a -> m b) -> t a -> m ()
forM_ :: (Monad m, Foldable t) => t a -> (a -> m b) -> m ()
```

Простые примеры

```
ghci> sequence_ [print 1, print 2]
1
2
ghci> mapM_ print [1,2]
1
2
```

Пример: поиск в ассоциативном списке

В заданном ассоциативном списке хранится информация о некоторой книге. Сформировать из этих данных список, содержащий название книги (ключ “title”), имя автора (ключ “author”) и название издательства (ключ “publisher”). Если информации в исходном списке недостаточно, результатом должно быть Nothing.

Исходные данные

```
book :: [(String, String)]
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Решение

```
info :: Maybe [String]
info = traverse (flip lookup book)
                ["title", "author", "publisher"]
```

Пример: дерево как Functor, Foldable и Traversable

```
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)

instance Functor Tree where
  fmap f Empty = Empty
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Node l a r) = Node (fmap f l) (f a) (fmap f r)

instance Foldable Tree where
  foldr f b Empty = b
  foldr f b (Leaf a) = f a b
  foldr f b (Node l a r) = foldr f (f a (foldr f b r)) l

instance Traversable Tree where
   traverse f Empty = pure Empty
   traverse f (Leaf x) = Leaf <$> f x
   traverse f (Node l k r) =
                  Node <$> traverse f l <*> f k <*> traverse f r
```

Использование реализованных экземпляров

```
tree :: Tree String
tree = Node (Leaf "4") "5" (Node Empty "10" (Leaf "7"))

tree2 :: Maybe (Tree Integer)
tree2 = traverse readMaybe tree
```

- Если что-то не преобразуется, то Applicative обеспечит возврат Nothing

```
sumTree = fold <$> fmap Sum <$> tree2
elems = fold <$> fmap (:[]) <$> tree2
printTree = sequence_ $ mapM_ print <$> tree2
```

- Каковы типы и значения sumTree, elems и printTree?

Содержание

Задача

Вычислить НОД двух заданных чисел, пользуясь алгоритмом Евклида и сообщая о значениях на промежуточных шагах.

```
gcd' :: Int -> Int -> Int
gcd' a 0 = a
gcd' a b = gcd' b (a `mod` b)
```

Решение с использованием Control.Monad.Writer

```
gcd'' :: Int -> Int -> Writer [(Int, Int)] Int
gcd'' a 0 = tell [(a,0)] >> return a
gcd'' a b = tell [(a,b)] >> gcd'' b (a `mod` b)
```

```
ghci> gcd' 102 876
6
ghci> runWriter $ gcd'' 102 876
(6,[(102,876),(876,102),(102,60),(60,42),(42,18),(18,6),(6,0)])
```

Монада Writer (упрощённо)

Экземпляр класса Monad

```
newtype Writer w a = Writer { runWriter :: (a, w) }

instance Monoid w => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f = let (Writer (y, v1)) = f x
                           in Writer (y, v `mappend` v1)
```

Вспомогательные функции для работы с монадой Writer

```
tell :: w -> m ()
writer :: (a, w) -> m a
runWriter :: Writer w a -> (a, w)
execWriter :: Writer w a -> w
```

- Область применения: журнализация вычислений.
- Эффект: не нужно заботиться о накоплении записей в журнале.

Подсчёт числа шагов в алгоритме Евклида

```
import Control.Monad.Writer
import Data.Monoid

gcd''' :: Int -> Int -> Writer (Sum Int) Int
gcd''' a 0 = tell (Sum 1) >> return a
gcd''' a b = tell (Sum 1) >> gcd''' b (a `mod` b)
```

```
ghci> getSum $ execWriter $ gcd''' 102 876
7
```

Содержание

Монада Reader

Пример вычисления в монаде Reader

```
comp :: Reader String Int
comp = ask >>= return.length
```

Использование

```
ghci> runReader comp "hello"
5
ghci> runReader comp "world!"
6
```

Монада Reader (упрощённо)

Экземпляр класса Monad

```
newtype Reader r a = R { runReader :: r -> a }

instance Monad (Reader r) where
    return a = R $ \_ -> a
    m >>= k = R $ \r -> runReader (k (runReader m r)) r
```

Вспомогательные функции для работы с монадой Reader

```
ask :: m r
asks :: (r -> a) -> m a
runReader :: Reader r a -> r -> a
```

- Область применения: вычисления в некотором окружении.
- Эффект: не нужно заботиться о передаче параметров.

Пример использования монады Reader

```
toString :: Reader Integer String
toString = do
    n <- ask
    return $ "Число " ++ show n

adder :: Integer -> Reader Integer Integer
adder a = do
    n <- ask
    return $ a + n

multiplier :: Integer -> Reader Integer Integer
multiplier a = do
    n <- ask
    return $ a * n
```

Пример использования монады Reader

```haskell
doubler :: Reader Integer Integer
doubler = multiplier 2

doAll :: Reader Integer String
doAll = do
  s <- toString
  n1 <- adder 10
  n2 <- doubler
  ns <- mapM multiplier [2..5]
  return $ s ++ ": " ++ show [n1, n2] ++ "; " ++ show ns
```

Пример использования монады Reader

```
main = do
    e <- (read . head) `fmap` getArgs
    putStrLn $ runReader doAll e
```

```
$ ./reader 2
Число 2: [12,4]; [4,6,8,10]
$ ./reader 3
Число 3: [13,6]; [6,9,12,15]
$ ./reader 100
Число 100: [110,200]; [200,300,400,500]
```

Пример использования монады Reader

```
main = getArgs >>= mapM_ (putStrLn . runReader doAll . read)
```

```
$ ./reader 1 2 3
Число 1: [11,2]; [2,3,4,5]
Число 2: [12,4]; [4,6,8,10]
Число 3: [13,6]; [6,9,12,15]
$ ./reader 1 2 3 4
Число 1: [11,2]; [2,3,4,5]
Число 2: [12,4]; [4,6,8,10]
Число 3: [13,6]; [6,9,12,15]
Число 4: [14,8]; [8,12,16,20]
```

Пример: передача конфигурационной информации

```
data Config = ...

subwork :: Reader Config ()
subwork = do
  cfg <- ask
  ...

work :: Reader Config ()
work = do
  ...
  subwork
  ...

main = getArgs >>= loadConfig >>= print . runReader work
```

Содержание

Монада State

- Область применения: вычисления с сохранением и изменением состояния.

Задача

Реализовать операции со стеком: push и pop. Состояние в данном случае — это стек со всем содержимым.

```
type Stack = [Int]

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

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

Пример использования стека

```
stackManip :: Stack -> (Int, Stack)
stackManip stack =
    let
        ((), newStack1) = push 3 stack
        (a , newStack2) = pop newStack1
    in pop newStack2
```

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

- Состояние между вызовами операций передаётся вручную.

Использование монады State

```haskell
import Control.Monad.State

push :: Int -> State Stack ()
push x = do
    xs <- get
    put (x:xs)

pop :: State Stack Int
pop = do
    (x:xs) <- get
    put xs
    return x
```

Использование монады State

```
stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

Использование монады State

```
stackStuff :: State Stack ()
stackStuff = do
    a <- pop
    if a == 5
       then push 5
       else do
          push 3
          push 8
```

```
ghci> runState stackStuff [9,0,2,1,0]
((),[8,3,0,2,1,0])
```

Монада State (упрощённо)

Экземляр класса Monad

```
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
```

Вспомогательные функции для работы с монадой State

```
get :: m s
put :: s -> m ()
state :: (s -> (a, s)) -> m a
runState :: State s a -> s -> (a, s)
execState :: State s a -> s -> s
evalState :: State s a -> s -> a
```

Вычисление выражения в обратной польской нотации

```
ghci> evalRPN "4 19 2 * +"
42
```

```haskell
type Stack = [Int]

push :: Int -> State Stack ()
push x = state (\xs -> ((), x:xs))

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

evalRPN xs = head $ execState (traverse step $ words xs) []
  where
    step "+" = processTops (+)
    step "*" = processTops (*)
    step  n  = push (read n)
    processTops op = op <$> pop <*> pop >>= push
```

Случайность и монада State

Пример с подбрасыванием монет из шестой лекции

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in (firstCoin, secondCoin, thirdCoin)
```

Случайность и монада State

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

```
randomSt :: (RandomGen g, Random a) => State g a
randomSt = state random
```

Решение с do-блоком

```
threeCoins :: State StdGen (Bool, Bool, Bool)
threeCoins = do
  a <- randomSt
  b <- randomSt
  c <- randomSt
  return (a, b, c)
```

Решение в аппликативном стиле

```
threeCoins =  (,,) <$> randomSt <*> randomSt <*> randomSt
```

```
ghci> newStdGen >>= print . evalState threeCoins
(True,True,False)
```

Монады Reader, Writer, State

Тип

- Reader
- Writer
- State

Класс типов

- MonadReader (ask, asks)
- MonadWriter (writer, tell)
- MonadState (state, put, get)

«Функции-пускатели»

- runReader
- runWriter, execWriter
- runState, evalState, execState

Дан текстовый файл, в каждой строке которого записано комплексное число (мнимая и вещественная части, разделённые пробелами). Напечатать все числа, модуль которых не превосходит заданное число. Имя файла и максимум модуля должны задаваться параметрами командной строки.

Решение

```
import System.Environment

absComplex :: String -> Double
absComplex xs = sqrt (re*re + im*im)
  where
    [re, im] = map read $ words xs

main = do
  [fname, max] <- getArgs
  text <- readFile fname
  let m = read max
  putStrLn $ unlines $ filter ((<=m).absComplex) $ lines text
```