CS314. Функциональное программирование

Лекция 12. Специальные монады

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

27 октября 2017 г.

Содержание

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

```haskell
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

```haskell
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

```
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

Содержание

Вычисление чисел Фибоначчи

```haskell
fib' :: Int -> Integer
fib' 0 = 0
fib' 1 = 1
fib' n = fib' (n-1) + fib' (n-2)
```

- Долгие вычисления.
- Большой расход памяти.

```haskell
fib'' :: Int -> Integer
fib'' n = fst $ fib2 n
  where
    fib2 0 = (0, 1)
    fib2 n =
       let (a, b) = fib2 (n-1)
       in (b, a+b)
```

- Большой расход памяти.

Монада ST (state-transformer, state-thread)

```
data ST s a
```

- s не обозначает конкретный тип состояния, это просто метка, позволяющая различать вычисления с разными состояниями (вычислительные потоки с состоянием) и запрещающая их смешивать;
- a — это тип результата вычислений.

```
runST :: (forall s. ST s a) -> a
```

Изменяемые переменные (модуль Data.STRef)

```
data STRef s a
```

```
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
modifySTRef' :: STRef s a -> (a -> a) -> ST s ()
```

Эффективное вычисление чисел Фибоначчи

```haskell
fibST :: Int -> ST s Integer
fibST n = do
  a <- newSTRef 0
  b <- newSTRef 1
  replicateM_ n $ do
      x <- readSTRef a
      y <- readSTRef b
      writeSTRef a y
      writeSTRef b $! x + y
  readSTRef a
```

- Две ссылки a и b.
- Повторяемое n раз действие (тело цикла).
- Строгое вычисление x+y (операция $!).
- Извлечение результата вычисления из переменной a.

```haskell
fib :: Int -> Integer
fib n = runST (fibST n)
```

- Чистая функция!

Примеры

```
ghci> fib' 25
75025
(0.29 secs, 85928376 bytes)
ghci> fib'' 55
139583862445
(0.01 secs, 2062704 bytes)
ghci> fib 55
139583862445
(0.00 secs, 0 bytes)
ghci> fib 155
1105603071560908172376327542123 45
(0.00 secs, 1058472 bytes)
```

Изменяемые массивы в монаде ST

```
newListArray ::
        (MArray a e m, Ix i) => (i, i) -> [e] -> m (a i e)
readArray ::
        (MArray a e m, Ix i) => a i e -> i -> m e
writeArray ::
        (MArray a e m, Ix i) => a i e -> i -> e -> m ()
```

```
runSTArray ::
     Ix i => (forall s . ST s (STArray s i e)) -> Array i e
```

Пример вычисления (обмен значениями элементов)

```
import Data.STRef
import Control.Monad.ST

import Data.Array
import Data.Array.ST
import Data.Array.MArray

swapElems :: Ix i => i -> i -> STArray s i e -> ST s ()
swapElems i j arr = do
    vi <- readArray arr i
    vj <- readArray arr j

    writeArray arr i vj
    writeArray arr j vi
```

Пример использования

```
test :: Int -> Int -> [a] -> [a]
test i j xs = elems $ runSTArray $ do
    arr <- newListArray (0, length xs - 1) xs
    swapElems i j arr
    return arr
```

```
ghci> test 1 3 [0,1,2,3,4]
[0,3,2,1,4]
ghci> test 4 2 [0,1,2,3,4]
[0,1,4,3,2]
```

Более сложные примеры

- Сортировки методами пузырька, вставки, поиска.
- Быстрая сортировка Хоара (настоящая, inplace).