Монады State и Cont, утилиты

Монада State

Итак, общая идея состоит в том, что если к данной нужной нам функции вида `f :: a -> b`, а особенно к цепочке вложенных функций (т.е. композиции)

```
f1 >.> f2 >.> f3 >.> ... >.> fn
```

или в более традиционной записи на Haskell:

```
fn . ... . f3 . f2 . f1
```

где

```
f1 :: a0 -> a1
f2 :: a1 -> a2
...
fn :: a(n-1) -> an
```

мы хотели бы добавить возможность чтение/записи некоторого глобального состояния, т.е. глобальной переменной в терминах традиционного программирования.

Глобальных переменных в Haskell нет, поэтому мы можем добавить к каждой функции ещё одну «лишнюю» переменную, которая будет переносить внутрь функции информацию о таком состоянии и это значение (возможно, изменённое функцией) должно быть добавлено к возвращаемому результату. Таким образом, тип наших функций будет изменён так:

```
f1 :: (a0,s) -> (a1,s)
f2 :: (a1,s) -> (a2,s)
...
fn :: (a(n-1),s) -> (an,s)
```

или даже так:

```
f1 :: a0 -> s -> (a1,s)
f2 :: a1 -> s -> (a2,s)
...
fn :: a(n-1) -> s -> (an,s)
```

но в таком случае, мы должны связывать цепочку функций не обычной композицией (прямой или обратной), а более сложной функцией, которая сама будет распаковывать пары результатов и передавать два аргумента в следующую функцию.

Вот по этому пути и пошли создатели монады `State`. Фактически, монадические функции, которые работают в монаде `State`, возвращают пару из типа `(b,s)`, и принимают два значения, принадлежащие типам `a` и `s`. Но делают это не напрямую, а скрытно, с помощью специального API так, что у программиста создаётся иллюзия, что его функция принимает на вход значение типа `a`, возвращает значение типа `b` (точнее, в случае монад типа `m b`), а обращение к состоянию напоминает обращение к базе данных внутри тела задаваемой функции (будет ниже пример !!!)

Далее я начну описывать формальности и особенности реализации монады `State`, а также некоторые простейшие примеры её использования и описания монадических функций в различном стиле.

Монада State

Мотивация

Нам необходимы вычисления, которые умеют поддерживать состояния.

Стратегия связывания:

Связывание пропускает параметр состояния через последовательность связанных функций, так что одно и то же значение состояния никогда не используется дважды, создавая иллюзию обновления на месте.

Полезно для построения вычислений из последовательностей операций, которые требуют общего состояния.

Чисто функциональный язык не может обновлять значения на месте, потому что тогда нарушается «ссылочная прозрачность». Обычная идиома для моделирования таких вычислений с состоянием состоит в том, чтобы «пропустить» параметр состояния через последовательность функций:

```
data MyType = MT Int Bool Char Int deriving Show

makeRandomValue :: StdGen -> (MyType, StdGen)
makeRandomValue g = let (n,g1) = randomR (1,100) g
                        (b,g2) = random g1
                        (c,g3) = randomR ('a','z') g2
                        (m,g4) = randomR (-n,n) g3
                    in (MT n b c m, g4)
```

Этот подход работает, но такой код может быть подвержен ошибкам, неряшлив и сложен в обслуживании. Монада `State` скрывает перемещение параметра состояния внутри операции связывания, одновременно делая код легче для написания, легче для чтения и для модификации.

Чтобы пример заработал необходимо добавить импорт System.Random и вызов:

```
import System.Random

data MyType = MT Int Bool Char Int deriving Show
...
test = makeRandomValue (mkStdGen 100)
```

что даст

```
> (MT 22 False 'g' 11,1478585369 1872071452)
```

wiki.haskell: All_About_Monads. The_State_monad

Тип данных и воплощение класса

Тип `State s a` — это тип вычисления с состоянием, которое манипулирует состоянием типа `s` и имеет результат типа `a`. Иными словами, есть некоторая путаница в терминах:

хотя мы задаём тип `State s a`, под состоянием мы чаще всего будем иметь в виду то, что ниже обозначает `s`: и как переменная типа, и как значение.

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

Воплощение класса **`Monad`** будет таким:

```
instance Monad (State s) where
    return a         = State $ \s -> (a,s)
    (State ff) >>= g = State $ \s ->
         let (y,s') = ff s
         in runState (g y) s'
```

Как и модуль `Control.Monad.Writer`, модуль `Control.Monad.State` не экспортирует свой конструктор значений. Если вы хотите взять вычисление с состоянием и обернуть его в `State`, используйте функцию `state`, которая делает то же самое, что делал бы конструктор `State`.

Цель использования функции **`return`** состоит в том, чтобы взять значение (и ничего больше не вычислять в этом случае) и создать минимальный контекст (но это всё будет называться «вычисление») с парой, которая всегда содержит это значение в качестве своего результата. Поэтому мы просто создаём анонимную функцию `\s -> (a, s)` и оборачиваем её тэгом `State`. Значение `a` представляет результат «вычисления с состоянием», а состояние `s` представляет собой собственно «состояние», которое тут тоже не меняется. Его мы получим внешним образом (а изначально при запуске `runState`) и потом будем передавать дальше по цепочке вычисления.

Отметим, что функция, определена от двух аргументов: `a` и `s`. При этом аргумент `a` «лежит на поверхности», а второй аргумент `s` глубоко спрятан конструктором `State`. Таким образом, формально тип **`return`**

```
return :: a -> State s a
```

становится как-бы от одного аргумента…

Это определение показывает, что применение **`return`** (в монаде `State`) к аргументу `a` создаёт функцию преобразования состояния, которая на самом деле не изменяет состояние вообще, но которая «возвращает» значение `a`, переданное в самом начале. И важно, что функция с тэгом `State` в правой части уравнения выше и есть «монадическое значение» или «трансформер состояния», как его иногда называют в этом случае. Если для монад **`Maybe`** и **`List`**, которые мы изучили раньше, монадическим значением было «нечто» (как правило числа, строки или иные обычные объекты) *в контейнере*, то *теперь это «нечто» стало функцией*.

Таким образом, если ранее монадическими функциями были, например, функции такого типа `f :: a -> Maybe b`, то в случае монады `State` станут такими

```
f :: a -> State s a
f x = State (\st -> (some func body) -> (x', st'))
```

где `x' st'` — результаты вычисления (значение и состояние) функцией `f`.

Рассмотрим теперь определение связывания в такой форме:

```
mv >>= g = State (\st ->
               let (State ff) = mv
                   (y, st')   = ff st
                   (State gg) = g y
               in gg st')
```

и разберём его по шагам:

1. во второй строке мы фактически распаковываем монадическое значение `mv` и полученной распакованной функции присваиваем имя `ff`;
2. в третье строке запускаем эту функцию со значением состояния `ff st` и результат передаём в пару `(y,st')`, где `y` — новое значение, а `st'` — новое состояние;
3. в четвёртой строке мы распаковываем частичное применение монадической функции `g y` (выглядит как запуск функции `g` с аргументом `y`) и полученной распакованной функции присваиваем имя `gg`;
4. в последней строке осуществляем запуск этой функции со значением состояния, которое мы получили на втором шаге `gg st'`.

7.3.1 Формальное описание State

Это, немного другое определение из начала раздела, несколько короче, хотя по сути выполняется то же самое, используя распаковщик `runState`:

```
(State ff) >>= g = State $ \s ->
   let (y,s') = ff s
   in runState (g y) s'
```

Пример работы «в лоб»

Для понимания работы монады `State` «изнутри» реализуем простой пример удвоения числа с передачей лога. Только теперь логом будет состояние.

Будем вновь пытаться распаковать-запаковать передаваемое значение. Для распаковки заведём такую знакомую функцию:

```
import Control.Monad.State

extract :: State s a -> s -> (a,s)
extract (State f) = f
```

К сожалению мы не можем сделать именно таким образом, так как конструктор `State` не экспортируется. Но мы можем сделать ещё проще, так как есть специально для этого деструктор `runState`:

```
import Control.Monad.State
extract = runState
```

Далее мы определяем функцию `dblSt`, которая по сути удваивает входящий аргумент `x`. Вторым аргументом, через λ-выражение мы передаём значение состояния `s`. И потом оборачиваем всё это конструктором `State` (с помощью функции `state`):

```
dblSt :: Int -> State String Int
dblSt x =
  state $ \s -> (2*x,s)
```

Просто для понимания, напомним, что, например, вот такая функция:

```
f x = \y -> (x,y)
```

является функцией двух аргументов:

```
Prelude> :t f
f :: a -> b -> (a, b)
```

Но «усложнив себе жизнь» оборачиванием

```
import Control.Monad.State
f :: a -> State b a
f x = state $ \y -> (x,y)
```

(ну или тип в общем случае):

```
Prelude Control.Monad.State> :t f
f :: MonadState s m => a -> m a
```

мы глубоко прячем второй аргумент `y`, и он будет доступен только после специальной распаковки.

Вернёмся к нашему упражнению. Чтобы воспользоваться нашей монадической функцией `dblSt`, мы в соответствии с её типом:

```
dblSt :: Int -> State String Int
```

должны сначала применить её к целому значению:

```
*Main> :t (dblSt 3)
(dblSt 3) :: State String Int
```

Потом сделать распаковку `extract $ dblSt`

```
*Main> :t extract (dblSt 3)
extract (dblSt 3) :: String -> (Int, String)
```

Затем применить ко втором аргументу и получить пару вида (результат, состояние):

```
*Main> :t extract (dblSt 3) "hello"
extract (dblSt 3) "hello" :: (Int, String)
```

Ну, или всё вместе:

```
test str = (extract $ dblSt 3) str
testhi = test "hi"
```

и в GHCi:

```
*Main> testhi
(6,"hi")
```

Утилиты класса `MonadState`

Показанное ниже определение использует классы многопараметрических типов и `funDeps` (функциональные зависимости), которые не являются стандартными для Haskell-2010. Нет необходимости полностью понимать эти детали, чтобы использовать монаду `State`.

Как и в случае почти со всеми предыдущими монадами, имеется дополнительный функционал для работы с монадами.

Модуль `Control.Monad.State` определяет класс типов под названием `MonadState`, в котором помимо прочих присутствуют две весьма полезные функции: `get` и `put`.

Для монады `State` функция `get` реализована вот так:

```
get = state $ \s -> (s,s)
```

Она просто получает (внешним образом) текущее состояние и представляет его в качестве результата.

Функция `put` принимает некоторое состояние в виде явного аргумента и создаёт функцию с состоянием, которая заменяет им текущее состояние (Липовача, с.433-434):

```
put s = state $ \_ -> ((),s)
```

Другими словами, функция `put`, фактически игнорируя входной параметр `(\_)` (т.е. игнорируя полученное внешним образом состояние), вернёт значение `()`, но помимо этого, переданный ей параметр-состояние поместит в «условное хранилище». Иначе говоря, изменит это состояние на переданное в качестве параметра значение.

И ещё три функции для запуска вычислений внутри монады `State`:

```
runState :: State s a -> s -> (a, s)
runState (State f) init_st = f init_st

evalState :: State s a -> s -> a
evalState mv init_st = fst (runState mv init_st)

execState :: State s a -> s -> s
execState mv init_st = snd (runState mv init_st)
```

Основной является функция `runState`, а функции `evalState` и `execState` являются производными от неё, возвращая конечное значение-результат или только последнее значение состояния соответственно. Функция `runState` возвращает пару `(a,s)`, т.е. и результат, и состояние. По своей сути эта функция `runState` (и две производные функции соответственно) распаковывает, т.е. снимает тэг `State` с монадического значения — функции-трансформатора состояния `f`, а потом запускает её с начальным значение состояния `init_st`. И тогда, если к этому моменту у нас уже создана некая монадическая функция, например, `f'`, то `f = f' a`. Применение монадической функции к значению `a` и является по сути монадическим значением. Другими словами, здесь в описании мы используем «очень хитрое» частичное применение.

Использовали определения из

http://mvanier.livejournal.com/5406.html

http://mvanier.livejournal.com/5846.html

Примеры

Из лекции-3:

```
import Control.Monad.State

fact' :: Integer -> State Integer Integer
-- тип состояния - Integer, тип результата - тоже Integer
fact' 0 = do
    acc <- get -- получаем накопленный результат
    return acc -- возвращаем его
fact' n = do
    acc <- get -- получаем аккумулятор
    put (acc * n) -- домножаем его на n и сохраняем
    fact' (n - 1) -- продолжаем вычисление факториала

-- fact :: Integer -> Integer
fact n = fst $ runState (fact' n) 1
-- начальное значение состояния = 1
```

где термином *начальное состояние*, *накопленный результат* и *аккумулятор* означаем одно и то же.

Вот версия с предыдущим упражнением, которое мы подробно разобрали выше, только теперь с утилитами:

```
dblSt2 :: Int -> State String Int
dblSt2 x = do
  s <- get
  put s
  return (2*x)

dblSt3 :: Int -> State String Int
dblSt3 x = return (2*x)

test2  = (runState $ dblSt2 3) "hi"
test3  = runState (dblSt2 3) "hi" -- the same
```

с таким отображением в GHCi:

```
*Main> test2
(6,"hi")
*Main> test3
(6,"hi")
```

Рассмотрим ещё ряд небольших примеров. Вот пара функций для работы со стеком `pop` и `push` и их произвольное использование.

```
import Control.Monad.State
```

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

testpop = (runState $ pop) $ [10..20]

push :: Int -> State [Int] Int

push x = do
   xs <- get
   put (x:xs)
   return x

somecalc x = do
    y <- pop
    z <- pop
    t <- pop
    let w = z*t + y - x
    push w
    return w

testrun = (runState $ somecalc 33)  $ [10..20]
testev  = (evalState $ somecalc 33) $ [10..20]
testex  = (execState $ somecalc 33) $ [10..20]
```

вот с таким ясным выводом в GHCi:

```
> testpop
(10,[11,12,13,14,15,16,17,18,19,20])
> testrun
(142,[142,13,14,15,16,17,18,19,20])
> testev
142
> testex
[142,13,14,15,16,17,18,19,20]
```

Начиная с ghc версии 8.65 (8.63??) пример не работает, указывая на возможность провала паттерна

```
(x:xs) <- get
```

если стек пуст и невозможность обработки ошибки (**fail**) в этом случае. Экспериментально было найдено 2 возможных решения: либо определяем pop так:

```
pop = do
    lst <- get
    put $ tail lst
    return $ head lst
```

либо доопределяем

```
import Control.Monad.State
import qualified Control.Monad.Fail as Fail
import Data.Functor.Identity

instance Fail.MonadFail Data.Functor.Identity.Identity where
  fail = error
```

и далее оставляем всё как прежде.

Решение по мотивам статей:

Control.Monad.Fail

Monad transformers: Implementation of a stack machine with MaybeT (State Stack) (see 2nd answ.)

No instance for (Control.Monad.Fail.MonadFail Data.Functor.Identity.Identity) (since 2019, ghc 8.6.3 ??)

https://hackage.haskell.org/package/base-4.11.0.0/docs/Data-Functor-Identity.html#t:Identity

Кроме того, в текущих версиях (9.2.8) пакет `mtl`, содержащий модуль `Control.Monad.State` стал «скрытым» и требует явного указания при запуске ghc и ghci:

```
ghci stack.hs -package mtl
```

И пример из мотивационной части:

```
import System.Random

data MyType = MT Int Bool Char Int deriving Show

makeRandomValue :: StdGen -> (MyType, StdGen)
makeRandomValue g = let (n,g1) = randomR (1,100) g
                        (b,g2) = random g1
                        (c,g3) = randomR ('a','z') g2
                        (m,g4) = randomR (-n,n) g3
                    in (MT n b c m, g4)

test = makeRandomValue (mkStdGen 100)
```

мы теперь можем переписать следующим образом — заголовок, кроме импортирования модуля `Control.Monad.State` остаётся прежним:

```
import System.Random
import Control.Monad.State

data MyType = MT Int Bool Char Int deriving Show
```

Далее, сделаем обёртку над функцией **`randomR`** так, чтобы она стала монадической функцией-трансформером состояния, а само состояние соответствовало использованию рандомного генератора и было спрятано (с доступом только через функции `get-put`):

```
randomRS :: (Random b) => (b, b) -> State StdGen b
randomRS (x1,x2) = do
  g <- get
```

```
  let (y,g') = randomR (x1,x2) g
  put g'
  return y
```

Теперь, вся последовательность обработки стала значительно проще и более похожа на аналогичные комбинации в обычных императивных языках программирования.

```
makeRandomValue2 :: State StdGen MyType
makeRandomValue2 = do
  n <- randomRS (1,100)
  b <- randomRS (False,True)
  c <- randomRS ('a','z')
  d <- randomRS (-n,n)
  return (MT n b c d)

test2 = (runState makeRandomValue2) (mkStdGen 100)
```

При запуске в GHCi функции ведут себя аналогичным образом:

```
*Main> test
(MT 22 False 'g' 11,1478585369 1872071452)
*Main> test2
(MT 22 False 'g' 11,1478585369 1872071452)
```

Полезные ссылки по теме

Monday Morning Haskell: State Monad

О монаде IO с точки зрения State

Ранее мы уже обсуждали систему ввода-вывода Haskell, связь этой системы с монадой `IO`, а также различные приёмы и утилиты по работе с файлами, стандартными устройствами (STDIN-STDOUT). С практической точки зрения этого вполне достаточно для программирования на Haskell. Однако, вы можете встретить в Интернете утверждения, что система ввода-вывода (несмотря на то, что большей частью сделана в виде примитивов, глубоко спрятанных в системе) также основана на понятии техники из мира монад, и самое интересное, на использовании чистых функций. Как это может быть? Ведь мы всегда считали, что именно монада `IO` это «проклятое место» обитания «нечистых функций» («действий» или «акций» в нашей терминологии).

Обсудим монаду `IO` с точки зрения ранее изученной монады `State`. Что бы было, если система ввода-вывода состояла из чистых функций в обычном нашем понимании?

Предположим, мы хотим сделать реализацию утилиты `getchar`. Каков тип мог бы у неё быть, если бы следовали по пути языков программирования «Си и Ко»:

```
getchar :: Char
```

Используем её для следующего определения, предполагая получить два символа со стандартного устройства ввода:

```
get2chars = [getchar, getchar]
```

Так как компилятор Haskell рассматривает все функции как чистые, он может избежать «ненужных» вызовов `getchar` и дважды использовать одно возвращаемое значение:

```
get2chars = let x = getchar in [x, x]
```

Это не то, что мы хотели. И как можно было бы решить эту проблему? Например, можно было бы ввести фиктивный параметр, чтобы сделать вызовы отличающимися друг от друга:

```
getchar :: Int -> Char
get2chars = [getchar 1, getchar 2]
```

Это решит проблему последовательности, но не решит главную проблему: `getchar 1` всё равно не станет чистой функцией, так как результат будет зависеть от того, что реально нажмёт пользователь!

Вместо параметра **`Int`** мы могли бы попробовать использовать `Time`, реальное изменяемое время

```
getchar :: Time -> Char
get2chars (t1,t2) = [getchar t1, getchar t2]
```

Это уже лучше, но всё равно мы можем получить проблему в многопользовательских окружениях, когда несколько человек одновременно вызывают эту «функцию» с разными нажатиями, или это будут делать несколько процессов.

Решение было найдено как раз в стиле изученной выше монады `State`. Предполагается, что внутри конструируемой монады **`IO`** каждый вызов монадической функции, каждая её утилита, будут получать некий параметр особого типа `RealWorld` и вместе со своим результатом будет также передавать изменённый параметр дальше, в точности таким же образом, как это делали монадические функции в монаде `State` с передаваемым состоянием какого-то типа `s`.

Утрированно (это не Haskell, ошибочная запись, но понятная :) это означает…

```
getChar :: RealWorld -> (Char, RealWorld)

main :: RealWorld -> ((), RealWorld)
main world0 = let (a, world1) = getChar world0
                  (b, world2) = getChar world1
              in ((), world2)

action1 >> action2 = action
  where
    action world0 = let (a, world1) = action1 world0
                        (b, world2) = action2 world1
                    in (b, world2)

(>>=) :: IO a -> (a -> IO b) -> IO b
(action >>= reaction) world0 =
   let (a, world1) = action world0
       (b, world2) = reaction a world1
   in (b, world2)
```

```
return :: a -> IO a
return a world0  =  (a, world0)
```

Формально же это означает как раз

```
instance Monad IO where
    return a         = IO $ \world0 -> (a,world0)
```

и примерно так

```
action >>= reaction = IO (\world0 ->
                let (IO ff) = action
                    (a, world1)   = ff world0
                    (IO gg) = reaction a
                in gg world1)
```

где вместо типа `State RealWorld a` мы можем писать что-то типа **`IO a`** (так как `RealWorld` будет уже неизменным типом)

В реальности, ситуация немного не такая и сложнее. Но такое представление даёт понимание «внутренней механики» монады **`IO`**.

wiki.haskell: IO inside

Where is the realWorld# defined?

What are the definitions for »= and return for the IO monad?

What is the IO type in Haskell

Primitive Haskell

A Problem With I/O

Unraveling the mystery of the IO monad

IO in Haskell

А.Холомьёв. Монада IO

The IO monad

Bartosz Milewski. 3.a The Tao of Monad. Simple I/O

Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. A History of Haskell: Being Lazy With Class. 7. Monads and input/output

IO monad realized in 1965

Монада Continuation

Общее представление о продолжениях

Как говорит википедия:

> Продолжение (англ. continuation) представляет состояние программы в определённый момент, которое может быть сохранено и использовано для перехода в это состояние. Продолжения содержат всю информацию, чтобы продолжить выполнения программы с определённой точки. Состояние глобальных

переменных обычно не сохраняется, однако для функциональных языков это несущественно (выборочное сохранение/восстановление значений глобальных объектов в Scheme достигается отдельным механизмом `dynamic-wind`). Продолжения похожи на `goto` Бейсика или `setjmp/longjmp` Си, так как также позволяют перейти в любое место программы. Но продолжения, в отличие от `goto`, позволяют перейти только в участок программы с определённым состоянием, которое должно быть сохранено заранее, в то время, как `goto` позволяет перейти в участок программы с неинициализированными переменными.

Scheme был первым промышленным языком, в котором реализованы полноценные продолжения.

Более наглядно, продолжение — это «вся оставшаяся часть программы от данной точки», или «функция, которая никогда не возвращает управление в точку своего вызова». В процессе изучения функционального программирования многие испытывают трудности с пониманием сущности продолжений. Традиционное объяснение этого понятия сводится к «расширению (усложнению) понятия сопрограммы», но в педагогическом смысле такое объяснение считается бесполезным. Причина трудности понимания заключается в том, что продолжения фактически представляют собой альтернативное обоснование понятия «поведения» («вызова» в самом широком понимании), т.е. иную семантическую модель, и в этом смысле начальный переход от «обычного» функционального программирования к программированию с интенсивным использованием продолжений можно сравнить с начальным переходом от императивного программирования к функциональному.

Центральное понятие: `callcc` — это функция высшего порядка, позволяющая абстрагировать динамический контекст имеющейся функции в виде другой функции, которая и называется «продолжением».

Небольшое введение на примерах

Обычные функции:

```
square :: Int -> Int
square x = x*x

incr :: Int -> Int
incr x = x+1

func :: Int -> Int
func x = square (incr x)
```

А теперь в стиле продолжений:

```
square_cps :: Int -> (Int -> r) -> r
square_cps x k = k (x*x)

incr_cps :: Int -> (Int -> r) -> r
incr_cps x k = k (x+1)
```

```
func_cps :: Int -> (Int -> r) -> r
func_cps x k = incr_cps x $ \inc ->
               square_cps inc $ \sq ->
               k sq
```

Теперь функции кроме непосредственно аргументов принимают на вход функцию, которая будет применена к результату. Это и есть продолжение. С помощью продолжений мы можем соединять функции, что и происходит в `func_cps`. Сначала выполняется `incr_cps`, и её результат «попадает» в продолжение (`\inc -> ...`), потом вызывается `square_cps`, чей результат передаётся продолжению (`\sq -> ...`), который, наконец, отдаётся самому внешнему продолжению `k`.

Продолжения здесь имеют тип (`Int -> r`) так как не обязательно, что продолжение вернёт `Int`. Например, чтобы вывести результат на консоль, мы можем передать `print` в качестве продолжения:

```
main = func_cps 5 print
```

см. Продолжения в Haskell

Теперь другой пример.

```
add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x^2

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

После модификации для возврата приостановленных вычислений, `pythagoras` выглядит так:

```
add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pythagoras_cps :: Int -> Int -> ((Int -> r) -> r)
pythagoras_cps x y = \k ->
  square_cps x $ \x_squared ->
  square_cps y $ \y_squared ->
  add_cps x_squared y_squared $ k
```

Схема работы примера `pythagoras_cps`:

1. возводим в квадрат `x` и передаём результат в «продолжение» (`\x_squared -> ...`);
2. возводим в квадрат `y` и передаём результат в «продолжение» (`\y_squared -> ...`);
3. складываем `x_squared` and `y_squared` и передаём результат в «продолжение» самого верхнего уровня `k`.

И мы можем запусть эту программу в GHCi:

```
*Main> pythagoras_cps 3 4 print
25
```

см. wikibooks: Continuation passing style

Мотивация

Расчёты, которые могут быть прерваны и возобновлены.

Стратегия связывания:

Привязка функции к монадическому значению создаёт новое продолжение, которое использует функцию как продолжение монадического вычисления.

Монада продолжения представляет вычисления в «стиле прохождения продолжения» (eng. CPS). В стиле с передачей продолжения результат не возвращается, а передаётся другой функции, получаемой в качестве параметра (продолжение). Вычисления строятся из последовательностей вложенных продолжений, оканчивающихся окончательным продолжением (часто идентификатором), которое даёт конечный результат. Поскольку продолжения — это функции, которые только ещё подлежат дальнейшим вычислениям, манипулирование функциями продолжения может давать сложные (и запутанные) эффекты, такие как прерывание вычислений в середине, прерывание части вычислений, перезапуск вычислений и перемешивание выполнения вычислений.

Прежде чем использовать монаду Continuation, убедитесь, что у вас есть чёткое понимание стиля прохождения продолжения, и что продолжения представляют собой лучшее решение вашей конкретной проблемы проектирования. Многие алгоритмы, которые требуют продолжения в других языках, не требуют их в Haskell из-за ленивой семантики Haskell. Злоупотребление монадой Continuation может привести к коду, который невозможно понять и поддерживать.

Control.Monad.Cont

All_About_Monads. The_Continuation_monad

Продолжение — это состояние программы в определённый момент, которое мы потом можем использовать, чтобы вернуться в то состояние.

С помощью продолжений можно реализовать обработку исключений, подобие `goto` и множество других вещей напоминающих императивные конструкции.

Также, используя продолжения можно улучшить производительность программы, убирая ненужные «обёртывания» и сопоставления с образцом.

Продолжения в Haskell

The Mother of all Monads (first publ. 2008)

The Mother of all Monads (2012)

wikipedia: Continuation passing style

Паттерны использования «call with current continuation»

Продолжение всемирной паутины

Making Haskell programs faster and smaller

Тип данных и воплощение класса

```
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }
-- r is the final result type of the whole computation

instance Monad (Cont r) where
  return a       = Cont $ \k -> k a
-- i.e. return a = -> k a
  (Cont c) >>= f =
    Cont $ \k -> c (\a -> runCont (f a) k)
```

Здесь `return a` — это `Cont`, который применяет полученное внешним образом (a изначально при запуске `runCont`) продолжение к значению `a`.

Оператор `>>=` можно описать и так:

```
m >>= f  =
  Cont $ \k -> runCont m (\a -> runCont (f a) k)
```

Здесь `Cont` который запускает (`runCont` просто «распаковывает» `Cont`, освобождая функцию) `m` с продолжением `(\a -> runCont (f a) k)`, которая может быть получит результат вычисления, и передаст его в `a` (а может и не получит, ведь функция может проигнорировать продолжение). Потом, `a` будет применено с функцией `f`, чтобы получить другой `Cont`, который, в свою очередь, будет запущен с самым внешним продолжением `k`.

Продолжения в Haskell

Чтобы нам самим стало понятнее, перепишем указанные ранее программы с использованием введённых монадических конструкций:

см. раздел Небольшое введение на примерах

```
import Control.Monad
import Control.Monad.Cont

square_Cont :: Int -> Cont r Int
square_Cont x = return (x*x)

incr_Cont :: Int -> Cont r Int
incr_Cont x = return (x+1)

func_Cont :: Int -> Cont r Int
func_Cont x = do inc <- incr_Cont x
                 sq  <- square_Cont inc
                 return sq

test1 = runCont (func_Cont 5) print
test2 = runCont (incr_Cont >=> square_Cont $ 5) print
test3 = runCont ((return 5) >>= incr_Cont >>= square_Cont) print
test4 = runCont (incr_Cont 5 >>= square_Cont) print
```

Это чистые функции (если не подмешан код из IO)!

Полезные утилиты работы с состояниями

Класс Control.Monad.Cont содержит утилиту `callCC`. Рассмотрим её детальнее, вот тип `callCC`:

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
```

(на самом деле, в классе определена более общая сигнатура):

```
callCC :: ((a -> m b) -> m a) -> m a
```

А воплощение этого метода для `Cont r`:

```
instance MonadCont (Cont r) where
 callCC f =
   Cont $ \k -> runCont (f (\a -> Cont $ \_ -> k a)) k
```

Начинается она по обычному, с оборачивания функции в стиле продолжений в `Cont`. Потом `(f (\a -> Cont $ \_ -> k a))` запускается с продолжением `k`. `(\a -> Cont $ \_ -> k a)` — это функция, которая берёт что-то и возвращает `Cont`, игнорирующий своё продолжение и использующий `k` взамен.

Рассмотрим такую модификацию примера, который мы уже выше разбирали:

```
func_Cond :: Int -> Cont r Int
func_Cond x =
    callCC $ \exit -> do
      if (x>0)
        then
          do
            inc <- incr_Cont x
            sq  <- square_Cont inc
            return sq
        else exit 0 -- "Bad x"
```

Тогда запуск с положительными числами пройдёт как обычно, а с отрицательными мы фактически вызовем исключение:

```
testCond = runCont (func_Cond (-5)) print
```

т.е., в GHCi:

```
*Main> testCond
0
```

Если заменить возвращаемое значение на текстовое, то можно сделать более информационный вывод (изменение последних двух строк):

```
            return $ show sq
        else exit $ "Bad: " ++ (show x)
```

Используем одну из полезных монадических утилит, перепишем в более компактный код:

```
func_Cond2 x =
    callCC $ \exit -> do
      when (x<=0) (exit $ "Bad: " ++ (show x))
      inc <- incr_Cont x
      sq  <- square_Cont inc
      return $ show sq

testCond2 = runCont (func_Cond2 (-5)) print
```

Тогда на прогонке получим:

```
*Main> testCond2
"Bad: -5"
```

Ещё один интересный пример можно глянуть тут:

Example 2: Using callCC

.

```
whatsYourName :: String -> String
whatsYourName name =
  (`runCont` id) $ do                        -- 1
    response <- callCC $ \exit -> do         -- 2
      validateName name exit                 -- 3
      return $ "Welcome, " ++ name ++ "!"    -- 4
    return response                          -- 5

validateName name exit = do
  when (null name) (exit "You forgot to tell me your name!")
```

Есть ещё утилита, `getCC'`…

https://habr.com/ru/post/127040/

Полезные общие утилиты для работы с монадами

Мы уже рассматривали на прошлой лекции удобные функции для работы с монадами из класса **`MonadPlus`**. В этот раз рассмотрим ещё ряд функций-утилит, более общих.

Монадические вычисления с условием

Для выполнения монадических вычислений с условием предусмотрены две функции. Функция **`when`** принимает логический аргумент и монадическое вычисление с типом `()` и выполняет вычисление только тогда, когда логический аргумент имеет значение **`True`**. Функция **`unless`** делает то же самое, за исключением того, что выполняет вычисления, если только логический аргумент не равен **`True`**.

Фактически, это аналог усечённого **`if`** из мира императивного программирования.

```
when :: (Monad m) => Bool -> m () -> m ()
when p s = if p then s else return ()
```

```
unless :: (Monad m) => Bool -> m () -> m ()
unless p s = when (not p) s
```

Пример работы:

```
> when (1 == 1) (print "OK")
"OK"
```

Как видим, в последовательности монадических вычислений вполне можно использовать и полную версию **if..then..else..** в том числе и без присваивания.

Лифтинг

Функция **liftM** `f m` позволяет немонадической функции `f` опрерировать на контексте монады (с монадическим значением) `m`. Является аналогом функции `liftA` для аппликативных функторов, функции `fmap` для функторов и даже **map** для списков.

Вот как реализована функция **liftM**:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= (\x -> return (f x))
```

Или с использованием нотации **do**:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = do
  x <- m
  return (f x)
```

(буквы `m` в первой и в следующих строках несут разный смысл: в первой это конструктор типа (вроде **Maybe**), в последующих — монадическое значение)

Примеры:

```
ghci> liftM sin (Just 0)
Just 0.0
ghci> liftM (replicate 10) ['a']
["aaaaaaaaaa"]

ghci> liftM (*3) (Just 8)
Just 24
ghci> fmap (*3) (Just 8)
Just 24
ghci> runWriter $ liftM not $ Writer (True,"hello")
(False,"hello")
ghci> runWriter $ fmap not $ Writer (True,"hello")
(False,"hello")
ghci> runState (liftM (+100) pop) [1,2,3,4]
(101,[2,3,4])
ghci> runState (fmap (+100) pop) [1,2,3,4]
(101,[2,3,4])
```

где функция pop была определена, когда мы тренировались со стеком.

Есть ещё варианты **`liftM2`**, **`liftM3`**, **`liftM4`**,… с многими аргументами.

И функция **`ap`**… аналогичная `<*>` …

join