Конкретные монады и их приложения

Монада Maybe

Мотивация

Монада **Maybe** воплощает стратегию объединения цепочки вычислений, каждое из которых может возвращать нечто существенное или **Nothing**, при этом заканчивая цепочку досрочно, если какой-либо шаг приводит к результату **Nothing**. Это полезно, когда вычисление влечёт за собой последовательность шагов, которые зависят друг от друга, и в которых некоторые шаги могут не вернуть значение.

wiki.haskell: The Maybe monad

Тип данных и воплощение класса

```
data Maybe a = Nothing | Just a
  deriving ( Eq, Ord, Read, Show )

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

Ещё раз об интуитивных представлениях

На прошлой лекции мы основное внимание уделили «монадической композиции монадических функций». Посмотрим как это могло бы работать в этом случае.

Попробуем так:

```
f >=> g = f .> extract .> g
  where
    extract (Just t) = t
```

К сожалению, мы получим проблему, если результатом работы функции `f` будет значение **Nothing**. Мы могли бы передать какое-нибудь дефолтное значение, но это нарушит всю стратегию в целом. Дело в том, что вторая сторона стратегии вычисления в монаде **Maybe** как раз и заключалась в том, чтобы при получении значения **Nothing** передавать его по цепочке без дальнейших вычислений. Поэтому, простая схема

```
f .> extract .> g
```

нас не устроит. Нас устроит что-то типа такого

```
(.>) = flip (.)
extract (Just t) = t
infixl 9 .>

f >=> g = \x ->
  if (f x) == Nothing
    then Nothing
    else (f .> extract .> g) x
```

ну или так

```
f >=> g = \x ->
  if f x == Nothing
    then Nothing
    else g $ extract $ f x
  where
    extract (Just t) = t
```

В реальности, конечно, существует определение

```
f >=> g = \x -> (f x >>= g)
```

на уровне класса (задано в пакете Control.Monad) и определения воплощения оператора `>>=` для полиморфного типа данных **`Maybe a`**.

Полезные примеры

```
f :: Int -> Maybe Int
f x = if x `mod` 2 == 0 then Nothing else Just (2*x)

g :: Int -> Maybe Int
g x = if x `mod` 3 == 0 then Nothing else Just (3*x)

h :: Int -> Maybe Int
h x = if x `mod` 5 == 0 then Nothing else Just (5*x)
```

Создадим теперь новую функцию `k`, которая будет монадической композицией этих трёх функций, с тем же самым типом:

```
k :: Int -> Maybe Int
```

Эта функция умножает входное число на 30, если оно не делится целочисленно на 2, 3 и 5 (а если делится, функция вернёт **`Nothing`**).

В монадическом стиле мы можем это оформить тремя способами:

```
k = f >=> g >=> h

k x = f x >>= g >>= h

k x = do y <- f x
         z <- g y
         h z
```

Но можно было бы задать эту композицию и без монадических идей, правда, выглядело бы это не слишком хорошо

```
k x = case f x of
        Nothing -> Nothing
        Just y  -> case g y of
                     Nothing -> Nothing
                     Just z  -> h z
```

Но в любом случае, это была бы *чистая функция*, что с монадическим вычислениями, что без таковых! И для нас тут важно, что польза от использования функциональности монады **`Maybe`** скорее эстетическая, т.е. состоит в том, что мы уменьшаем объём кода, делаем его опрятнее.

Ещё Одно Руководство по Монадам (часть 4: Монада **`Maybe`** и монада списка)

MonadPlus

На прошлой лекции мы уже с вами разбирали, что сами по себе монады должны быть моноидами относительно композиции Клейсли **`>=>`** и **`return`**, и должны выполнятся три монадных закона, которые и означают указанную моноидальность. Но кроме этого, для некоторых монад вводится ещё дополнительная моноидальная функциональность. Это делается с помощью специального класса **`MonadPlus`**.

Класс типов **`MonadPlus`** предназначен для монад, которые также могут вести себя как моноиды (т.е. полугруппы (есть ассоциативность) с нейтральным элементом) относительно специфичных операций. Вот его определение:

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

Функция **`mzero`** является аналогом функции `mempty` из класса типов `Monoid`, а функция **`mplus`** соответствует функции `mappend`.

Сам класс **`MonadPlus`** нам будет интересен ради некоторых очень интересных и полезных функций-утилит, которые возможны для некоторых из рассматриваемых монад. Сами функции рассмотрим чуть дальше, а ниже — некоторые ссылки по теме и большое количество различных обсуждений на тему концептуального пересеченя с идеей моноидов в монадах.

wikibooks: MonadPlus

wiki.haskell: MonadPlus

Специально по поводу воплощения для **`IO`** (ранее 20-го года, похоже это было не так):

MonadPlus definition for Haskell IO

MonadPlus IO isn't a monoid

В интернетах разгорелась нешуточная дискуссия о математических и классовых связях монад, монад-с-плюсом и моноидов:

reddit: Monoid vs MonadPlus

Why MonadPlus and not Monad + Monoid?

Monoid vs MonadPlus [duplicate]

Must mplus always be associative? Haskell wiki vs. Oleg Kiselyov

Jonathan M. Sterling. Unifying Monoids and Monads with Polymorphic Kinds

И возможно, в грядущем Хаскеле будет их объединение или реформа:

wiki.haskell: MonadPlus reform proposal

Disambiguate MonadPlus

AMP-corollary: `Alternative`/**`MonadPlus`**

Воплощение класса **`MonadPlus`** для монады **`Maybe`**

```
instance MonadPlus Maybe where
    mzero             = Nothing
    Nothing `mplus` x = x
    x `mplus` _       = x
```

Функция `guard`

Функция **`guard`** определена в контексте **`MonadPlus`** следующим образом:

```
guard :: (MonadPlus m) => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Функция **`guard`** принимает значение типа **`Bool`**. Если это значение равно **`True`**, функция **`guard`** берёт пустой кортеж `()` и помещает его в минимальный контекст, который по-прежнему является успешным. Если значение типа **`Bool`** равно **`False`**, функция **`guard`** создаёт монадическое значение с неудачей в вычислениях. Вот эта функция в действии для монады **`Maybe`** `()`:

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

> «…Ключ к пониманию этой функции — закон для монад с нулём и плюсом (т.е. реализующих класс MonadPlus), который гласит:

```
mzero >>= f == mzero
```

(где **`mzero`** в зависимости от монады может быть пустым списком `[]` или значением **`Nothing`** — В.В.)

> Таким образом, размещение функции **`guard`** в последовательности монадических операций, если значение аргумента будет **`False`**, заставит последующие операции быть **`mzero`**.…»
>
> Соответственно, если будет получено значение **`True`**, то **`guard`** вернёт значение `()` в монадической обёртке, однако, в типичном использовании мы просто перейдём к следующей строке с операцией.

Таким образом, выражение с **`guard`** становится монадической функцией, которая проверяет булево условие, и возвращает либо **`Just`** `()` (в случае с монадой **`Maybe`**, или другие обёртки с другими монадами), переходя на следующую строку в **`do`**-выражении; либо возвращается **`Nothing`** (или другое значение **`mzero`** для другой монады), и дальнейшие вычисления проваливаются (сворачиваются).

Рассмотрим пример. Выше была определена такая функция:

```
f :: Int -> Maybe Int
f x = if x `mod` 2 == 0 then Nothing else Just (2*x)
```

Тогда мы можем сделать функцию `ff`:

```
ff x = do
         y <- f x
         guard (y `elem` [6,10,22])
         return(y)
```

которая проверяет выполнение дополнительного условия, и в случае удачи возвращает обёрнутое значение `y`. Типы у функций будут следующие:

```
f  :: Int -> Maybe Int
ff :: Int -> Maybe Int
```

При запуске в GHCi получим:

```
*Main> ff 3
Just 6
*Main> f 3
Just 6
*Main> f 4
Nothing
*Main> ff 4
Nothing
```

Ещё раз к функции **`guard`** мы вернёмся в главе про монаду списков.

Монада Either

Ранее определённая монада **`Maybe`** используется, чтобы добавить к значениям контекст возможной неудачи. Вспомним, значениями монадических функций, работающих с **`Maybe`**, может быть **`Just`** `<smth>` либо **`Nothing`**. Как бы это ни было полезно, всё, что нам известно, когда у нас есть значение **`Nothing`** — это состоявшийся факт неудачи: туда не втиснуть больше информации, сообщающей нам, что именно произошло.

Мотивация

Тип **`Either`** `e a` позволяет нам включать контекст (описание) возможной неудачи в наши значения. С его помощью можно прикреплять значения к неудаче, чтобы они могли описать, что именно пошло не так, либо предоставить другую полезную информацию относительно ошибки. Значение типа **`Either`** `e a` может быть либо значением **`Right`** (правильный ответ и успех) либо значением **`Left`** (неудача). Сам тип мы уже рассматривали в 8-й лекции, там же были определены и некоторые полезные функции для работы с этим типом.

Тип данных и воплощение класса

```
data  Either a b  =  Left a | Right b
  deriving ( Eq, Ord, Read, Show )
```

Воплощение класса **Monad** для этого типа данных похоже на экземпляр для типа **Maybe**

```
instance Monad (Either e) where
   return x = Right x
   Right x >>= f = f x
   Left err >>= _ = Left err
   fail msg = Left msg
```

Data.Either

Функция **return** принимает значение и помещает его в минимальный контекст по умолчанию. Она оборачивает наше значение в конструктор **Right**, потому что мы используем его для представления успешных вычислений, т.е. там, где присутствует результат. Это очень похоже на определение метода **return** для типа **Maybe**.

Оператор >>= проверяет два возможных случая: **Left** и **Right**. В случае **Right** к значению внутри него применяется функция f, подобно случаю **Just**, где к его содержимому просто применяется функция. В случае ошибки сохраняется значение **Left** вместе с его содержимым, которое описывает неудачу.

Например, если мы хотим реализовать версию безопасного деления на 0 в целых числах (**Int**), мы могли бы написать что-то типа такого кода:

```
safe_divide :: Int -> Int -> Either String Int
safe_divide _ 0 = Left ”divide␣by␣zero”
safe_divide i j = Right (i `div` j)
```

и в ghci:

```
ghci> 36 `safe_divide` 6
Right 6
ghci> 1 `safe_divide` 0
Left ”divide␣by␣zero”
```

Yet Another Monad Tutorial (part 5: error-handling monads)

Но как говорит сам Майк, у нас тут есть хорошая новость, что «если вычисление проваливается, то мы можем получить информацию о причине», и есть плохая новость, что «эти конструкции сложны в использовании»:

```
-- f i j k = i + (j / k)
f :: Int -> Int -> Int -> Either String Int
f i j k =
  case j `safe_divide` k of
    Left msg -> Left msg
    Right r -> Right (i + r)
```

Так, если мы увеличиваем число возможных ошибочных ситуаций

```
safe_divide :: Int -> Int -> Either String Int
safe_divide _ 0 = Left ”divide␣by␣zero”
safe_divide i j | i `mod` j /= 0 = Left ”not␣divisible”
safe_divide i j = Right (i `div` j)
```

то вырастает сложность перебора для их отслеживания:

```
divide :: Int -> Int -> Either String Int
divide i j =
  case i `safe_divide` j of
      Left ”divide␣by␣zero” -> Left ”divide␣by␣zero”
      Left ”not␣divisible”  -> Right (i `div` j)
      Right k               -> Right k
```

Более хаскелевский способ:

```
data ArithmeticError = DivideByZero | NotDivisible
    -- could add more cases here
      deriving Show
```

соответственно изменим:

```
safe_divide :: Int -> Int -> Either ArithmeticError Int
safe_divide _ 0 = Left DivideByZero
safe_divide i j | i `mod` j /= 0 = Left NotDivisible
safe_divide i j = Right (i `div` j)

divide :: Int -> Int -> Either ArithmeticError Int
divide i j = case i `safe_divide` j of
               Left DivideByZero -> Left DivideByZero
               Left NotDivisible -> Right (i `div` j)
               Right k           -> Right k
```

На этом месте мы сделаем остановку. Тема обработки ошибок в Haskell чрезвычайно многоуровневая, путанная, см, напр.

8 ways to report errors in Haskell

mvanier.livejournal: Error-handling computations

10. Error Handling

Кроме того, эта тема постоянно находится в развитии, существуют параллельные и даже «перпендикулярные» в каком-то смысле проекты. Например, в модуле Control.Monad.Except можно встретить такое предупреждение на грани юмора:

> **Warning**
>
> Please do not confuse `ExceptT` and `throwError` with `Exception` / `SomeException` and **`catch`**, respectively. The latter are for exceptions built into GHC, by default, and are mostly used from within the **`IO`** `monad`. They do not interact with the “exceptions” in this package at all. This package allows you to define a new kind of exception control mechanism which does not necessarily need your code to be placed in the **`IO`** `monad`.
>
> In short, all “catching” mechanisms in this library will be unable to catch exceptions thrown by functions in the `Control.Exception` module, and vice-versa.

Немного об Except

В завершение темы обработки ошибок, рассмотрим пример использования нового базового модуля Control.Monad.Except, ——— (2021: уже есть).

Основной конструкцией в нём является монадный трансформер `ExceptT` и производная монада `Except`, в некотором смысле обёртка на монадой **`Either`**. Модуль содержит ряд утилит, которые позволяют создавать, выбрасывать и обрабатывать исключения в более-менее стандартном виде.

```
throwError :: e -> Except e a
runExcept  :: Except e a -> Either e a
```

Например

```
import Control.Monad.Except

type Err = String

safeDiv :: Int -> Int -> Except Err Int
safeDiv a 0 = throwError "Divide by zero"
safeDiv a b = return (a `div` b)

example :: Either Err Int
example = runExcept $ do
  x <- safeDiv 2 3
  y <- safeDiv 2 0
  return (x + y)
```

И на практике получим:

```
*Main> example
Left "Divide by zero"
```

Более-менее современные ссылки

Michael Snoyman. Exceptions Best Practices in Haskell.

Michael Snoyman. Safe exception handling

о `ExceptT`

Stephen Diehl. What I Wish I Knew When Learning Haskell: ExeptT

Control.Monad.Except

Control.Exception

stackoverflow: Adapting Error to Except

Control.Monad.Except not properly handling thrown exceptions

специфические:

lift Either to ExceptT automatically

Control.Exception.Safe, why do ExceptT and Either behave so differently?

Rethinking MonadError

Стандартный модуль обработки ошибок в GHC

Control.Exception

Вот отсюда, минимальный пример чтения файла с возможностью ошибки:

```
import Control.Exception
import System.IO

f = ”test.txt”

main = do
  str <- catch (readFile f)
          (\e -> do
            let err = show (e :: IOException)
            hPutStr stderr
              (”Warning:␣Couldn't␣open␣” ++ f ++ ”:␣” ++ err)
            return ””
          )
  print str
```

Уже устаревшие, но полезные ссылки:

Скандальная правда об обработке исключений в Haskell

8 ways to report errors in Haskell

Yet Another Monad Tutorial (part 6: more on error-handling monads)

Error Checking and Exceptions

Exceptions and monad transformers

Composing Exceptions with MonadError

Catching all exceptions

10. Error Handling

errors-1.0: Simplified error handling

Real World Haskell. Chapter 19. Error handling

Michael Snoyman. Exceptions in continuation-based monads

Общий вывод о работе с ошибками и исключениями.

В работе с ошибками и исключениями в Haskell следует придерживаться примерно следующего правила:

– по возможности используем **`Maybe`** (напр., при неудачном поиске) и **`Either`** (если нам нужна некоторая информация о неудаче);

– для сложных случаев в «чистых функциях» и монадах с ними используем `Control.Monad.Except` (т.е. `ExceptT`)

– для обработки и перехватывания ошибок в вводе-выводе и др. побочных ситауциях используем `Control.Exception`

Монада `Writer`

Мотивация

Монада `Writer` предусмотрена для значений, к которым присоединено другое значение, ведущее себя наподобие журнала. Монада `Writer` позволяет нам производить вычисления, в то же время обеспечивая слияние всех журнальных значений в одно, которое затем присоединяется к результату. Например, мы могли бы снабдить наши значения строками, которые объясняют, что происходит, возможно, для отладочных целей.

Например, пусть у нас есть функция

```
dbling :: Int -> Int
dbling x = 2 * x
```

Мы можем применить её несколько раз:

```
*Main> dbling . dbling $ 3
12
```

Но теперь мы решили, что хорошо бы иметь журнал того, что происходит с этой функцией:

```
dbling' :: Int -> (Int, String)
dbling' x = (2 * x, "we␣has␣dubbled␣␣" ++ (show x))
```

Однако, теперь система типов не даст нам возможность многократного применения `dbling'`:

```
*Main> (dbling' . dbling') 2

<interactive>:2:11: error:
    * Couldn't match expected type `a -> Int'
                  with actual type `(Int, String)'
```

Можно всё переписать вручную:

```
resVal = (val2, log1 ++ log2)
    where (val1, log1) = dbling' 8
          (val2, log2) = dbling' val1
```

Но, во-первых, это совсем не похоже на простую композицию, а во-вторых, придётся заново переписывать, если нам надо проводить вычисления с другими функциями или с большим их числом.

Тип данных и воплощение класса

Метка поля `runWriter` (её можно рассматривать как функцию-распаковщик) используется в определении типа, потому что она следует стилю определения монады, который явно представляет значения монады как вычисления. В этом стиле монадическое вычисление строится с использованием монадических операторов, а затем значение вычисления извлекается с использованием функции вида `run***`. (Один из примеров уже был

выше рассмотрен при знакомстве с монадой Except. Для дальнейшего знакомства с этим стилем методов в будущем изучим монаду State).

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

instance (Monoid w) => Monad (Writer w) where
  return a             = Writer (a,mempty)
  (Writer (a, w)) >>= f =
      let (a', w') = runWriter $ f a
        in Writer (a', w `mappend` w')
```

Монада `Writer` создаёт пару `(value,log)`, где тип для **`log`** должен быть моноидом. Функция **`return`** просто возвращает (создаёт) значение вместе с пустым логом. Биндинг `>>=` выполняет связанную функцию, используя в качестве входных данных текущее значение, и добавляет сообщение, полученное от монадической функции `f`, в существующий лог.

Основным типом журнала для обычного разработчика будет **`String`**, являющийся моноидом с функцией `(++)`. Но разумеется, можно использовать любой моноидальный подходящий тип. Например, суммировать значения.

Важной деталью, как уже указано выше, будет проведение вычислений с журналированием внутри запуска функции `runWriter` (хотя именно здесь это будет не очень обязательно, эта монада всё ещё доступна к прямым манипуляциям).

Полезные примеры

Теперь мы можем организовать композицию!

```
import Control.Monad.Writer

dblingWri :: Int -> Writer String Int
dblingWri x =
  writer (2*x, "we␣has␣dubbled␣␣" ++ (show x) ++ ";␣")

dbldbl x = return x >>= (dblingWri >=> dblingWri)

dbldbl' = dblingWri >=> dblingWri

dbldbl'' x =
  do x1 <- dblingWri x
     x2 <- dblingWri x1
     return x2
```

Отметим, по «историческим причинам», так как давно вместо монад используются монадные трансформеры, вместо конструктора `Writer` мы в коде примера используем функцию `writer`, выполняющую эту же роль.

Применение любой из этих функций даст такой вывод (кстати, `WriterT` как конструктор типов и значений, вновь напоминает о трансформерах, см. далее):

```
*Main> dbldbl'' 3
WriterT (Identity (12,"we␣has␣dubbled␣␣3;
␣we␣has␣dubbled␣␣6;␣"))
```

Тип ниже мы указали сами

```
dblingWri :: Int -> Writer String Int
```

А вот эти типы уже автоматически выведены:

```
dbldbl :: Int -> WriterT String Data.Functor.Identity.Identity Int

dbldbl' :: Int -> WriterT String Data.Functor.Identity.Identity Int

dbldbl'' ::
  Int -> WriterT String Data.Functor.Identity.Identity Int
```

А для того, чтобы избежать длинных строк в типах выше, можно сделать явный импорт

```
import Control.Monad.Writer
import Data.Functor.Identity
...
```

тогда типы буду выглядеть проще:

```
dblingWri :: Int -> Writer String Int
dbldbl :: Int -> WriterT String Identity Int
dbldbl' :: Int -> WriterT String Identity Int
dbldbl'' :: Int -> WriterT String Identity Int
dblingTel :: Int -> Writer String Int
dbll :: Int -> WriterT String Identity Int
```

Тут стоит отметить, что тип `WriterT String Identity Int` в современной хаскеловской «монадологии» и задаёт то, что мы понимаем как `Writer String Int` (фактически являясь его синонимом) с помощью трансформера `WriterT`. Но об этом позже.

Наконец, вот вычисления с журналированием внутри функции `runWriter`:

```
*Main> dbldbl 3
WriterT (Identity (12,"we␣has␣dubbled␣␣3;␣we␣has␣dubbled␣␣6;␣"))
*Main> runWriter $ dbldbl 3
(12,"we␣has␣dubbled␣␣3;␣we␣has␣dubbled␣␣6;␣")
*Main> fst $ runWriter $ dbldbl 3
12
*Main> snd $ runWriter $ dbldbl 3
"we␣has␣dubbled␣␣3;␣we␣has␣dubbled␣␣6;␣"
```

Утилиты класса `MonadWriter`

Показанные ниже определения используют класс многопараметрических типов и `funDeps` (функциональные зависимости), которые не являются стандартными для Haskell-2010. Но нам нет необходимости полностью понимать эти детали, чтобы использовать монаду `Writer`.

```
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
    pass   :: m (a, w -> w) -> m a
    listen :: m a -> m (a,w)
    tell   :: w -> m ()
```

```
instance (Monoid w) => MonadWriter w (Writer w) where
    pass   (Writer ((a,f),w)) = Writer (a, f w)
    listen (Writer (a,w))     = Writer ((a,w), w)
    tell   s                  = Writer ((),s)
```

Класс `MonadWriter` предоставляет ряд удобных функций для работы с монадой `Writer`. Самым простым и полезным является метод `tell`, который помещает одну или несколько записей в журнал (лог). Функция `listen` превращает входное значение `Writer` (возвращающий значение `a`, и производящий журналирование `w`) в изменённый Writer (производит значение `(a,w)` и также производит журналирование `w`). Это позволяет вычислениям «прослушивать» вывод журнала, генерированный монадой `Writer`.

Функция `pass` немного сложнее. Он преобразует `Writer`, (который производит значение `(a,f)`, и журналирует `w`) в `Writer` (который производит значение `a`, и журналирует обработанное значение `f w`). Таким образом, монадные вычисления внутри монады `Writer` получают дополнительную функциональность анализировать собственный журнал по ходу вычисления и преобразовывать его.

Примеры с утилитами

```
dblingTel :: Int -> Writer String Int
dblingTel x =
  do tell $ "we␣has␣dubbled␣␣" ++ (show x) ++ ";␣"
     return (2*x)

dbll = dblingTel >=> dblingTel
```

```
*Main> dbll 3
WriterT (Identity (12,"we␣has␣dubbled␣␣3;␣we␣has␣dubbled␣␣6;␣"))
*Main> runWriter $ dbll 3
(12,"we␣has␣dubbled␣␣3;␣we␣has␣dubbled␣␣6;␣")
*Main> fst $ runWriter $ dbll 3
12
*Main> snd $ runWriter $ dbll 3
"we␣has␣dubbled␣␣3;␣we␣has␣dubbled␣␣6;␣"
```

Важно!

Следует соблюдать осторожность при использовании списка в качестве моноида для `Writer`, так как может иметь место снижение производительности, связанное с операцией `mappend`, по мере увеличения выходных данных. В этом случае более подходящим выбором будут структуры данных, которые поддерживает более оптимальные операции добавления.

Кроме того!

Для реальных приложений не рекомендуется применять функционал монады `Write` (ситуация в чём-то аналогична простой, но неэффективной структуре списков), так как он совершенно необоснованно хранит весь лог в оперативной памяти. Для реальной работы рекомендуется, напр.:

fast-logger: A fast logging system

kazu-yamamoto/logger

Monday Morning Haskell: Reader and Writer Monads

Монада List

Мотивация

В лекции об аппликативных функторах мы обсуждали возможность функций, возвращающих список значений и их комбинацию:

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

Монада **`List`** представляет стратегию объединения цепочки недетерминированных вычислений, применяя операции ко всем возможным значениям на каждом шаге. Это полезно, когда вычисления должны иметь дело с неоднозначностью. В этом случае стратегия позволяет изучить все возможности, пока не будет разрешена неоднозначность.

Тип данных и воплощение класса

За основу взят тип списка и следующее определение

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

Функция **`return`** принимает значение и помещает его в минимальный контекст по умолчанию, который по-прежнему возвращает это значение. Другими словами, функция **`return`** создаёт список, который содержит только одно это значение в качестве своего результата. Это полезно, когда нам нужно просто обернуть обычное значение в список, чтобы оно могло взаимодействовать с недетерминированными значениями.

Суть операции `>>=` состоит в получении монадического значения (списка в нашем случае) и передаче его в функцию, которая принимает обычное значение и возвращает список значений (т.е., недетерминированный результат). Результат также является недетерминированным (общим списком), и он представляет все возможные результаты получения элементов из списка `xs` и передачи их функции `f` за счёт использования функции объединения списков **`concat`**.

Полезные примеры

Здесь анонимная функция применяется к каждому элементу `[3,4,5]`:

```
ghci> [3,4,5] >>= \x -> [x, -x]
[3,-3,4,-4,5,-5]
```

и мы получаем список списков `[[3,-3],[4,-4],[5,-5]]`. В итоге «сглаживаем список». И таким образом, мы применили (условно) недетерминированную функцию к недетерминированному значению.

Есть у списков и аналог «неудач» **Nothing** — это пустой список `[]`. Посмотрим, как он будет обрабатываться:

```
ghci> [] >>= \x -> [True, False, True]
[]
ghci> [1,2,3] >>= \x -> []
[]
```

Рассмотрим теперь композицию таких неоднозначных функций:

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

[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Чтобы лучше понять её работу, надо рассмотреть эту же запись со скобками (см. лекцию-13 о монадных законах и do-синтаксисе).

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

Эта же самая конструкция в do-нотации (по Липовачу):

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

Такая запись делает чуть более очевидным то, что образец `n` принимает каждое значение из списка `[1,2]`, а образец `ch` — каждое значение из списка `['a','b']`, при этом применить эти образцы (как мы обычно говорим, переменные) можно в любой последующей строке.

Воплощение класса `MonadPlus` для монады `List`

Поскольку списки являются моноидами, а также монадами, их можно сделать экземпляром этого класса типов:

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

Для списков функция **mzero** представляет недетерминированное вычисление, которое вообще не имеет результата, т.е. неуспешно окончившееся вычисление. Функция **mplus** сводит два недетерминированных значения в одно.

Функция `guard`

Раз у нас есть выполнение условий монады **MonadPlus** для списков, то мы можем задействовать функцию **guard**:

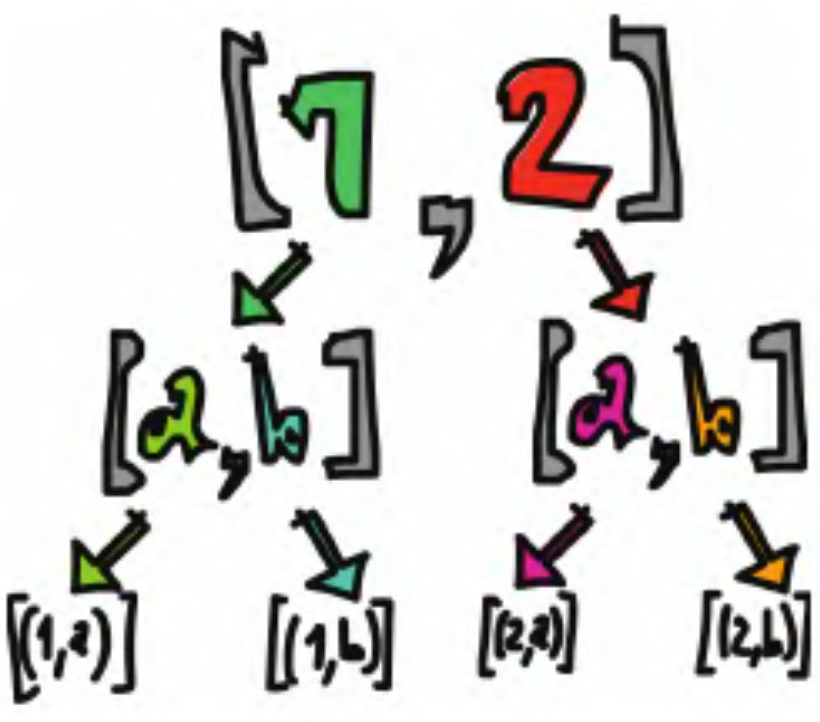

Рис. 1: Уровни обработки

```
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

В списковой монаде мы используем её для фильтрации недетерминированных вычислений:

```
> [1..50] >>= \x -> (guard ('7' `elem` show x) >> return x)

[7,17,27,37,47]
```

Вот этот пример, переписанный в нотации **do**:

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

> Если функция **guard** срабатывает успешно, результатом, находящимся в ней, будет пустой кортеж. Поэтому дальше мы используем операцию >>, чтобы игнорировать этот пустой кортеж и предоставить что-нибудь другое в качестве результата. Однако если функция **guard** не срабатывает успешно, функция **return** впоследствии тоже не сработает успешно, потому что передача пустого списка функции с помощью операции >>= всегда даёт в результате пустой список. Функция **guard** просто говорит: «Если это значение типа **Bool** равно **False**, верни неуспешное окончание вычислений прямо здесь. В противном случае создай успешное значение, которое содержит в себе значение-

пустышку ()». Всё, что она делает, — позволяет вычислению продолжиться. (*Липовача* с.398)

Пример. Рассмотрим чуть более содержательный пример с нахождением разложения числа на пару множителей.

```
solvDoz = do
  x <- [1..3]
  y <- [4..12]
  guard (x*y == 12)
  return (x,y)
```

при вычислении получим:

```
*Main> solvDoz
[(1,12),(2,6),(3,4)]
```

(по мотивам Monads in 15 minutes: Backtracking and Maybe).

Нотация do и генераторы списков

Для нашего понимания списков, пример выше с семёркой мог бы быть переписан в традиционном виде:

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

а второй пример так:

```
[(x,y) | x <- [1..3], y <- [4..12], x*y == 12]
```

И фильтрация в генераторе списков — это то же самое, что использование функции **guard**.

Таким образом, для списков у нас есть фактически два способа представления их обработки.

Кстати, предыдущий пример с картинкой (рис.1) мог бы быть переписан таким образом:

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

[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

На самом деле генераторы списков являются просто синтаксическим сахаром для использования списков как монад. В конечном счёте генераторы списков и списки, используемые в нотации **do**, переводятся в использование операции >>= для осуществления вычислений, которые обладают недетерминированностью.

Была даже идея (Philip Wadler. Comprehending Monads) использовать синтаксис генераторов списка для произвольных монад. Она не вошла в стандарт, но реализована в качестве специального расширения:

```
{-# LANGUAGE MonadComprehensions #-}

test = [ x + y | x <- Just 1, y <- Just 2 ]
```

```
*Main> test
Just 3
```

в традиционном виде это было бы так:

```
test = do
  x <- Just 1
  y <- Just 2
  return (x+y)
```

Philip Wadler. Comprehending Monads

6.2.8. Monad comprehensions

Monad comprehensions

Bringing Back Monad Comprehensions

24 Days of GHC Extensions: List Comprehensions

Более реальные примеры использования неоднозначности функций

Канонический пример использования монады **List** — анализ неоднозначных грамматик. В приведённых ниже фрагментах показан лишь небольшой пример разбора данных на шестнадцатеричные значения, десятичные значения и слова, содержащие только буквы. Обратите внимание, что шестнадцатеричные цифры перекрывают как десятичные, так и буквенные символы, что приводит к неоднозначной грамматике. Например, «dead» является допустимым шестнадцатеричным значением (57005 в десятичной системе) и словом, а «10» является десятичным значением 10 и шестнадцатеричным значением 16.

```
import Control.Monad
import Data.Char

-- we can parse three different types of terms
data Parsed = Digit Integer | Hex Integer | Word String
                deriving Show
```

Это парсер для разбора 16-ричных чисел

```
-- attempts to add a character to the parsed representation of a hex digit
parseHexDigit :: Parsed -> Char -> [Parsed]
parseHexDigit (Hex n) c =
   if isHexDigit c then
       return (Hex ((n*16) + (toInteger (digitToInt c))))
     else mzero
parseHexDigit _ _ = mzero
```

Парсер для разбора 10-тичных чисел

```
-- attempts to add a character to the parsed representation of a decimal
digit
parseDigit :: Parsed -> Char -> [Parsed]
parseDigit (Digit n) c =
   if isDigit c then
```

```
       return (Digit ((n*10) + (toInteger (digitToInt c))))
      else mzero
parseDigit _ _ = mzero
```

Парсер для разбора слов:

```
-- attempts to add a character to the parsed representation of a word
parseWord :: Parsed -> Char -> [Parsed]
parseWord (Word s) c =
   if isAlpha c then
       return (Word (s ++ [c]))
      else mzero
parseWord _  _ = mzero
```

Итоговый, объединённый парсер:

```
-- tries to parse the digit as a hex value,
-- a decimal value and a word
-- the result is a list of possible parses
parse :: Parsed -> Char -> [Parsed]
parse p c =
  (parseHexDigit p c) `mplus`
  (parseDigit p c)    `mplus`
  (parseWord p c)
```

Применение парсера в контексте монады списка (неоднозначных функций)

```
-- parse an entire String and return a list of the possible parsed values
parseArg :: String -> [Parsed]
parseArg s =
   do
     init <- (return (Hex 0))   `mplus`
             (return (Digit 0)) `mplus`
             (return (Word ""))
     foldM parse init s
```

О функции **foldM** можно посмотреть в документации (возможно, позже поговорим о таких монадических утилитах):

Control-Monad: foldM

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

Функция **foldM** аналогична **foldl**, за исключением того, что её результат встроен в монаде (инкапсулируется). Обратите внимание, что **foldM** работает слева направо над аргументами списка. Это может быть проблемой, когда (>>) и «сворачиваемая» функция не коммутативны.

```
foldM f a1 [x1, x2, ..., xm]

==
```

```
do
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```

Если требуется выполнение справа налево, входной список должен быть перевёрнут (reversed).

Примечание: **`foldM`** — то же самое, что `foldlM`.

И вот как это выглядит в работе:

```
*Main> parseArg "dead"
[Hex 57005, Word "dead"]
*Main> parseArg "100"
[Hex 256, Digit 100]
```

Ещё пример. НДКА

Определение. *Недетерминированным конечным автоматом* (сокращённо НДКА) называется упорядоченная пятёрка $\mathfrak{A} = \langle Q, A, \Delta, q_0, F\rangle$, состоящая из следующих объектов:

а) $Q = \{q_0, \dots, q_m\}$ — конечный алфавит *внутренних состояний* автомата;

б) $A = \{a_0, \dots, a_n\}$ — конечный *входной алфавит* автомата;

в) $\Delta : Q \times A \to P(Q)$, где $P(Q)$ — множество всех подмножеств Q, *функция переходов*;

г) q_0 — начальное состояние;

д) $F \subseteq Q$ — множество *выделенных (конечных) состояний*.

Определение. *Путём* в недетерминированном конечном автомате $\mathfrak{A} = \langle Q, A, \Delta, q_0, F\rangle$ назовём любую конечную последовательность $(r_0, s_0, r_1, \dots, s_k, r_k)$ такую, что $r_0, \dots, r_k \in Q$, $s_1, \dots, s_k \in A$ и $r_{i+1} \in \Delta(r_i, s_{i+1})$.

Определение. Говорят, что НДКА $\mathfrak{A} = \langle Q, A, \Delta, q_0, F\rangle$ *распознаёт слово* $s_1 s_2 \dots s_k \in A^*$, если существует такая последовательность состояний $q_0 = r_0, r_1, \dots, r_k$ такая, что

$$\begin{aligned} r_1 &\in \Delta(r_0, s_1),\\ r_2 &\in \Delta(r_1, s_2),\\ &\vdots \quad\quad \vdots\\ r_k &\in \Delta(r_{k-1}, s_k) \end{aligned}$$

и при этом $r_k \in F$.

Иначе говоря, слово $w = s_1 s_2 \dots s_k$ распознаётся автоматом, если в нём существует хотя бы один путь

$$q = r_0 \xrightarrow{s_1} r_1 \xrightarrow{s_2} \dots \xrightarrow{s_k} r_k \in F$$

такой, что он начинается в начальном состоянии q_0, вдоль его дуг читается слово $s_1 s_2 \ldots s_k$ (в недетерминированном автомате такой путь определяется неоднозначно по слову w) и заканчивается в некотором выделенном состоянии.

Пример. Пусть автомат $\mathfrak{A} = \langle Q, A, \Delta, q_0, F \rangle$ задаётся следующим образом: $A = \{a, b\}$, $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, а функция переходов задаётся соотношениями

$$\begin{aligned} &\Delta(q_0, a) = \{q_0\}, && \Delta(q_1, a) = \varnothing, && \Delta(q_2, a) = \varnothing, \\ &\Delta(q_0, b) = \{q_0, q_1\}, && \Delta(q_1, b) = \{q_2\}, && \Delta(q_2, b) = \varnothing. \end{aligned}$$

И у него будет следующее графическое изображение:

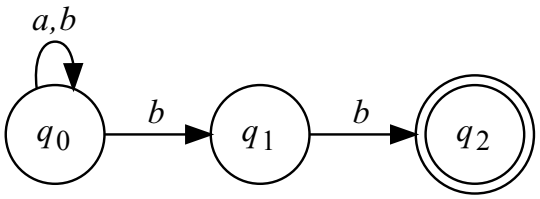

Рис. 2: Иллюстрация к примеру

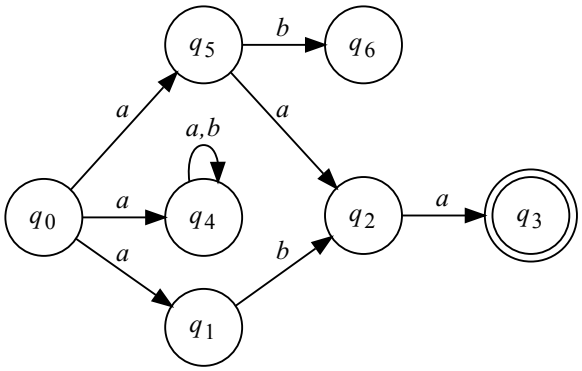

Рис. 3: Пример-2. НДКА

Ссылки по теме

Eric Kidd. Monads in 15 minutes: Backtracking and Maybe