Монады

Введение

В функциональном языке мы фактически избавлены от необходимости последовательного исполнения команд (в императивном стиле). Даже само понятие команд в чистом функциональном языке заменено на функции и их композиции.

Но как уже было оповещено, у нас есть необходимость взаимодействия с внешним миром и у нас есть необходимость организации некоторых вычислений в императивном стиле.

Для этого разработчики языка Haskell внедрили императивный язык. Но он внедрён не отдельным чужеродным составом, а особой эмуляцией, средствами и способами самого языка Haskell. Правда, для этого пришлось задействовать такое понятие как *монады* из области теории категорий.

Что дают нам монады

Чистая композиция

Вернёмся минут на 5 в чистый Haskell и его истокам.

Из основ λ-исчисления, мы помним, что для чистых функций вида

```
f :: a -> b
```

(заданных, например, с помощью лямбда-выражений или другими способами в Haskell) мы должны в языке обеспечить «применение функции» к значениям. В Haskell это можно сделать просто:

```
f 2
```

(если `a` это был тип вроде **`Int`**).

Но в Haskell существует специальный оператор `$`, это оператор применения функции. У него следующий тип:

```
($) :: (a -> b) -> a -> b
```

(напомним, в Haskell символьные инфиксные операторы эквивалентны функциям с тем же именем, заключённым в круглые скобки. Так, запись `f $ 2` эквивалентна записи `($) f 2`).

И он обеспечивает применение функции явным образом.

```
f 2       --> вернёт 4
f $ 2     --> тоже вернёт 4
($) f 2   --> и здесь вернет 4
```

Имея наименьший приоритет и правую ассоциативность, он ещё побочно избавляет нас от лишних скобок — и часто, именно это его основная роль!

```
sin (cos x) == sin $ cos x == sin $ cos $ x
```

Ради интереса мы можем задать оператор «обратного применения», назовём его `$>`, и пусть он принимает те же аргументы в обратном порядке:

```
($>) :: a -> (a -> b) -> b
x $> f = f x
```

т.е., то же самое, что и `f $ x` (только нужно установить низший приоритет и левую ассоциативность):

```
infixl 0 $>
```

Можем читать это как «оператор берёт значение `x`, применяет функцию к нему и возвращает результат». Если вы знакомы с UNIX-системами, вы могли заметить, что юниксовый конвейер (пайп, `|`) работает сходным образом. Вы передаете ему некоторые данные, а он применяет к ним идущую следом программу. Мы можем работать с операторами применения функции когда это удобно и нужно, хотя чаще всего мы их не используем, просто подставляем аргументы в функции.

Теперь, когда мы обсудили применении функций, следующая важная тема — это композиция функций.

Предположим, что у нас есть две функции `f` и `g`, а также значение `x` следующего вида:

```
x :: a
f :: a -> b
g :: b -> c
```

где `a`, `b`, `c` — некоторые типы. Мы можем сделать следующее: взять значение `x`, применить к нему функцию `f` (получив бы значение типа `b`), и затем к результату применить функцию `g`. Значение `x` типа `a` преобразовалось бы к значению типа `b`, а затем то, что получилось, было бы преобразовано к значению типа `c`. Записать на Haskell это проще, чем сказать:

```
g (f x)
```

Но работать это будет только в том случае, если типы `f` и `g` совместимы, т.е., если результат функции `f` имеет тот же тип, что и у аргумента функции `g` (в нашем случае это тип `b`). Применение одной функции к другой можно трактовать и другим способом: мы берём две функции `f` и `g` типов, соответственно, `a -> b` и `b -> c`, и создаём третью функцию типа `a -> c`. Применяя её к аргументу `x`, мы получим результат типа `c`. Эта идея с объединением двух функций в третью называется композицией функций. В Haskell даже определён простой оператор композиции функций:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

Операторы `($)` и `(.)` можно выразить друг через друга, например так:

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

f $ x = (f . id) x
```

И ещё мы можем написать «оператор обратной композиции» `.>`:

```
(.>) :: (a -> b) -> (b -> c) -> (a -> c)
f .> g = \x -> g (f x)
f .> g = \x -> g $ f $ x

f .> g = \x -> x $> f $> g
f .> g = \x -> f x $> g

infixl 9 .>
```

Или выразить через оператор композиции:

```
(.>) :: (a -> b) -> (b -> c) -> (a -> c)
f .> g = g . f
```

Или с помощью функции

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \x y -> f y x
```

можно сделать так:

```
(.>) = flip (.)
```

Композиция функций может показаться не таким уж великим делом, — в реальности же это один из основных пунктов функционального программирования. Композиция позволяет связывать существующие функции в более сложные функции, опуская ручную работу с аргументами. И вместо того, чтобы говорить «`h` — это функция, которая получается сначала вычислением функции `y = f(x)`, а затем вычислением функции `h = g(y)`», мы просто говорим: «`h` — это функция, которую мы получаем, применяя сначала `f`, а затем `g`». Без промежуточных сущностей код становится более кратким и высокоуровневым. Представьте себе, что вам потребовалось вызвать десять функций одну за другой. Если бы вы записывали промежуточные результаты, это вылилось бы в что-то подобное:

```
f11 x =
   let
     x2 = f1 x
     x3 = f2 x2
     x4 = f3 x3
     x5 = f4 x4
     x6 = f5 x5
     x7 = f6 x6
     x8 = f7 x7
     x9 = f8 x8
     x10 = f9 x9
     x11 = f10 x10
   in
     x11
```

Весьма утомительно, правда? А теперь посмотрим на композицию функций:

```
f11 = f10 . f9 . f8 . f7 . f6 . f5 . f4 . f3 . f2 . f1
```

или, то же самое:

```
f11 = f1 .> f2 .> f3 .> f4 .> f5 .> f6 .> f7 .> f8 .> f9 .> f10
```

Это не только короче, но и более интуитивно. («Применяя f1, затем f2, затем f3 и так далее, мы получим f11»). Кстати, этот способ записи функций с использованием композиции и без аргументов, называется «бесточечным стилем». Ирония в том, что в «бесточечном стиле» оператор «точка» (.) очень даже используется, сильнее, чем в обычном коде. Тут правильнее было бы сказать «безаргументный стиль», а не «бесточечный», так как мы опускаем аргументы функций.

Yet Another Monad Tutorial (part 1: basics)

С использованием оператора применения:

```
f11 x = x $> f1 .> f2 .> f3 .> f4 .> f5 .> f6 .> f7 .> f8 .> f9 .> f10
```

или даже

```
f11 x = x $> f1 $> f2 $> f3 $> f4 $> f5 $> f6 $> f7 $> f8 $> f9 $> f10
```

А если сравнить с конвейером в стиле UNIX, то будет выглядеть примерно так:

```
cat file.txt | filter1 | filter2 | filter3 > file2.txt
```

Монадный подход

Монадные значения и монадная функция

Выше было указано, что суть монад в том, чтобы обобщить понятие композиции и применения функций в виде вычислений, которые отличаются от вычислений в чистых функциях. Из определения монад следует, что мы получаем некие «расширенные функции», которые делают что-то ещё, помимо простого вычисления над входным значением. На схематическом псевдо-Haskell-языке мы могли бы записать эти «расширенные функции» так:

```
f :: a --[something else]--> b
```

где `f` — расширенная функция, `a` — тип аргумента, `b` — тип результата, а «что-то ещё» специфично для разных понятий вычислений. В Haskell за словами «понятие вычислений» кроются, в частности, и монады. Мы можем понимать «расширенные функции» как «монадические функции». Это не стандартная терминология, так будем говорить, чтобы отличить их от обычных чистых функций.

В Haskell используется такая форма записи для сигнатур монадических функций:

```
f :: a -> m b
```

для любой монады `m`. Запись значит, что есть некая функция `f`, которая принимает значение типа `a` и возвращает значение типа `m b`. В Haskell `m` обязан быть конструктором типа — специальной функцией на типах: она берёт аргумент (являющийся типом) и возвращает тип. Ещё говорят в таких случаях о *полиморфных* типах. Таковы, например, списки.

Монады, как они есть в Haskell — это конструкторы типов, производящие новый тип оборачиванием вокруг старого. И монада **`IO`**, фактически, является конструктором типа, с помощью которого производятся такие типы как **`IO Bool`**, **`IO Int`**, **`IO Float`**, **`IO Char`**, **`IO String`** и т.д. Это всё валидные типы в Haskell. Сходным образом для монады **`Maybe`** конструируются валидные типы **`Maybe Bool`**, **`Maybe Int`** и т.д.

Итак,

Существует знакомое вам понятие «чистая функция», т.е. такая функция, которая не делает ничего «особенного», а только преобразует входное значение одного типа в выходное значение другого типа (а может быть, того же самого).

И обозначены некоторые особые функции, которые делают что-то ещё помимо преобразования одних значений в другие. Это «что-то ещё» позволяет оперировать вводом/-выводом с файлами или консолью, генерировать исключения, взаимодействовать с глобальным или локальным состоянием, оно может вернуть результат или завершиться с ошибкой, или даже вернуть много результатов. Все эти особые функции представлены монадами, и называем их «монадическими функциями». Понятие монадической функции должно быть достаточно интуитивным, потому что каждый программист постоянно с ними работает, не подозревая об этом.

А что мы можем сказать о сущности «монадических значений»?

Ответ таков: ***Они не представляют собой ничего реально интуитивного!*** Интуитивно понятие монадической функции (той, которая делает что-то ещё кроме конвертирования одних данных в другие). Концепция «монадического значения» вовсе не интуитивна. Просто в Haskell так принято обозначать выходные значения монадических функций.

Тем не менее, в литературе по Haskell вы можете обнаружить два общих способа объяснить монадические значения:

1. Монадическое значение типа `m a` (для некоторой монады `m`) — это особый вид «действия», которое что-то выполняет и возвращает значение типа `a`. Суть действия зависит от каждой конкретной монады.
2. Монадическое значение типа `m a` (для некоторой монады `m`) — это такой контейнер, в котором хранится значение типа `a`.

Изучать монады через размышления о монадических значениях — обычно считается неверным подходом, а верным — через размышления о монадических функциях. Большая часть монад вовсе не контейнеры, хотя некоторые могут вести себя и как контейнеры тоже.

Монадная композиция

Аналогии…

Теперь мы более-менее готовы перевести разговор на «монадные рельсы» ;)

Прежде всего отметим, что в случае монадических функций мы не можем применить обычный оператор композиции `(.)`, так как возникает проблема с согласованием типов. Например, если хотим организовать композицию чистых функций с типами

```
p :: a -> b
q :: b -> c
r :: a -> c
```

операторы композиции легко дают возможность стыковки:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
r = q . p

(.>) :: (a -> b) -> (b -> c) -> (a -> c)
r = p .> q
```

Но если у нас монадические функции

```
f :: a -> IO b
g :: b -> IO c
h :: a -> IO c
```

то при комбинации `g . f` или `f .> g` получим ошибку типизации.

Нам нужна специальная операция монадической композиции, которую в дальнейшем обозначим как `>=>`. У неё следующий тип:

```
(a -> m b) -> (b -> m c) -> (a -> m c)
```

и такое определение:

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

(обычно, этот оператор доступен при подключении пакета Control.Monad).

Она работает для любой монады, в том числе для монады `IO`. Подставляя `IO`, мы получим соответствующее определение функции:

```
(a -> IO b) -> (b -> IO c) -> (a -> IO c)
```

Мы могли бы её использовать для композиции монадических функций `f` и `g`. Тип функции `h` был бы корректным:

```
f :: a -> IO b
g :: b -> IO c
h :: a -> IO c
h = f >=> g
```

Вот план, по которому она могла бы быть задана:

1. принять исходное значение типа `a`;
2. применить функцию `f` к нему (обычное применение функций) и получить результат типа `IO b`;
3. взять значение типа `IO b` от функции `f` и извлечь значение типа `b`;
4. взять значение типа `b` и применить функцию `g` к нему (опять обычное применение функций), чтобы получить значение типа `IO c`, которое и есть искомый результат.

Единственное, что мы не можем ещё сделать, это шаг (3): получить значение типа `b` из значения типа `IO b`. Давайте придумаем функцию `extract`, которую бы мы могли использовать для извлечения. Вот её тип:

```
extract :: IO b -> b
```

А если обобщить на все монады, получим:

```
extract :: m b -> b
```

И если бы `extract` была бы у нас, тогда мы могли бы сделать таким образом:

```
h = f .> extract .> g
```

Оказывается, что если бы такая функция существовала, она бы нивелировала все преимущества монад и чистого функционального программирования! Есть причина, по которой нам нужны монады. Мы хотим хранить специальные понятия вычислений (монадические функции) отдельно от чистых функций, потому что иначе не было бы никаких гарантий, что чистые функции — чистые. Это важный момент.

Вот пример, допустим такой универсальный экстрактор у нас есть. Тогда, мы могли бы написать такой код:

```
appendLine :: String -> String
appendLine str = str ++ extract getLine
```

и формально функция `appendLine` стала бы чистой, но по факту её результат не был бы математически предсказуем и зависел бы от действий пользователя.

How to extract value from monadic action

На самом деле, для некоторых монад есть эквивалент функции `extract`, что не влечет за собой никаких проблем. **Однако обобщённая на все монады функция `extract` запрещена**.

Таким образом, для каждой конкретной монады, реализация этого плана (иначе говоря, воплощение) делается индивидуально, и мы разберём некоторые детали таких реализаций на следующих лекциях. Наиболее «мистической» останется монада **`IO`**, и для неё мы примем, что детали реализации такой распаковки фактически скрыты на уровне компилятора.

N.B. *Отметим, что «план с распаковкой» помимо всего прочего не учитывает наличие особых стратегий при вычислениях в каждых монадах, что может оказаться весьма важным в конкретных монадах!!*

Определения монадных композиций и применения

Начнём разбирать теперь оператор монадного применения.

Монадический оператор применения мы можем записать двумя способами (как и классические `$`, `$>`, т.е. в две стороны). Первый способ — это так называемый bind-оператор `>>=` с типом

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

который является аналогом обычного оператора применения `$>`. Тривиально задаётся монадический оператор применения, принимающий аргументы в обратом порядке:

```
(=<<) :: (a -> m b) -> m a -> m b
 f =« x  = x »= f
```

или так:

```
(=«) = flip (»=)
```

И стандартный монадический оператор композиции `>=>` (Kleisli composition operator):

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

Это должно нам напомнить такую композицию обычных функций:

```
f .> g = \x -> f x $> g
```

И аналог `(.)`:

```
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(<=<) = flip (>=>)
```

Итак, мы определили монадические операторы применения и композиции для любого порядка операндов, как мы это делали с обычными (немонадическими) операторами. На практике, однако, Haskell-программисты более всего используют оператор `>>=`.

Особая операция return

> Если вы программируете в основном в императивном стиле (т.е. например, на языке Си и подобных), слово **`return`** может показаться вам несколько раздражительным. Просто запомните, что это не ключевое слово в Haskell, и тут ничего не возвращается из функции. Постарайтесь не думать о **`return`** как о **`return`** в императивном языке программирования (Еще Одно Руководство по Монадам (часть 2)).

Тут, правда, есть ещё одна тонкость: когда мы с вами тренировались в императивном программировании на Haskell и встречали **`return`**, то заметили, что часто его поведение всё-таки немного похоже на таковое в императивных языках. Пришло время понять, что всё-таки эта операция делает.

Рассмотрим следующий сценарий. Пусть вам потребовалось соединить монадическую функцию с немонадической. Иными словами, у вас есть такие функции:

```
f :: a -> m b    -- монадическая
g :: b -> c      -- немонадическая
```

Проблема в следующем. Вы не можете использовать обычную функцию композиции для `f` и `g`, потому что тип `m b` — не то же самое, что тип `b`. И монадическая композиция тоже не подходит, — она ничего не знает о типе `b -> c`, ей нужен монадический тип `b -> m c`. Что вы будете делать?

Если бы у вас была функция `extract`, разобранная ранее, вы бы могли соединить две функции таким вот образом:

```
h :: a -> c
h = f .> extract .> g
```

но мы уже выяснили, что это невозможно. Иными словами, нам запрещено комбинировать немонадическую функцию и монадическую, чтобы получить немонадическую (потому что это нарушило бы чистоту всего языка). А разрешено нам комбинировать монадическую и немонадическую, если в результате снова получится монадическая функция. Примерно вот так:

```
h :: a -> m c
h = f [somehow composed with] g
```

т.е., мы смогли бы сделать то, что нам надо и воспользоваться монадической композицией, если бы сумели преобразовать обычную функцию в монадическую. И мы это смогли бы сделать, если бы у нас была бы возможность преобразовать обычное значение в монадическое (как говорят, с минимальным контекстом). И такая операция-преобразователь существует, и она как раз и называется **`return`**. У нее следующий тип:

```
return :: a -> m a
```

где `a` — любой тип, и `m` — любой монадический конструктор типа. Функция **`return`** конвертирует обычное значение в соответствующее монадическое значение для любой монады `m`, и это всё, что она делает.

Таким образом, требуемую задачу мы могли бы решить следующим образом:

```
h = f >=> (return . g)
h = f >=> (g .> return)
```

Название «return» на самом деле пришло из понимания монадических значений как «действий». В этом смысле функция **`return`** берет простое значение и производит монадическое значение, которое является по сути «акцией» или «действием», с помощью которого в свою очередь уже «что-то делается», а в результате возвращается незменённое значение (но упакованное или обёрнутое каким-то образом). Стоит заметить также, что фактически **`return`** — монадическая функция. Сложив эти две идеи, можно сделать вывод, что **`return`** — монадическая версия тождественной (identity, или **`id`**) функции (эта функция отображает значение в само себя, т.е. `\x -> x`).

Есть ещё один довольно своеобразный момент с **`return`**. Сказано, что тип **`return`** выглядит как `a -> m a`. Когда мы говорим, например, **`return`** `10`, какой будет тип этого выражения? Он может быть **`IO Int`**, **`Maybe Int`**, **`[Int]`** или ещё какой-нибудь монадический **`Int`**. Откуда нам знать, что за монада тут стоит?

В Haskell смысл **`return`** `10` определяется контекстом. Валидатор типов (type checker) должен убедиться, что функции получают аргументы с нужными типами, так что если **`return`** `10` передан в функцию, где ожидается значение **`IO Int`**, то **`return`** `10` будет трактоваться как **`IO Int`**. Иначе говоря, значение выражения **`return`** `10` зависит от типа, в контексте которого оно используется. Если пожелаете, вы можете явно задать тип выражения **`return`** `10` с помощью, например, такой записи

```
return 10 :: IO Int
```

(но это редко бывает нужно).

Ну и теперь мы можем описать цепочку монадических функций в стиле цепочек чистых функций:

```
f11 = f1 .> f2 .> f3 .> f4 .> f5 .> f6 .> f7 .> f8 .> f9 .> f10

f11 x = x $> f1 .> f2 .> f3 .> f4 .> f5 .> f6 .> f7 .>
  f8 .> f9 .> f10

f11 x = x $> f1 $> f2 $> f3 $> f4 $> f5 $> f6 $> f7 $>
  f8 $> f9 $> f10
```

Теперь это будет так:

```
f11 = f1 >=> f2 >=> f3 >=> f4 >=> f5 >=> f6 >=> f7 >=>
  f8 >=> f9 >=> f10

f11 x = return x >>= f1 >=> f2 >=> f3 >=> f4 >=> f5 >=> f6 >=> f7
  >=> f8 >=> f9 >=> f10

f11 x = return x >>= f1 >>= f2 >>= f3 >>= f4 >>= f5 >>= f6 >>= f7
  >>= f8 >>= f9 >>= f10
```

для некоторых записей, возможно, потребуются скобки (это связано с определением `>=>` как **`infixr`** `1`, вероятно по ошибке… `:(`):

```
f11 x = x $> (f1 .> f2 .> f3 .> f4 .> f5 .> f6 .> f7
  .> f8 .> f9 .> f10)

f11 x = return x >>= (f1 >=> f2 >=> f3 >=> f4 >=> f5 >=> f6 >=> f7
  >=> f8 >=> f9 >=> f10)
```

Или реализовать «бесточечный стиль»:

```
f11  = f1 .> f2 .> f3 .> f4 .> f5 .> f6 .> f7
  .> f8 .> f9 .> f10

f11 = f1 >=> f2 >=> f3 >=> f4 >=> f5 >=> f6 >=> f7
  >=> f8 >=> f9 >=> f10
```

Ну, или ввести синоним оператора (`>=>`) с нужным нам приоритетом:

```
import Control.Monad

(>->) = (>=>)
infixl 9 >->

f11 x = return x >>= f1 >-> f2 >-> f3 >-> f4 >-> f5 >-> f6 >-> f7
  >-> f8 >-> f9 >-> f10
```

Класс Monad

Теперь мы готовы дать формальное определение класса Monad, который и определяет «правила игры» с монадами.

Изначальное и традиционное определение класса монад было сделано примерно в таком виде:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Монады являются расширенной версией аппликативных функторов (т.е. являются и функторами, и аппликативными функторами, о которых мы ранее уже говорили), и поэтому следовало бы указать (как в современной версии описания класса **Monad** и делается):

```
class (Applicative m) =>  Monad m where
```

но мы пока начнём с изучения традиционной версии, и, как пишет Липовача (с.374):

> когда появился язык Haskell, людям не пришло в голову, что аппликативные функторы хорошо подходят для этого языка.

Иными словами, монады в языке появились значительно раньше, решая кардинально сложную проблему, часть из которой мы уже описали.

Замечание. Отметим здесь, что хотя введённый класс **Monad** в принципе похож на такой класс как:

```
class Eq a where
  (==) :: a -> a -> Bool
```

о котором мы говорили в одной из ранних лекций, и его воплощения определялись примерно таким образом:

```
instance Eq Int where
  (==) = intEquals
```

где тип встроенной функции `intEquals` такой: `(Int -> Int -> Bool)`,

```
instance Eq Float where
  (==) = floatEquals
```

где тип встроенной функции `floatEquals`: `(Float -> Float -> Bool)`, — тут присутствует некоторая особенность или даже странность.

Странность с классом типов Monad в том, что он не таков, как **Eq**. **Monad** является «классом конструкторов» (constructor class), для которого экземпляры (обозначенные как `m`) не типы, но конструкторы типов; мы как раз уже убедились, что все монады должны быть конструкторами типов. Вот так мы задаём экземпляр класса конструкторов (для примера взята монада **Maybe**):

```
instance Monad Maybe where
  (>>=)  = {- версия (>>=) для Maybe -}
  return = {- версия return для Maybe-}
```

Для обычных классов и для классов конструкторов выбрана одна и та же запись, что может вас немного запутать, но немного практики — и всё встанет на свои места.

В Haskell нет особого синтаксиса, чтобы отличать классы конструктора от классов типов. Это выводится из сигнатур метода.

> There's no particular syntax to distinguish constructor classes from type classes. It's inferred from the method signatures. Constructor class

Для этого вводится специальное понятие видов (*kinds*) и мы можем посмотреть на вид переменной типа `a` и конструктора типов `m`. Они будут соответственно определяться как `*` и `* -> *`. Мы можем посмотреть на их конкретные значения:

```
Prelude> :k Int
Int :: *
Prelude> :k Maybe
Maybe :: * -> *
Prelude>
```

Собственно, с функторами и аппликативными функторами мы могли наблюдать то же самое, только не акцентировали этот момент.

wikipedia: Higher-kinded polymorphism

Constructor classes

Близко, но не совсем:

Data.Kind

What is Constraint in kind signature

The Constraint kind

Constraint Kind

Класс **`Monad`** вводит ещё несколько функций, его более полная формулировка выглядит так:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
    (>>) :: m a -> m b -> m b
    fail :: String -> m a
```

Функция **`fail`** изначально введена как примитивный способ оповещения об ошибке. Она вызывается, когда оператор `>>=` не может связать значение типа `a` и функцию типа `a -> m b` из-за ошибок сопоставления. Мы не будем вдаваться в детали этого механизма. В большинстве случаев беспокоиться о **`fail`** не нужно.

Оператор `>>` немного интереснее. У него такой тип:

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

Этот оператор представляет собой монадический оператор последовательности. В частности, это вариант монадического применения (`>>=` или «bind»), который отбрасывает распакованное значение типа `a` перед выполнением «действия» типа `m b`. Он определен следующим образом:

```
mv1 >> mv2  =  mv1 >>= (\_ -> mv2)
```

Мы можем видеть здесь, что любое значение, распакованное из монадического значения `mv1`, отвергается, и потом возвращается финальное монадическое значение `mv2`. Оператор бывает полезным, когда тип распакованного значения равен `()`, т.е., является пустым типом. Хорошим примером можно считать функцию **`putStrLn`**:

```
putStrLn :: String -> IO ()
```

Представьте, что вы хотите напечатать две строки, одну за другой, со знаками конца строки после каждой. Можно так:

```
putStrLn "This is string 1." >> putStrLn "This is string 2."
```

И почему же это работает? Посмотрим на типы:

```
(putStrLn "This is string 1.") :: IO ()
(putStrLn "This is string 2.") :: IO ()
```

Оператор `>>` комбинирует два монадических значения типа **`IO`** `()` в одно результирующее монадическое значение типа **`IO`** `()`. Давайте возьмем оператор `>>` и рассмотрим его для нашего случая:

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

Если `m` — это **`IO`**, и `a`, и `b` — `()`, то получим:

```
(>>) :: IO () -> IO () -> IO ()
```

По записи можно сказать, что, вероятно, оператор `>>` выполняет подряд два «действия» — печать строк.

Кстати, чтобы легче было понять, как действует этот оператор, рассмотрим его «немонадический» аналог:

```
Prelude> x |> y = x $> (\_ -> y)
Prelude> 3 |> 'a'
'a'
Prelude> :t (|>)
(|>) :: t1 -> t2 -> t2
Prelude>
```

Видим, что роль оператора `(|>)` сводится к простой связке: первое значение игнорируется, и возвращаем второе значение в качестве ответа. При этом организуется «фейковая» функция, которая ничего не принимает на вход, возвращая только требуемое значение.

Законы монад, do-синтаксис

Хотя в предыдущем разделе была сделана достаточно полная попытка объяснить, зачем были введены монады и как они работают, описание было бы неполным, если бы мы не зафиксировали *законы монад*.

Так как монады пришли из чистой математики (как моноиды, функторы и другие структуры), то они вынуждены подчиняться тем математическим аксиомам, которые их описывают. Мы не будем смотреть математический формализм и теоретико-категорное происхождение монад, однако опишем законы монад, так как проверка их выполнения является обязанностью программиста-разработчика. Haskell не проверяет монадные законы,

единственное, что проверяется, — это чтобы типы определений **return** и >>= были корректными.

Вряд ли нам самим придётся вводить и разрабатывать собственные монады, и таким образом, доказывать выполнение этих законов для наших собственных монад. Но, возможно, в обучающих целях, мы проверим выполнение этих законов для монад, которые бы будем рассматривать.

Вот определение трёх монадных законов, выраженных в терминах монадической композиции (оператор (>=>) — это монадический оператор композиции функций (fish operator)):

```
1.    return >=> f ≡  f
2.    f >=> return ≡  f
3. (f >=> g) >=> h ≡  f >=> (g >=> h)
```

Знак ≡ здесь означает скорее равенство в математическом смысле, для программиста же это означает, что значение слева, легко может быть заменено значением справа в любом месте программного кода (Monad laws).

О чем эти законы говорят нам?

Законы 1 и 2 говорят, чем должен быть **return**: это единица (нейтральный элемент) для монадической композиции функций (первое правило утверждает, что **return** — левая единица, а второе — что правая). Другими словами, композиция монадической функции f и **return** (в любом порядке) просто возвращает функцию f. Аналогами можно считать 0 — нейтральный элемент для функции сложения целых чисел, и 1 — нейтральный элемент для целочисленной функции умножения; в каждом из случаев нейтральный элемент, соединенный с обычным значением при помощи соответствующей функции, просто вернет назад это значение.

Закон 3 гласит, что монадическая функция композиции ассоциативна: когда мы хотим комбинировать три монадические функции f, g, h, то не важно, какие две мы соединим первыми. Это аналог того, что сложение и умножение тоже ассоциативны в применении к целым числам.

Вам не кажутся эти законы смутно знакомыми?

Во-первых, это уже знакомые нам законы математического моноида для оператора (>=>).

А во-вторых, взглянем на соответствующие «законы», которым удовлетворяет обычная функция композиции:

```
1. id . f ≡         f
2. f  . id ≡        f
3. (f . g) . h ≡    f . (g . h)
```

или для оператора .>:

```
1. id .> f ≡         f
2. f  .> id ≡        f
3. (f .> g) .> h ≡   f .> (g .> h)
```

где **id** — нейтральный элемент, единица. Композиция функции с единицей слева или справа даст снова ту же функцию, и функция композиции ассоциативна. Монадическая функция композиции должна быть ассоциативна, и **return** должен быть монадическим эквивалентом единичной функции, чтобы поведение монадической композиции было таким же предсказуемым, как и поведение обычной композиции.

Какое значение у этих законов с точки зрения программиста? Так как мы желаем, чтобы наши монады вели себя разумно, наши определения **return** и `>>=` должны удовлетворять этим законам.

Законы монад в версии `>>=`

Несмотря на ясность, монадическую композицию Клейсли (`>=>`) как-то не принято слишком широко использовать. Да и сам оператор `>=>` не определяется напрямую в классе типов **Monad**, вместо этого определен оператор `>>=`, а оператор `>=>` выводится из него, как показано выше.

Кроме того, аналогично с обычной композицией функций (`.`), когда версия с оператором применения `$` оказывается более удобной на практике, позволяя легко оперировать переменной `x` — в практике монад более удобно использовать оператор «монадического применения» (`>>=`).

Так что если мы ограничиваем определения до операторов `>>=` и **return**, нам нужны монадные законы, содержащие только **return** и `>>=`. И в таком виде они подаются в большинстве книг и документаций по монадам в Haskell, несмотря на меньшую интуитивность, чем показано в прошлой секции.

Поэтому, сформулируем теперь законы монад в другой записи. В терминах оператора `>>=` и функции **return** монадные законы выглядят так:

```
1.   return x >>= f ≡ f x
2.    mv >>= return ≡ mv
3. (mv >>= f) >>= g ≡ mv >>= (\x -> (f x >>= g))
```

где типы различных значений такие:

```
mv :: m a
f  :: a -> m b
g  :: b -> m c
```

для некоторых типов `a`, `b`, `c` и какой-то монады `m`.

Монадные законы иногда можно использовать в коде, заменив длинное выражение более коротким (например, вместо **return** `x >>= f` можно писать просто `f x`). Однако, основная польза монадных законов в том, что они позволяют выводить определения **return** и `>>=` для конкретных монад.

Чуть ниже мы рассмотрим ещё пару банальных примеров.

Bartosz Milewski. Monads: Programmer's Definition

But why should monads obey these laws?

перевод: Монады: определение программиста, сам текст

A monad is just a monoid in the category of endofunctors, what's the problem?

Monday Morning Haskell: Monads

Monday Morning Haskell: Monad Laws

Monday Morning Haskell: Monads (and other Functional Structures)

do-синтаксис

Для удобства работы с монадным кодом (так как одна из целей введения такой концепции была возможность описания последовательного исполнения действий) введены синтаксические удобства в виде do-синтаксиса. Такою запись кода мы уже рассматривали, когда изучали императивное программирование на Haskell.

Дизайнеры языка Haskell заметили, что монадические определения часто трудно читать, и придумали действительно «приятный синтаксический сахар», с которым определения получаются более читаемыми.

В основе этого синтаксического сахара лежит наблюдение, что огромное количество операций в монадическом коде записывается в двух формах:

Форма 1.

```
-- mv :: m a
-- f :: a -> m b

mv >>= \x -> f x
```

Форма 2.

```
-- mv :: m a
-- mv2 :: m b

mv >> mv2
```

Нотация разрабатывалась с намерением сделать эти две формы легкочитаемыми. Она начинается с ключевого слова **do**, за которым следуют некоторые монадические операции. Так эти два наших примера будут записаны в **do**-нотации:

Форма 1, do-нотация.

```
do v <- mv
   f v
```

Форма 2, do-нотация.

```
do mv
   mv2
```

В форме 1 первая строка означает, что мы берем монадическое значение `mv` и «распаковываем» его в обычное под названием `v`. Вторая строка — это просто вычисление `f` от `v`. Результат строки `f v` является результатом всего выражения.

В форме 2 в первой строке «выполняется» монадическое значение («действие») `mv`. Во второй строке «выполняется» другое монадическое значение («действие») `mv2`. Таким

образом, мы имеем просто нотацию, которая увязывает в последовательность `mv` и `mv2`, как это делает оператор `>>`.

Компилятор в Haskell преобразовывает удобную do-нотацию в запись без **`do`** для формы 1 и формы 2. Это просто синтаксическое преобразование, а смысл обеих записей идентичен. Пример:

```
-- mv :: m a
-- v1 :: a
-- f :: a -> m b
-- v2 :: b
-- mv3 :: m c

do v1 <- mv
   v2 <- f v1
   mv3
   return v2
```

Это в точности то же самое, что и:

```
mv >>= (\v1 ->
  (f v1 >>= (\v2 ->
     (mv3 >>
        (return v2)))))
```

Или без скобок:

```
mv >>= \v1 ->
  f v1 >>= \v2 ->
    mv3 >> return v2
```

Кроме того, можно смешивать do-нотацию и обессахаренную нотацию в одном выражении. Вот так:

```
    do v1 <- mv
       v2 <- f v1
       mv3 >> return v2
```

Кроме того, обе формы можно смешивать в одном выражении каждой из нотаций. Иногда это полезно, но может часто стать причиной плохой читаемости кода.

Yet Another Monad Tutorial (part 3: The Monad Laws)

wikibooks: Haskell/ do notation

wiki.haskell: Monad/ do-notation

wikipedia: Monad/ Syntax_Sugar

wiki.haskell: Do notation considered harmful

Кстати, наша цепочка монадических функций теперь будет выглядеть так:

```
f11 x = do
          x2 <- f1 x
          x3 <- f2 x2
          x4 <- f3 x3
```

```
...
x9 <- f8 x8
x10 <- f9 x9
f11 x10
```

или последние строки можно оформить так:

```
x10 <- f9 x9
x11 <- f11 x10
return x11
```

что как раз даёт нам аналогию применения **`return`** о которой мы выше говорили!