Монады и do-синтаксис

Михаил Беляев

9 октября 2018 г.

Абстракции вычислений

Функция:

$$A \xrightarrow{f} B$$

Функтор:

$$\boxed{\text{A}} \xrightarrow{fmap(f)} \boxed{\text{B}}$$

Законы: $fmap(id)$ не должен ничего менять

Абстракции вычислений

Функция:

$$A \xrightarrow{f} B$$

Аппликативный функтор:

$$\boxed{\text{A}} \xrightarrow{fmap(f)} \boxed{\text{B}}$$

$$A \xrightarrow{pure} \boxed{\text{A}}$$

$$\boxed{\text{A}} \xrightarrow{\boxed{\text{f}}<*>} \boxed{\text{B}}$$

Законы: $pure(f) <*> x \equiv fmap(f)(x)$

Абстракции вычислений

Функция:

$$A \xrightarrow{f} B$$

Монада:

$$\boxed{\text{A}} \xrightarrow{fmap(f)} \boxed{\text{B}}$$

$$A \xrightarrow{return} \boxed{\text{A}}$$

$$\boxed{\boxed{\text{A}}} \xrightarrow{flatten} \boxed{\text{A}}$$

Законы: $flatten(return(x)) \equiv x$ (ну и по мелочи...)

Абстракции вычислений

Что из этого круче?

Абстракции вычислений

Что из этого круче?

Любая монада является аппликативным функтором, но не наоборот. Реализация?

Абстракции вычислений

Функция:

$$A \xrightarrow{g} \boxed{\text{B}}$$

Монада (более короткий, но менее понятный, набор):

$$A \xrightarrow{return} \boxed{\text{A}}$$

$$\boxed{\text{A}} \xrightarrow{bind(g)} \boxed{\text{B}}$$

Классы типов: монада

Монада — это параметризованный тип `m`, удовлетворяющий требованиям:

- Он является функтором
- Методы:

```
return :: a -> m a
flatten :: m (m a) -> m a
x >>= f = flatten (fmap f x)
```

- Работают законы:

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

Монада: формальное определение

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

Почему требует Applicative?

Applicative << Monad

```haskell
instance Applicative m where
    pure = return
    f <*> x = f >>= \f' -> x >>= \x' -> return (f' x')
```

Обратно в общем случае не получится

Проблема 1: Coalescing

Есть несколько функций, которые зависят друг от друга и используют результаты друг друга, но каждая из них может выдать ошибку.

```
x :: Maybe Int
y :: Maybe Int

xplusy =
    case x of
        Nothing -> Nothing
        Just x' ->
            case y of
                Nothing -> Nothing
                Just y' -> Just (x' + y')
```

Решение проблемы 1 с помощью монады `Maybe`

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

x :: Maybe Int
y :: Maybe Int
xplusy =
    x >>= \ x' ->
      y >>= \ y' ->
        return (x' + y')
```

Проблема 2: декартово произведение

Есть несколько списков, нужно скомбинировать их элементы «каждый с каждым»

```
x :: [Int]
y :: [Int]

xplusy = concat (map (\x' -> map (+x') y) x)
```

Решение проблемы 2 с помощью монады `List`

```haskell
instance Monad [] where
    lst >>= f = concat (map f lst)
    return x = [x]
    fail message = []

x :: [Int]
y :: [Int]
xplusy =
    x >>= \ x' ->
      y >>= \ y' ->
        return (x' + y')
```

Какие ещё штуки можно так комбинировать?

```
xplusy =
    x >>= \ x' ->
      y >>= \ y' ->
        return (x' + y')
```

- Вне FP: async, future, lazy
- Коллекции (но не все!)
- Потоки данных

Композиция с помощью `bind`

```
xplusy =
    x >>= \ x' ->
      y >>= \ y' ->
        return (x' + y')

f :: a -> m a
xplusytimez =
    x >>= \ x' ->
      y >>= \ y' ->
        (f y') >>= \ z' ->
           return $ x' + y' * z'
```

Do-синтаксис

```haskell
xplusytimez =
    x >>= \ x' ->
      y >>= \ y' ->
        (f y') >>= \ z' ->
           return $ x' + y' * z'

xplusytimez =
    do
       x' <- x
       y' <- y
       z' <- f y'
       return $ x' + y' * z'
```

Do-синтаксис

- Вычисления со *скрытым контекстом*
- Примеры такого контекста: декартово произведение, вычисления с обработкой ошибок
- Контекст не является чем-то фиксированным, он просто должен отвечать законам монады

Примеры монад: монада `Identity`

Монада с пустым контекстом

```
data Identity a = Identity a
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
instance Monad Identity where
    return x = Identity x
    (Identity x) >>= f = f x
    fail message = error message
```

Примеры монад: монада `Identity`

Вычисления в монаде `Identity` — это просто обычные вычисления

```
xplusy x y =
    do
        x' <- x
        y' <- y
        return $ x' + y'
xplusy (Identity 2) (Identity 3) === (Identity 5)
```

Примеры монад: монада `Maybe`

Контекст: вычисления с возможностью неудачи

```
data Maybe a = Just a | Nothing
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
instance Monad Maybe where
    return x = Just x
    (Just x) >>= f = f x
    Nothing  >>= f = Nothing
    fail message = Nothing
```

Примеры монад: монада `Maybe`

Вычисления в монаде Maybe — это обычные вычисления, но любое действие может «не получиться»

```
xplusy x y =
    do
        x' <- x
        y' <- y
        return $ x' + y'
xplusy (Just 2) (Just 3) === (Just 5)
xplusy (Just 42) (Nothing) === Nothing
```

Расширение монады Maybe: монада Error

Контекст: вычисления с возможностью неудачи

```
data Error a = Success a | Error String
instance Functor Error where
    fmap f (Success x) = Success (f x)
    fmap f (Error s) = Error s
instance Monad Error where
    return x = Success x
    (Success x) >>= f = f x
    (Error s)  >>= f = (Error s)
    fail message = Error message
```

Примеры монад: монада `List`

Контекст (по умолчанию): декартово произведение списков

```
instance Functor [] where
    fmap = map
instance Monad [] where
    return x = [x]
    (>>=) = flatMap
    fail message = []
```

Примеры монад: монада List

Вычисления в монаде List — это обычные вычисления, но любое действие комбинируется в аргументах «каждый с каждым»

```
xplusy x y =
    do
       x' <- x
       y' <- y
       return $ x' + y'
xplusy [0..2] [4,5] === [4,5,6,5,6,7]
xplusy [0..10000] [] === []
```

List comprehensions

```haskell
-- все сочетания чисел от 0 до 5
allComb = do
            x <- [0..5]
            y <- [0..5]
            return (x,y)
allComb = [(x,y) | x <- [0..5], y <- [0..5]]
```

Монады «ради монад»

- Монада `Reader`
- Монада `Writer`
- Монада `State`

Монада Reader

Контекст: вычисления с контекстом, который можно читать

```haskell
data Reader env a = Reader (env -> a)
runReader (Reader f) env = f env
ask :: Reader env env
ask = Reader (x -> x)
instance Monad (Reader env) where
   return x = Reader (\_ -> x)
   (Reader f) >>= g = Reader $ \x -> runReader (g (f x)) x
```

Монада Reader

Контекст: вычисления с контекстом

```
xplusenv x = do
               x' <- x
               env <- ask
               return $ x' + env
fortyTwoPlusEnv = xplusenv (return 42)
runReader fortyTwoPlusEnv 3 === 45
runReader fortyTwoPlusEnv 8 === 50
```

Монада `Writer`

Контекст: вычисления с контекстом, который можно писать

```haskell
data Writer env a = Writer (a, env)
runWriter (Writer x) = x
tell env = Writer ((), env)

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`

Контекст: вычисления с контекстом, в который можно писать

```
logFirst x y = do
                x' <- x
                y' <- y
                if (x' > y') then tell [x']
                             else return ()
                return $ x + y
foo = logFirst (return 5) (return 4)
fee = logFirst (return 13) (return 12)
bar = logFirst foo fee
runWriter foo === (9, [5])
runWriter fee === (25, [13])
runWriter bar === (34, [5,13,9])
```

Монада State

Контекст: вычисления с контекстом, который можно и читать, и писать

```haskell
data State env a = State (env -> (a, env))
runState (State f) = f
put newState = State $ \ _ -> ((), newState)
get = State $ \ current -> (current, current)
instance Monad (State s) where
    return a         = State $ \s -> (a,s)
    (State x) >>= f = State $ \s -> let (v,s') = x s in
                                    runState (f v) s'
```

Промежуточные итоги

- Абстракция монады позволяет на *чистых функциях* реализовать псевдо-изменяемое состояние с сохранением всех следующих особенностей:
  - Порядок выполнения операций
  - Независимость результата от точки в коде, в которой операция происходит

Промежуточные итоги

- Почему это по-прежнему функционально-чистый подход?
  - В рамках монадных операций «создать» контекст невозможно, для этого всегда нужны внешние функции
  - Сам контекст при этом изолирован в рамках набора монадных операций, для извлечения результатов так же всегда нужны внешние функции

Монада `IO`

Представим себе, что *весь окружающий программу мир* — это состояние, которое можно поместить в монаду а-ля `State`.

Только магической функции `runState` нет, вместо неё стандартный вызов через `main`.

Получим монаду `IO`.

Монада IO

Disclaimer: это не настоящий код!

```
data IO a = ...
instance Monad IO where
    return a = ... -> (world, a)
    io >>= f = (world, x) -> ... -> (getWorld f x, getValue f x)
```

Монада `IO`

Как войти в вычисления в монаде `IO`?
Через функцию `main :: IO ()`

Как достать значение из `IO x`?
Никак

Монада `IO`

Как войти в вычисления в монаде `IO`?
Через функцию `main :: IO ()`

Как достать значение из `IO x`?
Никак

Ну, почти...

Монады-трансформеры

Что делать, если вам нужны вычисления с множеством контекстов?

- Возможность ошибки + состояние
  - Парсеры
  - Разбор бинарных файлов
- Состояние + IO
- ...

Монады-трасформеры

- Берём монаду и реализуем все операции так, чтобы они прозрачно передавались монаде «внутри»
- Примеры:
  - `ErrorT m a`
  - `StateT s m a`
  - `ReaderT env m a`
  - `WriterT env m a`

```
type GetCtx a = State (Input, Position) a
type Parser a = ErrorT GetCtx a
```

Монады как абстракции вычислений

- Монада — это просто инструмент:
  - Композиции функций
  - Скрытия контекста
- Есть и другие инструменты

Больше абстракций!

Функция:

$$\boxed{A} \xrightarrow{g} B$$

Комонада:

$$\boxed{A} \xrightarrow{extract} A$$

$$\boxed{A} \xrightarrow{duplicate} \boxed{\boxed{A}}$$

$$\boxed{A} \xrightarrow{extend(g)} \boxed{B}$$

Что является монадой?

- Всё, что угодно, над чем:
  - Можно определить монадные операции
  - Работают монадные законы
  - И всё это можно для чего-нибудь использовать
- А как же рассказы про вездесущие монады?
  - Да никак, это просто инструмент

Что позволяют монады?

- Писать код в «процедурном» стиле, причем «процедурность» заложена в типе данных
- Можно писать функции, не зависящие не только от типа, но и **от контекста**
- И всё это надёжно изолировано от «чистого» кода

Другие примеры монад

- The free monad — монада, строящая дерево входящих в неё операций
  - Используется (например, в facebook) для DSL
- `STM` — Softare Transactional Memory
- `ST` — «облегчённая» версия `IO`

Другие примеры монад

Многие хорошо известные ООП-концепции тоже являются монадами:

- future / promise (асинхронное программирование)
- observable (реактивное программирование)
- coroutines
- и т.д.

http://kspt.icc.spbstu.ru/course/lang

belyaev@kspt.icc.spbstu.ru

КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ СПбПУ

ПОЛИТЕХ
Санкт-Петербургский политехнический университет Петра Великого