Монады

Функциональное программирование

2023 г.

Еще раз о функторах

```
(<$>) :: (a -> b) -> f a -> f b
(<*>) :: f(a -> b) -> f a -> f b
```

В каждом из этих случаев сами функции чистые. А если потребовать, чтобы функция возвращала контейнер?

Еще раз о функторах

```
(<$>) :: (a -> b) -> f a -> f b
(<*>) :: f(a -> b) -> f a -> f b
```

В каждом из этих случаев сами функции чистые. А если потребовать, чтобы функция возвращала контейнер?

Стрелка Клейсли

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

Операции над стрелкой Клейсли

- Способ упаковывать значение в контейнер.

```
?? :: a -> f a
```

- Способ строить композицию стрелок Клейсли.

```
?? :: (a -> f b) -> (b -> f c) -> (a -> f c)
```

Операции над стрелкой Клейсли

- Способ упаковывать значение в контейнер.

```
?? :: a -> f a
-- Этот оператор мы знаем как pure
```

- Способ строить композицию стрелок Клейсли.

```
?? :: (a -> f b) -> (b -> f c) -> (a -> f c)
```

Определение монады

```
class Applicative m => Monad m where
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
m1 >> m2 = m1 >>= \ _ -> m2
```

Как построить из чистой функции стрелку Клейсли, пользуясь оператором return?

Определение монады

```
class Applicative m => Monad m where
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
m1 >> m2 = m1 >>= \ _ -> m2
```

Как построить из чистой функции стрелку Клейсли, пользуясь оператором return?

```
toKleisli f = \x -> return (f x)
```

Bind и аппликация

```
(>>=) :: m a -> (a -> m b) -> m b
(<*>) :: f(a -> b) -> f a -> f b
```

Предположим, m, f — тривиальные контейнеры (отсутствуют). Какие λ-термы имеют типы, соответствующие >>=, <*>?

Bind и аппликация

```
(>>=) :: m a -> (a -> m b) -> m b
(<*>) :: f(a -> b) -> f a -> f b
```

Предположим, m, f — тривиальные контейнеры (отсутствуют). Какие λ-термы имеют типы, соответствующие >>=, <*>?

```
($) :: (a -> b) -> a -> b
(&) :: a -> (a -> b) -> b
(&) = flip ($)
> (+1) $ (*3) $ (+2) $ 5
> 5 & (+2) & (*3) & (+1)
```

Applicative Monad Proposal

Монада — класс, дочерний для Applicative (а Applicative — для Functor). Поэтому при определении любой монады придется определять не только return и bind, а еще fmap, pure, app. Стандартный способ определения:

```
import Control.Applicative
import Control.Monad (liftM, ap)
instance Functor MyMonad where
    fmap = liftM

instance Applicative MyMonad where
-- Здесь лучше явно писать определение return
    pure = return
    (<*>) = ap
```

Определение монады

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

Альтернативное определение:

```
join :: M M a -> M a
ap :: M (a -> b) -> M a -> M b
```

В Scala — метод FlatMap, но монада обязательно подразумевает ещё return, а также выполнение законов!

- `return a >>= h` ≡ `h a`
- `m >>= return` ≡ `m`
- `(m >>= g)>>= h` ≡ `m >>= (\x -> g x >>= h)`

Поэтому не все классы с FlatMap — монады (с учётом замыканий с эффектами). См. «монада» Future (поэтому в Haskell IO инкапсулирован в монаду).

Монадическая логика (Lax Logic)

Распределение доступа (в технических системах — распределение задержек).

Правила вывода (+ стандартные правила естественного вывода в минимальной логике):

- $\dfrac{A}{\bigcirc A}$ (Return)
- $\dfrac{A \Rightarrow \bigcirc B}{\bigcirc A \Rightarrow \bigcirc B}$ (Bind)

Сохраняет возможность извлечь монадический оператор из вывода типов! Обратим внимание на инвертированный порядок аргументов.

Пример разбора задачи на монадическое соответствие

Построим оператор `join` из операторов `>>=` и `return`.

$*\ \bigcirc\bigcirc A$ (тип терма x)

(Тут нужно придумать, как объединить контейнеры, оборачивающие x*)*

$\bigcirc A$

$\bigcirc\bigcirc A \Rightarrow \bigcirc A$ (тип терма λx. . . .)

Единственный оператор, умеющий заглядывать внутрь контейнера — это `bind`. Он требует два аргумента: контейнерный типа `M a` и монадическую функцию типа `a -> M b`. Поскольку нужно заглянуть внутрь только одной из двух контейнерных оболочек, в роли типа `a` выступает также монада `M a'`. Значит, нам нужна стрелка типа `M a' -> M a'`. Это функция `id`, вывод которой мы уже умеем строить.

Пример разбора задачи на монадическое соответствие

$* \bigcirc \bigcirc A$ (тип терма x)

$* \bigcirc A$ (тип терма y)

$\bigcirc A$ (просто возвращаем сам y)

$\bigcirc A \Rightarrow \bigcirc A$ (тип терма $\lambda y.y$)

(Аргументы $\mathtt{bind}$*-оператора построены, осталось применить их в правильном порядке)*

$\bigcirc A$

$\bigcirc \bigcirc A \Rightarrow \bigcirc A$ (тип терма $\lambda x. \ldots$)

Пример разбора задачи на монадическое соответствие

$* \bigcirc \bigcirc A$ (тип терма x)

$* \bigcirc A$ (тип терма y)

$\bigcirc A$ (просто возвращаем сам y)

$\bigcirc A \Rightarrow \bigcirc A$ (тип терма $\lambda y.y$)

(Аргументы `bind`*-оператора построены, осталось применить их в правильном порядке)*

$\bigcirc A$ (тип терма `x>>=(\lambda y.y)`)

$\bigcirc \bigcirc A \Rightarrow \bigcirc A$ (тип терма $\lambda x.(x{>>=}(\lambda y.y))$)

Пример разбора задачи на монадическое соответствие

$* \bigcirc \bigcirc A$ (тип терма x)

$* \bigcirc A$ (тип терма y)

$\bigcirc A$ (просто возвращаем сам y)

$\bigcirc A \Rightarrow \bigcirc A$ (тип терма $\lambda y.y$)

(Аргументы `bind`-оператора построены, осталось применить их в правильном порядке)

$\bigcirc A$ (тип терма x>>=$(\lambda y.y)$)

$\bigcirc \bigcirc A \Rightarrow \bigcirc A$ (тип терма $\lambda x.(x$>>=$(\lambda y.y))$)

С помощью построения вывода в модальной логике мы просто решили уравнение в ФВП, получив ответ:
`join x = x >>= id`.

Задачи

- Построить вывод $(A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B$.
- Построить вывод $\bigcirc(A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B$.
- Построить вывод $(A \Rightarrow \bigcirc B) \Rightarrow (B \Rightarrow \bigcirc C) \Rightarrow (A \Rightarrow \bigcirc C)$.

А как определить через них <*>?
Последняя задача — тип т.н. «стрелки Клейсли» (ещё одно альтернативное определение монады).

Задачи

- Построить вывод $(A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B$.
- Построить вывод $\bigcirc(A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B$.
- Построить вывод $(A \Rightarrow \bigcirc B) \Rightarrow (B \Rightarrow \bigcirc C) \Rightarrow (A \Rightarrow \bigcirc C)$.

fmap для монад определяется через bind и return. Как?

```
liftM f m = m >>= \i -> return (f i)
```

А как определить через них <*>?

Последняя задача — тип т.н. «стрелки Клейсли» (ещё одно альтернативное определение монады).

Ещё раз о моделях Крипке

Если время конечно, то в каждом «последнем» мире, если A нет, то A «нет никогда», т.е. выполнено $A \vee \neg A$. Таким образом, каждая классически верная формула верна конструктивно, но только «после второго пришествия».

Что значит «верна в каждом конечном мире»? В модальностях что-то подобное «необходимо, что возможно». Сотрём модальности и получим, что $\Box(\neg(\Box \neg A))$ превратится в $\neg\neg A$.

Теорема Гливенко

Пропозициональная формула Φ доказуема классически $\Leftrightarrow$ $\neg\neg\Phi$ доказуема конструктивно (интуиционистски).

Ещё одна трансформация в монаду: Φ погружается в $(\Phi \Rightarrow R) \Rightarrow R$ (относительное отрицание). Легко проверить, что это именно монада, а не ко-монада.

Шуточное определение

Классика жанра

"Monads are just monoids in the category of endofunctors".

Моноид — полугруппа M с единицей $\langle \circ, \mathrm{id} \rangle$. Типы объектов моноида: $\circ :: \mathrm{M} \times \mathrm{M} \to \mathrm{M}$, $\mathrm{id} :: 1 \to \mathrm{M}$.

Законы моноида:

- $a \circ (b \circ c) = (a \circ b) \circ c$
- $a \circ \mathrm{id} = \mathrm{id} \circ a = a$

Операторы и законы монады

```
fmap :: (a -> b) -> m a -> m b
join :: m m a -> m a
return :: a -> m a
join . fmap join = join . join
join . pure = join . fmap pure = id
```

После η-конверсии join и return приобретают типы $\circ$ и id

Простой пример монады

Зададим контейнер, который только упаковывает значение.

```
-- В фигурных скобках -- имя поля типа
newtype Identity a
        = Identity { runId :: a }
instance Monad Identity where
  return x = Identity x
  Identity x >>= k = k x
```

И стрелки Клейсли для его частного случая:

```
idp x = Identity ((+1) x)
idd x = Identity ((*2) x)
> runId $ idp 3
> runId $ idp 3 >>= idd
> runId $ idp 3 >>= idd >>= idp
```

Простой пример монады

Зададим контейнер, который только упаковывает значение.

```
-- В фигурных скобках -- имя поля типа
newtype Identity a
        = Identity { runId :: a }
instance Monad Identity where
  return x = Identity x
  Identity x >>= k = k x
```

И стрелки Клейсли для его частного случая:

```
idp x = Identity ((+1) x)
idd x = Identity ((*2) x)
> runId $ idp 3
> runId $ idp 3 >>= idd
> runId $ idp 3 >>= idd >>= idp
```

Bind — левоассоциативная операция!

Законы класса монад

```
return a >>= k = k a -- левая единица
m >>= return = m -- правая единица
-- ассоциативность композиции
(m >>= k) >>= k' = m >>= (\x -> k x >>= k')
```

Например, сравним:

```
> runId $ idp 3 >>= idd >>= idp
> runId $ idp 3 >>= (\x -> idd x >>= idp)
```

Последовательное выполнение

```
wrap0 = idp 3 >>= idd >>= idp >>= return

wrap1 = idp 3 >>= (\x ->
    idd x >>= (\y ->
    idp y >>= \z ->
    return z))

-- А можно и так
wrap2 = idp 3 >>= (\x ->
    idd x >>= (\y ->
    idp y >>= \z ->
    return (x,y,z)))
```

Можно использовать обычное let-связывание.

```
wrap3 = let i=3 in idp i >>= (\x ->
    idd x >>= (\y ->
    idp y >>= \z ->
    return (i,x,y,z)))
```

do-нотация

```
do {e} = e
do {e; other} = e >> do {other}
do {p <- e; other} = e >>= \p -> do {other}
do {let v = exp; other} = let v = exp in do {other}
```

Значащие отступы:

```
-- Полная запись
do {let i = 3; x <- idp i; return (i,x)}
-- Упрощенная запись
do
    let i = 3
    x <- idp i
    y <- idd x
    return (i,x,y)
```

Монада Maybe

```
instance Monad Maybe where
  return = Just
  (Just x) >>= k = k x
  Nothing >>= _ = Nothing
  (Just _) >> m = m
  Nothing >> _ = Nothing
```

А как определить join в Maybe?

Монада Maybe

```
instance Monad Maybe where
  return = Just
  (Just x) >>= k = k x
  Nothing >>= _ = Nothing
  (Just _) >> m = m
  Nothing >> _ = Nothing
```

А как определить join в Maybe?
Рассмотрим поиск в ассоциативном списке.

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
> lookup 4 [(1, "f"), (2, "s"), (3, "t")]
```

Работа с монадой Maybe

```
type Name = String
type DataBase = [(Name, Name)]
fathers, mothers :: DataBase
fathers =
  [("Bill","John"),("Ann","John"),("John","Peter")]
mothers =
  [("Bill","Jane"),("Ann","Jane"),("John","Alice"),
  ("Jane", "Dorothy"),("Alice","Mary")]
getM, getF :: Name -> Maybe Name
getM person = lookup person mothers
getF person = lookup person fathers
```

Что делают эти вычисления?

```
> getF "Bill" >>= getM >>= getM
> do { f <- getF "Bill"; m <- getM f; getM m }
```

Список как монада

```
instance Monad [] where
  return x = [x]
  xs >>= k = concat (map k xs)
```

- Что будет, если взять xs >>= k = (map k xs)?
- Как определяется >> в монаде списков?
- Как определить join в монаде списков?

Список как монада

```
instance Monad [] where
  return x = [x]
  xs >>= k = concat (map k xs)
```

- Что будет, если взять xs >>= k = (map k xs)?
- Как определяется >> в монаде списков?
- Как определить join в монаде списков?

Пример применения:

```
do {a <- [1..3]; b <- [a..3]; return (a,b)}
```

Лабораторная работа 3

1. Тип мультимножеств (не обязательно экземпляров класса `Eq`).
   ```
   data Multiset a = [(int, a)]
   ```
2. Тип списков с доступом с двух сторон.
   ```
   data DoubleList a
    = Item a | Cons a (DoubleList a) | Snoc (DoubleList a) a
   ```
3. Тип деревьев вычислений.
   ```
   data EvalTree b a =
    Leaf a | (Node (b -> b -> a) (EvalTree b a) (EvalTree b a))
   ```
4. Тип деревьев ассоциативных вычислений.
   ```
   data EvalATree b a = Leaf a | (Node ([b] -> a) [EvalATree b a])
   ```
5. Тип многомерных списков.
   ```
   data SuperList a
   = List [Either [a] (SuperList a)] | Item [a]
   ```
6. Тип деревьев с накоплением сообщений об ошибочных значениях.
   ```
   data ErrorTree a = Either [a] (AuxTree a)
   data AuxTree a = Leaf a | Branch (ErrorTree a) (ErrorTree a)
   ```
7. Тип чтения в элемент продолжения.
   ```
   data ContReader r d a = ContR ((d -> a) -> r) -> r
   ```