Функциональное программирование

Лекция 5

Степан Львович Кузнецов

НИУ ВШЭ, факультет компьютерных наук

Классы типов

- Классы типов *(type classes)* служат для реализации ad hoc полиморфизма.

Классы типов

- Классы типов *(type classes)* служат для реализации ad hoc полиморфизма.
- Принадлежность типа некоторому классу влечёт необходимость реализации для этого типа функций, свойственных данному классу.

Классы типов

- Классы типов *(type classes)* служат для реализации ad hoc полиморфизма.
- Принадлежность типа некоторому классу влечёт необходимость реализации для этого типа функций, свойственных данному классу.
- При этом, в отличие от классов в ООП, сам класс типов абстрактен, и содержит только объявления функций («методов») класса.

Классы типов

- Классы типов *(type classes)* служат для реализации ad hoc полиморфизма.
- Принадлежность типа некоторому классу влечёт необходимость реализации для этого типа функций, свойственных данному классу.
- При этом, в отличие от классов в ООП, сам класс типов абстрактен, и содержит только объявления функций («методов») класса.
    - Таким образом, класс в смысле ООП будет состоять из класса типов и его конкретной реализации (как правило, алгебраического типа).

Классы типов

- Классы типов *(type classes)* служат для реализации ad hoc полиморфизма.
- Принадлежность типа некоторому классу влечёт необходимость реализации для этого типа функций, свойственных данному классу.
- При этом, в отличие от классов в ООП, сам класс типов абстрактен, и содержит только объявления функций («методов») класса.
    - Таким образом, класс в смысле ООП будет состоять из класса типов и его конкретной реализации (как правило, алгебраического типа).
    - Есть и *наследование* (абстрактных) классов типов: класс-наследник расширяет исходный класс новыми функциями.

Классы типов

- Пример: класс «числовых» типов Num.

```
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in ‘GHC.Num’
```

Классы типов

- Пример: класс «числовых» типов `Num`.

```haskell
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in ‘GHC.Num’
```

  - Следующие типы реализуют этот класс: `Int`, `Integer` (“big int”), `Float`, `Double`

Классы типов

- Пример: класс «числовых» типов `Num`.

```haskell
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in ‘GHC.Num’
```

  - Следующие типы реализуют этот класс: `Int`, `Integer` ("big int"), `Float`, `Double`
- «Тип» для `Num`, равный `* -> Constraint`, это *сорт (kind).*

Классы типов

- Пример: класс «числовых» типов `Num`.

```haskell
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in ‘GHC.Num’
```

  - Следующие типы реализуют этот класс: `Int`, `Integer` (“big int”), `Float`, `Double`
- «Тип» для `Num`, равный `* -> Constraint`, это *сорт (kind)*.
- Если «применить» `Num` к переменной по типам a, то получится ограничение *(constraint)*: `Num a => ...`

Классы типов

- Ограничения учитываются при выведении типов.

Классы типов

- Ограничения учитываются при выведении типов.
- Это происходит одновременно с унификацией: например, если в контексте появляется +, то на соответствующие типы a накладывается ограничение `Num a`.

Классы типов

- Ограничения учитываются при выведении типов.
- Это происходит одновременно с унификацией: например, если в контексте появляется +, то на соответствующие типы a накладывается ограничение `Num a`.
- В простом случае,
  `(\x y -> x + 2 * y) :: Num a => a -> a -> a`
  — это условие на переменную по типам.

Классы типов

- Ограничения учитываются при выведении типов.
- Это происходит одновременно с унификацией: например, если в контексте появляется +, то на соответствующие типы a накладывается ограничение `Num a`.
- В простом случае,
  `(\x y -> x + 2 * y) :: Num a => a -> a -> a`
  — это условие на переменную по типам.
- Однако тип может стать и сложным (за счёт унификации):

```
(\f g x -> (f + g) x)
  :: Num (t1 -> t2) => (t1 -> t2) -> (t1 -> t2) -> t1 -> t2
```

Классы типов

- Ограничения учитываются при выведении типов.
- Это происходит одновременно с унификацией: например, если в контексте появляется +, то на соответствующие типы a накладывается ограничение `Num a`.
- В простом случае,

  ```
  (\x y -> x + 2 * y) :: Num a => a -> a -> a
  ```

  — это условие на переменную по типам.
- Однако тип может стать и сложным (за счёт унификации):

  ```
  (\f g x -> (f + g) x)
    :: Num (t1 -> t2) => (t1 -> t2) -> (t1 -> t2) -> t1 -> t2
  ```

  - Для этого нужна прагма

    ```
    {-# LANGUAGE FlexibleContexts #-}
    ```

Классы типов

- Функция, конечно, не число, но операции можно определить, например, покоординатно:

```haskell
{-# LANGUAGE FlexibleContexts, NoMonomorphismRestriction #-}

instance (Num a, Num b) => Num (a -> b) where
  f + g = \x -> (f x + g x)
  f * g = \x -> (f x * g x)
  abs f = \x -> (abs (f x))
  signum f = \x -> (signum (f x))
  fromInteger n = \x -> fromInteger n
  negate f = \x -> (negate (f x))

fff = (\f g x -> (f + g*g) x)
```

Классы типов

- Функция, конечно, не число, но операции можно определить, например, покоординатно:

```haskell
{-# LANGUAGE FlexibleContexts, NoMonomorphismRestriction #-}

instance (Num a, Num b) => Num (a -> b) where
  f + g = \x -> (f x + g x)
  f * g = \x -> (f x * g x)
  abs f = \x -> (abs (f x))
  signum f = \x -> (signum (f x))
  fromInteger n = \x -> fromInteger n
  negate f = \x -> (negate (f x))

fff = (\f g x -> (f + g*g) x)
```

  - Здесь появляется ещё одна прагма: `NoMonomorphismRestriction`.

Классы типов

- Функция, конечно, не число, но операции можно определить, например, покоординатно:

```haskell
{-# LANGUAGE FlexibleContexts, NoMonomorphismRestriction #-}

instance (Num a, Num b) => Num (a -> b) where
  f + g = \x -> (f x + g x)
  f * g = \x -> (f x * g x)
  abs f = \x -> (abs (f x))
  signum f = \x -> (signum (f x))
  fromInteger n = \x -> fromInteger n
  negate f = \x -> (negate (f x))

fff = (\f g x -> (f + g*g) x)
```

  - Здесь появляется ещё одна прагма: `NoMonomorphismRestriction`.
  - Без неё тип для `fff` будет не полиморфным, а конкретизируется, и при её различном использовании выведение типов не сработает.

Параметрические классы типов

- Бывают и более сложные классы типов.

Параметрические классы типов

- Бывают и более сложные классы типов.
- Рассмотрим классы типов сорта `(* -> *) -> Constraint`

Параметрические классы типов

- Бывают и более сложные классы типов.
- Рассмотрим классы типов сорта `(* -> *) -> Constraint`
- Сорт `* -> *` содержит *однопараметрические конструкторы типов.*

Параметрические классы типов

- Бывают и более сложные классы типов.
- Рассмотрим классы типов сорта (`* -> *`) `-> Constraint`
- Сорт `* -> *` содержит *однопараметрические конструкторы типов.*
- Например, таковы уже известный нам `Maybe` и конструктор типа списков `[ ]`: каждый из них по типу-аргументу строит новый тип.

Параметрические классы типов

- Бывают и более сложные классы типов.
- Рассмотрим классы типов сорта `(* -> *) -> Constraint`
- Сорт `* -> *` содержит *однопараметрические конструкторы типов.*
- Например, таковы уже известный нам `Maybe` и конструктор типа списков `[ ]`: каждый из них по типу-аргументу строит новый тип.
    - Заметим, что конструктор типов (например, `Maybe`) не следует путать с конструктором объектов алгебраического типа, таким как `Just`. Последний имеет тип (а не сорт) `a -> Maybe a`

Параметрические классы типов

- Бывают и более сложные классы типов.
- Рассмотрим классы типов сорта `(* -> *) -> Constraint`
- Сорт `* -> *` содержит *однопараметрические конструкторы типов.*
- Например, таковы уже известный нам `Maybe` и конструктор типа списков `[ ]`: каждый из них по типу-аргументу строит новый тип.
    - Заметим, что конструктор типов (например, `Maybe`) не следует путать с конструктором объектов алгебраического типа, таким как `Just`. Последний имеет тип (а не сорт) `a -> Maybe a`
    - Сложные сорта содержат не только алгебраические типы: например, `(->) :: * -> * -> *`

Параметрические классы типов

- Бывают и более сложные классы типов.
- Рассмотрим классы типов сорта `(* -> *) -> Constraint`
- Сорт `* -> *` содержит *однопараметрические конструкторы типов.*
- Например, таковы уже известный нам `Maybe` и конструктор типа списков `[ ]`: каждый из них по типу-аргументу строит новый тип.
  - Заметим, что конструктор типов (например, `Maybe`) не следует путать с конструктором объектов алгебраического типа, таким как `Just`. Последний имеет тип (а не сорт) `a -> Maybe a`
  - Сложные сорта содержат не только алгебраические типы: например, `(->) :: * -> * -> *`
- В `(* -> *) -> Constraint` тоже живут классы типов.

Параметрические классы типов

- Например, класс типов `Foldable` включает однопараметрические типы, для которых определено «сворачивание»:

```
type Foldable :: (* -> *) -> Constraint
class Foldable t where
  ...
  foldl :: (b -> a -> b) -> b -> t a -> b
  ...
```

Параметрические классы типов

- Например, класс типов `Foldable` включает однопараметрические типы, для которых определено «сворачивание»:

```
type Foldable :: (* -> *) -> Constraint
class Foldable t where
  ...
  foldl :: (b -> a -> b) -> b -> t a -> b
  ...
```

- Естественная реализация (instance) класса `Foldable` — это тип списков `[a]`.

Параметрические классы типов

- Например, класс типов `Foldable` включает однопараметрические типы, для которых определено «сворачивание»:

```
type Foldable :: (* -> *) -> Constraint
class Foldable t where
  ...
  foldl :: (b -> a -> b) -> b -> t a -> b
  ...
```

- Естественная реализация (instance) класса `Foldable` — это тип списков `[a]`.
- Здесь `foldl op w [x,y,z]` означает `((w `op` x) `op` y) `op` z`:

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x in foldl f z' xs
```

Параметрические классы типов

- Например, класс типов `Foldable` включает однопараметрические типы, для которых определено «сворачивание»:

```
type Foldable :: (* -> *) -> Constraint
class Foldable t where
  ...
  foldl :: (b -> a -> b) -> b -> t a -> b
  ...
```

- Естественная реализация (instance) класса `Foldable` — это тип списков `[a]`.
- Здесь `foldl op w [x,y,z]` означает `((w `op` x) `op` y) `op` z`:

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x in foldl f z' xs
```

- Однако `Maybe` также `Foldable` (угадайте, как там работает `foldl`).

- Систему типов можно представить как очень большой граф, где вершины — типы, а стрелки (рёбра) — функции между ними.

«Категория типов»

- Систему типов можно представить как очень большой граф, где вершины — типы, а стрелки (рёбра) — функции между ними.
    - Этот граф ориентированный, в нём есть петли и кратные рёбра.

«Категория типов»

- Систему типов можно представить как очень большой граф, где вершины — типы, а стрелки (рёбра) — функции между ними.
    - Этот граф ориентированный, в нём есть петли и кратные рёбра.
- В математике такие графы, с естественными условиями на стрелки (композиция, существование петли-«единицы»), называются **категориями.**

«Категория типов»

- Систему типов можно представить как очень большой граф, где вершины — типы, а стрелки (рёбра) — функции между ними.
    - Этот граф ориентированный, в нём есть петли и кратные рёбра.
- В математике такие графы, с естественными условиями на стрелки (композиция, существование петли-«единицы»), называются **категориями.**
- Вершины называются *объектами* категории, а стрелки — *морфизмами.*

«Категория типов»

- В Haskell’е система типов содержит функциональные («стрельчатые») типы, т.е. совокупность морфизмов между двумя объектами сама представляется как объект.

«Категория типов»

- В Haskell'е система типов содержит функциональные («стрельчатые») типы, т.е. совокупность морфизмов между двумя объектами сама представляется как объект.
- Категории с таким свойством называются **декартово замкнутыми.**

«Категория типов»

- В Haskell'е система типов содержит функциональные («стрельчатые») типы, т.е. совокупность морфизмов между двумя объектами сама представляется как объект.
- Категории с таким свойством называются **декартово замкнутыми.**
- Ближайший пример из математики: категория SET множеств и функций между ними.

«Категория типов»

- В Haskell'е система типов содержит функциональные («стрельчатые») типы, т.е. совокупность морфизмов между двумя объектами сама представляется как объект.
- Категории с таким свойством называются **декартово замкнутыми.**
- Ближайший пример из математики: категория SET множеств и функций между ними.
- Огрубляя, можно мыслить типы как множества, но есть и тонкие отличия.

«Категория типов»

- В Haskell'е система типов содержит функциональные («стрельчатые») типы, т.е. совокупность морфизмов между двумя объектами сама представляется как объект.
- Категории с таким свойством называются **декартово замкнутыми.**
- Ближайший пример из математики: категория SET множеств и функций между ними.
- Огрубляя, можно мыслить типы как множества, но есть и тонкие отличия.
    - Например, объединение $A \cup B$ множеств содержит как подмножества A и B, а для его ближайшего аналога в системе типов, конструкции `Either`, имеются только функции-вложения `Left` и `Right`.

«Категория типов»

- Из теории категорий «импортированы» некоторые идеи в функциональном программировании, о которых пойдёт речь дальше.

«Категория типов»

- Из теории категорий «импортированы» некоторые идеи в функциональном программировании, о которых пойдёт речь дальше.
- Аккуратное изложение самой теории категорий целью этого курса не является.

«Категория типов»

- Из теории категорий «импортированы» некоторые идеи в функциональном программировании, о которых пойдёт речь дальше.
- Аккуратное изложение самой теории категорий целью этого курса не является.
- Нам достаточно будет категорного взгляда «с высоты птичьего полёта», при котором большие и сложные типы представляются точками.

- Из теории категорий «импортированы» некоторые идеи в функциональном программировании, о которых пойдёт речь дальше.
- Аккуратное изложение самой теории категорий целью этого курса не является.
- Нам достаточно будет категорного взгляда «с высоты птичьего полёта», при котором большие и сложные типы представляются точками.
- Итак, теоретико-категорные конструкции *мотивируют* конструкции в системе типов, в т.ч. *монады.*

Функторы

- Сами категории можно также воспринимать как алгебраические структуры, и между ними можно строить «морфизмы».

Функторы

- Сами категории можно также воспринимать как алгебраические структуры, и между ними можно строить «морфизмы».
- Такие «морфизмы категорий» называются *функторами.*

Функторы

- Сами категории можно также воспринимать как алгебраические структуры, и между ними можно строить «морфизмы».
- Такие «морфизмы категорий» называются *функторами.*
- Функтор $F : \mathscr{C} \Rightarrow \mathscr{D}$ отображает объекты в объекты, морфизмы в морфизмы, при этом сохраняя начало/конец:

$$\begin{tikzcd} A \arrow[r, Rightarrow, "F"] \arrow[d, "f"'] & FA \arrow[d, "Ff"] \\ B \arrow[r, Rightarrow, "F"] & FB \end{tikzcd}$$

Функторы

- Сами категории можно также воспринимать как алгебраические структуры, и между ними можно строить «морфизмы».
- Такие «морфизмы категорий» называются *функторами.*
- Функтор $F : \mathscr{C} \Rightarrow \mathscr{D}$ отображает объекты в объекты, морфизмы в морфизмы, при этом сохраняя начало/конец:

$$\begin{tikzcd} A \arrow[r, Rightarrow, "F"] \arrow[d, "f"'] & FA \arrow[d, "Ff"] \\ B \arrow[r, Rightarrow, "F"] & FB \end{tikzcd}$$

- При этом функтор «уважает» композицию и единицу: $F(f \circ g) = (Ff) \circ (Fg)$ и $F\mathbf{1}_A = \mathbf{1}_{FA}$.

Функторы

- Кабы не логические парадоксы, можно было бы говорить о «категории всех категорий», где морфизмы — это функторы.

Функторы

- Кабы не логические парадоксы, можно было бы говорить о «категории всех категорий», где морфизмы — это функторы.
- У нас одна категория — абстрактная категория, приближённо описывающая систему типов языка.

Функторы

- Кабы не логические парадоксы, можно было бы говорить о «категории всех категорий», где морфизмы — это функторы.
- У нас одна категория — абстрактная категория, приближённо описывающая систему типов языка.
- Поэтому нас будут интересовать *эндофункторы* $F : \mathscr{C} \Rightarrow \mathscr{C}$ (т.е. действующие внутри одной категории).

Функторы

- Кабы не логические парадоксы, можно было бы говорить о «категории всех категорий», где морфизмы — это функторы.
- У нас одна категория — абстрактная категория, приближённо описывающая систему типов языка.
- Поэтому нас будут интересовать *эндофункторы* $F : \mathscr{C} \Rightarrow \mathscr{C}$ (т.е. действующие внутри одной категории).
- С точки зрения системы типов (эндо)функтор должен реализоваться как *преобразование типов* и связанная с ним *функция на функциях.*

Функторы

- Пример: взятие списка элементов данного типа, a ⇒ [a].

Функторы

- Пример: взятие списка элементов данного типа, a ⇒ [a].
- Соответствующее преобразование на функциях
  ```
  map :: (a -> b) -> [a] -> [b]
  ```

Функторы

- Пример: взятие списка элементов данного типа, a ⇒ `[a]`.
- Соответствующее преобразование на функциях
  `map :: (a -> b) -> [a] -> [b]`
- Близкая математическая конструкций — функтор $\mathscr{P}$ в категории SET, сопоставляющий множеству множество всех его подмножеств.

Функторы

- Пример: взятие списка элементов данного типа, `a` ⇒ `[a]`.
- Соответствующее преобразование на функциях
  `map :: (a -> b) -> [a] -> [b]`
- Близкая математическая конструкций — функтор $\mathscr{P}$ в категории SET, сопоставляющий множеству множество всех его подмножеств.
  - Для $f : A \to B$ и $X \subseteq A$ имеем $\mathscr{P} f : X \mapsto \{f(x) \mid x \in X\}$.

Функторы

- Пример: взятие списка элементов данного типа, `a` ⇒ `[a]`.
- Соответствующее преобразование на функциях
  `map :: (a -> b) -> [a] -> [b]`
- Близкая математическая конструкций — функтор $\mathscr{P}$ в категории SET, сопоставляющий множеству множество всех его подмножеств.
  - Для $f : A \to B$ и $X \subseteq A$ имеем $\mathscr{P} f : X \mapsto \{f(x) \mid x \in X\}$.
  - Можно рассмотреть $\mathscr{P}_{\text{fin}}$ — взятие множества конечных подмножеств.

Функторы

- Пример: взятие списка элементов данного типа, a $\Rightarrow$ [a].
- Соответствующее преобразование на функциях
  `map :: (a -> b) -> [a] -> [b]`
- Близкая математическая конструкций — функтор $\mathscr{P}$ в категории SET, сопоставляющий множеству множество всех его подмножеств.
  - Для $f : A \to B$ и $X \subseteq A$ имеем $\mathscr{P} f : X \mapsto \{f(x) \mid x \in X\}$.
  - Можно рассмотреть $\mathscr{P}_{\mathrm{fin}}$ — взятие множества конечных подмножеств.
  - Кстати, список в Haskell'е не обязательно конечный.

Функторы

- Пример: взятие списка элементов данного типа, a ⇒ [a].
- Соответствующее преобразование на функциях
  `map :: (a -> b) -> [a] -> [b]`
- Близкая математическая конструкций — функтор $\mathscr{P}$ в категории SET, сопоставляющий множеству множество всех его подмножеств.
  - Для $f: A \to B$ и $X \subseteq A$ имеем $\mathscr{P}f: X \mapsto \{f(x) \mid x \in X\}$.
  - Можно рассмотреть $\mathscr{P}_{\text{fin}}$ — взятие множества конечных подмножеств.
  - Кстати, список в Haskell'е не обязательно конечный.
  - Отличие множества от списка — во множестве не имеет значения порядок и кратность элементов.

Функторы

- Пример: взятие списка элементов данного типа, a ⇒ [a].
- Соответствующее преобразование на функциях

  ```haskell
  map :: (a -> b) -> [a] -> [b]
  ```

- Близкая математическая конструкций — функтор $\mathscr{P}$ в категории SET, сопоставляющий множеству множество всех его подмножеств.
  - Для $f : A \to B$ и $X \subseteq A$ имеем $\mathscr{P} f : X \mapsto \{ f(x) \mid x \in X \}$.
  - Можно рассмотреть $\mathscr{P}_{\mathrm{fin}}$ — взятие множества конечных подмножеств.
  - Кстати, список в Haskell'е не обязательно конечный.
  - Отличие множества от списка — во множестве не имеет значения порядок и кратность элементов.
  - К функтору $\mathscr{P}_{\mathrm{fin}}$ мы ещё вернёмся.

Функторы

- В общем случае понятие функтора реализуется как параметрический класс типов `Functor`

```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Функторы

- В общем случае понятие функтора реализуется как параметрический класс типов `Functor`

```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- При этом реализация этого класса — это не обязательно алгебраический тип, т.е. `f a` не обязательно является «новым» типом, построенным из `a`.

Функторы

- В общем случае понятие функтора реализуется как параметрический класс типов `Functor`

```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- При этом реализация этого класса — это не обязательно алгебраический тип, т.е. `f a` не обязательно является «новым» типом, построенным из `a`.
- Пример:

```
instance Functor ((->) r)
```

Функторы

- В общем случае понятие функтора реализуется как параметрический класс типов `Functor`
```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```
- При этом реализация этого класса — это не обязательно алгебраический тип, т.е. `f a` не обязательно является «новым» типом, построенным из a.
- Пример:
```
instance Functor ((->) r)
```
- Этот функтор преобразует тип a в `r -> a` (r — фиксированный тип-параметр).

Функторы

- В общем случае понятие функтора реализуется как параметрический класс типов `Functor`
```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```
- При этом реализация этого класса — это не обязательно алгебраический тип, т.е. `f a` не обязательно является «новым» типом, построенным из a.
- Пример:
```
instance Functor ((->) r)
```
- Этот функтор преобразует тип a в `r -> a` (r — фиксированный тип-параметр).
  - Здесь `fmap h = \g z -> h (g z)`

Функторы

- В общем случае понятие функтора реализуется как параметрический класс типов `Functor`

```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- При этом реализация этого класса — это не обязательно алгебраический тип, т.е. `f a` не обязательно является «новым» типом, построенным из `a`.
- Пример:

```
instance Functor ((->) r)
```

- Этот функтор преобразует тип `a` в `r -> a` (`r` — фиксированный тип-параметр).
  - Здесь `fmap h = \g z -> h (g z)`
  - `a` ⇒ `a -> r` — это *контравариантный функтор* (переворачивает стрелки).

Условия функториальности

- От типа, реализующего класс `Functor`, требуется соблюдение условий функториальности: $F(f \circ g) = (Ff) \circ (Fg)$ и $F\mathbf{1}_A = \mathbf{1}_{FA}$.

Условия функториальности

- От типа, реализующего класс `Functor`, требуется соблюдение условий функториальности: $F(f \circ g) = (Ff) \circ (Fg)$ и $F\mathbf{1}_A = \mathbf{1}_{FA}$.
- В терминах `fmap`: `fmap (f . g) = (fmap f) . (fmap g)` и `fmap (\x -> x) = \x -> x`.

Условия функториальности

- От типа, реализующего класс `Functor`, требуется соблюдение условий функториальности: $F(f \circ g) = (Ff) \circ (Fg)$ и $F\mathbf{1}_A = \mathbf{1}_{FA}$.
- В терминах fmap: `fmap (f . g) = (fmap f) . (fmap g)` и `fmap (\x -> x) = \x -> x`.
- При работе с типом класса `Functor` можно предполагать, что эти равенства выполнены, однако, к сожалению, они не *верифицируются* в самом Haskell'е.

Условия функториальности

- От типа, реализующего класс `Functor`, требуется соблюдение условий функториальности: $F(f \circ g) = (Ff) \circ (Fg)$ и $F\mathbf{1}_A = \mathbf{1}_{FA}$.
- В терминах `fmap`:
  `fmap (f . g) = (fmap f) . (fmap g)` и
  `fmap (\x -> x) = \x -> x`.
- При работе с типом класса `Functor` можно предполагать, что эти равенства выполнены, однако, к сожалению, они не *верифицируются* в самом Haskell'е.
- Например, для списков можно определить альтернативный `fmap' = \f -> reverse . (map f)`, для которого эти равенства неверны.

Функтор

- Вернёмся к примеру a ⇒ `r -> a`. «Внутри» `r -> a` находится бесконечно много элементов типа a, по одному для каждого `z :: r`.

Функтор

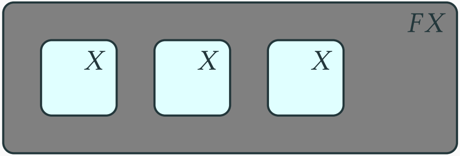
- Вернёмся к примеру a ⇒ `r -> a`. «Внутри» `r -> a` находится бесконечно много элементов типа a, по одному для каждого `z :: r`.
- То же происходит для функтора взятия списка или множества.

Функтор

- Вернёмся к примеру a ⇒ `r -> a`. «Внутри» `r -> a` находится бесконечно много элементов типа a, по одному для каждого `z :: r`.
- То же происходит для функтора взятия списка или множества.
- Уже знакомый нам конструктор типов `Maybe` также является функтором, и в `Maybe a` лежит либо один элемент, либо ни одного элемента типа a.

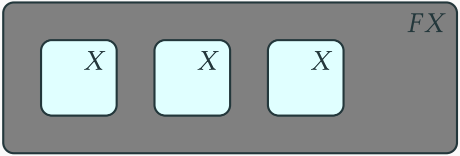
«Чёрный ящик»

- Таким образом, если X — тип, а F — функтор, то FX — это «чёрный ящик», в котором каким-то образом спрятаны элементы типа X, к которым можно применить функцию $f : X \to Y$.



«Чёрный ящик»

- Таким образом, если X — тип, а F — функтор, то FX — это «чёрный ящик», в котором каким-то образом спрятаны элементы типа X, к которым можно применить функцию $f: X \to Y$.



- Однако чего-то не хватает: непонятно, как «положить» что-то в FX, и вообще, как создать объект типа FX.

Монада

- *Монада M* — это эндофунктор с дополнительными операциями.

Монада

- *Монада* M — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.

Монада

- *Монада M* — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
    - `return :: Monad m => a -> m a`

Монада

- *Монада* M — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
    - `return :: Monad m => a -> m a`
- Вторая более сложная и позволяет «поднимать» аргумент функции, ведущей в монаду. Для $f : X \to MY$ эта операция даёт $f^* : MX \to MY$.

Монада

- *Монада M* — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
    - `return :: Monad m => a -> m a`
- Вторая более сложная и позволяет «поднимать» аргумент функции, ведущей в монаду. Для $f : X \to MY$ эта операция даёт $f^* : MX \to MY$.
    - `(>>=) :: Monad m => m a -> (a -> m b) -> m b`

Монада

- *Монада M* — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
    - `return :: Monad m => a -> m a`
- Вторая более сложная и позволяет «поднимать» аргумент функции, ведущей в монаду. Для $f : X \to MY$ эта операция даёт $f^* : MX \to MY$.
    - `(>>=) :: Monad m => m a -> (a -> m b) -> m b`
- При этом должны выполняться определённые условия, о которых чуть позже.

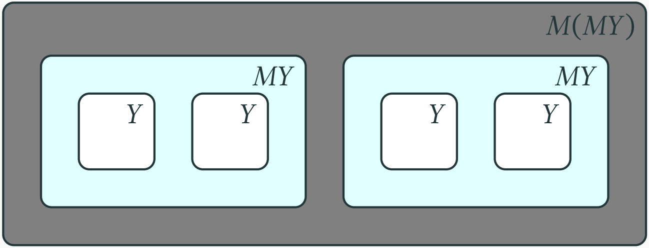
Монада

- *Монада M* — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
    - `return :: Monad m => a -> m a`
- Вторая более сложная и позволяет «поднимать» аргумент функции, ведущей в монаду. Для $f : X \to MY$ эта операция даёт $f^* : MX \to MY$.
    - `(>>=) :: Monad m => m a -> (a -> m b) -> m b`
- При этом должны выполняться определённые условия, о которых чуть позже.
- Набор $(M, \eta, {}^*)$ в теории категорий называется *тройкой Клейсли*, а собственно монадой — другая, но эквивалентная конструкция.

- Операция *, или `>>=`, соответствует идее, что можно внутри монады войти ещё раз в монаду, и остаться в той же монаде.

- Операция *, или `>>=`, соответствует идее, что можно внутри монады войти ещё раз в монаду, и остаться в той же монаде.
- Пример: `[1,2,3] >>= (\x -> [x,x])` даёт `[1,1,2,2,3,3]`

- Операция *, или >>=, соответствует идее, что можно внутри монады войти ещё раз в монаду, и остаться в той же монаде.
- Пример: `[1,2,3] >>= (\x -> [x,x])` даёт `[1,1,2,2,3,3]`
- Если бы вместо f^* применили M как функтор, то получилось бы $Mf : MX \to M(MY)$.



- Таким образом, f^* можно свести к более простой операции «разглаживания» двойной монады в одинарную, $\mu_X : M(MY) \to MY$.

- Таким образом, f^* можно свести к более простой операции «разглаживания» двойной монады в одинарную, $\mu_X : M(MY) \to MY$.
- Тогда $f^* = \mu \circ (Mf)$.

- Таким образом, f^* можно свести к более простой операции «разглаживания» двойной монады в одинарную, $\mu_X : M(MY) \to MY$.
- Тогда $f^* = \mu \circ (Mf)$.
- Собственно, в теории категорий именно эндофунктор M, оснащённый семействами морфизмов η и μ, удовлетворяющий огромному количеству условий корректности, и называют монадой.

Условия монады

Для тройки Клейсли $(M, \eta, {}^*)$ условия формулируются намного короче:

$\eta_X^* = \mathbf{1}_{MX}$

$MX \to MX$

```
(z >>= return) = z
```

$f^* \circ \eta_X = f$

$X \xrightarrow{\eta} MX \xrightarrow{f^*} MY$

```
(return x >>= f) = (f x)
```

$g^* \circ f^* = (g^* \circ f)^*$

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

$MX \xrightarrow{f^*} MY \xrightarrow{g^*} MZ$

$X \xrightarrow{f} MY \xrightarrow{g^*} MZ$

Выражение остальных операций

- Через η и * можно выразить всё остальное.

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}^*_{MY}$.

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}^*_{MY}$.
    - `join = \x -> (x >>= (\y -> y))`

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}^*_{MY}$.
    - `join = \x -> (x >>= (\y -> y))`
- Преобразование на морфизмах (функтор), $Mf : MX \to MY$ (для $f : X \to Y$) также выражается: $Mf = (\eta_Y \circ f)^*$.

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}^*_{MY}$.
    - `join = \x -> (x >>= (\y -> y))`
- Преобразование на морфизмах (функтор), $Mf : MX \to MY$ (для $f : X \to Y$) также выражается: $Mf = (\eta_Y \circ f)^*$.
    - `fmap = (\f -> (\z -> (z >>= \x -> return (f x))))`

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}^*_{MY}$.
    - `join = \x -> (x >>= (\y -> y))`
- Преобразование на морфизмах (функтор), $Mf : MX \to MY$ (для $f : X \to Y$) также выражается: $Mf = (\eta_Y \circ f)^*$.
    - `fmap = (\f -> (\z -> (z >>= \x -> return (f x))))`
    - При этом $Mg \circ Mf = (\eta_Z \circ g)^* \circ (\eta_Y \circ f)^* = ((\eta_Z \circ g)^* \circ \eta_Y \circ f)^* = (\eta_Z \circ g \circ f)^* = M(g \circ f)$.

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}^*_{MY}$.
  - `join = \x -> (x >>= (\y -> y))`
- Преобразование на морфизмах (функтор), $Mf : MX \to MY$ (для $f : X \to Y$) также выражается: $Mf = (\eta_Y \circ f)^*$.
  - `fmap = (\f -> (\z -> (z >>= \x -> return (f x))))`
  - При этом $Mg \circ Mf = (\eta_Z \circ g)^* \circ (\eta_Y \circ f)^* = ((\eta_Z \circ g)^* \circ \eta_Y \circ f)^* = (\eta_Z \circ g \circ f)^* = M(g \circ f)$.
  - Таким образом, в определении монады не нужно требовать, что это функтор.

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}_{MY}^*$.
  - `join = \x -> (x >>= (\y -> y))`
- Преобразование на морфизмах (функтор), $Mf : MX \to MY$ (для $f : X \to Y$) также выражается: $Mf = (\eta_Y \circ f)^*$.
  - `fmap = (\f -> (\z -> (z >>= \x -> return (f x))))`
  - При этом $Mg \circ Mf = (\eta_Z \circ g)^* \circ (\eta_Y \circ f)^* = ((\eta_Z \circ g)^* \circ \eta_Y \circ f)^* = (\eta_Z \circ g \circ f)^* = M(g \circ f)$.
  - Таким образом, в определении монады не нужно требовать, что это функтор.
- Более того, каждая монада — это *аппликативный функтор* с операцией применения функции внутри монады: `(<*>) :: m (a -> b) -> m a -> m b`

Выражение остальных операций

- Через η и * можно выразить всё остальное.
- Так, $\mu : M(MY) \to MY$, или `join`, выражается так: $\mu = \mathbf{1}^*_{MY}$.
    - `join = \x -> (x >>= (\y -> y))`
- Преобразование на морфизмах (функтор), $Mf : MX \to MY$ (для $f : X \to Y$) также выражается: $Mf = (\eta_Y \circ f)^*$.
    - `fmap = (\f -> (\z -> (z >>= \x -> return (f x))))`
    - При этом $Mg \circ Mf = (\eta_Z \circ g)^* \circ (\eta_Y \circ f)^* = ((\eta_Z \circ g)^* \circ \eta_Y \circ f)^* = (\eta_Z \circ g \circ f)^* = M(g \circ f)$.
    - Таким образом, в определении монады не нужно требовать, что это функтор.
- Более того, каждая монада — это *аппликативный функтор* с операцией применения функции внутри монады:
  `(<*>) :: m (a -> b) -> m a -> m b`
    - `h <*> t = h >>= \f -> fmap f t`

Монада IO

- При вычислении последовательности функций «внутри» монады сама монада каждый раз заменяется на новую.

Монада IO

- При вычислении последовательности функций «внутри» монады сама монада каждый раз заменяется на новую.
- Таким образом поддерживается порядок вычислений, что приближает работу в монаде к императивному языку.

Монада IO

- При вычислении последовательности функций «внутри» монады сама монада каждый раз заменяется на новую.
- Таким образом поддерживается порядок вычислений, что приближает работу в монаде к императивному языку.
    - Явно это выражается в т.н. do-нотации, о которой чуть позже.

Монада IO

- При вычислении последовательности функций «внутри» монады сама монада каждый раз заменяется на новую.
- Таким образом поддерживается порядок вычислений, что приближает работу в монаде к императивному языку.
    - Явно это выражается в т.н. do-нотации, о которой чуть позже.
- В частности, с помощью специальной монады можно реализовать взаимодействие с «внешним миром» (побочные эффекты вычислений), в частности, ввод-вывод.

Монада IO

- При вычислении последовательности функций «внутри» монады сама монада каждый раз заменяется на новую.
- Таким образом поддерживается порядок вычислений, что приближает работу в монаде к императивному языку.
    - Явно это выражается в т.н. do-нотации, о которой чуть позже.
- В частности, с помощью специальной монады можно реализовать взаимодействие с «внешним миром» (побочные эффекты вычислений), в частности, ввод-вывод.
- Эта специальная монада называется `IO`.

Монада IO

- Объект типа `IO a` можно понимать как объект типа `a`, помещённый в большой и страшный внешний мир.

Монада IO

- Объект типа `IO a` можно понимать как объект типа `a`, помещённый в большой и страшный внешний мир.

«Ёжик в тумане», Союзмультфильм

Монада IO

- Грубое приближение монады `IO` — это взятие пары с контекстом: “(a, `RealWorld`)”. При операциях с монадой состояние `RealWorld` может меняться.

Монада IO

- Грубое приближение монады `IO` — это взятие пары с контекстом: “(a, `RealWorld`)”. При операциях с монадой состояние `RealWorld` может меняться.
- Есть операция `return`, погружающая объект в окружение `IO` («выпускающая во внешний мир»), а вот обратного преобразования нет.

Монада IO

- Грубое приближение монады `IO` — это взятие пары с контекстом: “(a, `RealWorld`)”. При операциях с монадой состояние `RealWorld` может меняться.
- Есть операция `return`, погружающая объект в окружение `IO` («выпускающая во внешний мир»), а вот обратного преобразования нет.
- `putChar :: Char -> IO ()` берёт символ и возвращает новый «мир», в котором растворился (был напечатан в консоли) этот символ.

Монада IO

- Грубое приближение монады `IO` — это взятие пары с контекстом: “(a, `RealWorld`)”. При операциях с монадой состояние `RealWorld` может меняться.
- Есть операция `return`, погружающая объект в окружение `IO` («выпускающая во внешний мир»), а вот обратного преобразования нет.
- `putChar :: Char -> IO ()` берёт символ и возвращает новый «мир», в котором растворился (был напечатан в консоли) этот символ.
- `getChar :: IO Char` — мы можем получить символ из внешнего мира, но только внутри монады.

Монада IO

- Грубое приближение монады `IO` — это взятие пары с контекстом: “(a, `RealWorld`)”. При операциях с монадой состояние `RealWorld` может меняться.
- Есть операция `return`, погружающая объект в окружение `IO` («выпускающая во внешний мир»), а вот обратного преобразования нет.
- `putChar :: Char -> IO ()` берёт символ и возвращает новый «мир», в котором растворился (был напечатан в консоли) этот символ.
- `getChar :: IO Char` — мы можем получить символ из внешнего мира, но только внутри монады.
- Достать его из монады мы не можем, но можем работать с ним внутри `IO` с помощью `>>=`.

Монада IO

- Например: `getChar >>= (\x -> (putChar x >> putChar x))`

Монада IO

- Например: `getChar >>= (\x -> (putChar x >> putChar x))`
- Здесь `>>` — это версия `>>=`, игнорирующая аргумент (`putChar` всё равно ничего не выдаёт, а настоящий `x` ранее абстрагирован).

Монада IO

- Например: `getChar >>= (\x -> (putChar x >> putChar x))`
- Здесь `>>` — это версия `>>=`, игнорирующая аргумент (`putChar` всё равно ничего не выдаёт, а настоящий `x` ранее абстрагирован).
- В процессе выполнения программы, содержащей `IO`, объекты типов `IO a` остаются временно невычисленными, как задумки.

Монада IO

- Например: `getChar >>= (\x -> (putChar x >> putChar x))`
- Здесь `>>` — это версия `>>=`, игнорирующая аргумент (`putChar` всё равно ничего не выдаёт, а настоящий x ранее абстрагирован).
- В процессе выполнения программы, содержащей `IO`, объекты типов `IO a` остаются временно невычисленными, как задумки.
- Например, если мы где-то напишем `putChar 'a'`, то символ не будет тут же напечатан.

Монада IO

- Например: `getChar >>= (\x -> (putChar x >> putChar x))`
- Здесь `>>` — это версия `>>=`, игнорирующая аргумент (`putChar` всё равно ничего не выдаёт, а настоящий x ранее абстрагирован).
- В процессе выполнения программы, содержащей `IO`, объекты типов `IO a` остаются временно невычисленными, как задумки.
- Например, если мы где-то напишем `putChar 'a'`, то символ не будет тут же напечатан.
- Вместо этого нужно дождаться, пока соберётся «главный» объект типа `IO ()`, и уже при его вычислении все операции с внешним миром будут выполнены, причём в правильном порядке.

do-нотация

- *do-нотация* — это альтернативный синтаксис работы с `>>=` и `>>`, делающий код похожим на императивный.

do-нотация

- *do-нотация* — это альтернативный синтаксис работы с `>>=` и `>>`, делающий код похожим на императивный.
- Пример:

```haskell
main :: IO ()
main = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

do-нотация

- *do-нотация* — это альтернативный синтаксис работы с `>>=` и `>>`, делающий код похожим на императивный.
- Пример:

```haskell
main :: IO ()
main = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

- do-нотация раскрывается так. «Команды», у которых нет возвращаемого значения, соединяются с помощью `>>`. Если возвращаемое значение есть: `x <- ...`, то пишется `... >>= \x -> ...`

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.

do-нотация

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.
- Более того, в «присваивании» `<-` можно (как в императивных языках) использовать одно и то же имя несколько раз.

do-нотация

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.
- Более того, в «присваивании» `<-` можно (как в императивных языках) использовать одно и то же имя несколько раз.
    - При этом более раннее забывается, поскольку переменная связана более глубокой лямбдой: $\lambda x.(\ldots\lambda x.(\ldots x\ldots)\ldots)$.

do-нотация

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.
- Более того, в «присваивании» `<-` можно (как в императивных языках) использовать одно и то же имя несколько раз.
    - При этом более раннее забывается, поскольку переменная связана более глубокой лямбдой: $\lambda x.(\ldots\lambda x.(\ldots x\ldots)\ldots)$.
- Пример:

```
main =
  putStrLn "What's your name?" >>
  getLine >>=
  \name -> putStrLn $ "Hello, " ++ name ++ "!"
```

do-нотация

- do-нотация работает не только с `IO`, но и с любой другой монадой.

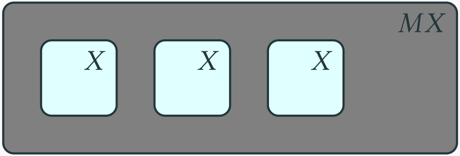
do-нотация

- do-нотация работает не только с `IO`, но и с любой другой монадой.
- Например, вот такой код рекурсивно генерирует все булевы наборы данной длины:

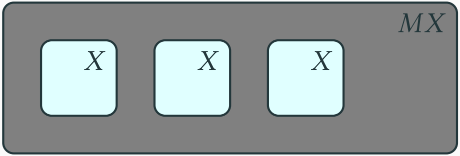
```
allVals 0 = [[]]
allVals n = do
  v <- allVals (n-1)
  [True:v, False:v]
```

Монады

- Итак, *монада* — это абстрактная конструкция преобразования типов: X преобразуется в MX, при этом внутри «монадического» объекта типа MX в некотором «живут» элементы исходного типа X:



- Итак, *монада* — это абстрактная конструкция преобразования типов: X преобразуется в MX, при этом внутри «монадического» объекта типа MX в некотором «живут» элементы исходного типа X:



- Внутри монады можно применять функцию к исходному типу — $f : X \to Y$ «поднимается» до $Mf : MX \to MY$, т.е. монада является функтором.

Монады

- Более того, f сама может создавать монадический объект, и этот объект помещается в исходную монаду.

Монады

- Более того, f сама может создавать монадический объект, и этот объект помещается в исходную монаду.
- А именно, $f : X \to MY$ даёт $f^* : MX \to MY$. Другое обозначение: $f^*(t) = (t \mathbin{\texttt{>>=}} f)$.

Монады

- Более того, f сама может создавать монадический объект, и этот объект помещается в исходную монаду.
- А именно, $f : X \to MY$ даёт $f^* : MX \to MY$. Другое обозначение: $f^*(t) = (t \mathbin{\texttt{>>=}} f)$.
- Можно определить f^* через операцию «разглаживания двойной монады» $\mu : MMY \to MY$, а именно, $f^* = \mu \circ f$.

Монады

- Более того, f сама может создавать монадический объект, и этот объект помещается в исходную монаду.
- А именно, $f : X \to MY$ даёт $f^* : MX \to MY$. Другое обозначение: $f^*(t) = (t \mathbin{>>=} f)$.
- Можно определить f^* через операцию «разглаживания двойной монады» $\mu : MMY \to MY$, а именно, $f^* = \mu \circ f$.
- Наконец, в монаду можно помещать объект «чистого» типа, $\eta : X \to MX$.

Монады

- Более того, f сама может создавать монадический объект, и этот объект помещается в исходную монаду.
- А именно, $f : X \to MY$ даёт $f^* : MX \to MY$. Другое обозначение: $f^*(t) = (t \texttt{>>=} f)$.
- Можно определить f^* через операцию «разглаживания двойной монады» $\mu : MMY \to MY$, а именно, $f^* = \mu \circ f$.
- Наконец, в монаду можно помещать объект «чистого» типа, $\eta : X \to MX$.
- Для корректно определённых монад эти операции должны удовлетворять нескольким естественным соотношениям.

Монады

- Основный примером является монада `IO`, осуществляющая взаимодействие с «внешним миром».

Монады

- Основный примером является монада `IO`, осуществляющая взаимодействие с «внешним миром».
- Рассмотрим ещё два примера использования монад — для моделирования недетерминированных и вероятностных вычислений.

Монады

- Основный примером является монада `IO`, осуществляющая взаимодействие с «внешним миром».
- Рассмотрим ещё два примера использования монад — для моделирования недетерминированных и вероятностных вычислений.
- Общая идея: помещение внутрь монады помещает вычисления в некоторое своеобразное окружение.