

Лекция 9: функторы, монады и все-все-все

Функциональное программирование на Haskell

Алексей Романов

27 апреля 2023 г.

МИЭТ

Роды типов

- У значений есть типы, а у типов — роды (kinds).
- Все обычные типы (`Int`, `Maybe Bool`, `(Int, Int)`, ...) имеют род `*` (синоним — `Type`).

Роды типов

- У значений есть типы, а у типов — роды (kinds).
- Все обычные типы (`Int`, `Maybe Bool`, `(Int, Int)`, ...) имеют род `*` (синоним — `Type`).
- Пример более сложного рода — `Maybe`. Он принимает тип рода `*` и возвращает тип рода `*`:
`ghci> :kind Maybe`

Роды типов

- У значений есть типы, а у типов — роды (kinds).
- Все обычные типы (`Int`, `Maybe Bool`, `(Int, Int)`, ...) имеют род `*` (синоним — `Type`).
- Пример более сложного рода — `Maybe`. Он принимает тип рода `*` и возвращает тип рода `*`:

```
ghci> :kind Maybe
Maybe :: * -> *
```

Роды типов

- У значений есть типы, а у типов — роды (kinds).
- Все обычные типы (`Int`, `Maybe Bool`, `(Int, Int)`, ...) имеют род `*` (синоним — `Type`).
- Пример более сложного рода — `Maybe`. Он принимает тип рода `*` и возвращает тип рода `*`:
`ghci> :kind Maybe`
`Maybe :: * -> *`
- Определите род:
 - `Either (data Either a b = Left a | Right b)`

Роды типов

- У значений есть типы, а у типов — роды (kinds).
- Все обычные типы (`Int`, `Maybe Bool`, `(Int, Int)`, ...) имеют род `*` (синоним — `Type`).
- Пример более сложного рода — `Maybe`. Он принимает тип рода `*` и возвращает тип рода `*`:

```
ghci> :kind Maybe
Maybe :: * -> *
```
- Определите род:
 - `Either (data Either a b = Left a | Right b)`
 - `Shape (type Shape f = f ())`

Роды типов

- У значений есть типы, а у типов — роды (kinds).
- Все обычные типы (`Int`, `Maybe Bool`, `(Int, Int)`, ...) имеют род `*` (синоним — `Type`).
- Пример более сложного рода — `Maybe`. Он принимает тип рода `*` и возвращает тип рода `*`:

```
ghci> :kind Maybe
Maybe :: * -> *
```
- Определите род:
 - `Either (data Either a b = Left a | Right b)`
 - `Shape (type Shape f = f ())`
- В стандартном Haskell все роды строятся из `*` и `->`, в GHC всё сложнее (`Kind polymorphism`, `Unboxed type kinds`, `Datatype promotion`, `The Constraint kind`).

Частичное применение типов

- Конструкторы типов могут быть частично применены (как функции).
- Если `Either :: * -> * -> *`, то `Either Int`

Частичное применение типов

- Конструкторы типов могут быть частично применены (как функции).
- Если `Either :: * -> * -> *`, то `Either Int :: * -> *`. `Either a` — тоже.

Частичное применение типов

- Конструкторы типов могут быть частично применены (как функции).
- Если `Either :: * -> * -> *`, то `Either Int :: * -> *`. `Either a` — тоже.
- Типы кортежей, функций и списков можно писать в префиксном виде:
`(,) a b`, `(->) a b`, `[] a`.

Частичное применение типов

- Конструкторы типов могут быть частично применены (как функции).
- Если `Either :: * -> * -> *`, то `Either Int :: * -> *`. `Either a` — тоже.
- Типы кортежей, функций и списков можно писать в префиксном виде:
`(,) a b`, `(->) a b`, `[] a`.
- И применять частично: `(->) Int` читается как «функции из `Int`» и имеет род

Частичное применение типов

- Конструкторы типов могут быть частично применены (как функции).
- Если `Either :: * -> * -> *`, то `Either Int :: * -> *`. `Either a` — тоже.
- Типы кортежей, функций и списков можно писать в префиксном виде:
`(,) a b`, `(->) a b`, `[] a`.
- И применять частично: `(->) Int` читается как «функции из `Int`» и имеет род `* -> *`.

Частичное применение типов

- Конструкторы типов могут быть частично применены (как функции).
- Если `Either :: * -> * -> *`, то `Either Int :: * -> *`. `Either a` — тоже.
- Типы кортежей, функций и списков можно писать в префиксном виде:
`(,) a b`, `(->) a b`, `[] a`.
- И применять частично: `(->) Int` читается как «функции из `Int`» и имеет род `* -> *`.
- Синонимы типов (как `Shape` с прошлого слайда) всегда должны быть применены полностью.

Частичное применение типов

- Конструкторы типов могут быть частично применены (как функции).
- Если `Either :: * -> * -> *`, то `Either Int :: * -> *`. `Either a` — тоже.
- Типы кортежей, функций и списков можно писать в префиксном виде:
`(,) a b`, `(->) a b`, `[] a`.
- И применять частично: `(->) Int` читается как «функции из `Int`» и имеет род `* -> *`.
- Синонимы типов (как `Shape` с прошлого слайда) всегда должны быть применены полностью.
- Для частичного применения нужен `newtype Shape f = Shape (f ())`.

- Сравним типы нескольких функций:
- `map :: (a -> b) -> [a] -> [b]`

Варианты `map`

- Сравним типы нескольких функций:
- `map :: (a -> b) -> [a] -> [b]`
- `mapMb :: (a -> b) -> Maybe a -> Maybe b`
(`mapMaybe` называется другая функция)

Варианты `map`

- Сравним типы нескольких функций:
- `map :: (a -> b) -> [a] -> [b]`
- `mapMb :: (a -> b) -> Maybe a -> Maybe b`
(`mapMaybe` называется другая функция)
- `mapMap :: (a -> b) -> Map k a -> Map k b`
(она же `Data.Map.{Lazy/Strict}.map`)

Варианты `map`

- Сравним типы нескольких функций:
- `map :: (a -> b) -> [a] -> [b]`
- `mapMb :: (a -> b) -> Maybe a -> Maybe b`
(`mapMaybe` называется другая функция)
- `mapMap :: (a -> b) -> Map k a -> Map k b`
(она же `Data.Map.{Lazy/Strict}.map`)
- `mapSnd :: (a -> b) -> (c, a) -> (c, b)`

Варианты `map`

- Сравним типы нескольких функций:
- `map :: (a -> b) -> [a] -> [b]`
- `mapMb :: (a -> b) -> Maybe a -> Maybe b`
(`mapMaybe` называется другая функция)
- `mapMap :: (a -> b) -> Map k a -> Map k b`
(она же `Data.Map.{Lazy/Strict}.map`)
- `mapSnd :: (a -> b) -> (c, a) -> (c, b)`

Варианты `map`

- Сравним типы нескольких функций:
- `map :: (a -> b) -> [a] -> [b]`
- `mapMb :: (a -> b) -> Maybe a -> Maybe b`
(`mapMaybe` называется другая функция)
- `mapMap :: (a -> b) -> Map k a -> Map k b`
(она же `Data.Map.{Lazy/Strict}.map`)
- `mapSnd :: (a -> b) -> (c, a) -> (c, b)`
- Видим, что все они имеют вид

Варианты `map`

- Сравним типы нескольких функций:
- `map :: (a -> b) -> [a] -> [b]`
- `mapM :: (a -> b) -> Maybe a -> Maybe b`
(`mapMaybe` называется другая функция)
- `mapMap :: (a -> b) -> Map k a -> Map k b`
(она же `Data.Map.{Lazy/Strict}.map`)
- `mapSnd :: (a -> b) -> (c, a) -> (c, b)`
- Видим, что все они имеют вид
`forall a b. (a -> b) -> f a -> f b` для
разных `f` (`f` не под `forall!`).

Функторы

- Можем обобщить все функции с предыдущего слайда, введя класс типов

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a          -> f b -> f a
  (<$) = fmap . const
```

- Здесь `f` — переменная типа сорта

Функторы

- Можем обобщить все функции с предыдущего слайда, введя класс типов

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a          -> f b -> f a
  (<$) = fmap . const
```

- Здесь `f` — переменная типа сорта `* -> *`.
- Смысл `fmap g xs`: применить функцию `g` ко всем значениям типа `a` «внутри» `xs`, не меняя структуры.
- `(<$)` — частный случай `fmap`, который иногда может быть реализован напрямую.

Функторы

- Можем обобщить все функции с предыдущего слайда, введя класс типов

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a          -> f b -> f a
  (<$) = fmap . const
```

- Здесь `f` — переменная типа сорта `* -> *`.
- Смысл `fmap g xs`: применить функцию `g` ко всем значениям типа `a` «внутри» `xs`, не меняя структуры.
- `(<$)` — частный случай `fmap`, который иногда может быть реализован напрямую.
- `(<$>)` — синоним `fmap` как оператор.

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

`fmap id xs == xs`

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

`fmap id xs == xs`

- Этот закон также формулируется как `fmap id == id`.

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

`fmap id xs == xs`

- Этот закон также формулируется как `fmap (id @a) == id @(f a)`.

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

`fmap id xs == xs`

- Этот закон также формулируется как `fmap (id @a) == id @(f a)`.
- Второй закон функторов:

`fmap (g . h) xs ==`

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

```
fmap id xs == xs
```

- Этот закон также формулируется как `fmap (id @a) == id @(f a)`.
- Второй закон функторов:

```
fmap (g . h) xs == fmap g (fmap h xs)
```

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

```
fmap id xs == xs
```

- Этот закон также формулируется как `fmap (id @a) == id @(f a)`.

- Второй закон функторов:

```
fmap (g . h) xs == fmap g (fmap h xs)
```

```
fmap (g . h)      ==
```

Законы функторов

- Что значит «не меняя структуры»?
- Например, рассмотрим случай `fmap id`. Чему должно быть равно `fmap id xs`?

```
fmap id xs == xs
```

- Этот закон также формулируется как `fmap (id @a) == id @(f a)`.
- Второй закон функторов:

```
fmap (g . h) xs == fmap g (fmap h xs)
```

```
fmap (g . h)      == fmap g . fmap h
```


Примеры функторов

- Очень многие типы (рода `* -> *`) являются функторами. Например:

```
instance Functor Maybe where
```

```
-- fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap f (Just x) =
```

Примеры функторов

- Очень многие типы (рода `* -> *`) являются функторами. Например:

```
instance Functor Maybe where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  =
```

Примеры функторов

- Очень многие типы (рода `* -> *`) являются функторами. Например:

```
instance Functor Maybe where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  = Nothing
```

Примеры функторов

- Очень многие типы (рода `* -> *`) являются функторами. Например:

```
instance Functor Maybe where
```

```
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
    fmap f (Just x) = Just (f x)
```

```
    fmap _ Nothing  = Nothing
```

- ```
instance Functor [] where
```

```
 fmap =
```

# Примеры функторов

- Очень многие типы (рода `* -> *`) являются функторами. Например:

```
instance Functor Maybe where
```

```
 -- fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
 fmap f (Just x) = Just (f x)
```

```
 fmap _ Nothing = Nothing
```

- ```
instance Functor [] where
```

```
    fmap = map
```

- Не можем определить

```
instance Functor [] where
```

```
    fmap f xs = []
```

Примеры функторов

- Очень многие типы (рода `* -> *`) являются функторами. Например:

```
instance Functor Maybe where
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap f (Just x) = Just (f x)
    fmap _ Nothing  = Nothing
```

- `instance Functor [] where`
`fmap = map`

- Не можем определить
`instance Functor [] where`
`fmap f xs = []`

Типы сходятся (проверьте!), но законы нарушены.

Ещё примеры функторов

- Частично применённые кортежи:

```
instance Functor ((,) c) where
  -- fmap :: (a -> b) -> (c, a) -> (c, b)
  fmap f (z, x) =
```

Ещё примеры функторов

- Частично применённые кортежи:

```
instance Functor ((,) c) where
  -- fmap :: (a -> b) -> (c, a) -> (c, b)
  fmap f (z, x) = (z, f x)
```


Ещё примеры функторов

- Частично применённые кортежи:

```
instance Functor ((,) c) where
  -- fmap :: (a -> b) -> (c, a) -> (c, b)
  fmap f (z, x) = (z, f x)
```

- `data Pair a = Pair a a`
`instance Functor Pair where`
`-- fmap :: (a -> b) -> Pair a -> Pair b`
`fmap f (Pair x y) =`

Ещё примеры функторов

- Частично применённые кортежи:

```
instance Functor ((,) c) where
  -- fmap :: (a -> b) -> (c, a) -> (c, b)
  fmap f (z, x) = (z, f x)
```

- `data Pair a = Pair a a`
instance Functor Pair where
-- fmap :: (a -> b) -> Pair a -> Pair b
fmap f (Pair x y) = Pair (f x) (f y)

Ещё примеры функторов

- Частично применённые кортежи:

```
instance Functor ((,) c) where
  -- fmap :: (a -> b) -> (c, a) -> (c, b)
  fmap f (z, x) = (z, f x)
```

- `data Pair a = Pair a a`

```
instance Functor Pair where
  -- fmap :: (a -> b) -> Pair a -> Pair b
  fmap f (Pair x y) = Pair (f x) (f y)
```

- Дома (начните с уточнения типа `fmap`):

```
instance Functor (Either c) where ...
instance Functor ((->) c) where ...
```

... и нефункторов

- Не все типы можно сделать функторами!

Рассмотрим

```
newtype Pred a = Pred (a -> Bool)
instance Functor Pred where
  -- fmap :: (a -> b) -> Pred a -> Pred b
  fmap g (Pred p) =
```

... и нефункторов

- Не все типы можно сделать функторами!

Рассмотрим

```
newtype Pred a = Pred (a -> Bool)
instance Functor Pred where
  -- fmap :: (a -> b) -> Pred a -> Pred b
  fmap g (Pred p) = Pred (\x -> ???)
```

... и нефункторов

- Не все типы можно сделать функторами!

Рассмотрим

```
newtype Pred a = Pred (a -> Bool)
```

```
instance Functor Pred where
```

```
-- fmap :: (a -> b) -> Pred a -> Pred b
```

```
fmap g (Pred p) = Pred (\x -> ???)
```

`g :: a -> b` не к чему применить: у нас нет ничего типа `a`!

... и нефункторов

- Не все типы можно сделать функторами!

Рассмотрим

```
newtype Pred a = Pred (a -> Bool)
```

```
instance Functor Pred where
```

```
-- fmap :: (a -> b) -> Pred a -> Pred b
```

```
fmap g (Pred p) = Pred (\x -> ???)
```

`g :: a -> b` не к чему применить: у нас нет ничего типа `a`!

- Общее правило: `instance Functor F` можно определить, если `a` не входит в определение `F` а слева от нечётного числа `->` (**объяснение**).

... и нефункторов

- Не все типы можно сделать функторами!

Рассмотрим

```
newtype Pred a = Pred (a -> Bool)
```

```
instance Functor Pred where
```

```
-- fmap :: (a -> b) -> Pred a -> Pred b
```

```
fmap g (Pred p) = Pred (\x -> ???)
```

`g :: a -> b` не к чему применить: у нас нет ничего типа `a`!

- Общее правило: `instance Functor F` можно определить, если `a` не входит в определение `F` а слева от нечётного числа `->` (**объяснение**).
- По этому правилу, функтор ли ниже?

```
newtype Tricky a = T ((a -> Int) -> a)
```


Доказательство законов для экземпляра `Functor`

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`

Доказательство законов для экземпляра `Functor`

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`
- Шаг 1: `pair = Pair x y`, где `x, y ::`

Доказательство законов для экземпляра `Functor`

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`
- Шаг 1: `pair = Pair x y`, где `x, y :: a`
`fmap id pair ==`
`fmap id (Pair x y) ==`

Доказательство законов для экземпляра `Functor`

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`
- Шаг 1: `pair = Pair x y`, где `x, y :: a`
`fmap id pair ==`
`fmap id (Pair x y) ==` -- определение `fmap`

Доказательство законов для экземпляра `Functor`

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`
- Шаг 1: `pair = Pair x y`, где `x, y :: a`
`fmap id pair ==`
`fmap id (Pair x y) ==` -- определение `fmap`
`Pair (id x) (id y) ==`

Доказательство законов для экземпляра `Functor`

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`
- Шаг 1: `pair = Pair x y`, где `x, y :: a`
`fmap id pair ==`
`fmap id (Pair x y) ==` -- определение `fmap`
`Pair (id x) (id y) ==` -- определение `id`

Доказательство законов для экземпляра **Functor**

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`
- Шаг 1: `pair = Pair x y`, где `x, y :: a`
`fmap id pair ==`
`fmap id (Pair x y) ==` -- определение `fmap`
`Pair (id x) (id y) ==` -- определение `id`
`Pair x y ==`
`pair`

Доказательство законов для экземпляра **Functor**

- Возьмём для примера `instance Functor Pair`.
Нужно доказать, что для него:
- $\forall a :: *, \text{pair} :: \text{Pair } a$
`fmap id pair == pair`
- Шаг 1: `pair = Pair x y`, где `x, y :: a`
`fmap id pair ==`
`fmap id (Pair x y) ==` -- определение `fmap`
`Pair (id x) (id y) ==` -- определение `id`
`Pair x y ==`
`pair`
- $\forall a, b, c :: *, \text{pair} :: \text{Pair } a, h :: a \rightarrow b, g :: b \rightarrow c$
`fmap (g . h) pair == fmap g (fmap h pair)`

Использование функторов

- Предскажите результаты:

(+ 3) <\$> Just 1

(+ 3) <\$> Nothing

(+ 3) <\$> [1, 2, 3]

Использование функторов

- Предскажите результаты:

```
(+ 3) <$> Just 1
```

```
(+ 3) <$> Nothing
```

```
(+ 3) <$> [1, 2, 3]
```

- ```
void :: Functor f => f a -> f ()
void x = ???
```

```
void [1, 2]
```

## Чего не могут функторы?

- Удобно считать, что `fmap` берёт функцию `a -> b` и «поднимает» её в тип `f a -> f b`.
- Можно ли с её помощью поднять функцию двух аргументов? Т.е. можно ли реализовать

```
fmap2 :: Functor f => (a -> b -> c) -> f a -> f b -> f c
fmap2 g fx fy = ???
```

## Чего не могут функторы?

- Удобно считать, что `fmap` берёт функцию `a -> b` и «поднимает» её в тип `f a -> f b`.
- Можно ли с её помощью поднять функцию двух аргументов? Т.е. можно ли реализовать  
`fmap2 :: Functor f => (a -> b -> c) -> f a -> f b -> f c`  
`fmap2 g fx fy = ???`
- Нет! Единственное, с чего мы можем начать:

## Чего не могут функторы?

- Удобно считать, что `fmap` берёт функцию `a -> b` и «поднимает» её в тип `f a -> f b`.
- Можно ли с её помощью поднять функцию двух аргументов? Т.е. можно ли реализовать  
`fmap2 :: Functor f => (a -> b -> c) -> f a -> f b -> f c`  
`fmap2 g fx fy = ???`
- Нет! Единственное, с чего мы можем начать:  
`fmap g`. Она имеет тип

## Чего не могут функторы?

- Удобно считать, что `fmap` берёт функцию `a -> b` и «поднимает» её в тип `f a -> f b`.
- Можно ли с её помощью поднять функцию двух аргументов? Т.е. можно ли реализовать  
`fmap2 :: Functor f => (a -> b -> c) -> f a -> f b -> f c`  
`fmap2 g fx fy = ???`
- Нет! Единственное, с чего мы можем начать:  
`fmap g`. Она имеет тип `f a -> f (b -> c)`.
- Применив к `fx` (тип подходит), получим `fmap g fx :: f (b -> c)`.

## Чего не могут функторы?

- Удобно считать, что `fmap` берёт функцию `a -> b` и «поднимает» её в тип `f a -> f b`.
- Можно ли с её помощью поднять функцию двух аргументов? Т.е. можно ли реализовать  
`fmap2 :: Functor f => (a -> b -> c) -> f a -> f b -> f c`  
`fmap2 g fx fy = ???`
- Нет! Единственное, с чего мы можем начать:  
`fmap g`. Она имеет тип `f a -> f (b -> c)`.
- Применив к `fx` (тип подходит), получим `fmap g fx :: f (b -> c)`.
- Но `f (b -> c)` и `f b` скомбинировать уже не получится.

# Аппликативные функторы

- Если добавить к функторам метод получения `f` с из `f (b -> c)` и `f b`, и метод поднятия любого значения, получится понятие аппликативного функтора:

```
class Functor f => Applicative f where
 pure :: a -> f a
 infixl 4 <*>, *>, <*>
 (<*>) :: f (a -> b) -> f a -> f b
 (*>) :: f a -> f b -> f b
 (<*>) :: f a -> f b -> f a
```

- `(*>)` и `<*>` имеют определения по умолчанию.
- Теперь можно определить

```
liftA2 :: Applicative f => (a -> b -> c) ->
 f a -> f b -> f c
liftA2 g fx fy = ???
```



# Законы аппликативных функторов

- У **Applicative** 5 законов:
- `pure id <*> v = v`

`pure f <*> pure x = pure (f x)`

`u <*> pure y = pure ($ y) <*> u`

`u <*> (v <*> w) = pure (.) <*> u <*> v <*> w`

`fmap g x = pure g <*> x`

# Моноидальные функторы

- Эти законы не очень удобны и их трудно запомнить. Но мы можем использовать другой класс:
- ```
class Functor f => Monoidal f where  
  unit :: f ()  
  (**) :: f a -> f b -> f (a,b)
```
- Он взаимовыразим с аппликативными функторами (т.е. `pure` и `<*>` выражаются через `unit` и `(**)` и наоборот).
- Законы (\simeq : с точностью до изоморфизма):

```
unit ** v  $\simeq$  v  
u ** unit  $\simeq$  u  
u ** (v ** w)  $\simeq$  (u ** v) ** w
```
- Упражнение: записать точные их версии

Примеры аппликативных функторов

- Многие функторы являются аппликативными:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure =
```

Примеры аппликативных функторов

- Многие функторы являются аппликативными:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure = Just
```

```
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
  Just g <*> Just x  =
```

Примеры аппликативных функторов

- Многие функторы являются аппликативными:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure = Just
```

```
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
  Just g <*> Just x  = Just (g x)
```

```
  _      <*> _      =
```

Примеры аппликативных функторов

- Многие функторы являются аппликативными:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure = Just
```

```
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
  Just g <*> Just x  = Just (g x)
```

```
  _      <*> _      = Nothing
```

Примеры аппликативных функторов

- Многие функторы являются аппликативными:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure = Just
```

```
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
  Just g <*> Just x  = Just (g x)
```

```
  _      <*> _      = Nothing
```

- Для списков есть 2 варианта. Стандартный:

```
instance Applicative [] where
```

```
  -- pure :: a -> [a]
```

```
  pure x =
```

Примеры аппликативных функторов

- Многие функторы являются аппликативными:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure = Just
```

```
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
  Just g <*> Just x  = Just (g x)
```

```
  _      <*> _      = Nothing
```

- Для списков есть 2 варианта. Стандартный:

```
instance Applicative [] where
```

```
  -- pure :: a -> [a]
```

```
  pure x = [x]
```

```
  -- (<*>) :: [a -> b] -> [a] -> [b]
```

```
  gs <*> xs =
```


Примеры аппликативных функторов

- Многие функторы являются аппликативными:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure = Just
```

```
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
  Just g <*> Just x  = Just (g x)
```

```
  _      <*> _      = Nothing
```

- Для списков есть 2 варианта. Стандартный:

```
instance Applicative [] where
```

```
  -- pure :: a -> [a]
```

```
  pure x = [x]
```

```
  -- (<*>) :: [a -> b] -> [a] -> [b]
```

```
  gs <*> xs = [g x | g <- gs, x <- xs]
```

Примеры аппликативных функторов

- Второй аппликативный функтор для списков (определён в `Control.Applicative`):

```
newtype ZipList a = ZipList { getZipList :: [a] }
instance Applicative ZipList where
    (ZipList fs) <*> (ZipList xs) = ZipList (zipWith (
    pure x = ???
```

Примеры аппликативных функторов

- Второй аппликативный функтор для списков (определён в `Control.Applicative`):

```
newtype ZipList a = ZipList { getZipList :: [a] }  
instance Applicative ZipList where  
    (ZipList fs) <*> (ZipList xs) = ZipList (zipWith (  
        pure x = ???
```

- Подсказка: реализация определяется законом
`pure id <*> v = v`

Примеры аппликативных функторов

- Второй аппликативный функтор для списков (определён в `Control.Applicative`):

```
newtype ZipList a = ZipList { getZipList :: [a] }  
instance Applicative ZipList where  
    (ZipList fs) <*> (ZipList xs) = ZipList (zipWith (  
        pure x = ???
```

- Подсказка: реализация определяется законом
`pure id <*> v = v`
- Ответ:
`pure x = ZipList (repeat x)`

Пример неаппликативного функтора

- `instance Applicative ((,) c) where`
 `-- pure :: a -> (c, a)`
 `pure = ???`

Пример неаппликативного функтора

- `instance Applicative ((,) c) where`
 `-- pure :: a -> (c, a)`
 `pure = ???`

Невозможно определить без ограничений на `c`.

Пример неаппликативного функтора

- `instance Applicative ((,) c) where`
 `-- pure :: a -> (c, a)`
 `pure = ???`

Невозможно определить без ограничений на `c`.

- А какие ограничения нужны?

Пример неаппликативного функтора

- `instance Applicative ((,) c) where`
 `-- pure :: a -> (c, a)`
 `pure = ???`

Невозможно определить без ограничений на `c`.

- А какие ограничения нужны?

```
instance Monoid c => Applicative ((,) c) where
  pure x = (mempty, x)
  (c1, g) <*> (c2, x) =
```


Пример неаппликативного функтора

- `instance Applicative ((,) c) where`
 `-- pure :: a -> (c, a)`
 `pure = ???`

Невозможно определить без ограничений на `c`.

- А какие ограничения нужны?

```
instance Monoid c => Applicative ((,) c) where
  pure x = (mempty, x)
  (c1, g) <*> (c2, x) = (c1 <> c2, g x)
```

Что можно сделать с аппликативными функторами

- Предскажите результаты:

```
pure (+) <*> [2, 3, 4] <*> pure 4
```

```
[(+1), (*2)] <*> [2, 3]
```

```
[1, 2] *> [3, 4, 5]
```

```
[1, 2] <*> [3, 4, 5]
```

```
ZipList [1, 2] *> ZipList [3, 4, 5]
```

Что можно сделать с аппликативными функторами

- Предскажите результаты:

```
pure (+) <*> [2, 3, 4] <*> pure 4
```

```
[(+1), (*2)] <*> [2, 3]
```

```
[1, 2] *> [3, 4, 5]
```

```
[1, 2] <*> [3, 4, 5]
```

```
ZipList [1, 2] *> ZipList [3, 4, 5]
```

- Реализуйте

```
when :: Applicative f => Bool -> f () -> f ()
```

```
sequenceAL :: Applicative f => [f a] -> f [a]
```

- У второй есть обобщённая версия `sequenceA`, определяющая ещё один класс `Traversable`.

... И ЧТО НЕЛЬЗЯ

- Структура результата аппликативного вычисления зависит только от структуры аргументов, а не от значений внутри них.
- Т.е. мы не можем определить функцию
`ifA :: Applicative f => f Bool -> f a -> f a -> f a`
так, чтобы выполнялось
`ifA [True] [1, 2] [3] == [1, 2]`
`ifA [False] [1, 2] [3] == [3]`
(в случае списков структура — число элементов).

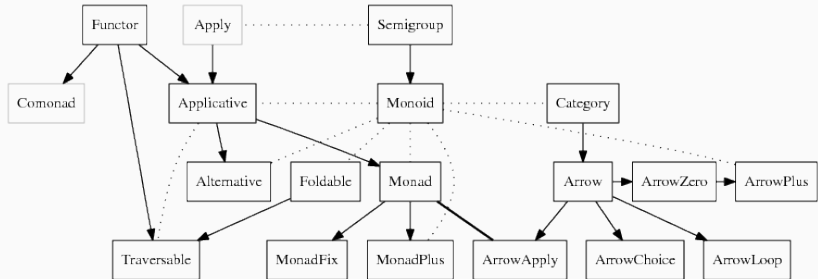
... И ЧТО НЕЛЬЗЯ

- Структура результата аппликативного вычисления зависит только от структуры аргументов, а не от значений внутри них.
- Т.е. мы не можем определить функцию
`ifA :: Applicative f => f Bool -> f a -> f a -> f a`
так, чтобы выполнялось
`ifA [True] [1, 2] [3] == [1, 2]`
`ifA [False] [1, 2] [3] == [3]`
(в случае списков структура — число элементов).
- Попробуйте написать какие-нибудь варианты `ifA` по типу и убедиться в этом.

... И ЧТО НЕЛЬЗЯ

- Структура результата аппликативного вычисления зависит только от структуры аргументов, а не от значений внутри них.
- Т.е. мы не можем определить функцию
`ifA :: Applicative f => f Bool -> f a -> f a -> f a`
так, чтобы выполнялось
`ifA [True] [1, 2] [3] == [1, 2]`
`ifA [False] [1, 2] [3] == [3]`
(в случае списков структура — число элементов).
- Попробуйте написать какие-нибудь варианты `ifA` по типу и убедиться в этом.
- Эту проблему решают монады, о которых будет следующая лекция.

Диаграмма классов типов высших родов



- [Typeclassopedia](#) (диаграмма на предыдущем слайде оттуда). Особенно посмотрите на [Foldable](#), [Traversable](#) и [Alternative](#).
- [Good examples of Not a Functor/Applicative/Monad?](#)
- [Functors, Applicatives, And Monads In Pictures.](#)
- Глава [Applicative Functors](#) в Wikibook.