Функтори и монади

Трифон Трифонов

Функционално програмиране, 2018/19 г.

16 януари 2019 г.

Класове от по-висок ред

- Досега разглеждахме *класове* от типове, които имат сходно поведение (`Eq`, `Read`, `Show`, `Enum`, `Measurable`, `Num`, ...).

Класове от по-висок ред

- Досега разглеждахме *класове* от типове, които имат сходно поведение (`Eq`, `Read`, `Show`, `Enum`, `Measurable`, `Num`, ...).
- Разглеждахме и *типови конструктори*, които позволяват дефиниране на параметризирани (генерични) типове (`Maybe`, `[]`, `BinTree`, `Tree`, `IO`, ...).

Класове от по-висок ред

- Досега разглеждахме *класове* от типове, които имат сходно поведение (`Eq`, `Read`, `Show`, `Enum`, `Measurable`, `Num`, ...).
- Разглеждахме и *типови конструктори*, които позволяват дефиниране на параметризирани (генерични) типове (`Maybe`, `[]`, `BinTree`, `Tree`, `IO`, ...).
- Нека да разгледаме *клас от типови конструктори*, които имат някаква обща характеристика.

Класове от по-висок ред

- Досега разглеждахме *класове* от типове, които имат сходно поведение (`Eq`, `Read`, `Show`, `Enum`, `Measurable`, `Num`, ...).
- Разглеждахме и *типови конструктори*, които позволяват дефиниране на параметризирани (генерични) типове (`Maybe`, `[]`, `BinTree`, `Tree`, `IO`, ...).
- Нека да разгледаме *клас от типови конструктори*, които имат някаква обща характеристика.
- **Пример:** Има ли нещо общо, което можем да правим с `[]`, `BinTree` и `Tree`?

Класове от по-висок ред

- Досега разглеждахме *класове* от типове, които имат сходно поведение (`Eq`, `Read`, `Show`, `Enum`, `Measurable`, `Num`, ...).
- Разглеждахме и *типови конструктори*, които позволяват дефиниране на параметризирани (генерични) типове (`Maybe`, `[]`, `BinTree`, `Tree`, `IO`, ...).
- Нека да разгледаме *клас от типови конструктори*, които имат някаква обща характеристика.
- **Пример:** Има ли нещо общо, което можем да правим с `[]`, `BinTree` и `Tree`?
- Нещо, което не зависи от *типа* на елементите в тези контейнери?

Примери за класове от конструктори

- **Пример:** Има ли нещо общо, което можем да правим с `[]`, `BinTree` и `Tree`?

Примери за класове от конструктори

- **Пример:** Има ли нещо общо, което можем да правим с `[]`, `BinTree` и `Tree`?
- Можем да намираме брой елементи

```
class Countable c where
  count :: c a -> Integer
```

Примери за класове от конструктори

- **Пример:** Има ли нещо общо, което можем да правим с `[]`, `BinTree` и `Tree`?
- Можем да намираме брой елементи

```
class Countable c where
  count :: c a -> Integer
```

- Можем да намерим списък от всички елементи

```
class Listable c where
  elements :: c a -> [a]
```

Примери за класове от конструктори

- **Пример:** Има ли нещо общо, което можем да правим с `[]`, `BinTree` и `Tree`?
- Можем да намираме брой елементи

```
class Countable c where
  count :: c a -> Integer
```

- Можем да намерим списък от всички елементи

```
class Listable c where
  elements :: c a -> [a]
```

- Можем да приложим функция над всеки елемент

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

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`
- `(,) a`

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`
- `(,) a`
- `Either a`

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`
- `(,) a`
- `Either a`
- `[]`

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`
- `(,) a`
- `Either a`
- `[]`
- `BinTree`

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`
- `(,) a`
- `Either a`
- `[]`
- `BinTree`
- `Tree`

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`
- `(,) a`
- `Either a`
- `[]`
- `BinTree`
- `Tree`
- `(->) r`

Функтори

Дефиниция

Класът `Functor` в Haskell се състои от типовите конструктори f, за които може да се дефинира `fmap :: (a -> b) -> f a -> f b`.

За удобство операцията `<$>` е инфиксен вариант на `fmap`.

Примери за функтори:

- `Maybe`
- `(,) a`
- `Either a`
- `[]`
- `BinTree`
- `Tree`
- `(->) r`
- `IO`

Странни функторни екземпляри

Пример: да разгледаме екземпляра

```
data Pill a = BluePill a | RedPill a
instance Functor Pill where
  fmap f (BluePill x) = RedPill (f x)
  fmap f (RedPill  x) = BluePill (f x)
```

Странни функторни екземпляри

Пример: да разгледаме екземпляра

```
data Pill a = BluePill a | RedPill a
instance Functor Pill where
  fmap f (BluePill x) = RedPill (f x)
  fmap f (RedPill  x) = BluePill (f x)
```

Проблем №1:

- `fmap id (BluePill 2) = RedPill 2`

Странни функторни екземпляри

Пример: да разгледаме екземпляра

```
data Pill a = BluePill a | RedPill a
instance Functor Pill where
  fmap f (BluePill x) = RedPill (f x)
  fmap f (RedPill  x) = BluePill (f x)
```

Проблем №1:

- `fmap id (BluePill 2) = RedPill 2`
- `fmap` с "празна" функция променя структурата на функтора!

Странни функторни екземпляри

Пример: да разгледаме екземпляра

```
data Pill a = BluePill a | RedPill a
instance Functor Pill where
  fmap f (BluePill x) = RedPill (f x)
  fmap f (RedPill  x) = BluePill (f x)
```

Проблем №1:

- `fmap id (BluePill 2) = RedPill 2`
- `fmap` с "празна" функция променя структурата на функтора!

Проблем №2:

Странни функторни екземпляри

Пример: да разгледаме екземпляра

```haskell
data Pill a = BluePill a | RedPill a
instance Functor Pill where
  fmap f (BluePill x) = RedPill (f x)
  fmap f (RedPill  x) = BluePill (f x)
```

Проблем №1:

- `fmap id (BluePill 2) = RedPill 2`
- fmap с "празна" функция променя структурата на функтора!

Проблем №2:

- `fmap (+3) (BluePill 3) = RedPill 6`

Странни функторни екземпляри

Пример: да разгледаме екземпляра

```
data Pill a = BluePill a | RedPill a
instance Functor Pill where
  fmap f (BluePill x) = RedPill (f x)
  fmap f (RedPill  x) = BluePill (f x)
```

Проблем №1:

- `fmap id (BluePill 2) = RedPill 2`
- `fmap` с "празна" функция променя структурата на функтора!

Проблем №2:

- `fmap (+3) (BluePill 3) = RedPill 6`
- `fmap (+1) (fmap (+2) (BluePill 3)) = BluePill 6`

Странни функторни екземпляри

Пример: да разгледаме екземпляра

```
data Pill a = BluePill a | RedPill a
instance Functor Pill where
  fmap f (BluePill x) = RedPill (f x)
  fmap f (RedPill  x) = BluePill (f x)
```

Проблем №1:

- `fmap id (BluePill 2) = RedPill 2`
- `fmap` с "празна" функция променя структурата на функтора!

Проблем №2:

- `fmap (+3) (BluePill 3) = RedPill 6`
- `fmap (+1) (fmap (+2) (BluePill 3)) = BluePill 6`
- Има значение колко поред функции ще приложим!

Функторни закони

Дефиниция

Функтор наричаме екземпляр на класа *Functor* такъв, че:

1. `fmap id` $\Longleftrightarrow$ `id` (запазване на идентитета)
2. `fmap f . fmap g` $\Longleftrightarrow$ fmap (f . g) (дистрибутивност относно композиция)

Функторни закони

Дефиниция

Функтор наричаме екземпляр на класа *Functor* такъв, че:

1. `fmap id` ⟺ `id` (запазване на идентитета)
2. `fmap f . fmap g` ⟺ fmap (f . g) (дистрибутивност относно композиция)

Функторните закони ни дават гаранция, че реализацията на `fmap` е "неутрална" към функтора и променя стойностите в него само и единствено на базата на подадената функция `f`.

Функторни закони

Дефиниция

Функтор наричаме екземпляр на класа *Functor* такъв, че:

1. `fmap id` $\Longleftrightarrow$ `id` (запазване на идентитета)
2. `fmap f . fmap g` $\Longleftrightarrow$fmap (f . g) (дистрибутивност относно композиция)

Функторните закони ни дават гаранция, че реализацията на `fmap` е “неутрална” към функтора и променя стойностите в него само и единствено на базата на подадената функция `f`.
Всички примерни екземпляри (освен `Pill`) удовлетворяват функторните закони.

Функторни закони

Дефиниция

Функтор наричаме екземпляр на класа *Functor* такъв, че:

1. `fmap id` $\Longleftrightarrow$ `id` (запазване на идентитета)
2. `fmap f . fmap g` $\Longleftrightarrow$fmap (f . g) (дистрибутивност относно композиция)

Функторните закони ни дават гаранция, че реализацията на `fmap` е "неутрална" към функтора и променя стойностите в него само и единствено на базата на подадената функция `f`.

Всички примерни екземпляри (освен `Pill`) удовлетворяват функторните закони.

Можем да мислим, че `fmap` "повдига" функцията `f` от елементи към функтори.

fmap с двуаргументни функции

Можем ли да използваме fmap за "повдигане" на двуаргументна функция?

`fmap` с двуаргументни функции

Можем ли да използваме `fmap` за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` $\longrightarrow$?

fmap с двуаргументни функции

Можем ли да използваме `fmap` за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` $\longrightarrow$ Грешка!

fmap с двуаргументни функции

Можем ли да използваме `fmap` за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` ⟶ Грешка!

Проблем: `fmap (+) (Just 3) :: ?` $f\ b = Maybe\ (Int \to Int)$

$(+) :: \underbrace{Int}_{a} \to \underbrace{Int \to Int}_{b}$

$Just\ 3 :: \underbrace{Maybe}_{f}\ \underbrace{Int}_{a}$

$f = Maybe$

`fmap` с двуаргументни функции

Можем ли да използваме `fmap` за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` $\longrightarrow$ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

fmap с двуаргументни функции

Можем ли да използваме fmap за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` $\longrightarrow$ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

Получаваме функтор над функция, която не можем директно да приложим над функтор над стойност!

fmap с двуаргументни функции

Можем ли да използваме fmap за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` $\longrightarrow$ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

Получаваме функтор над функция, която не можем директно да приложим над функтор над стойност!

Идея: Да разбием fmap на две части:

`fmap` с двуаргументни функции

Можем ли да използваме `fmap` за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` $\longrightarrow$ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

Получаваме функтор над функция, която не можем директно да приложим над функтор над стойност!

Идея: Да разбием `fmap` на две части:

- повдигане на функтор над функция към функция над функтори

Maybe (Int -> Int) -> Maybe Int -> Maybe Int

fmap с двуаргументни функции

Можем ли да използваме fmap за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` $\longrightarrow$ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

Получаваме функтор над функция, която не можем директно да приложим над функтор над стойност!

Идея: Да разбием fmap на две части:

- повдигане на функтор над функция към функция над функтори
  - `f (a -> b) -> f a -> f b`

fmap с двуаргументни функции

Можем ли да използваме fmap за “повдигане” на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` ⟶ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

Получаваме функтор над функция, която не можем директно да приложим над функтор над стойност!

Идея: Да разбием fmap на две части:

- повдигане на функтор над функция към функция над функтори
    - `f (a -> b) -> f a -> f b`
- повдигане на обикновена функция към функтор над функция

fmap с двуаргументни функции

Можем ли да използваме fmap за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` ⟶ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

Получаваме функтор над функция, която не можем директно да приложим над функтор над стойност!

Идея: Да разбием fmap на две части:

- повдигане на функтор над функция към функция над функтори
  - `f (a -> b) -> f a -> f b`
- повдигане на обикновена функция към функтор над функция
  - `(a -> b) -> f (a -> b)`

fmap с двуаргументни функции

Можем ли да използваме `fmap` за "повдигане" на двуаргументна функция?

Пример: `fmap (+) (Just 3) (Just 5)` ⟶ Грешка!

Проблем: `fmap (+) (Just 3) :: Maybe (Int -> Int)`

Получаваме функтор над функция, която не можем директно да приложим над функтор над стойност!

Идея: Да разбием `fmap` на две части:

- повдигане на функтор над функция към функция над функтори
  - `f (a -> b) -> f a -> f b`
- повдигане на обикновена функция към функтор над функция
  - `(a -> b) -> f (a -> b)`

Функторите, които поддържат такова разлагане на `fmap` наричаме *апликативни*.

Класът Applicative

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

Класът Applicative

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

Можем да дефинираме `fmap f a = pure f <*> a`.

Класът Applicative

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

Можем да дефинираме `fmap f a = pure f <*> a`.

Примери за апликативни функтори:

- Maybe

Класът Applicative

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

Можем да дефинираме `fmap f a = pure f <*> a`.

Примери за апликативни функтори:

- `Maybe`
- `Either a`

Класът Applicative

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

Можем да дефинираме `fmap f a = pure f <*> a`.

Примери за апликативни функтори:

- `Maybe`
- `Either a`
- `[]`

Класът Applicative

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

Можем да дефинираме `fmap f a = pure f <*> a`.

Примери за апликативни функтори:

- `Maybe`
- `Either a`
- `[]`
- `ZipList`

Класът Applicative

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

Можем да дефинираме `fmap f a = pure f <*> a`.

Примери за апликативни функтори:

- `Maybe`
- `Either a`
- `[]`
- `ZipList`
- `(->) r`

Класът Applicative

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

Можем да дефинираме `fmap f a = pure f <*> a`.

Примери за апликативни функтори:

- `Maybe`
- `Either a`
- `[]`
- `ZipList`
- `(->) r`
- `IO`

Функции за апликативни функтори

- ```
  liftA2 :: (Applicative f) =>
            (a -> b -> c) -> f a -> f b -> f c
  ```
  ```

Функции за апликативни функтори

- `liftA2 :: (Applicative f) =>`
  `(a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор

Функции за апликативни функтори

- `liftA2 :: (Applicative f) =>`
  `(a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`

Функции за апликативни функтори

- `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` $\longrightarrow$?

Функции за апликативни функтори

- `liftA2 :: (Applicative f) =>`
  `(a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` $\longrightarrow$ `[12,22,32,13,23,33]`

Функции за апликативни функтори

- `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` $\longrightarrow$ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`

Функции за апликативни функтори

- `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` ⟶ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`
  - повдига списък от функтори до функтор над списък

Функции за апликативни функтори

- `liftA2 :: (Applicative f) =>`
  `(a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` ⟶ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`
  - повдига списък от функтори до функтор над списък
  - `sequenceA [] = pure []`
  - `sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)`

f a   f [a]

(:) :: a -> [a] -> [a]

f   f   f

Функции за апликативни функтори

- `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:** `liftA2 (+) [2,3] [10,20,30]` $\longrightarrow$ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`
  - повдига списък от функтори до функтор над списък
  - `sequenceA [] = pure []`
  - `sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)`
  - `sequenceA = foldr (liftA2 (:)) (pure [])`

Функции за апликативни функтори

- `liftA2 :: (Applicative f) =>`
  `(a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` ⟶ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`
  - повдига списък от функтори до функтор над списък
  - `sequenceA [] = pure []`
  - `sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)`
  - `sequenceA = foldr (liftA2 (:)) (pure [])`
  - **Пример:**
    `sequenceA [Just 2, Just 3, Just 5]` ⟶ ?

Функции за апликативни функтори

- `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` ⟶ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`
  - повдига списък от функтори до функтор над списък
  - `sequenceA [] = pure []`
  - `sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)`
  - `sequenceA = foldr (liftA2 (:)) (pure [])`
  - **Пример:**
    `sequenceA [Just 2, Just 3, Just 5]` ⟶ `Just [2,3,5]`

Функции за апликативни функтори

- `liftA2 :: (Applicative f) =>`
  `(a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:**
    `liftA2 (+) [2,3] [10,20,30]` ⟶ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`
  - повдига списък от функтори до функтор над списък
  - `sequenceA [] = pure []`
  - `sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)`
  - `sequenceA = foldr (liftA2 (:)) (pure [])`
  - **Пример:**
    `sequenceA [Just 2, Just 3, Just 5]` ⟶ `Just [2,3,5]`
  - **Пример:** `sequenceA [Just 2, Nothing, Just 5]` ⟶ ?

Функции за апликативни функтори

- `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`
  - повдига двуаргументна функция над функтор
  - `liftA2 f a b = f <$> a <*> b`
  - **Пример:** `liftA2 (+) [2,3] [10,20,30]` ⟶ `[12,22,32,13,23,33]`
- `sequenceA :: (Applicative f) => [f a] -> f [a]`
  - повдига списък от функтори до функтор над списък
  - `sequenceA [] = pure []`
  - `sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)`
  - `sequenceA = foldr (liftA2 (:)) (pure [])`
  - **Пример:** `sequenceA [Just 2, Just 3, Just 5]` ⟶ `Just [2,3,5]`
  - **Пример:** `sequenceA [Just 2, Nothing, Just 5]` ⟶ `Nothing`

Закони за апликативни функтори

Дефиниция

Апликативен функтор наричаме екземпляр на класа `Applicative`, за който:

1. `pure f <*> x` $\Longleftrightarrow$ `fmap f x`

Закони за апликативни функтори

Дефиниция

Апликативен функтор наричаме екземпляр на класа `Applicative`, за който:

1. `pure f <*> x` $\Longleftrightarrow$ `fmap f x`
2. `pure id <*> v` $\Longleftrightarrow$ `v`

Закони за апликативни функтори

Дефиниция

Апликативен функтор наричаме екземпляр на класа `Applicative`, за който:

1. `pure f <*> x` $\Longleftrightarrow$ `fmap f x`
2. `pure id <*> v` $\Longleftrightarrow$ `v`
3. `pure (.) <*> u <*> v <*> w` $\Longleftrightarrow$ `u <*> (v <*> w)`

Закони за апликативни функтори

Дефиниция

Апликативен функтор наричаме екземпляр на класа `Applicative`, за който:

1. `pure f <*> x` $\Longleftrightarrow$ `fmap f x`
2. `pure id <*> v` $\Longleftrightarrow$ `v`
3. `pure (.) <*> u <*> v <*> w` $\Longleftrightarrow$ `u <*> (v <*> w)`
4. `pure f <*> pure x` $\Longleftrightarrow$ `pure (f x)`

Закони за апликативни функтори

Дефиниция

Апликативен функтор наричаме екземпляр на класа `Applicative`, за който:

1. `pure f <*> x` $\Longleftrightarrow$ `fmap f x`
2. `pure id <*> v` $\Longleftrightarrow$ `v`
3. `pure (.) <*> u <*> v <*> w` $\Longleftrightarrow$ `u <*> (v <*> w)`
4. `pure f <*> pure x` $\Longleftrightarrow$ `pure (f x)`
5. `u <*> pure y` $\Longleftrightarrow$ `pure ($ y) <*> u`

Операцията “свързване” (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:

Операцията "свързване" (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`

Операцията "свързване" (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори

Операцията “свързване” (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори
  - `(+) <$> [1,2] <*> [10,20]` $\longrightarrow$ `[11,12,21,22]`

Операцията "свързване" (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори
  - `(+) <$> [1,2] <*> [10,20]` $\longrightarrow$ `[11,12,21,22]`
- Но как можем да превърнем *функция, връщаща функтор* във функция над функтори?

Операцията "свързване" (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори
  - `(+) <$> [1,2] <*> [10,20]` $\longrightarrow$ `[11,12,21,22]`
- Но как можем да превърнем *функция, връщаща функтор* във функция над функтори?
  - `(\x -> [1..x]) =<< [3,4]` $\longrightarrow$ `[1,2,3,1,2,3,4]`

Операцията “свързване” (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори
  - `(+) <$> [1,2] <*> [10,20]` $\longrightarrow$ `[11,12,21,22]`
- Но как можем да превърнем *функция, връщаща функтор* във функция над функтори?
  - `(\x -> [1..x]) =<< [3,4]` $\longrightarrow$ `[1,2,3,1,2,3,4]`
  - Искаме структурата на функтора-резултат да може да зависи от стойността във функтора-параметър!

Операцията "свързване" (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори
  - `(+) <$> [1,2] <*> [10,20]` $\longrightarrow$ `[11,12,21,22]`
- Но как можем да превърнем *функция, връщаща функтор* във функция над функтори?
  - `(\x -> [1..x]) =<< [3,4]` $\longrightarrow$ `[1,2,3,1,2,3,4]`
  - Искаме структурата на функтора-резултат да може да зависи от стойността във функтора-параметър!
  - `(=<<) :: (a -> f b) -> f a -> f b`

Операцията “свързване” (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
    - `(+3) <$> [1,2]` ⟶ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори
    - `(+) <$> [1,2] <*> [10,20]` ⟶ `[11,12,21,22]`
- Но как можем да превърнем *функция, връщаща функтор* във функция над функтори?
    - `(\x -> [1..x]) =<< [3,4]` ⟶ `[1,2,3,1,2,3,4]`
    - Искаме структурата на функтора-резултат да може да зависи от стойността във функтора-параметър!
    - `(=<<) :: (a -> f b) -> f a -> f b`
    - По-често се използва операцията “свързване” (bind) с разменени аргументи:

Операцията "свързване" (bind)

- Функторите ни позволяваха да превърнем *функция над елементи* във функция над функтори:
  - `(+3) <$> [1,2]` $\longrightarrow$ `[4,5]`
- Апликативните функтори ни позволяваха да превърнем *функтор над функция* към функция над функтори
  - `(+) <$> [1,2] <*> [10,20]` $\longrightarrow$ `[11,12,21,22]`
- Но как можем да превърнем *функция, връщаща функтор* във функция над функтори?
  - `(\x -> [1..x]) =<< [3,4]` $\longrightarrow$ `[1,2,3,1,2,3,4]`
  - Искаме структурата на функтора-резултат да може да зависи от стойността във функтора-параметър!
  - `(=<<) :: (a -> f b) -> f a -> f b`
  - По-често се използва операцията "свързване" (bind) с разменени аргументи:
  - `(>>=) :: f a -> (a -> f b) -> f b`

Класът Monad

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

  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  x >> y   =   x >>= \_ -> y
```

Класът Monad

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

  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  x >> y   =   x >>= \_ -> y
```

Примери за монади:

- Maybe

Класът Monad

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

  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  x >> y   =   x >>= \_ -> y
```

Примери за монади:

- Maybe
- []

Класът Monad

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

  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  x >> y   =   x >>= \_ -> y
```

Примери за монади:

- `Maybe`
- `[]`
- `(->) r`

Класът Monad

```haskell
class (Applicative m) => Monad m where
  return :: a -> m a
  return = pure

  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  x >> y   =   x >>= \_ -> y
```

Примери за монади:

- Maybe
- []
- (->) r
- IO

Класът Monad

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

  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  x >> y   =   x >>= \_ -> y
```

Примери за монади:

- `Maybe`
- `[]`
- `(->) r`
- `IO`

`do` синтаксисът работи за всички екземпляри на `Monad`!

Монадни функции (1)

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

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
  - `fmap` за монади

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
  - fmap за монади
  - `liftM f m = m >>= (\x -> return $ f x)`

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
  - fmap за монади
  - `liftM f m = m >>= (\x -> return $ f x)`
- `ap :: (Monad m) => m (a -> b) -> m a -> m b`

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
    - fmap за монади
    - `liftM f m = m >>= (\x -> return $ f x)`
- `ap :: (Monad m) => m (a -> b) -> m a -> m b`
    - <*> за монади

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
    - fmap за монади
    - `liftM f m = m >>= (\x -> return $ f x)`
- `ap :: (Monad m) => m (a -> b) -> m a -> m b`
    - <*> за монади
    - `ap mf m = mf >>= (\f -> m >>= (\x -> return $ f x))`

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
  - fmap за монади
  - `liftM f m = m >>= (\x -> return $ f x)`
- `ap :: (Monad m) => m (a -> b) -> m a -> m b`
  - <*> за монади
  - `ap mf m = mf >>= (\f -> m >>= (\x -> return $ f x))`
- `liftM2::(Monad m) => (a -> b -> c) -> m a -> m b -> m c`

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
  - fmap за монади
  - `liftM f m = m >>= (\x -> return $ f x)`
- `ap :: (Monad m) => m (a -> b) -> m a -> m b`
  - <*> за монади
  - `ap mf m = mf >>= (\f -> m >>= (\x -> return $ f x))`
- `liftM2::(Monad m) => (a -> b -> c) -> m a -> m b -> m c`
  - liftA2 за монади

Монадни функции (1)

- `liftM :: (Monad m) => (a -> b) -> m a -> m b`
  - fmap за монади
  - `liftM f m = m >>= (\x -> return $ f x)`
- `ap :: (Monad m) => m (a -> b) -> m a -> m b`
  - <*> за монади
  - `ap mf m = mf >>= (\f -> m >>= (\x -> return $ f x))`
- `liftM2::(Monad m) => (a -> b -> c) -> m a -> m b -> m c`
  - liftA2 за монади

```
liftM2 f m1 m2 = m1 <<= (\x1 ->
                 m2 <<= (\x2 ->
                 return $ f x1 x2))
```

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - “слива” двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ "опаковани" булеви стойности

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - “слива” двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ “опаковани” булеви стойности
  - Резултатът е “опакованите” елементи на списъка

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме (>>=) чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ "опаковани" булеви стойности
  - Резултатът е "опакованите" елементи на списъка
  - `powerset = filterM (\x -> [True,False])`

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - “слива” двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ “опаковани” булеви стойности
  - Резултатът е “опакованите” елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - “слива” двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ “опаковани” булеви стойности
  - Резултатът е “опакованите” елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`
  - Натрупва елементи от списък с монадна операция

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - “слива” двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ “опаковани” булеви стойности
  - Резултатът е “опакованите” елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`
  - Натрупва елементи от списък с монадна операция
  - Натрупването е ляво (итеративен процес, подобно на `foldl`)

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ "опаковани" булеви стойности
  - Резултатът е "опакованите" елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`
  - Натрупва елементи от списък с монадна операция
  - Натрупването е ляво (итеративен процес, подобно на `foldl`)
  - ```
    boundSum lim = foldM (\x y -> if x+y < lim
                                   then Just (x+y) else Nothing) 0
    ```
    ```

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ "опаковани" булеви стойности
  - Резултатът е "опакованите" елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`
  - Натрупва елементи от списък с монадна операция
  - Натрупването е ляво (итеративен процес, подобно на `foldl`)
  - ```
    boundSum lim = foldM (\x y -> if x+y < lim
                                   then Just (x+y) else Nothing) 0
    ```
  - `boundSum 60 [1..10]` $\longrightarrow$?
    ```

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ "опаковани" булеви стойности
  - Резултатът е "опакованите" елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`
  - Натрупва елементи от списък с монадна операция
  - Натрупването е ляво (итеративен процес, подобно на `foldl`)
  - ```
    boundSum lim = foldM (\x y -> if x+y < lim
                                  then Just (x+y) else Nothing) 0
    ```
  - `boundSum 60 [1..10]` ⟶ `Just 55`
    ```

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ "опаковани" булеви стойности
  - Резултатът е "опакованите" елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`
  - Натрупва елементи от списък с монадна операция
  - Натрупването е ляво (итеративен процес, подобно на `foldl`)
  - ```
    boundSum lim = foldM (\x y -> if x+y < lim
                                  then Just (x+y) else Nothing) 0
    ```
  - `boundSum 60 [1..10]` $\longrightarrow$ `Just 55`
  - `boundSum 50 [1..10]` $\longrightarrow$?
    ```

Монадни функции (2)

- `join :: (Monad m) => m (m a) -> m a`
  - "слива" двойната опаковка в единична
  - `join mm = mm >>= id`
  - Можем да дефинираме `(>>=)` чрез `join` и `fmap`:
  - `m >>= f = join (fmap f m)`
- `filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]`
  - Филтрира с предикат, връщащ "опаковани" булеви стойности
  - Резултатът е "опакованите" елементи на списъка
  - `powerset = filterM (\x -> [True,False])`
- `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`
  - Натрупва елементи от списък с монадна операция
  - Натрупването е ляво (итеративен процес, подобно на `foldl`)
  - ```
    boundSum lim = foldM (\x y -> if x+y < lim
                                  then Just (x+y) else Nothing) 0
    ```
  - `boundSum 60 [1..10]` $\longrightarrow$ `Just 55`
  - `boundSum 50 [1..10]` $\longrightarrow$ `Nothing`
    ```

Монадни закони

Дефиниция

Монада наричаме инстанция на класа `Monad`, за която:

1. `return x >>= f` $\Longleftrightarrow$ `f x` (ляв идентитет)

Монадни закони

Дефиниция

Монада наричаме инстанция на класа `Monad`, за която:

1. `return x >>= f` $\Longleftrightarrow$ `f x` (ляв идентитет)
2. `m >>= return` $\Longleftrightarrow$ `m` (десен идентитет)

Монадни закони

Дефиниция

Монада наричаме инстанция на класа `Monad`, за която:

1. `return x >>= f` $\Longleftrightarrow$ `f x` (ляв идентитет)
2. `m >>= return` $\Longleftrightarrow$ `m` (десен идентитет)
3. `(m >>= f) >>= g` $\Longleftrightarrow$ `m >>= (\x -> f x >>= g)` (асоциативност)

Монадни закони

Дефиниция

Монада наричаме инстанция на класа `Monad`, за която:

1. `return x >>= f` $\Longleftrightarrow$ `f x` (ляв идентитет)
2. `m >>= return` $\Longleftrightarrow$ `m` (десен идентитет)
3. `(m >>= f) >>= g` $\Longleftrightarrow$ `m >>= (\x -> f x >>= g)` (асоциативност)

Композиция на монадни функции:

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g   =   \x -> g x >>= f
```

Монадни закони

Дефиниция

Монада наричаме инстанция на класа Monad, за която:

1. `return x >>= f` $\Longleftrightarrow$ `f x` (ляв идентитет)
2. `m >>= return` $\Longleftrightarrow$ `m` (десен идентитет)
3. `(m >>= f) >>= g` $\Longleftrightarrow$ `m >>= (\x -> f x >>= g)` (асоциативност)

Композиция на монадни функции:

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g   =   \x -> g x >>= f
```

Монадните закони чрез композиция:

1. `f <=< return` $\Longleftrightarrow$ `f` (ляв идентитет)
2. `return <=< f` $\Longleftrightarrow$ `f` (десен идентитет)
3. `f <=< (g <=< h)` $\Longleftrightarrow$ `(f <=< g) <=< h` (асоциативност)