

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`, ...).
- 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?
- 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`.

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)
```

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 $fmap\ id \iff id$ (запазване на идентитета)
- 2 $fmap\ f \ .\ fmap\ g \iff fmap\ (f \ .\ g)$ (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?

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
 - $f (a \rightarrow b) \rightarrow f a \rightarrow f b$
- повдигане на обикновена функция към функтор над функция
 - $(a \rightarrow b) \rightarrow f (a \rightarrow b)$

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
```

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`
 - повдига дваргументна 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 $\text{pure } f \langle * \rangle x \iff \text{fmap } f \ x$
- 2 $\text{pure } \text{id} \langle * \rangle v \iff v$
- 3 $\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w \iff u \langle * \rangle (v \langle * \rangle w)$
- 4 $\text{pure } f \langle * \rangle \text{pure } x \iff \text{pure } (f \ x)$
- 5 $u \langle * \rangle \text{pure } y \iff \text{pure } (\$ \ y) \langle * \rangle u$

Операцията “свързване” (bind)

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

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
```

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`
 - `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`
 - “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`
 - `boundSum 50 [1..10] -> Nothing`

Монадни закони

Дефиниция

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

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

Композиция на монадни функции:

$$(<=<) :: (\text{Monad } m) \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c)$$

$$f <=< g = \backslash x \rightarrow g\ x >>= f$$

Монадните закони чрез композиция:

- 1 `f <=< return` \iff `f` (ляв идентитет)
- 2 `return <=< f` \iff `f` (десен идентитет)
- 3 `f <=< (g <=< h)` \iff `(f <=< g) <=< h` (асоциативност)