Funktori

Slobodan Jelić

Sveučilište J. J. Strossmayera u Osijeku, Odjel za matematiku

20. prosinca 2017.

Funktori

- prisjetimo do sad viđenih obrazaca programiranja

```
inc :: [Int] -> [Int]
inc [] = []
inc (n:ns) = n+1 : inc ns

sqr :: [Int] -> [Int]
sqr [] = []
sqr (n:ns) = n^2 : sqr ns
```

- prisjetimo se funkcije višeg reda `map`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

- `inc` i `sqr` možemo definirati koristeći `map`:

```
inc1 :: [Int] -> [Int]
inc1 = map (+1)
sqr1 :: [Int] -> [Int]
sqr1 = map (^2)
```

Funktori

- koncept mapiranja funkcije na elemente liste može se generalizirati na širok spektar *parametriziranih tipova*

Definicija

Funktori *su klase tipova koje podržavaju mapirajuće funkcije. Mapirajuće funkcije su one funkcije koje se primjenjuju na elemente tipa.*

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

- da bi određeni parametrizirani tip bio instanca klase `Functor` mora podržavati funkciju `fmap` koja uzima funkciju tipa `a -> b` i strukturu tipa `f a` čiji su elementi tipa `a`, primjenjuje tu funkciju na svaki element te strukture i daje strukturu tipa `f b`

Funktori I

- listu možemo shvatiti kao funktor, odnosno klasu parametriziranih tipova na kojoj je `fmap` jednak `map`

```
instance Functor [] where
        -- fmap :: (a -> b) -> [a] -> [b]
        fmap = map
```

- tip `Maybe a` koji predstavlja tip `a` u slučaju uspjeha ili neuspjeha, možemo pridružiti klasi `Functor`

```
data Mozda a = Nista | Samo a deriving Show

instance Functor Mozda where
     -- fmap :: (a -> b) -> Mozda a -> Mozda b
     fmap _ Nista = Nista
     fmap g (Samo x) = Samo (g x)
```

Funktori II

- tip `Tree a` predstavlja tip binarnog stabla u kojem listovi sadrže podatke

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
            deriving Show

instance Functor Tree where
    -- fmap :: (a -> b) -> Tree a -> Tree b
    fmap g (Leaf x) = Leaf (g x)
    fmap g (Node l r) = Node (fmap g l) (fmap g r)

fmap (+1) (Node (Leaf 1) (Leaf 2))
```

- `IO` se može razmatrati kao funktor iako ga ne vidimo kao parametrizirani tip jer predstavlja ulazno/izlazne akcije

```
instance Functor IO where
      -- fmap :: (a -> b) -> IO a -> IO b
      fmap g mx = do { x <- mx; return (g x)}
```

Funktori

- funkcija `fmap` može se koristiti za procesiranje elemenata bilo kog tipa koji je funktor
- mogu se definirati generičke funkcije za bilo koji funktorski tip

```
inc2 :: Functor f => f Int -> f Int
inc2 = fmap (+1)

inc2 (Node (Leaf 1) (Leaf 2))
```

Funktori

Svojstva funktora

- funktori zadovoljavaju sljedeća svojstva:

```
fmap id = id

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

- `fmap` "čuva" funkciju identitete. Koji su tipovi funkcije `id` na lijevoj i desnoj strani?
- primjena `fmap` na kompoziciju funkcija jednaka je kompoziciji funkcija dobivenih primjenom `fmap`

Funktori

Svojstva funktora

- Primjer funktora koji ne zadovoljava prethodno navedena svojstva

```
instance Functor [] where
        -- fmap :: (a -> b) -> f a -> f b
        fmap g [] = []
        fmap g (x:xs) = fmap g xs ++ [g x]

fmap id [1,2]

id [1,2]

fmap (not . even) [1,2]

(fmap not . fmap even) [1,2]
```

Primjenjivi funktori (eng. Applicatives)

- moguće je **primijeniti** funkciju na **više** argumenata

```
fmap0 :: a -> f a

fmap1 :: (a -> b) -> f a -> f b

fmap2 :: (a -> b -> c) -> f a -> f b -> f c

fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
.
.
.
```

- `fmap1` je jednaka `fmap`, a `fmap0` je degenerativan slučaj jer funkcija koja se primjenjuje nema argumenata

Primjenjivi funktori (eng. Applicatives)

- ovakav pristup zahtjeva deklariranje svake od verzija klase `Functor` iako sve slijede isti obrazac
- problem se javlja jer takvih definicija može biti beskonačno mnogo
- često ne znamo unaprijed koliko puta će biti potrebno primijeniti danu funkciju
- koristit ćemo **kaskadiranje**
- generalizirat ćemo primjenu funkcije na argument `(a -> b) -> a -> b`

Primjenjivi funktori (eng. Applicatives) I

- `fmap` za funkcije s proizvoljnim brojem argumenata može se definirati na sljedeći način:

```
pure :: a -> f a

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

- `pure` konvertira vrijednost tipa `a` u strukturu upa `f a`
- `<*>` je generalizacija primjene funkcije u kojoj su funkcija, argument i rezultat unutar strukture `f`
- koristimo ih na sljedeći način:

```
pure g <*> x1 <*> x2 <*> ... <*> xn
```

- ovo je aplikativni stil koji generalizira primjenu funkcije na proizvoljnom broju argumenata:

```
g x1 x2 ... xn
```

gdje je funkcija `g` kaskadna funkcija koja prima n argumenata tipova `a1`,`a2`,...,`an` i vraća rezultat tipa `b`

Primjenjivi funktori (eng. Applicatives) II

- u aplikativnom stilu funkciju svaki argument je tipa `f ai`, a rezultat je tipa `f b`:

```
fmap0 :: a -> f a
fmap0 = pure

fmap1 :: (a -> b) -> f a -> f b
fmap1 g x = pure g <*> x

fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap2 g x y = pure g <*> x <*> y

fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
fmap3 g x y z = pure g <*> x <*> y <*> z

.
.
.
```

Primjenjivi funktori (eng. Applicatives)

- u Haskellu ovu klasu funktora, koji podržavaju `pure` i `<*>`, nazivamo **primjenjivi funktori**

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

Primjeri primjenjivih funktora

- u Haskellu ovu klasu funktora, koji podržavaju `pure` i `<*>`, nazivamo **primjenjivi funktori**

```
instance Applicative Mozda where
    -- pure :: a -> Mozda a
    pure = Samo

    -- (<*>) :: Mozda (a -> b) -> Mozda a -> Mozda b
    Nista <*> _ = Nista
    (Samo g) <*> mx = fmap g mx
```

Primjeri primjenjivih funktora I

- standardni `prelude` sadrži sljedeću deklaraciju liste kao primjenjivog funktora:

```
instance Applicative [] where
        -- pure :: a -> [a]
        pure x = [x]

        -- (<*>) :: [a -> b] -> [a] -> [b]
        gs <*> xs = [g x | g <- gs, x <- xs]

pure (+1) <*> [1,2,3,4,5]

pure (+) <*> [1] <*> [2]

pure (*) <*> [1,2] <*> [3,4]
```

- tip `[a]` vidimo kao generalizaciju tipa `Maybe a` koja dozvoljava više različitih rezultata u slučaju uspjeha
- ukoliko je jedna od listi prazna rezultat je neuspjeh, odnosno prazna lista

Primjeri primjenjivih funktora II

- posljednji primjer daje sve moguće načine primjene operatora plus na operande iz `[1,2]` i `[3,4]`