

Programare declarativă

Functori aplicativi¹

Ioana Leuştean
Traian Florin Şerbănuţă

Departamentul de Informatică, FMI, UNIBUC
traian.serbanuta@unibuc.ro

7 decembrie 2017

¹bazat pe [Learn You a Haskell for Great Good](#)

Problemă

- Dat fiind un **Functor** f putem “ridica” funcție $h :: a \rightarrow b$ la o funcție $\text{fmap } f :: f\ a \rightarrow f\ b$
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente
E.g., cum trecem de la $h :: a \rightarrow b \rightarrow c$ la $h' :: f\ a \rightarrow f\ b \rightarrow f\ c$
- putem încerca să folosim fmap

Problemă

- Dat fiind un **Functor** f putem “ridica” funcție $h :: a \rightarrow b$ la o funcție $\text{fmap } f :: f\ a \rightarrow f\ b$
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente
E.g., cum trecem de la $h :: a \rightarrow b \rightarrow c$ la $h' :: f\ a \rightarrow f\ b \rightarrow f\ c$
- putem încerca să folosim fmap
- Dar, deoarece $h :: a \rightarrow (b \rightarrow c)$, avem că
 $\text{fmap } h :: f\ a \rightarrow f\ (b \rightarrow c)$
- Putem aplica $\text{fmap } h$ la o valoare $ca :: f\ a$ și obținem
 $\text{fmap } h\ ca :: f\ (b \rightarrow c)$

Problemă

Cum transformăm un obiect din $h (b \rightarrow c)$ într-o funcție $f\ b \rightarrow f\ c$?

- **ap** $:: f\ (b \rightarrow c) \rightarrow (f\ b \rightarrow f\ c)$, sau, ca operator
- $(\langle * \rangle) :: f\ (b \rightarrow c) \rightarrow f\ b \rightarrow f\ c$

Merge pentru funcții cu oricâte argumente

Problemă

Dată fiind o funcție $f :: a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a$ și computațiile $ca_1 :: f\ a_1, \dots, ca_n :: f\ a_n$, vrem să „aplicăm” funcția f pe rând computațiilor ca_1, \dots, ca_n pentru a obține o computație finală $ca :: f\ a$.

- funcția $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
- funcția $(<*>) :: f\ (b \rightarrow c) \rightarrow f\ b \rightarrow f\ c$ cu „proprietăți bune”

Soluție

Merge pentru funcții cu oricâte argumente

Problemă

Data fiind o funcție $f :: a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a$ și computațiile $ca_1 :: f\ a_1, \dots, can :: f\ a_n$, vrem să „aplicăm” funcția f pe rând computațiilor ca_1, \dots, can pentru a obține o computație finală $ca :: f\ a$.

- funcția $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
- funcția $(<*>) :: f\ (b \rightarrow c) \rightarrow f\ b \rightarrow f\ c$ cu „proprietăți bune”

Soluție

$fmap\ h :: f\ a_1 \rightarrow f\ (a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

$fmap\ h\ ca_1 :: f\ (a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

$fmap\ h\ ca_1\ <*>\ ca_2 :: f\ (a_3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

...

$fmap\ h\ ca_1\ <*>\ ca_2\ <*>\ ca_3\ \dots\ <*>\ can :: f\ a$

Clasa de tipuri Applicative

Definiție

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

- Orice instanță a lui Applicative trebuie să fie instanță a lui **Functor**
- **pure** “ridică” o valoare într-o colecție minimală care conține acea valoare și nimic mai mult!
- **(<*>)** ia o colecție de care funcții și o colecție de argumente pentru funcții și obține colecția rezultatelor aplicării funcțiilor asupra argumentelor.

Proprietate importantă

- $\text{fmap } f \ x == \text{pure } f \ \text{<*>} \ x$
- Se definește operatorul **(<\$>)** prin $\text{(<$>)} = \text{fmap}$

Tipul opțiune

```
Main> pure "Hey" :: Maybe String
```

```
Just "Hey"
```

```
Main> (++) <$> (Just "Hey ") <*> (Just "You!")
```

```
Just "Hey You!"
```

```
Main> let mDiv x y = if y == 0 then Nothing else Just (x '
    div ' y)
```

```
Main> let f x = 4 + 10 'div' x
```

```
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Tipul opțiune

```

Main> pure "Hey" :: Maybe String
Just "Hey"
Main> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
Main> let mDiv x y = if y == 0 then Nothing else Just (x `
    div y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x

```

Instanță pentru tipul opțiune

```

instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    Just f <*> x = fmap f x

```


Tipul eroare

```
Main> pure "Hey" :: Either a String
Right "Hey"
Main> (++) <$> (Right "Hey ") <*> (Right "You!")
Right "Hey You!"
Main> let mDiv x y = if y == 0 then Left "Division by 0!"
    else Right (x 'div' y)
Main> let f x = 4 + 10 'div' x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Tipul eroare

```

Main> pure "Hey" :: Either a String
Right "Hey"
Main> (++) <$> (Right "Hey ") <*> (Right "You!")
Right "Hey You!"
Main> let mDiv x y = if y == 0 then Left "Division by 0!"
      else Right (x 'div' y)
Main> let f x = 4 + 10 'div' x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x

```

Instanță pentru tipul eroare

```

instance Applicative (Either a) where
  pure = Right
  Left e <*> _ = Left e
  Right f <*> x = fmap f x

```

Tipul listă

```

Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"] ["Hello world", "Hello happiness", "Goodbye
    world", "Goodbye happiness"]
Main> [(+), (*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]

```

Tipul listă

```

Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"] ["Hello world", "Hello happiness", "Goodbye
    world", "Goodbye happiness"]
Main> [(+), (*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]

```

Instanță pentru liste

```

instance Applicative [] where
    pure x = [x]
    fs    <*> xs = [f x | f <- fs, x <- xs]

```

Tipul funcțiilor de sursă dată

```
data Exp = Lit Int | Var String | Exp :+: Exp
```

```
type Env = [(String, Int)]
```

```
find :: String -> (Env -> Int)
```

```
find x env = head [i | (y,i) <- env, y == x]
```

```
eval :: Exp -> (Env -> Int)
```

```
eval (Lit i) = pure i
```

```
eval (Var x) = find x
```

```
eval (e1 :+: e2) = (+) <$> eval e1 <*> eval e2
```

Tipul funcțiilor de sursă dată

```
data Exp = Lit Int | Var String | Exp :+: Exp
type Env = [(String, Int)]
```

```
find :: String -> (Env -> Int)
find x env = head [i | (y,i) <- env, y == x]
```

```
eval :: Exp -> (Env -> Int)
eval (Lit i) = pure i
eval (Var x) = find x
eval (e1 :+: e2) = (+) <$> eval e1 <*> eval e2
```

Instanță pentru tipul funcțiilor de sursă dată

```
instance Applicative ((->) t) where
  pure :: a -> (t -> a)
  pure x = \ _ -> x
  (<*>) :: (t -> (a -> b)) -> (t -> a) -> (t -> b)
  f <*> g = \ x -> f x (g x)
```

Liste ca fluxuri de date.

```
newtype ZList a = ZList { get :: [a]}
```

```
> get $ max <$> ZList [1,2,3,4,5,3] <*> ZList [5,3,1,2]
[5,3,3,4]
```

```
> get $ (+) <$> ZList [1,2,3] <*> ZList [100,100..]
[101,102,103]
```

```
> get $ (,,) <$> ZList "dog" <*> ZList "cat" <*> ZList "rat"
[( 'd', 'c', 'r' ), ( 'o', 'a', 'a' ), ( 'g', 't', 't' )]
```

Liste ca fluxuri de date.

```
newtype ZList a = ZList { get :: [a]}
```

```
> get $ max <$> ZList [1,2,3,4,5,3] <*> ZList [5,3,1,2]
[5,3,3,4]
```

```
> get $ (+) <$> ZList [1,2,3] <*> ZList [100,100..]
[101,102,103]
```

```
> get $ (,,) <$> ZList "dog" <*> ZList "cat" <*> ZList "rat"
[( 'd', 'c', 'r' ), ( 'o', 'a', 'a' ), ( 'g', 't', 't' )]
```

Instanță pentru ZipList

```
instance Functor ZipList where
```

```
    fmap f (ZipList xs) = ZipList (fmap f xs)
```

```
instance Applicative ZipList where
```

```
    pure x = repeat x
```

```
    ZipList fs <*> ZipList xs =
```

```
        ZipList (zipWith (\ f x -> f x) fs xs)
```


Proprietăți ale functorilor aplicativi

identitate $\text{pure id } \langle * \rangle v = v$

compoziție $\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w = u \langle * \rangle (v \langle * \rangle w)$

homomorfism $\text{pure } f \langle * \rangle \text{pure } x = \text{pure } (f \ x)$

interschimbare $u \langle * \rangle \text{pure } y = \text{pure } (\$ \ y) \langle * \rangle u$

Consecință: $\text{fmap } f \ x == f \ \langle \$ \rangle x == \text{pure } f \ \langle * \rangle x$