Programare declarativă

Functori aplicativi[1]

Traian Florin Șerbănuță
Ioana Leuștean

Departamentul de Informatică, FMI, UB
traian.serbanuta@fmi.unibuc.ro
ioana@fmi.unibuc.ro

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

Problemă

```
Prelude> negate 3
-3

Prelude> negate (Just 3)
<interactive>:27:1: error:...

Prelude> negate <$> Just 3
Just (-3)
```

- Dat fiind un **Functor** f putem "ridica" funcție h :: a -> b la o funcție h' :: f a -> f b folosind fmap, i.e. h'=fmap f

Problemă

```
Prelude> negate 3
-3

Prelude> negate (Just 3)
<interactive>:27:1: error:...

Prelude> negate <$> Just 3
Just (-3)
```

- Dat fiind un **Functor** f putem "ridica" funcție h :: a -> b la o funcție h' :: f a -> f b folosind fmap, i.e. h'=fmap f
- Ce se întâmplă dacă avem o funcție cu mai multe argumente? E.g., cum trecem de la h :: a -> b -> c la h' :: f a -> f b -> f c

Problemă

- Ce se întâmplă dacă avem o funcție cu mai multe argumente?
  E.g., cum trecem de la `h :: a -> b -> c` la `h' :: f a -> f b -> f c`

```
Prelude> (Just 3) + (Just 3)
<interactive>:29:1: error:...
```

Problemă

- Ce se întâmplă dacă avem o funcție cu mai multe argumente?
  E.g., cum trecem de la h :: a -> b -> c la h' :: f a -> f b -> f c

```
Prelude> (Just 3) + (Just 3)
<interactive>:29:1: error:...
```

- Putem încerca să folosim fmap:
  - din h :: a -> (b -> c) obținem fmap h :: f a -> f (b -> c)

```
Prelude> :t fmap (+)
fmap (+) :: (Num a, Functor f) => f a -> f (a -> a)
```

  - putem aplica fmap h la o valoare ca :: f a și obținem
    fmap h ca :: f (b -> c)

```
Prelude> :t fmap (+) (Just 3)
fmap (+) (Just 3) :: Num a => Maybe (a -> a)
```

Problemă

- Dat fiind un **Functor** f putem "ridica" funcție h :: a -> b la o funcție h' :: f a -> f b folosind fmap, i.e. h'=fmap f
- Ce se întâmplă dacă avem o funcție cu mai multe argumente? E.g., cum trecem de la h :: a -> b -> c la h' :: f a -> f b -> f c
- Putem încerca să folosim fmap:

Problemă

- Dat fiind un **Functor** f putem "ridica" funcție h :: a -> b la o funcție h' :: f a -> f b folosind fmap, i.e. h'=fmap f
- Ce se întâmplă dacă avem o funcție cu mai multe argumente? E.g., cum trecem de la h :: a -> b -> c la h' :: f a -> f b -> f c
- Putem încerca să folosim fmap:
  - din h :: a -> (b -> c) obținem fmap h :: f a -> f (b -> c)
  - putem aplica fmap h la o valoare ca :: f a și obținem fmap h ca :: f (b -> c)

Problemă

Cum transformăm un obiect din f (b -> c) într-o funcție f b -> f c?

- **ap** :: f (b -> c) -> (f b -> f c), sau, ca operator
- (<*>) :: f (b -> c) -> f b -> f c

Problemă

Problemă

Cum transformăm un obiect din f (b -> c) într-o funcție f b -> f c?

- **ap** :: f (b -> c) -> (f b -> f c)

```
Prelude Control.Monad> :t ap
ap :: Monad m => m (a -> b) -> m a -> m b
```

- (<*>) :: f (b -> c) -> f b -> f c

```
Prelude> :t (<*>)
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

```
Prelude Control.Monad> ap (fmap (+) (Just 3) )(Just 4)
Just 7
```

```
Prelude> (fmap (+) (Just 3) ) <*>(Just 4)
Just 7
```

Operatorii (<*>) și (<$>)

Problemă

Cum transformăm un obiect din f (b -> c) într-o funcție f b -> f c?

- (<*>) :: f (b -> c) -> f b -> f c

```
Prelude> (fmap (+) (Just 3) ) <*>(Just 4)
Just 7
```

- se definește operatorul (<$>) prin (<$>) = fmap

```
Prelude> (+) <$> (Just 3) <*> (Just 4)
Just 7
Prelude> (+) <$> Nothing <*> (Just 4)
Nothing
Prelude> (*) <$> (Just 3) <*> (Just 4)
Just 12
```

Merge pentru funcții cu oricâte argumente

Problemă

Dată fiind o funcție h :: a1 -> a2 -> a3 -> ... -> an -> a și computațiile ca1 :: f a1, . . . , can :: f an, vrem să „aplicăm” funcția *h* pe rând computațiilor ca1, . . . , can pentru a obține o computație finală ca :: f a.

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

Soluție

Merge pentru funcții cu oricâte argumente

Problemă

Dată fiind o funcție h :: a1 -> a2 -> a3 -> ... -> an -> a și computațiile ca1 :: f a1, . . . , can :: f an, vrem să „aplicăm” funcția *h* pe rând computațiilor ca1, . . . , can pentru a obține o computație finală ca :: f a.

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

Soluție

fmap h :: f a1 -> f (a2 -> a3 -> ... -> an -> a)

fmap h ca1 :: f (a2 -> a3 -> ... -> an -> a)

fmap h ca1 <*> ca2 :: f (a3 -> ... -> an -> a)

. . .

fmap h ca1 <*> ca2 <*> ca3 ... <*> 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

- fmap f x == pure f <*> 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
Main> mF 0
Main> Nothing
```

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
Main> mF 0
Main> Nothing
```

Instanță pentru tipul opțiune

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

Tipul eroare

```
Main> pure "Hey" :: Either e 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 e 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 e) where
  pure = Right
  Left f <*> _ = Left f
  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]

>:t (,,)
(,,) :: a -> b -> c -> (a, b, c)
> 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 ZList where
  fmap f (ZList xs) = ZList (fmap f xs)
```

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 ZList where
  fmap f (ZList xs) = ZList (fmap f xs)

instance Applicative ZList where
  pure x = ZList (repeat x)   -- repeat x =[x,x,x,...]
  ZList fs <*> ZList xs =
    ZList (zipWith (\ f x -> f x) fs xs)
```

Proprietăți ale functorilor aplicativi

Identitate `pure id <*> v = v`

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

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

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

Consecință: `fmap f x == f <$> x == pure f <*> x`