

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C07-C08

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Functori

Va aduceti aminte functia

```
map :: (a -> b) -> [a] -> [b]
```

Problema. Putem generaliza aceasta functie la alte tipuri parametrizate?

Definiție

```
class Functor f where
```

```
    fmap :: (a -> b) -> f a -> f b
```

Data fiind o funcție $f :: a \rightarrow b$ și $ca :: f\ a$, `fmap` produce $cb :: f\ b$ obținută prin transformarea rezultatelor produse de computația `ca` folosind funcția `f` (și doar atât!)

Instanță pentru liste

```
instance Functor [] where
```

```
    fmap = map
```

Clasa de tipuri Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

Instanță pentru tipul optiune

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
instance Functor Maybe where
```

```
  fmap f Nothing = Nothing
```

```
  fmap f (Just x) = Just (f x)
```

Clasa de tipuri Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

Instanță pentru tipul eroare

```
fmap :: (a -> b) -> Either e a -> Either e b
```

```
instance Functor (Either e) where
```

```
  fmap _ (Left x) = Left x
```

```
  fmap f (Right y) = Right (f y)
```

Clasa de tipuri Functor

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

```
data Arbore a = Nil  
             | Nod a Arbore Arbore
```

Instanță pentru tipul arbore

fmap :: (a -> b) -> Arbore a -> Arbore b

```
instance Functor Arbore where  
    fmap f Nil = Nil  
    fmap f (Nod x l r) = Nod (f x) (fmap f l) (fmap f r)
```

Clasa de tipuri Functor

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

Tipul funcțiilor de sursă dată **t -> a** (parametric in a)

Instanță pentru tipul funcție

```
fmap :: (a -> b) -> (t -> a) -> (t -> b)
```

```
instance Functor (->) a where
```

```
  fmap f g = f . g  -- sau, mai simplu, fmap = (.)
```


Example

```
Prelude> fmap (*2) [1..3]
```

```
[2,4,6]
```

```
Prelude> fmap (*2) (Just 200)
```

```
Just 400
```

```
Prelude> fmap (*2) Nothing
```

```
Nothing
```

```
Prelude> fmap (*2) (+100) 4
```

```
208
```

```
Prelude> fmap (*2) (Right 6)
```

```
Right 12
```

```
Prelude> fmap (*2) (Left 135)
```

```
Left 135
```

```
Prelude> (fmap . fmap) (+1) [Just 1, Just 2, Just 3]
```

```
[Just 2, Just 3, Just 4]
```

Proprietăți ale functorilor

- Argumentul **f** al lui **Functor f** definește o transformare de tipuri
 - **f a** este tipul **a** transformat prin functorul **f**
- **fmap** definește transformarea corespunzătoare a funcțiilor
 - **fmap :: (a -> b) -> (f a -> f b)**

Contractul lui **fmap**

- **fmap f ca** e obținută prin transformarea rezultatelor produse de computația **ca** folosind funcția **f** (și doar atât!)
- Abstractizat prin două legi:
 - identitate** **fmap id == id**
 - compunere** **fmap (g . h) == fmap g . fmap h**

Invalidarea contractului - identitate

```
data WhoCares a = ItDoesnt
                | Matter a
                | WhatThisIsCalled
deriving (Eq, Show)
```

Instanta a clasei Functor care invalideaza conditia de conservare a identitatii:

```
instance Functor WhoCares where
    fmap _ ItDoesnt = WhatThisIsCalled
    fmap _ WhatThisIsCalled = ItDoesnt
    fmap f (Matter a) = Matter (f a)
```

```
Prelude> fmap id ItDoesnt
WhatThisIsCalled
Prelude> id ItDoesnt
ItDoesnt
```

Validarea contractului - identitate

```
data WhoCares a = ItDoesnt
                | Matter a
                | WhatThisIsCalled
deriving (Eq, Show)
```

Instanta a clasei Functor care valideaza conditia de conservare a identitatii:

```
instance Functor WhoCares where
    fmap _ ItDoesnt = ItDoesnt
    fmap _ WhatThisIsCalled = WhatThisIsCalled
    fmap f (Matter a) = Matter (f a)
```

```
Prelude> fmap id ItDoesnt
ItDoesnt
Prelude> id ItDoesnt
ItDoesnt
```

Invalidarea contractului - compunere

```
data CountingBad a =  
    Heisenberg Int a  
    deriving (Eq, Show)
```

Instanta a clasei Functor care invalideaza conditia de conservare a compunerii:

```
instance Functor CountingBad where  
    fmap f (Heisenberg n a) = Heisenberg (n+1) (f a)
```

```
Prelude> oneWhoKnocks = Heisenberg 0 "Uncle"
```

```
Prelude> f = (++ " Jesse")
```

```
Prelude> g = (++ " lol")
```

```
Prelude> fmap (f . g) oneWhoKnocks
```

```
Heisenberg 1 "Uncle lol Jesse"
```

```
Prelude> fmap f . fmap g $ oneWhoKnocks
```

```
Heisenberg 2 "Uncle lol Jesse"
```

Validarea contractului - compunere

```
data CountingBad a =  
    Heisenberg Int a  
    deriving (Eq, Show)
```

Instanta a clasei Functor care valideaza conditia de conservare a compunerii:

```
instance Functor CountingBad where  
    fmap f (Heisenberg n a) = Heisenberg n (f a)
```

```
Prelude> oneWhoKnocks = Heisenberg 0 "Uncle"
```

```
Prelude> f = (++ " Jesse")
```

```
Prelude> g = (++ " lol")
```

```
Prelude> fmap (f . g) oneWhoKnocks
```

```
Heisenberg 0 "Uncle lol Jesse"
```

```
Prelude> fmap f . fmap g $ oneWhoKnocks
```

```
Heisenberg 0 "Uncle lol Jesse"
```

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvoR5>

Seria 24: <https://questionpro.com/t/AT4NiZvmKW>

Seria 25: <https://questionpro.com/t/AT4qgZvoR8>

Functori applicativi

Va aduceti aminte functia

```
map :: (a -> b) -> [a] -> [b]
```

Problema. Putem generaliza aceasta functie la alte tipuri parametrizate?

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

Problemă

- Folosind **fmap** putem transforma o funcție **$h :: a \rightarrow b$** într-o funcție **$fmap\ h :: m\ a \rightarrow m\ b$**
- 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
 $h' :: m\ a \rightarrow m\ b \rightarrow m\ c$
- Putem încerca să folosim **fmap**

Problemă

- Folosind **fmap** putem transforma o funcție **$h :: a \rightarrow b$** într-o funcție **$fmap\ h :: m\ a \rightarrow m\ b$**
- 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 $h' :: m\ a \rightarrow m\ b \rightarrow m\ c$
- Putem încerca să folosim **fmap**
- Dar, deoarece **$h :: a \rightarrow (b \rightarrow c)$** , obținem **$fmap\ h :: m\ a \rightarrow m\ (b \rightarrow c)$**
- Putem aplica **fmap h** la o valoare **$ca :: m\ a$** și obținem **$fmap\ h\ ca :: m\ (b \rightarrow c)$**

Cum transformăm un obiect din $\mathbf{m}(\mathbf{b} \rightarrow \mathbf{c})$ într-o funcție $\mathbf{m} \mathbf{b} \rightarrow \mathbf{m} \mathbf{c}$?

- $\mathbf{ap} :: \mathbf{m}(\mathbf{b} \rightarrow \mathbf{c}) \rightarrow (\mathbf{m} \mathbf{b} \rightarrow \mathbf{m} \mathbf{c})$, sau, ca operator
- $(\langle * \rangle) :: \mathbf{m}(\mathbf{b} \rightarrow \mathbf{c}) \rightarrow \mathbf{m} \mathbf{b} \rightarrow \mathbf{m} \mathbf{c}$

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

$$ca_1 :: m\ a_1, ca_2 :: m\ a_2, \dots, ca_n :: m\ a_n,$$

vrem să „aplicăm” funcția f pe rând computațiilor ca_1, \dots, ca_n pentru a obține o computație finală $ca :: m\ a$.

Merge pentru funcții cu oricâte argumente

Date fiind

- $f :: a1 \rightarrow a2 \rightarrow a3 \rightarrow \dots \rightarrow a_n \rightarrow a$
- $ca1 :: m\ a1, ca2 :: m\ a2, \dots, can :: m\ a_n,$
- $fmap :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$
- $(<*>) :: m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$ cu „proprietăți bune”

Atunci

$fmap\ f :: m\ a1 \rightarrow m\ (a2 \rightarrow a3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

$fmap\ f\ ca1 :: m\ (a2 \rightarrow a3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

$fmap\ f\ ca1\ <*>\ ca2 :: m\ (a3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

...

$fmap\ f\ ca1\ <*>\ ca2\ <*>\ ca3\ \dots\ <*>\ can :: m\ a$

Clasa de tipuri Applicative

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

- Orice instanță a lui **Applicative** trebuie să fie instanță a lui **Functor**
- `pure` transformă o valoare într-o computație minimală care are acea valoare ca rezultat, și nimic mai mult!
- `(<*>)` ia o computație care produce funcții și o computație care produce argumente pentru funcții și obține o computație care produce rezultatele aplicării funcțiilor asupra argumentelor

Clasa de tipuri Applicative

```
class Functor m where
```

```
  fmap :: (a -> b) -> m a -> m b
```

```
class Functor m => Applicative m where
```

```
  pure  :: a -> m a
```

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

Proprietate importantă

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

$$\begin{aligned}(\$) &:: (a \rightarrow b) \rightarrow a \rightarrow b \\(<\$>) &:: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b \\(<*>) &:: m\ (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b\end{aligned}$$

```
class Functor m where
```

```
  fmap :: (a -> b) -> m a -> m b
```

```
class Functor m => Applicative m where
```

```
  pure :: a -> m a
```

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

```
instance Applicative Maybe where
```

```
  pure = Just
```

```
  Nothing <*> _ = Nothing
```

```
  Just f <*> x = fmap f x
```

Instante – Maybe

```
Prelude> pure "Hey" :: Maybe String  
Just "Hey"
```

Cum concatenam **Just** "Hey" cu **Just** "You!"?

Instante – Maybe

```
Prelude> pure "Hey" :: Maybe String  
Just "Hey"
```

Cum concatenam **Just "Hey"** cu **Just "You!"**?

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

Instante – Maybe

```
Prelude> pure "Hey" :: Maybe String  
Just "Hey"
```

Cum concatenam **Just "Hey"** cu **Just "You!"**?

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

- **(++) :: String -> (String -> String)**
- **Just "Hey_" :: Maybe String**
- **(<\$>) :: (a -> b) -> m a -> m b** (este fmap)
- **(++) <\$> (Just "Hey_") :: Maybe (String -> String)**
- **Just "You!" :: Maybe String**
- **(<*>) :: m (b -> c) -> m b -> m c**
- **Just "Hey_You!" :: Maybe String**

Instante – Maybe

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

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

- $(+)$:: **Int** \rightarrow **Int** \rightarrow **Int**
- `pure 4` :: **Maybe Int**
- $(\<\$>)$:: $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(+) \<\$> \text{pure } 4$:: **Maybe (Int \rightarrow Int)**
- `mDiv` :: **Int** \rightarrow **Int** \rightarrow **Maybe Int**
- `mDiv 10 x` :: **Maybe Int**
- $(\<*>)$:: $m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$

```
Prelude> mF 2
```

```
Just 9
```

```
class Functor m where
```

```
  fmap :: (a -> b) -> m a -> m b
```

```
class Functor m => Applicative m where
```

```
  pure :: a -> m a
```

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

```
instance Applicative (Either a) where
```

```
  pure = Right
```

```
  Left e   <*> _ = Left e
```

```
  Right f <*> x = fmap f x
```

Instante – Either

```
Prelude> pure "Hey" :: Either a String  
Right "Hey"
```

```
Prelude> (++) <$> (Right "Hey ") <*> (Right "You!")  
Right "Hey You!"
```

- **(++) :: String -> (String -> String)**
- **Right "Hey" :: Either a String**
- **(<\$>) :: (a -> b) -> m a -> m b** (este fmap)
- **(++) <\$> (Right "Hey ") :: Either a (String -> String)**
- **Right "You!" :: Either a String**
- **(<*>) :: m (b -> c) -> m b -> m c**
- **Right "Hey You!" :: Either a String**

Instante – Either

```
eDiv x y = if y == 0 then Left "Division by 0!"  
           else Right (x 'div' y)
```

```
eF x = (+) <$> pure 4 <*> eDiv 10 x
```

- $(+)$:: **Int** \rightarrow **Int** \rightarrow **Int**
- `pure 4` :: **Either String Int**
- $(\<\$>)$:: $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(+) \<\$> \text{pure } 4$:: **Either String (Int \rightarrow Int)**
- `eDiv` :: **Int** \rightarrow **Int** \rightarrow **Either String Int**
- `eDiv 10 x` :: **Either String Int**
- $(\<*>)$:: $m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$

```
Prelude> eF 2
```

```
Right 9
```

```
class Functor m where
```

```
  fmap :: (a -> b) -> m a -> m b
```

```
class Functor m => Applicative m where
```

```
  pure :: a -> m a
```

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

```
instance Applicative [] where
```

```
  pure x = [x]
```

```
  fs <*> xs = [f x | f <- fs, x <- xs]
```

Instante – Liste

```
Prelude> pure "Hey" :: [String]  
["Hey"]
```

```
Prelude> (++) <$> ["Hello ", "Goodbye "] <*> ["world"  
    , "happiness"]  
["Hello world", "Hello happiness", "Goodbye world", "  
    Goodbye happiness"]
```

- $(++) :: \text{String} \rightarrow (\text{String} \rightarrow \text{String})$
- $["\text{Hello}_\perp", "\text{Goodbye}_\perp"] :: [\text{String}]$
- $(\<\$>) :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(++) \<\$> ["\text{Hello}_\perp", "\text{Goodbye}_\perp"] :: [\text{String} \rightarrow \text{String}]$
- $["\text{world}", "\text{happiness}] :: [\text{String}]$
- $(\<*>) :: m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$

```
Prelude> (+) <$> [1,2] <*> [3,4]  
[4,5,5,6]
```

- $(+)$:: **Int** \rightarrow **Int** \rightarrow **Int**
- $[1,2]$:: [**Int**]
- $(\<\$>)$:: $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(\<*>)$:: $m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$
- $[(+), (*)]$ <*> $[1,2]$:: [**Int** \rightarrow **Int**]
- $[(+), (*)]$ <*> $[1,2]$ <*> $[3,4]$:: [**Int**]

```
Prelude> [(+) , (*)] <*> [1,2] <*> [3,4]  
[4,5,5,6,3,4,6,8]
```

- $(+), (*) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
- $[(+), (*)] :: [\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}]$
- $[1,2] :: [\text{Int}]$
- $(<*>) :: m (b \rightarrow c) \rightarrow m b \rightarrow m c$
- $[(+), (*)] <*> [1,2] :: [\text{Int} \rightarrow \text{Int}]$
- $[(+), (*)] <*> [1,2] <*> [3,4] :: [\text{Int}]$

```
Prelude> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]  
[55,80,100,110]
```

- $(*) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
- $[2,5,10] :: [\text{Int}]$
- $(<\$>) :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(*) <\$> [2,5,10] :: [\text{Int} \rightarrow \text{Int}]$
- $(<*>) :: m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$
- $(*) <\$> [2,5,10] <*> [8,10,11] :: [\text{Int}]$

```
Prelude> (*) <$> [2,5,10] <*> [8,10,11]  
[16,20,22,40,50,55,80,100,110]
```

Proprietăți ale functorilor aplicativi

```
class Functor m where
```

```
  fmap :: (a -> b) -> m a -> m b
```

```
class Functor m => Applicative m where
```

```
  pure :: a -> m a
```

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

- **identitate** pure id <*> v = v
- **compoziție** pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
- **homomorfism** pure f <*> pure x = pure (f x)

Consecință: $\text{fmap } f \ x == f \ \<\$> \ x == \text{pure } f \ \<*> \ x$

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvoSC>

Seria 24: <https://questionpro.com/t/AT4NiZvmNw>

Seria 25: <https://questionpro.com/t/AT4qgZvoSD>

Pe săptămâna viitoare!