

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C07

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Functori

Map

În cursurile trecute, am văzut funcția

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

Problema. Putem generaliza această funcție la alte tipuri parametrizate?

Clasa de tipuri Functor

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

Pentru o funcție $g :: a \rightarrow b$ și o valoare $ca :: f a$,
 $fmap$ produce $cb :: f b$,
obținută prin transformarea lui ca folosind funcția g (și doar atât!).

Instanță pentru liste

```
instance Functor [] where
    fmap = map
```

```
Prelude> fmap (*2) [1..3]
[2,4,6]
```

```
Prelude> map (*2) [1..3]
[2,4,6]
```

```
Prelude> fmap (++"1") ["1", "2", "3"]
["11", "21", "31"]
```

Instanță pentru Maybe

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

Instanță pentru Maybe

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

```
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

Exemple

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

```
Just 400
```

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

```
Nothing
```

```
Prelude> fmap (++"_world!") (Just "Hello")
```

```
Just "Hello_world!"
```

Instanță pentru Either

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

Instanță pentru Either e

```
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)
```

Exemple

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

```
Right 12
```

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

```
Left 135
```

Instanță pentru Tree

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

Instanță pentru Tree

```
data Tree a = Empty
            | Node a (Tree a) (Tree a)
```

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

```
instance Functor Tree where
```

```
    fmap f Empty = Empty
```

```
    fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

Instanță pentru tipul funcție

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

Instanță pentru tipul funcție

Tipul funcțiilor de sursă dată parametric în tipul a este $t \rightarrow a$.

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

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

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

Exemple

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

```
208
```

```
Prelude> (fmap . fmap) (+1) [Just 1, Just 2, Just 3]  
[Just 2, Just 3, Just 4]
```

fmap ca operator

```
fmap :: Functor f => (a -> b) -> f a -> f b  
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

Exemple

```
(+1) <$> [1,2,3]      ==> [2,3,4]  
not <$> Just False    ==> Just True
```

```
reverse . tail $      "hello"      ==> "olle"  
reverse . tail <$> Just "hello"    ==> Just "olle"  
-- echivalent cu  
fmap (reverse . tail) (Just "hello") ==> Just "olle"
```

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 g ca** e obținută prin transformarea rezultatelor produse de computația ca folosind funcția **g** (și doar atât!)
- Abstractizat prin două legi:

identitate $\text{fmap id} == \text{id}$

componere $\text{fmap (g . h)} == \text{fmap g . fmap h}$

Invalidarea contractului - identitate

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

Instanță a clasei Functor care invalidează condiția de conservare a identității:

```
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)
```

Instanță a clasei Functor care validează condiția de conservare a identității:

```
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 =  
    Sheldon Int a  
deriving (Eq, Show)
```

Instantă a clasei Functor care invalidează condiția de conservare a compunerii:

```
instance Functor BigKnockTheory where  
    fmap f (Sheldon n a) = Sheldon (n+1) (f a)
```

```
Prelude> howManyPauses = Sheldon 1 "knock_knock"  
Prelude> f = (++"_Penny")  
Prelude> g = (++"_knock")  
Prelude> fmap (f . g) howManyPauses  
Sheldon 2 "knock_knock_knock_Penny"  
Prelude> fmap f . fmap g $ howManyPauses  
Sheldon 3 "knock_knock_knock_Penny"
```

Validarea contractului - compunere

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

Instantă a clasei Functor care validează condiția de conservare a compunerii:

```
instance Functor BigKnockTheory where  
    fmap f (Sheldon n a) = Sheldon n (f a)
```

```
Prelude> howManyLaughs = Sheldon 0 "knock_\nknock"  
Prelude> f = (++"_Penny")  
Prelude> g = (++"_knock")  
Prelude> fmap (f . g) howManyLaughs  
Sheldon 0 "knock_\nknock_\nknock_\nPenny"  
Prelude> fmap f . fmap g $ howManyLaughs  
Sheldon 0 "knock_\nknock_\nknock_\nPenny"
```

Quiz Time!



<https://tinyurl.com/PF-C07-Quiz1>

Functori applicativi

Problema

Cum putem "concatena" **Just** "Hey" cu **Just** "You!"?

Vrem să obținem **Just** "Hey„You!"

Problema

Cum putem "concatena" **Just** "Hey" cu **Just** "You!"?

Vrem să obținem **Just** "Hey_You!"

```
Prelude> :t (++) <$> (Just "Hey_")
(++) <$> (Just "Hey_") :: Maybe (String -> String)
```

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

Problema

- 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?
De exemplu, 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
 $fmap\ h\ ca :: m\ (b \rightarrow c)$

Problema

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

- **ap** :: $m\ (b \rightarrow c) \rightarrow (m\ b \rightarrow m\ c)$, sau, ca operator
- **(<*>)** :: $m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$

Problema

Merge pentru funcții cu oricâte argumente!

Dată 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 $ca :: m\ a$.

Cazul general

Date fiind

- $f :: a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a$
- $ca_1 :: m\ a_1, ca_2 :: m\ a_2, \dots, ca_n :: m\ a_n,$
- $fmap :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$
- $(\langle * \rangle) :: m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c \text{ cu } \text{„proprietăți bune”}$

Atunci

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

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

$fmap\ f\ ca_1 \langle * \rangle ca_2 :: m\ (a_3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

...

$fmap\ f\ ca_1 \langle * \rangle ca_2 \langle * \rangle ca_3 \dots \langle * \rangle ca_n :: m\ a$

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

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

Instanță pentru Maybe

```
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
```

Instanță pentru Maybe

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

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

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

```
(++) :: String -> (String -> String)  
Just "Hey" :: Maybe String  
(<$>) :: (a -> b) -> m a -> m b  
(++) <$> (Just "Hey") :: Maybe (String -> String)  
Just "You!" :: Maybe String  
(<*>) :: m (a -> b) -> m a -> m b  
Just "HeyYou!" :: Maybe String
```

Instanță pentru Maybe

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

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

```
Prelude> mF 2  
Just 9
```

```
(+) :: Int -> Int -> Int  
pure 4 :: Maybe Int  
(<$>) :: (a -> b) -> m a -> m b  
(+) <$> pure 4 :: Maybe (Int -> Int)  
mDiv :: Int -> Int -> Maybe Int  
mDiv 10 x :: Maybe Int  
(<*>) :: m (b -> c) -> m b -> m c
```

Instanță pentru Either

```
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
```

Instanță pentru Either

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

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

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

Instanță pentru Either

```
eDiv x y = if y == 0 then Left "Division\u00e7 by \u00e70!"  
            else Right (x `div` y)  
  
eF x = (+) <$> pure 4 <*> eDiv 10 x
```

```
Prelude> eF 2  
Right 9
```

```
(+) :: Int -> Int -> Int  
pure 4 :: Either String Int  
(<$>) :: (a -> b) -> m a -> m b  
(+) <$> pure 4 :: Either String (Int -> Int)  
eDiv :: Int -> Int -> Either String Int  
eDiv 10 x :: Either String Int  
(<*>) :: m (b -> c) -> m b -> m c
```

Instanță pentru liste

```
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]
```

Instanță pentru liste

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

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

```
(++) :: String -> (String -> String)
["Hello", "Goodbye"] :: [String]
(<$>) :: (a -> b) -> m a -> m b
(++) <$> ["Hello", "Goodbye"] :: [String -> String]
["world", "happiness"] :: [String]
(<*>) :: m (b -> c) -> m b -> m c
```

Instanță pentru liste

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

```
(+) :: Int -> Int -> Int  
[1,2] :: [Int]  
(<$>) :: (a -> b) -> m a -> m b  
(<*>) :: m (b -> c) -> m b -> m c
```

Instanță pentru liste

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

```
(+) ,(*) :: Int -> Int -> Int
[(+),(*)] :: [Int -> Int -> Int]
[1,2] :: [Int]
(<*>) :: m (b -> c) -> m b -> m c
[(+),(*)] <*> [1,2] :: [Int -> Int]
[(+),(*)] <*> [1,2] <*> [3,4] :: [Int]
```

Recap

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

Quiz Time!



<https://tinyurl.com/PF-C07-Quiz2>

Pe săptămâna viitoare!