

# Programare declarativă

## Clasa de tipuri Functor

Ioana Leuştean  
Traian Florin Şerbănuţă

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

8 decembrie 2017

# Colecții generalizate

# Tipuri parametrizate — „cutii” conținând obiecte

## Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii”, recipiente care pot conține elemente de tipul dat ca argument.

## Exemple

- Clasa de tipuri opțiune asociază unui tip  $a$ , tipul **Maybe**  $a$ 
  - cutii goale: **Nothing**
  - cutii care țin un element  $x$  de tip  $a$ : **Just**  $x$
- Clasa de tipuri listă asociază unui tip  $a$ , tipul  $[a]$ 
  - cutii care țin 0, 1, sau mai multe elemente de tip  $a$ :  $[1, 2, 3]$ ,  $[]$ ,  $[5]$

# Tipuri parametrizate — „cutii”

## Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii”, recipiente care pot conține elemente de tipul dat ca argument.

## Exemplu: tip de date pentru arbori binari

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

- Un arbore este o „cutie” care poate ține 0, 1, sau mai multe elemente de tip a:  
Nod 3 Nil (Nod 4 (Nod 2 Nil Nil) Nil), Nil, Nod 3 Nil Nil

# Tipuri parametrizate — „cutii

## Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii”, recipiente care pot conține elemente de tipul dat ca argument.

## Exemplu: tipul funcțiilor de sursă dată

- $t \rightarrow a$  descrie o colecție (infinită) de obiecte de tip  $a$  indexate de obiecte de tip  $b$ ; pentru orice intrare de tip  $t$  produce un rezultat de tip  $a$ 
  - $(++ \text{ "!"} ) :: \text{String} \rightarrow \text{String}$  este colecția șirurilor terminate în “!”, indexate de prefixele lor
  - $\text{length} :: \text{String} \rightarrow \text{Int}$  este colecția numerelor naturale indexate de șiruri care au acea lungime
  - $\text{id} :: \text{String} \rightarrow \text{String}$  este o colecția șirurilor de calcul care produce șirul dat ca argument

# Clase de tipuri pentru transformarea colecțiilor?

## Analogie

- Colecțiile de mai sus, asemeni listelor conțin obiecte.
- Funcția **map** ::  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ 
  - modifică fiecare element al unei liste (și tipul acestuia)
  - pe baza unei funcții date ca argument
  - fără a modifica structura listei
  - **map** ::  $(a \rightarrow b) \rightarrow ([a] \rightarrow [b])$

## Observație

Ideea de transformare a elementelor unei colecții conform unei funcții poate fi generalizată la orice fel de colecție.

## Întrebare

Cum transformăm o transformare a unui element într-o transformare a unei colecții?

# Functori

# Problemă

## Formulare cu cutii

Dată fiind o funcție  $f :: a \rightarrow b$  și o cutie  $ca$  care conține elemente de tip  $a$ , vreau să obțin o cutie  $cb$  care conține elemente de tip  $b$  obținute prin transformarea elementelor din cutia  $ca$  folosind funcția  $f$  (și doar atât!)

## Exemplu — liste

Dată fiind o funcție  $f :: a \rightarrow b$  și o listă  $la$  de elemente de tip  $a$ , vreau să obțin o listă de elemente de tip  $b$  transformând fiecare element din  $la$  folosind funcția  $f$  (și doar atât!)



# Clasa de tipuri Functor

## Definiție

```
class Functor m where
```

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

Data fiind o funcție  $f :: a \rightarrow b$ , `fmap f` transformă colecții de tipul `m a` în colecții de tipul `m b`

- transformă fiecare element al colecției folosind `f`
- fără a afecta structura colecției

## Instanță pentru liste

```
instance Functor [] where
```

```
  fmap = map
```

# Clasa de tipuri Functor

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind f
- fără a afecta structura colecției

Instanță pentru tipul opțiune `fmap :: (a -> b) -> Maybe a -> Maybe b`

```
data Maybe a = Nothing  
              | Just a
```

# Clasa de tipuri Functor

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind  $f$
- fără a afecta structura colecției

Instanță pentru tipul opțiune  $\text{fmap} :: (a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b$

```
data Maybe a = Nothing  
  | Just a
```

```
instance Functor Maybe where
```

```
  fmap f Nothing = Nothing
```

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

# Clasa de tipuri Functor

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind f
- fără a afecta structura colecției

Instanță pentru tipul arbore `fmap :: (a -> b) -> Arbore a -> Arbore b`

```
data Arbore a = Nil
```

```
          | Nod a Arbore Arbore
```

# Clasa de tipuri Functor

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind  $f$
- fără a afecta structura colecției

Instanță pentru tipul arbore  $\text{fmap} :: (a \rightarrow b) \rightarrow \text{Arbore } a \rightarrow \text{Arbore } b$

```
data Arbore a = Nil
```

```
  | Nod a Arbore Arbore
```

```
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

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind `f`
- fără a afecta structura colecției

Instanță pentru tipul eroare `fmap :: (a -> b) -> Either e a -> Either e b`

```
data Either a b = Left a  
                | Right b
```

# Clasa de tipuri Functor

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind  $f$
- fără a afecta structura colecției

Instanță pentru tipul eroare  $\text{fmap} :: (a \rightarrow b) \rightarrow \text{Either } e \ a \rightarrow \text{Either } e \ b$

```
data Either a b = Left a  
               | Right b
```

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

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

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

# Clasa de tipuri Functor

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind  $f$
- fără a afecta structura colecției

Instanță pentru tipul funcție  $\text{fmap} :: (a \rightarrow b) \rightarrow (t \rightarrow a) \rightarrow (t \rightarrow b)$



# Clasa de tipuri Functor

## Instanțe

```
class Functor f where
```

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

- transformă fiecare element al colecției folosind  $f$
- fără a afecta structura colecției

Instanță pentru tipul funcție  $\text{fmap} :: (a \rightarrow b) \rightarrow (t \rightarrow a) \rightarrow (t \rightarrow b)$

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

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

# Example

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

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

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

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

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

```
Main> fmap (*2) (Left 1)
```

# Example

```
Main> fmap (*2) [1..3]
[2,4,6]
Main> fmap (*2) (Just 200)
Just 400
Main> fmap (*2) Nothing
Nothing
Main> fmap (*2) (+100) 4
208
Main> fmap (*2) (Right 6)
Right 12
Main> fmap (*2) (Left 135)
Left 135
```

# Proprietăți ale functorilor

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

## Contractul lui $fmap$

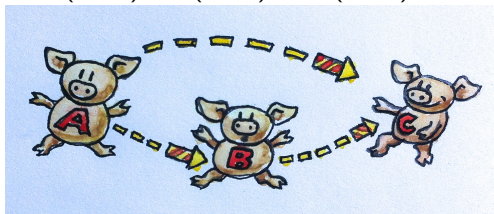
- $fmap\ f\ ca$  e obținută prin transformarea rezultatelor produse de computația  $ca$  folosind funcția  $f$  (și doar atât!)
- Abstractizat prin două legi:
  - identitate**  $fmap\ id == id$
  - compunere**  $fmap\ (g . f) == fmap\ g . fmap\ f$

# Categorii și Functori

# Categorii

O categorie  $\mathbb{C}$  este dată de:

- O clasă  $|\mathbb{C}|$  a **obiectelor**
- Pentru oricare două obiecte  $A, B \in |\mathbb{C}|$ ,  
o mulțime  $\mathbb{C}(A, B)$  a **săgeților** „de la  $A$  la  $B$ ”  
 $f \in \mathbb{C}(A, B)$  poate fi scris ca  $f : A \rightarrow B$
- Pentru orice obiect  $A$  o săgeată  $id_A : A \rightarrow A$  numită **identitatea** lui  $A$
- Pentru orice obiecte  $A, B, C$ , o operație de compunere a săgeților  
 $\circ : \mathbb{C}(B, C) \times \mathbb{C}(A, B) \rightarrow \mathbb{C}(A, C)$



Bartosz Milewski  
— Category: The  
Essence of Composition

- Compunerea este asociativă și are element neutru  $id$

# Exemplu: Categoria Set

- Obiecte: mulțimi
- Săgeți: funcții
- Identități: Funcțiile identitate
- Compunere: Compunerea funcțiilor

# Exemplu: Categoria **Hask**

- Obiectele: tipuri
- Săgețile: funcții între tipuri  
 $f :: A \rightarrow B$
- Identități: funcția polimorfică **id**

**Prelude**> :t id

**id** :: a -> a

- Compunere: funcția polimorfică **(.)**

**Prelude**> :t (.)

**(.)** :: (b -> c) -> (a -> b) -> a -> c



# Subcategorii ale lui Hask date de tipuri parametrizate

- Obiecte: o clasă restânsă de tipuri din  $\mathbf{Hask}$ 
  - Exemplu: tipuri de forma  $[a]$
- Săgeți: toate funcțiile din  $\mathbf{Hask}$  între tipurile obiecte
  - Exemple: **concat** ::  $[[a]] \rightarrow [a]$ , **words** ::  $[\mathbf{Char}] \rightarrow [\mathbf{String}]$ ,  
**reverse** ::  $[a] \rightarrow [a]$

## Exemple

**Liste** obiecte: tipuri de forma  $[a]$

**Optiuni** obiecte: tipuri de forma  $\text{Maybe } a$

**Arbori** obiecte: tipuri de forma  $\text{Arbore } a$

**Comenzi I/O** obiecte: tipuri de forma  $\text{IO } a$

**Funcții de sursă**  $t$  obiecte: tipuri de forma  $t \rightarrow a$

# De ce categorii?

## (Des)compunerea este esența programării

- Am de rezolvat problema  $P$
- O descompun în subproblemele  $P_1, \dots, P_n$
- Rezolv problemele  $P_1, \dots, P_n$  cu programele  $p_1, \dots, p_n$ 
  - Eventual aplicând recursiv procedura de față
- Compun rezolvările  $p_1, \dots, p_n$  într-o rezolvare  $p$  pentru problema inițială

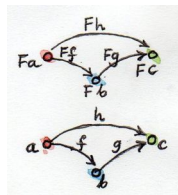
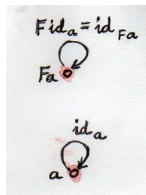
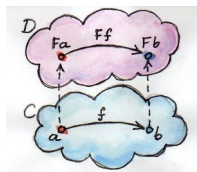
## Categoriile rezolvă problema compunerii

- Ne forțează să abstractizăm datele
- Se poate acționa asupra datelor doar prin săgeți (metode?)
- Forțează un stil de compunere independent de structura obiectelor

# Functori

Date fiind două categorii  $\mathbb{C}$  și  $\mathbb{D}$ , un functor  $F : \mathbb{C} \rightarrow \mathbb{D}$  este dat de

- O funcție  $F : |\mathbb{C}| \rightarrow |\mathbb{D}|$  de la obiectele lui  $\mathbb{C}$  la cele ale lui  $\mathbb{D}$
- Pentru orice  $A, B \in |\mathbb{C}|$ , o funcție  $F : \mathbb{C}(A, B) \rightarrow \mathbb{D}(F(A), F(B))$
- Compatibilă cu identitățile și cu compunerea
  - $F(id_A) = id_{F(A)}$  pentru orice  $A$
  - $F(g \circ f) = F(g) \circ F(f)$  pentru orice  $f : A \rightarrow B, g : B \rightarrow C, h = g \circ f$



Bartosz Milewski  
— Functors

# Functori în Haskell

În general un functor  $F : \mathbb{C} \rightarrow \mathbb{D}$  este dat de

- O funcție  $F : |\mathbb{C}| \rightarrow |\mathbb{D}|$  de la obiectele lui  $\mathbb{C}$  la cele ale lui  $\mathbb{D}$
- Pentru orice  $A, B \in |\mathbb{C}|$ , o funcție  $F : \mathbb{C}(A, B) \rightarrow \mathbb{D}(F(A), F(B))$
- Compatibilă cu identitățile și cu compunerea
  - $F(id_A) = id_{F(A)}$  pentru orice  $A$
  - $F(g \circ f) = F(g) \circ F(f)$  pentru orice  $f : A \rightarrow B, g : B \rightarrow C, h = g \circ f$

În Haskell o instanță **Functor** m este dată de

- Un tip  $m$  a pentru orice tip  $a$  (deci  $m$  trebuie să fie tip parametrizat)
- Pentru orice două tipuri  $a$  și  $b$ , o funcție

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

- Compatibilă cu identitățile și cu compunerea

`fmap id == id`

`fmap (g . f) == fmap g . fmap f`

pentru orice  $f :: a \rightarrow b$  și  $g :: b \rightarrow c$