Programare declarativă

Introducere în programarea funcțională folosind Haskell

Traian Florin Șerbănuță - seria 33
Ioana Leuștean - seria 34

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

Clasa Functor

Tipuri parametrizate — „cutii” conținând obiecte

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii” (recipiente, colecții) 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

Observăm că $f : B \to A$ poate fi gândită ca $\{f(x)\}_{x \in B}$

- b -> a descrie o colecție (infintă) de obiecte de tip *a* indexate de obiecte de tip *b*; orice intrare de tip b produce un rezultat de tip a
  - (++ "!") :: **String** -> **String** este colecția șirurilor terminate în “!”, indexate de prefixele lor

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 computatiilor I/O

comp :: **IO** a poate fi gandită ca o cutie care conține un element de tip a care însă nu este disponibil până la momentul execuției.

- **getChar** :: **IO Char** este o cutie în care în momentul execuției se va găsi caractererul următor introdus de la consolă.

Clase de tipuri pentru transformarea colecțiilor?

Analogie

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

Observație

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

Clase de tipuri pentru transformarea colecțiilor?

Analogie

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

Problemă

Formulare cu cutii

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

Exemplu — liste

Dată fiind o funcție `f :: a -> b` și o listă *la* de elemente de tip a, vreau să să obțin o lista 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
```

Dată fiind o funcție f :: a -> 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

Clasa de tipuri Functor

Definiție

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

Dată fiind o funcție f :: a -> 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
```

Kinds (tipuri de tipuri)

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

Observăm că m în definiția de mai sus este un constructor de tip.

Kinds (tipuri de tipuri)

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

Observăm că m în definiția de mai sus este un constructor de tip.

În Haskell, valorile sunt clasificate cu ajutorul *tipurilor*:

```
Prelude> :t "as"
"as" :: [Char]
```

Constructorii de tipuri sunt la rândul lor clasificați în *kind*-uri:

```
Prelude> :k Char
*     -- constructor de tip fara argumente
Prelude> :k []
[] :: * -> *   -- constructor de tip cu un argument
```

Kinds (tipuri de tipuri)

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

Observăm că m în definiția de mai sus este un constructor de tip.

În Haskell, valorile sunt clasificate cu ajutorul *tipurilor*:

```
Prelude> :t "as"
"as" :: [Char]
```

Constructorii de tipuri sunt la rândul lor clasificați în *kind*-uri:

```
Prelude> :k Char
*     -- constructor de tip fara argumente
Prelude> :k []
[] :: * -> *   -- constructor de tip cu un argument
```

În consecință, o instanță a clasei **Functor** trebuie să fie un *constructor de tip* care are *kind-ul* $*->*$.

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 optiune 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 optiune fmap :: (a -> b) -> **Maybe** a -> **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 fmap :: (a -> b) -> Arbore a -> 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 e a = Left e
                | Right 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 eroare fmap :: (a -> b) -> **Either** e a -> **Either** e b

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

instance Functor (Either e) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

Atenție! Un constructor de tip care este instanță a lui **Functor** trebuie să aibă un singur parametru.

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

Observăm că t -> a este (->) t a, deci
(->) t este tipul functiilor definite pe t.

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

Observăm că t -> a este (->) t a, deci
(->) t este tipul functiilor definite pe t.

```
instance Functor ((->) t) where
  fmap f g = f . g -- sau, mai simplu, fmap = (.)
```

Exemple

```
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)
```

Exemple

```
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
```

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 IO

fmap :: (a -> b) -> **IO** a-> **IO** 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 IO

fmap :: (a -> b) -> **IO** a-> **IO** b

```
instance Functor IO where
 fmap f action = do
                 result <- action
                 return (f result)
```

Vă amintiți notația **do** din cursul trecut!

(<$>) = fmap

În loc de fmap putem folosi operatorul infix <$>

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

Vă amintiți operatorul $> din cursul trecut?

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

În tipurile acestor operații se impune constrângerea ca f să fie un constructor din clasa **Functor**. Definițiile lor sunt

```
(<$) = fmap . const  -- (x <$) = fmap (const x)
($>) = flip (<$)
```

unde

```
const :: a -> b -> a
const x _ = x
```

(<$>), (<$), (>$)

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

Prelude Data.Functor> ('a':) <$> getLine
aaa
"aaaa"

Prelude Data.Functor> 3 <$ [2,3,4]
[3,3,3]
Prelude Data.Functor> [2,3,4] $> 3
[3,3,3]

Prelude Data.Functor> 3 <$ (putStr "abc")
abc3
Prelude Data.Functor>  (putStr "abc") $> 3
abc3
```

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 –> b) –>(m a –> 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**

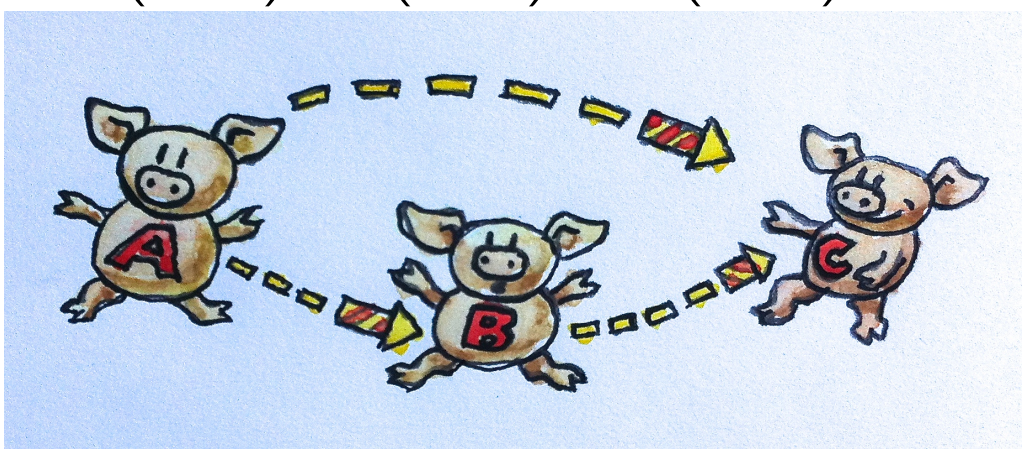
compunere fmap (g . f) == fmap g . fmap f

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 $f : A \to B$
- Pentru orice obiect A o săgeată $id_A : A \to A$ numită identitatea lui A
- Pentru orice obiecte A, B, C, o operație de compunere a săgeților
  $\circ : \mathbb{C}(B, C) \times \mathbb{C}(A, B) \to \mathbb{C}(A, C)$



Bartosz Milewski — Category: The Essence of Composition

- Compunerea este asociativă și are element neutru id

Exemplu: Categoria $\mathbb{S}\mathrm{et}$

- Obiecte: mulțimi
- Săgeți: funcții
- Identități: funcțiile identitate
- Compunere: compunerea funcțiilor

Exemplu: Categoria $\mathbb{S}$et

- Obiecte: mulțimi
- Săgeți: funcții
- Identități: funcțiile identitate
- Compunere: compunerea funcțiilor

Alte exemple de categorii:

- categoria grupurilor cu morfisme de grupuri;
- categoria algebrelor de o signatură dată cu morfismele corespunzătoare.

Exemplu: Categoria $\mathbb{H}$ask

- Obiectele: tipuri (a,b,c, ..)
- Săgețiile: funcții între tipuri

```
f :: a -> 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

Pentru orice constructor c de tipuri de kind `*` –> `*` putem defini subcategoria lui $\mathbb{H}\mathrm{ask}$ indusă de c, astfel:

- Obiecte: toate tipurile c t, unde t tip in $|\mathbb{H}\mathrm{ask}|$.
  - Exemplu: tipuri de forma [a]
- Săgeți: toate funcțiile din $\mathbb{H}$ask între tipurile obiecte
  - Exemple: **concat** :: [[a]] –> [a], **words** :: [**Char**] –> [**String**], **reverse** :: [a] –> [a]

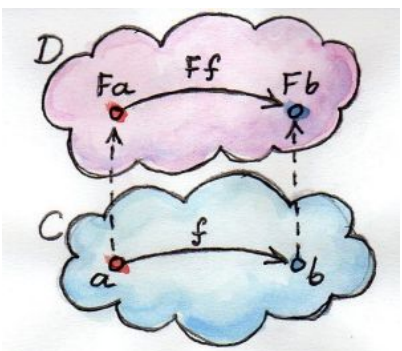
Exemple

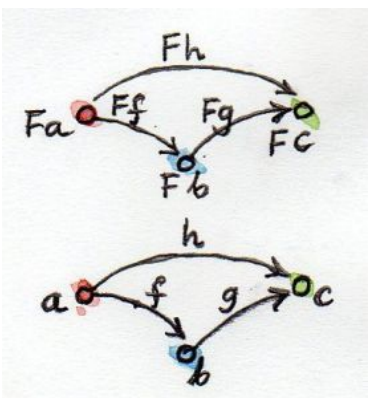
Liste	obiecte: tipuri de forma [a]
Optiuni	obiecte: tipuri de forma Maybe a
Arbori	obiecte: tipuri de forma Arbore a
Comenzi I/O	obiecte: tipuri de forma IO a
Funcții de sursă t	obiecte: tipuri de forma t -> a

Functori

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

- O funcție $F : |\mathbb{C}| \to |\mathbb{D}|$ de la obiectele lui $\mathbb{C}$ la cele ale lui $\mathbb{D}$
- Pentru orice $A, B \in |\mathbb{C}|$, o funcție $F : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$
- 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 $f : A \to B, g : B \to C, h = g \circ f$





Bartosz Milewski — Functors

Functori în Haskell

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

- O funcție $F : |\mathbb{C}| \to |\mathbb{D}|$ de la obiectele lui $\mathbb{C}$ la cele ale lui $\mathbb{D}$
- Pentru orice $A, B \in |\mathbb{C}|$, o funcție $F : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$
- 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 $f : A \to B, g : B \to C, h = g \circ f$

În Haskell o instanță **Functor** m definește un functor de la $\mathbb{H}\mathrm{ask}$ in subcategoria lui $\mathbb{H}\mathrm{ask}$ indusă de m care asociază

- Tipul m a pentru orice tip a (deci m trebuie sa 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 -> b` și `g :: b -> c`

Pe săptămâna viitoare!