Programare declarativă

Functori și categorii

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Cutii și computații

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.

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

Generalizare: Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemple

- **Maybe** a descrie rezultate de computații deterministe care pot eșua
  - computații care eșuează: **Nothing**
  - computații care produc un element de tipul dat: **Just** 4
- [**Int**] descrie liste de rezultate posibile ale unor computații nedeterministe
  - care pot produce oricare dintre rezultatele date: [1, 2, 3], [], [5]

Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemple

- **Either** e a descrie rezultate de tip a ale unor computații deterministe care pot eșua cu o eroare de tip *e*
  - **Right** 5 :: **Either** e **Int** reprezintă rezultatul unei computații reușite
  - **Left** "OOM":: **Either String** a reprezintă o excepție de tip **String**

Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemplu: tipul funcțiilor de sursă dată

- `t -> a` descrie computații care atunci când primesc o intrare de tip `t` produc un rezultat de tip `a`
  - `(++ "!")` :: **String** –> **String** este o computație care dat fiind un șir, îi adaugă un semn de exclamare
  - **length** :: **String** –> **Int** este o computație care dat fiind un șir, îi prduce lungimea acestuia
  - **id** :: **String** –> **String** este o computație care produce șirul dat ca argument

Clase de tipuri pentru cutii și computații?

Întrebare

Care sunt trăsăturile comune ale acestor tipuri parametrizate care pot fi gândite intuitiv ca cutii care conțin elemente / computații care produc rezultate?

Problemă

Putem proiecta clase de tipuri care descriu funcționalități comune tuturor acestor tipuri?

Functori

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!)

Formulare cu computații

Dată fiind o funcție f :: a −> b și o computație ca care produce rezultate de tip a, vreau să să obțin o computație cb care produce rezultate de tip b obținute prin transformarea rezultatelor produse de computația 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

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

Dată fiind o funcție f :: a -> b și ca :: m a, fmap produce cb :: m b obținută prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)

Instanță pentru liste

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

Clasa de tipuri Functor

Instanțe

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

Instanță pentru tipul optiune fmap :: (a –> b) –> **Maybe** a –> **Maybe** b

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

Clasa de tipuri Functor

Instanțe

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

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

Instanțe

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

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

Instanță pentru tipul funcție fmap :: (a -> b) -> (t -> a) -> (t -> b)

Clasa de tipuri Functor

Instanțe

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

Instanță pentru tipul funcție fmap :: (a -> b) -> (t -> a) -> (t -> b)

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

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**

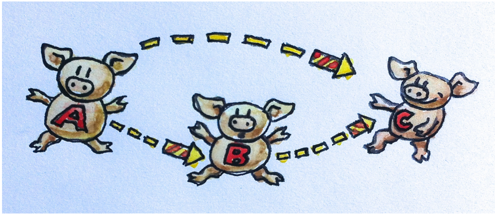
  compunere fmap (g . f) == fmap g . fmap f

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 $\mathbb{H}$ask

- Obiectele: tipuri
- 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

- Obiecte: o clasă restânsă de tipuri din $|\mathbb{H}\mathrm{ask}|$
  - 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
Funcții de sursă t	obiecte: tipuri de forma t -> a

De ce categorii?

(Des)compunerea este esența programării

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

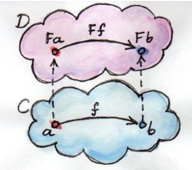
Categoriile rezolvă problema compunerii

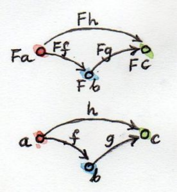
- Ne forțează să abstractizăm datele
- Se poate acționa asupra datelor doar prin săgeți (metode?)
- Forțează un stil de compunere independent de structura obiectelor

Functori

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

- Un tip 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