Programare declarativă

Functori, functori aplicativi[1]

Ioana Leuștean
Traian Florin Șerbănuță

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

15 decembrie 2020

[1] bazat pe Learn You a Haskell for Great Good

Colecții

Tipuri parametrizate reprezentând „colecții”

Idee

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

Exemple

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

Tipuri parametrizate reprezentând „colecții”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca reprezentând „colecții”, 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 „colecție” 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 reprezentând „computații”

O clasă largă de tipuri parametrizate pot reprezenta „contexte computaționale” 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 produce elemente de tip a sau 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**
- t –> a descrie computații care atunci când primesc orice 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 colecții și computații?

Întrebare

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

Problemă

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

Functori

Problemă

Formulare

Dată fiind o funcție f :: a –> b și o colecție/computație ca care conține/produce elemente de tip a, vreau să să obțin o colecție/computație cb care conține/produce elemente de tip b obținute prin transformarea elementelor conținute/produse de colecția ca fără a afecta structura colecției/computației.

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, fără a afecta structura listei (e.g., lungimea, ordinea elementelor).

Clasa de tipuri Functor

Definiție

```
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 elementelor conținute/produse de colecția/computația ca folosind funcția f (fără a afecta structura colecției/computației)

Instanță pentru liste

```
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

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

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)
```

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

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

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

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)
```

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 =
```

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
  - e.g., tipul colecțiilor cu elemente din a, tipul functiilor care au ca rezultat a, tipul operatiilor de citire care produc a
- 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 fără a modifica structura preexistentă în ca
- Abstractizat prin două legi:

identitate fmap **id** == **id**

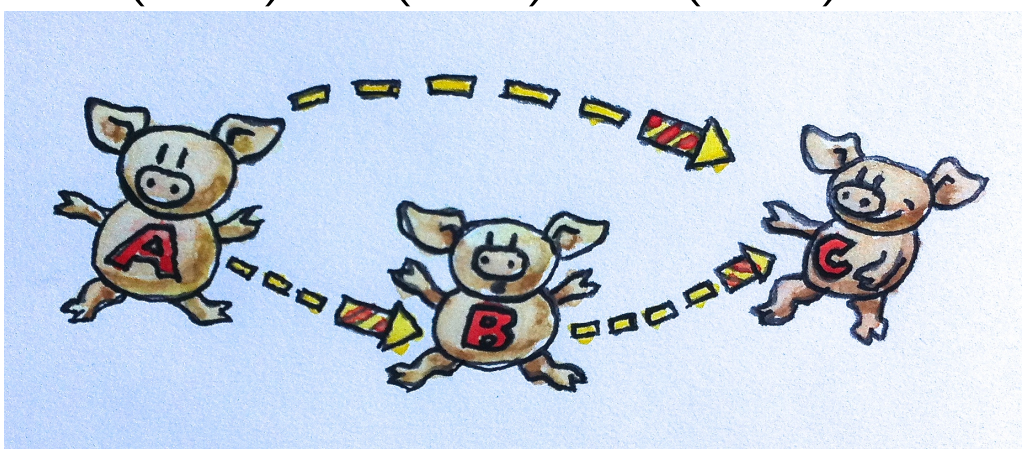
compunere fmap (g . f) == fmap g . fmap f

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 $\mathbb{S}$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

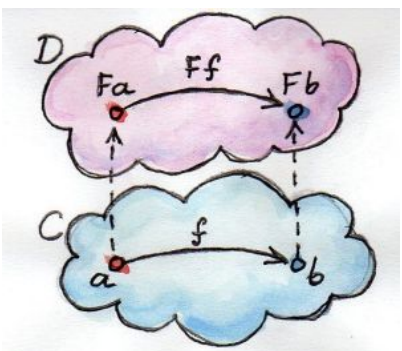
Categoriile rezolvă problema compunerii

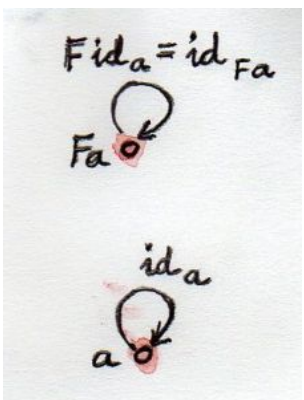
- Ne forțează să abstractizăm datele
- Se poate acționa asupra datelor doar prin săgeți (metode?)
- Forțează un stil de compunere independent de structura obiectelor

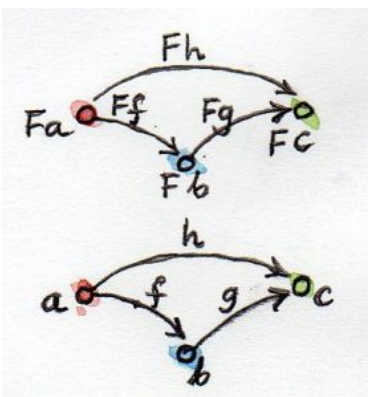
Functori

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

Functori aplicativi

Problemă

- Folosind fmap putem transforma o funcție h :: a –> b într-o funcție între colecții/computații fmap h :: m a –> m b
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente E.g., cum trecem de la h :: a –> b –> c la h' :: m a –> m b –> m c
- putem încerca să folosim fmap

Problemă

- Folosind fmap putem transforma o funcție h :: a –> b într-o funcție între colecții/computații fmap h :: m a –> m b
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente
  E.g., cum trecem de la h :: a –> b –> c la h' :: m a –> m b –> m c
- putem încerca să folosim fmap
- Dar, deoarece h :: a –> (b –> c), avem că
  fmap h :: m a –> m (b –> c)
- Putem aplica fmap h la o valoare ca :: m a și obținem
  fmap h ca :: m (b –> c)

Problemă

Cum transformăm un obiect din m (b –> c) într-o funcție m b –> m c?

- **ap** :: m (b –> c) –> (m b –> m c), sau, ca operator
- (<*>) :: m (b –> c) –> m b –> m c

Merge pentru funcții cu oricâte argumente

Problemă

Dată fiind o funcție f :: a1 –> a2 –> a3 –> ... –> an –> a și computațiile ca1 :: m a1, . . . , can :: m an, vrem să „aplicăm” funcția *f* pe rând computațiilor ca1, . . . , can pentru a obține o computație finală ca :: m a.

Soluție: Date fiind

- funcția fmap :: (a –> b) –> m a –> m b
- funcția (<*>) :: m (b –> c) –> m b –> m c cu „proprietăți bune”

Atunci

```
fmap f :: m a1 –> m (a2 –> a3 –> ... –> an –> a)
fmap f ca1 :: m (a2 –> a3 –> ... –> an –> a)
fmap f ca1 <*> ca2 :: m (a3 –> ... –> an –> a)
...
fmap f ca1 <*> ca2 <*> ca3 ... <*> can :: m a
```

Clasa de tipuri Applicative

Definiție

```
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

Proprietate importantă

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

Tipul opțiune

```
Main> pure "Hey" :: Maybe String
Just "Hey"
Main> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
Main> let mDiv x y = if y == 0 then Nothing else Just (x `
    div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Tipul opțiune

```
Main> pure "Hey" :: Maybe String
Just "Hey"
Main> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
Main> let mDiv x y = if y == 0 then Nothing else Just (x `
    div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Instanță pentru tipul opțiune

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f  <*> x = fmap f x
```

Tipul opțiune

```
Main> pure "Hey" :: Either a String
Right "Hey"
Main> (++) <$> (Right "Hey ") <*> (Right "You!")
Right "Hey You!"
Main> let mDiv x y = if y == 0 then Left "Division by 0!"
    else Right (x `div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Tipul opțiune

```
Main> pure "Hey" :: Either a String
Right "Hey"
Main> (++) <$> (Right "Hey ") <*> (Right "You!")
Right "Hey You!"
Main> let mDiv x y = if y == 0 then Left "Division by 0!"
    else Right (x `div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Instanță pentru tipul eroare

```
instance Applicative (Either a) where
  pure = Right
  Left e <*> _ = Left e
  Right f  <*> x = fmap f x
```

Tipul listă (computație nedeterministă)

```
Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"]["Hello world","Hello happiness","Goodbye
    world","Goodbye happiness"]
Main> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Tipul listă (computație nedeterministă)

```
Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"]["Hello world","Hello happiness","Goodbye
    world","Goodbye happiness"]
Main> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Instanță pentru tipul computațiilor nedeterministe (liste)

```
instance Applicative [] where
  pure x = [x]
  fs  <*> xs = [f x | f <- fs, x <- xs]
```

Tipul I/O

```
Main> (++) <$> getLine <*> getLine >>= putStrLn
hello
 world!
hello world!
```

Tipul I/O

```
Main> (++) <$> getLine <*> getLine >>= putStrLn
hello
 world!
hello world!
```

Instanță pentru tipul computațiilor I/O

```
instance Applicative IO where
  pure = return
  iof <*> iox = do
    f <- iof
    x <- iox
    return (f x)
```

Tipul funcțiilor de sursă dată

```
data Exp = Lit Int | Var String | Exp :+: Exp
type Env = [(String, Int)]

find :: String -> (Env -> Int)
find x env = head [i | (y,i) <- env, y == x]

eval :: Exp -> (Env -> Int)
eval (Lit i) = pure i
eval (Var x) = find x
eval (e1 :+: e2) = (+) <$> eval e1 <*> eval e2
```

Tipul funcțiilor de sursă dată

```
data Exp = Lit Int | Var String | Exp :+: Exp
type Env = [(String, Int)]

find :: String -> (Env -> Int)
find x env = head [i | (y,i) <- env, y == x]

eval :: Exp -> (Env -> Int)
eval (Lit i) = pure i
eval (Var x) = find x
eval (e1 :+: e2) = (+) <$> eval e1 <*> eval e2
```

Instanță pentru tipul funcțiilor de sursă dată

```
instance Applicative ((->) t) where
  pure :: a -> (t -> a)
  pure x = \ _ -> x
  (<*>) :: (t -> (a -> b)) -> (t -> a) -> (t -> b)
  f  <*> g = \ x -> f x (g x)
```

Liste ca fluxuri de date.

```
newtype ZList a = ZList { get :: [a]}

> get $ max <$> ZList [1,2,3,4,5,3] <*> ZList [5,3,1,2]
[5,3,3,4]
> get $ (+) <$> ZList [1,2,3] <*> ZList [100,100..]
[101,102,103]
> get $ (,,) <$> ZList "dog" <*> ZList "cat" <*> ZList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Liste ca fluxuri de date.

```
newtype ZList a = ZList { get :: [a]}

> get $ max <$> ZList [1,2,3,4,5,3] <*> ZList [5,3,1,2]
[5,3,3,4]
> get $ (+) <$> ZList [1,2,3] <*> ZList [100,100..]
[101,102,103]
> get $ (,,) <$> ZList "dog" <*> ZList "cat" <*> ZList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Instanță pentru ZipList

```
instance Functor ZipList where
  fmap f (ZipList xs) = ZipList (fmap f xs)

instance Applicative ZipList where
  pure x = repeat x
  ZipList fs <*> ZipList xs =
    ZipList (zipWith (\ f x -> f x) fs xs)
```

Proprietăți ale functorilor aplicativi

identitate `pure id <*> v = v`

compoziție `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

homomorfism `pure f <*> pure x = pure (f x)`

interschimbare `u <*> pure y = pure ($ y) <*> u`

Consecință: `fmap f x == f <$> x == pure f <*> x`