Programare funcțională

Introducere în programarea funcțională folosind Haskell
C09

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Tipuri parametrizate

Tipuri parametrizate — „cutii”

Idee

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

Exemple

- Multimea tipurilor opțiune asociază unui tip a, tipul **Maybe** a
  - cutii goale: **Nothing**
  - cutii care țin un element x de tip a: **Just** x
- Multimea tipurilor 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

Multe 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

Multe 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

Multe 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

Multe 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 produce 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 obțin o cutie *cb* care conține elemente de tip b obținute prin transformarea elementele din cutia ca folosind funcția `f` (și doar atât!)

Problemă

Formulare cu cutii

Dată fiind o funcție `f :: a -> 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 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 obțin o lista de elemente de tip b transformând fiecare element din *la* folosind funcția `f` (și doar atât!)

Problemă

Formulare cu computații

Dată fiind o funcție `f :: a -> b` și o computație *ca* care produce rezultate de tip a, vreau 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 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 f where
  fmap :: (a -> b) -> f a -> f b
```

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

Instanță pentru liste

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

Clasa de tipuri Functor

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

Instanță pentru tipul optiune

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

Clasa de tipuri Functor

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

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

Instanță pentru tipul eroare

fmap :: (a -> b) -> Either e a -> Either e b

Clasa de tipuri Functor

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

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

Instanță pentru tipul arbore

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

Clasa de tipuri Functor

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

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

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

Instanță pentru tipul funcție

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

Clasa de tipuri Functor

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

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

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

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)

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 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 . h) == fmap g . fmap h

Invalidarea contractului - identitate

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

Instanta a clasei Functor care invalideaza conditia de conservare a identitatii:

```
instance Functor WhoCares where
  fmap _ ItDoesnt = WhatThisIsCalled
  fmap f 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)
```

Instanta a clasei Functor care valideaza conditia de conservare a identitatii:

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

Instanta a clasei Functor care invalideaza conditia de conservare a compunerii:

```
instance Functor CountingBad where
  fmap f (Heisenberg n a) = Heisenberg (n+1) (f a)
```

```
Prelude> oneWhoKnocks = Heisenberg 0 "Uncle"
Prelude> f = (++" Jesse")
Prelude> g = (++" lol")
Prelude> fmap (f . g) oneWhoKnocks
Heisenberg 1 "Uncle lol Jesse"
Prelude> fmap f . fmap g $ oneWhoKnocks
Heisenberg 2 "Uncle lol Jesse"
```

Validarea contractului - compunere

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

Instanta a clasei Functor care valideaza conditia de conservare a compunerii:

```
instance Functor CountingBad where
  fmap f (Heisenberg n a) = Heisenberg n (f a)
```

```
Prelude> oneWhoKnocks = Heisenberg 0 "Uncle"
Prelude> f = (++" Jesse")
Prelude> g = (++" lol")
Prelude> fmap (f . g) oneWhoKnocks
Heisenberg 0 "Uncle lol Jesse"
Prelude> fmap f . fmap g $ oneWhoKnocks
Heisenberg 0 "Uncle lol Jesse"
```

Quiz time!

Seria 23: https://www.questionpro.com/t/AT4qgZqOwD

Seria 24: https://www.questionpro.com/t/AT4NiZqN7a

Seria 25: https://www.questionpro.com/t/AT4qgZqOwu

Categorii și Functori

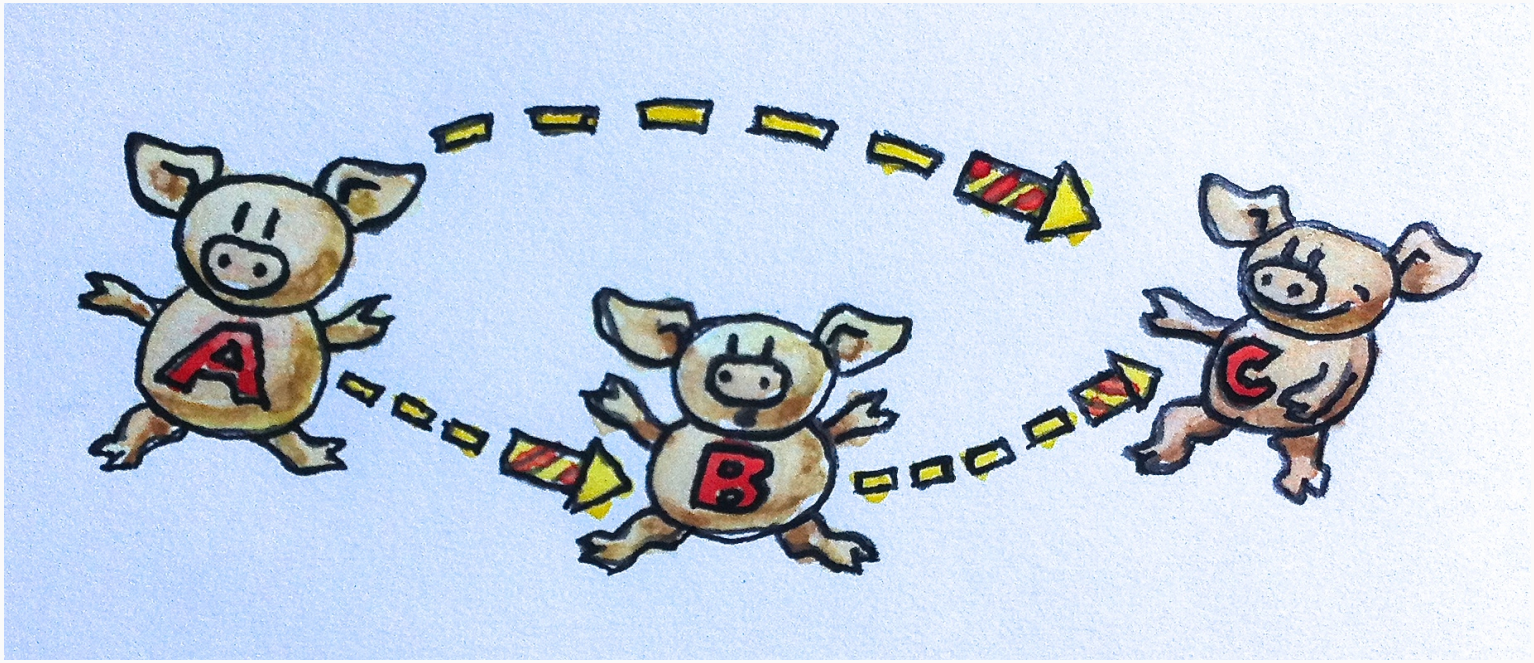
O categorie

- A category is an embarrassingly simple concept.
  Bartosz Milewski, Category Theory for Programmers
- Categorie = obiecte + sageti

O categorie

- A category is an embarrassingly simple concept.
  Bartosz Milewski, Category Theory for Programmers
- Categorie = obiecte + sageti
- Ingredient cheie: compunerea de sageti

O categorie



credits: Bartosz Milewski

O categorie

O categorie **C** consta in

- Obiecte:
- Sageti:

- Compunere:

O categorie

O categorie **C** consta in

- Obiecte: notate $A, B, C, \ldots$
- Sageti:

- Compunere:

O categorie

O categorie $\mathbf{C}$ consta in

- Obiecte: notate $A, B, C, \ldots$
- Sageti: pentru orice obiecte A si B, exista o multime de sageti $\mathbf{C}(A, B)$
  - notam $f \in \mathbf{C}(A, B)$ cu $f : A \to B$ sau $A \xrightarrow{f} B$
- Compunere:

O categorie

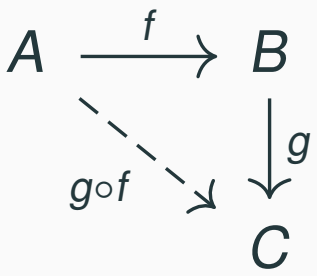
O categorie $\mathbf{C}$ consta in

- Obiecte: notate $A, B, C, \ldots$
- Sageti: pentru orice obiecte A si B, exista o multime de sageti $\mathbf{C}(A, B)$
  - notam $f \in \mathbf{C}(A, B)$ cu $f : A \to B$ sau $A \xrightarrow{f} B$
- Compunere: pentru orice sageti $f : A \to B$ si $g : B \to C$ exista o sageata $g \circ f : A \to C$

O categorie

O categorie $\mathbf{C}$ consta in

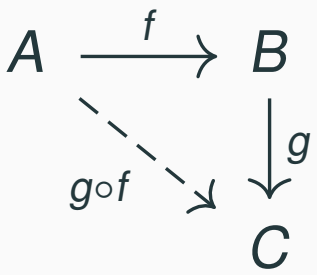
- Obiecte: notate $A, B, C, \ldots$
- Sageti: pentru orice obiecte A si B, exista o multime de sageti $\mathbf{C}(A, B)$
  - notam $f \in \mathbf{C}(A, B)$ cu $f : A \to B$ sau $A \xrightarrow{f} B$
- Compunere: pentru orice sageti $f : A \to B$ si $g : B \to C$ exista o sageata $g \circ f : A \to C$



O categorie

O categorie $\mathbf{C}$ consta in

- Obiecte: notate $A, B, C, \ldots$
- Sageti: pentru orice obiecte A si B, exista o multime de sageti $\mathbf{C}(A, B)$
  - notam $f \in \mathbf{C}(A, B)$ cu $f : A \to B$ sau $A \xrightarrow{f} B$
- Compunere: pentru orice sageti $f : A \to B$ si $g : B \to C$ exista o sageata $g \circ f : A \to C$

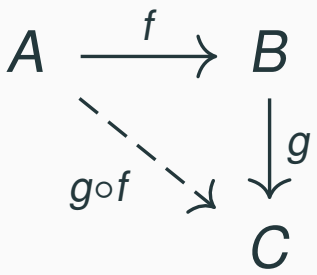


- Identitate: pentru orice obiect A exista o sageata $id_A : A \to A$

O categorie

O categorie $\mathbf{C}$ consta in

- Obiecte: notate $A, B, C, \ldots$
- Sageti: pentru orice obiecte A si B, exista o multime de sageti $\mathbf{C}(A, B)$
  - notam $f \in \mathbf{C}(A, B)$ cu $f : A \to B$ sau $A \xrightarrow{f} B$
- Compunere: pentru orice sageti $f : A \to B$ si $g : B \to C$ exista o sageata $g \circ f : A \to C$



- Identitate: pentru orice obiect A exista o sageata $id_A : A \to A$
- Axiome: pentru orice sageti $f : A \to B$, $g : B \to C$, si $h : C \to D$

$$h \circ (g \circ f) = (h \circ g) \circ f \qquad f \circ id_A = f = id_B \circ f$$

Exemplu - categoria de multimi

Categoria **Set** are

- Obiecte: multimi
- Sageti: functii
- Compunere: compunerea de functii
- Identitate: pentru orice multime A, functia identitate $id_A : A \to A$, $id_A(a) = a$
- Axiome: ✓

Monoizi

Un monoid **M** este o structura $\langle M, +, e \rangle$ astfel incat

- M este o multime
- $+ : M \times M \to M$ este asociativa
  (aka $(a + b) + c = a + (b + c)$ pentru orice $a, b, c \in M$)
- $e \in M$ este identitate pentru $+$
  (aka $e + a = a + e = a$ pentru orice $a \in M$)

Monoizi

Un monoid **M** este o structura $\langle M, +, e \rangle$ astfel incat

- M este o multime
- $+ : M \times M \to M$ este asociativa
  (aka $(a + b) + c = a + (b + c)$ pentru orice $a, b, c \in M$)
- $e \in M$ este identitate pentru $+$
  (aka $e + a = a + e = a$ pentru orice $a \in M$)

Monoizii sunt un concept extrem de putenic:

- Stau in spatele aritmeticii de baza
  - si adunarea, si inmultirea formeaza un monoid
- Sunt prezenti peste tot in programare
  - siruri de caractere, liste ...

Exemplu - categoria de monoizi

Categoria **Mon** are

- Obiecte: monoizi
- Sageti: morfisme de monoizi
  (aka functii care nu "strica" operatia de monoid)
- Compunerea: compunerea de morfisme de monoizi
- Identitatea: pentru orice obiect $\mathbf{M}$, $id_{\mathbf{M}} : M \to M$, $id_M(m) = m$
- Axiome: ✓

Exemplu - un monoid ca o categorie

Orice monoid $\mathbf{M} = \langle M, +, e \rangle$ este o categorie cu

Exemplu - un monoid ca o categorie

Orice monoid $\mathbf{M} = \langle M, +, e \rangle$ este o categorie cu

- Obiecte: un singur obiect $\square$

Exemplu - un monoid ca o categorie

Orice monoid $\mathbf{M} = \langle M, +, e \rangle$ este o categorie cu

- Obiecte: un singur obiect $\square$
- Sageti: elementele multimii M (i.e, $\mathbf{M}(\square, \square) = M$)

Exemplu - un monoid ca o categorie

Orice monoid $\mathbf{M} = \langle M, +, e \rangle$ este o categorie cu

- Obiecte: un singur obiect $\square$
- Sageti: elementele multimii M (i.e, $\mathbf{M}(\square, \square) = M$)
- Compunerea: operatia de monoid $+$

Exemplu - un monoid ca o categorie

Orice monoid $\mathbf{M} = \langle M, +, e \rangle$ este o categorie cu

- Obiecte: un singur obiect $\square$
- Sageti: elementele multimii M (i.e, $\mathbf{M}(\square, \square) = M$)
- Compunerea: operatia de monoid $+$
- Identitatea: identitatea monoidului e

Exemplu - un monoid ca o categorie

Orice monoid $\mathbf{M} = \langle M, +, e \rangle$ este o categorie cu

- Obiecte: un singur obiect $\square$
- Sageti: elementele multimii M (i.e, $\mathbf{M}(\square, \square) = M$)
- Compunerea: operatia de monoid $+$
- Identitatea: identitatea monoidului e
- Axiome:

$$h \circ (g \circ f) = (h \circ g) \circ f \qquad f \circ id_A = f = id_B \circ f$$
$$a + (b + c) = (a + b) + c \qquad a + e = a = e + a$$

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

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 $\mathbb{D}$

Functori

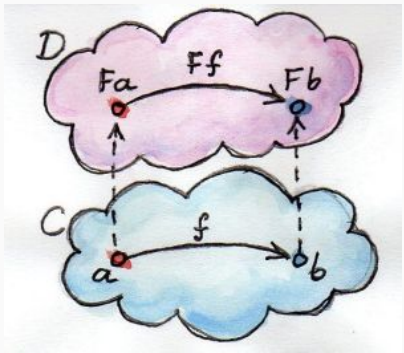
Date fiind două categorii $\mathbb{C}$ și $\mathbb{D}$, un functor $F : \mathbb{C} \to \mathbb{D}$ este dat de

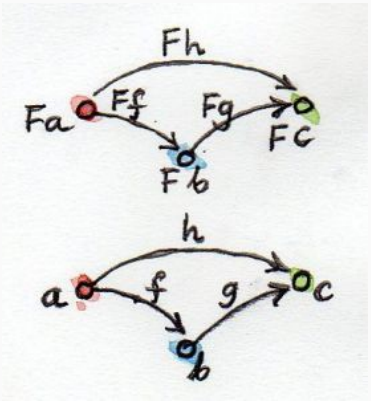
- O funcție $F : |\mathbb{C}| \to |\mathbb{D}|$ de la obiectele lui $\mathbb{C}$ la cele ale lui $\mathbb{D}$
- Pentru orice $A, B \in |\mathbb{C}|$, o funcție $F : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$

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 Haskell o instanță **Functor** f este dată de

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

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

- Compatibilă cu identitățile și cu compunerea

  ```
  fmap id == id
  fmap (g . h) == fmap g . fmap h
  ```

  pentru orice h :: a -> b și g :: b -> c

Pe săptămâna viitoare!