Programare funcțională

Introducere în programarea funcțională folosind Haskell
C07

Claudia Chiriță
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Examen parțial - săptămâna viitoare

- Test grilă pe foaie în cadrul cursului
- 3 puncte din nota finală, 15 întrebări grilă, 40 min
- Nu este obligatoriu și nu se poate reface
- O să afișăm pe Drive repartiția pe ore
- Materiale ajutătoare: suporturile de curs și de laborator, notițe
- Materiale ajutătoare doar în format fizic!
- Fără ceasuri, telefoane etc

Functori

Map

În cursurile trecute, am văzut funcția

```
map :: (a -> b) -> [a] -> [b]
```

Problemă. Putem generaliza această funcție la alte tipuri parametrizate?

Clasa de tipuri Functor

Definiție

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

Dată fiind o funcție g :: a -> b și ca :: f a, fmap produce cb :: f b obținută prin transformarea rezultatelor produse de computația ca folosind funcția g (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 constructorul de tipuri Maybe
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 constructorul de tipuri Either e
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

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

Instanță pentru constructorul de tipuri 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
```

Tipul funcțiilor de sursă dată **t -> a** (parametric în a)

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)

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)
```

Instanță a clasei Functor care invalideaza condiția de conservare a identității:

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

Prelude> fmap id ItDoesnt
WhatThisIsCalled
Prelude> id ItDoesnt
ItDoesnt
```

Validarea contractului - identitate

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

Instanță a clasei Functor care validează condiția de conservare a identității:

```haskell
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

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

Instanță a clasei Functor care invalidează condiția de conservare a compunerii:

```haskell
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

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

Instanță a clasei Functor care validează condiția de conservare a compunerii:

```haskell
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!

https://tinyurl.com/PF2023-C08-Quiz1

Functori aplicativi

Problemă

- Folosind **fmap** putem transforma o funcție **h :: a –> b** într-o funcție **fmap h :: m a –> m b**
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente?
  De exemplu, 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)**, obținem **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, ca2 :: m a2, . . . , can :: m an,

vrem să „aplicăm” funcția *f* pe rând computațiilor ca1, . . . , can pentru a obține o computație finală ca :: m a.

Merge pentru funcții cu oricâte argumente

Date fiind

- f :: a1 –> a2 –> a3 –> ... –> an –> a
- ca1 :: m a1, ca2 :: m a2, . . . , can :: m an,
- fmap :: (a –> b) –> m a –> m b
- (<*>) :: 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

```
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

Clasa de tipuri Applicative

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

class Functor m => Applicative m where
  pure :: a –> m a
  (<*>) :: m (a –> b) –> m a –> m b
```

Proprietate importantă

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

Functori aplicativi

```
  ($)  ::     (a –> b) –>     a –>     b
(<$>)  ::     (a –> b) –> m a –> m b
(<*>)  :: m (a –> b) –> m a –> m b
```

Instante – Maybe

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

class Functor m => Applicative m where
   pure :: a -> m a
  (<*>) :: m (a -> b) -> m a -> m b


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


Instante – Maybe

```haskell
Prelude> pure "Hey" :: Maybe String
Just "Hey"
```

Cum concatenam **Just** "Hey" cu **Just** "You!"?

```haskell
Prelude> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
```

- (++) :: **String** –> (**String** –> **String**)
- **Just** "Hey␣" :: **Maybe String**
- (<$>) :: (a –> b) –> m a –> m b (este fmap)
- (++) <$> (**Just** "Hey␣") :: **Maybe** (**String** –> **String**)
- **Just** "You!" :: **Maybe String**
- (<*>) :: m (b –> c) –> m b –> m c
- **Just** "Hey␣You!" :: **Maybe String**

Instante – Maybe

```haskell
mDiv x y = if y == 0 then Nothing
           else Just (x `div` y)

mF x = (+) <$> pure 4 <*> mDiv 10 x
```

- (+) :: **Int** –> **Int** –> **Int**
- pure 4 :: **Maybe Int**
- (<$>) :: (a –> b) –> m a –> m b (este fmap)
- (+) <$> pure 4 :: **Maybe** (**Int** –> **Int**)
- mDiv :: **Int** –> **Int** –> **Maybe Int**
- mDiv 10 x :: **Maybe Int**
- (<*>) :: m (b –> c) –> m b –> m c

```haskell
Prelude> mF 2
Just 9
```


Instante – Either

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

class Functor m => Applicative m where
   pure :: a -> m a
  (<*>) :: m (a -> b) -> m a -> m b


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


Instante – Either

```haskell
Prelude> pure "Hey" :: Either a String
Right "Hey"

Prelude> (++) <$> (Right "Hey ") <*> (Right "You!")
Right "Hey You!"
```

- (++) :: **String** –> (**String** –> **String**)
- **Right** "Hey␣" :: **Either** a **String**
- (<$>) :: (a –> b) –> m a –> m b (este fmap)
- (++) <$> (**Right** "Hey␣") :: **Either** a (**String** –> **String**)
- **Right** "You!" :: **Either** a **String**
- (<*>) :: m (b –> c) –> m b –> m c
- **Right** "Hey␣You!" :: **Either** a **String**

Instante – Either

```haskell
eDiv x y = if y == 0 then Left "Division by 0!"
           else Right (x `div` y)

eF x = (+) <$> pure 4 <*> eDiv 10 x
```

- (+) :: **Int** –> **Int** –> **Int**
- pure 4 :: **Either String Int**
- (<$>) :: (a –> b) –> m a –> m b (este fmap)
- (+) <$> pure 4 :: **Either String** (**Int** –> **Int**)
- eDiv :: **Int** –> **Int** –> **Either String Int**
- eDiv 10 x :: **Either String Int**
- (<*>) :: m (b –> c) –> m b –> m c

```haskell
Prelude> eF 2
Right 9
```

Instante – Liste

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

class Functor m => Applicative m where
   pure :: a -> m a
  (<*>) :: m (a -> b) -> m a -> m b

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


Instante – Liste

```
Prelude> pure "Hey" :: [String]
["Hey"]

Prelude> (++) <$> ["Hello ", "Goodbye "] <*> ["world"
    , "happiness"]
["Hello world","Hello happiness","Goodbye world","
   Goodbye happiness"]
```

- (++) :: **String** -> (**String** -> **String**)
- ["Hello␣", "Goodbye␣"] :: [**String**]
- (<$>) :: (a -> b) -> m a -> m b (este fmap)
- (++) <$> ["Hello␣", "Goodbye␣"] :: [**String** -> **String**]
- ["world", "happiness"] :: [**String**]
- (<*>) :: m (b -> c) -> m b -> m c

Instante – Liste

```
Prelude> (+) <$> [1,2] <*> [3,4]
[4,5,5,6]
```

- (+) :: **Int** -> **Int** -> **Int**
- [1,2] :: [**Int**]
- (<$>) :: (a -> b) -> m a -> m b (este fmap)
- (<*>) :: m (b -> c) -> m b -> m c

Instante – Liste

```
Prelude> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

- (+),(*) :: **Int** -> **Int** -> **Int**
- [(+),(*)] :: [**Int** -> **Int** -> **Int**]
- [1,2] :: [**Int**]
- (<*>) :: m (b -> c) -> m b -> m c
- [(+),(*)] <*> [1,2] :: [**Int** -> **Int**]
- [(+),(*)] <*> [1,2] <*> [3,4] :: [**Int**]

Instante – Liste

```
Prelude> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

- (*) :: **Int** -> **Int** -> **Int**
- [2,5,10] :: [**Int**]
- (<$>) :: (a -> b) -> m a -> m b (este fmap)
- (*) <$> [2,5,10] :: [**Int** -> **Int**]
- (<*>) :: m (b -> c) -> m b -> m c
- (*) <$> [2,5,10] <*> [8,10,11] :: [**Int**]

```
Prelude> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```


Quiz time!

https://tinyurl.com/PF2023-C08-Quiz2

Pe săptămâna viitoare!