Programare funcțională

Introducere în programarea funcțională folosind Haskell
C03

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Currying

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.

- Pentru $x \in A$ (arbitrar, fixat) definim

  $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.

  Funcția f_x se obține prin aplicarea parțială a funcției f.

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.

- Pentru $x \in A$ (arbitrar, fixat) definim

  $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.

  Funcția f_x se obține prin aplicarea parțială a funcției f.

  In mod similar definim *aplicarea parțială* pentru orice $y \in B$

  $f^y : A \to C$, $f^y(x) = z$ dacă și numai dacă $f(x, y) = z$.

Funcții în matematică

Exemplu

$A =$Int, $B = C =$String

$$f(x, y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

Funcții în matematică

Exemplu

$A =$Int, $B = C =$String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

- Fie $x \in$ Int arbitrar, fixat. Atunci f_x : String $\rightarrow$ String și
  - dacă $x <= 0$, atunci $f_x(y) = ""$ oricare y
  - dacă $x > 0$, atunci $f_x(y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \end{cases}$

Funcții în matematică

Exemplu

$A = $Int, $B = C = $String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

- Fie $x \in$ Int arbitrar, fixat. Atunci f_x : String $\to$ String și
  - dacă $x <= 0$, atunci $f_x(y) = ""$ oricare y
  - dacă $x > 0$, atunci $f_x(y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \end{cases}$
- Fie $y \in$ String arbitrar, fixat. Atunci f^y : Int $\to$ String și

$$f^y(x) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.
- Dacă notăm $B \to C \stackrel{not}{=} \{h : B \to C \mid h \text{ funcție}\}$ observăm că $f_x \in B \to C$ pentru orice $x \in A$.

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.
- Dacă notăm $B \to C \stackrel{not}{=} \{h : B \to C \mid h \text{ funcție}\}$ observăm că $f_x \in B \to C$ pentru orice $x \in A$.
- Asociem lui f funcția

  $cf : A \to (B \to C), \;\; cf(x) = f_x$

  Observăm că pentru fiecare element $x \in A$, funcția cf întoarce ca rezultat funcția $f_x \in B \to C$, adică

  $cf(x)(y) = z$ dacă și numai dacă $f(x, y) = z$

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.
- Dacă notăm $B \to C \stackrel{not}{=} \{h : B \to C \mid h \text{ funcție}\}$ observăm că $f_x \in B \to C$ pentru orice $x \in A$.
- Asociem lui f funcția

  $$cf : A \to (B \to C), \;\; cf(x) = f_x$$

  Observăm că pentru fiecare element $x \in A$, funcția cf întoarce ca rezultat funcția $f_x \in B \to C$, adică

  $$cf(x)(y) = z \text{ dacă și numai dacă } f(x, y) = z$$

Forma curry. Vom spune că funcția cf este *forma curry* a funcției f.

De la matematică la Haskell

Funcția f : Int $\times$ String $\rightarrow$ String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

poate fi definită în Haskell astfel:

```haskell
f :: (Int, String) -> String
f (n,s) = take n s
```

De la matematică la Haskell

Funcția f : Int × String → String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

poate fi definită în Haskell astfel:

```haskell
f :: (Int, String) -> String
f (n,s) = take n s
```

Observăm că:

```haskell
Prelude> let cf = curry f
Prelude> :t cf
cf :: Int -> String -> String
Prelude> f(1,"abc")
"a"
Prelude> cf 1 "abc"
"a"
```

Currying

"Currying" este procedeul prin care o funcție cu mai multe argumente este transformată într-o funcție care are un singur argument și întoarce o altă funcție.

- In Haskell toate funcțiile sunt în forma **curry**, deci au un singur argument.
- Operatorul $\to$ pe tipuri este asociativ la dreapta, adică tipul $a_1 \to a_2 \to \cdots \to a_n$ îl gândim ca $a_1 \to (a_2 \to \cdots (a_{n-1} \to a_n) \cdots)$.
- Aplicarea funcțiilor este asociativă la stânga, adică expresia $f\ x_1 \cdots x_n$ o gândim ca $(\cdots((f\ x_1)\ x_2) \cdots x_n)$.

Funcții și mulțimi

Teoremă. Mulțimile $(A \times B) \to C$ și $A \to (B \to C)$ sunt echipotente.

Funcții și mulțimi

Teoremă. Mulțimile $(A \times B) \to C$ și $A \to (B \to C)$ sunt echipotente.

Observație
Funcțiile curry și uncurry din Haskell stabilesc bijecția din teoremă:

```
Prelude> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
Prelude> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

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

- are două argumente, de tipuri (a -> b) și [a],
  adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

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

- are două argumente, de tipuri (a -> b) și [a],
  adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

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

Quiz time!

Seria 23: `https://www.questionpro.com/t/AT4qgZpP3R`

Seria 24: `https://www.questionpro.com/t/AT4NiZpPWu`

Seria 25: `https://www.questionpro.com/t/AT4qgZpP3X`

Operatori. Secțiuni

Operatorii sunt funcții cu două argumente

Operatorii în Haskell

- au două argumente
- pot fi apelați folosind notația infix
- pot fi definiți folosind numai "simboluri" (ex: *!*)
  - în definiția tipului operatorul este scris între paranteze

Operatorii sunt funcții cu două argumente

Operatorii în Haskell

- au două argumente
- pot fi apelați folosind notația infix
- pot fi definiți folosind numai "simboluri" (ex: *!*)
  - în definiția tipului operatorul este scris între paranteze

- Operatori predefiniți

```haskell
(||) :: Bool -> Bool -> Bool
 (:) :: a -> [a] -> [a]
 (+) :: Num a => a -> a -> a
```

Operatorii sunt funcții cu două argumente

Operatorii în Haskell

- au două argumente
- pot fi apelați folosind notația infix
- pot fi definiți folosind numai "simboluri" (ex: *!*)
  - în definiția tipului operatorul este scris între paranteze

- Operatori predefiniți

```haskell
(||) :: Bool -> Bool -> Bool
 (:) :: a -> [a] -> [a]
 (+) :: Num a => a -> a -> a
```

- Operatori definiți de utilizator

```haskell
(&&&) :: Bool -> Bool -> Bool   -- atentie la paranteze
True &&& b = b
False &&& _ = False
```

Funcții ca operatori

```
Prelude> mod 5 2
1
Prelude> 5 `mod` 2
1
```

Funcții ca operatori

```
Prelude> mod 5 2
1
Prelude> 5 `mod` 2
1
```

Operatorii care sunt definiți în formă infix, sunt apelați în formă prefix folosind paranteze

```
2 + 3 == (+) 2 3
```

Operatorii care sunt definiți în formă prefix, sunt apelați în formă infix folosind \` \` *(backtick)*

```
mod 5 2 == 5 `mod` 2
```

Funcții ca operatori

```
Prelude> mod 5 2
1
Prelude> 5 `mod` 2
1
```

Operatorii care sunt definiți în formă infix, sunt apelați în formă prefix folosind paranteze

```
2 + 3 == (+) 2 3
```

Operatorii care sunt definiți în formă prefix, sunt apelați în formă infix folosind \` \` *(backtick)*

```
mod 5 2 == 5 `mod` 2
```

```
elem :: a -> [a] -> Bool
Prelude> 1 `elem` [1,2,3]
True
```

Precedență și asociativitate

```
Prelude> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False
```

Precedență și asociativitate

```
Prelude> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False
True
```

Precedență și asociativitate

```
Prelude> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False
True
```

Precedence	Left associative	Non-associative	Right associative
9	!!		.
8			^, ^^, **
7	*, /, ‘**div**‘, ‘**mod**‘, ‘**rem**‘, ‘**quot**‘		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, ‘**elem**‘, ‘**notElem**‘	
3			&&
2			||
1	>>, >>=		
0			$, $!, ‘**seq**‘

Asociativitate

Operatorul - asociativ la stânga

```
5 - 2 - 1 == (5 - 2) - 1                -- /= 5 - (2 - 1)
```

Asociativitate

Operatorul - asociativ la stânga

```
5 - 2 - 1 == (5 - 2) - 1        -- /= 5 - (2 - 1)
```

Operatorul : asociativ la dreapta

```
5 : 2 : [] == 5 : (2 : [])
```

Asociativitate

Operatorul - asociativ la stânga

```
5 - 2 - 1 == (5 - 2) - 1                    -- /= 5 - (2 - 1)
```

Operatorul : asociativ la dreapta

```
5 : 2 : [] == 5 : (2 : [])
```

Operatorul ++ asociativ la dreapta

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
l1 ++ l2 ++ l3 ++ l4 ++ l5 == l1 ++ (l2 ++ (l3 ++ (l4 ++ l5)))
```

Asociativitate

Pentru a seta asociativitatea unui operator vom folosi următoarea notație

```
infix[l|r] NUMAR <operatori-separati-prin-virgula>
```

- **infix** - neasociativ
- **infixr** - asociativ la dreapta
- **infixl** - asociativ la stânga
- NUMAR - precedența (între 0 și 9)

Pentru a descoperi asociativitatea unui operator se poate folosi comanda `:i` în GHCI

Asociativitate - Exemple

```haskell
infix  4  ==, /=, <, <=, >=, >

infixr 3  &&
(&&) :: Bool -> Bool -> Bool

infixr 2  ||
(||) :: Bool -> Bool -> Bool

infixl 9 !!
(!!) :: [a] -> Int -> a

infixl 7 `div`
div :: Integral a => a -> a -> a
```

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).

Matematic, ele corespund aplicării parțiale a funcției op.

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).

Matematic, ele corespund aplicării parțiale a funcției op.

Aplicarea parțială.

Fie $f : A \times B \to C$ o funcție. În mod uzual scriem

$f(a, b) = c$ unde $a \in A$, $b \in B$ și $c \in C$.

Pentru $a \in A$ și $b \in B$ (arbitrare, fixate) definim

$f_a : B \to C$, $f_a(b) = c$ dacă și numai dacă $f(a, b) = c$,

$f^b : A \to C$, $f^b(a) = c$ dacă și numai dacă $f(a, b) = c$.

Funcțiile f_a și f_b se obțin prin aplicarea parțială a funcției f.

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).

- secțiunile lui ++ sunt (++ e) și (e ++)

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).

- secțiunile lui ++ sunt (++ e) și (e ++)

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
Prelude> (++ " world!") "Hello" -- atentie la paranteze
"Hello world!"
```

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).

- secțiunile lui ++ sunt (++ e) și (e ++)

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
Prelude> (++ " world!") "Hello" -- atentie la paranteze
"Hello world!"
Prelude> ++ " world!" "Hello"
error
```

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).

- secțiunile lui ++ sunt (++ e) și (e ++)

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
Prelude> (++ " world!") "Hello" -- atentie la paranteze
"Hello world!"
Prelude> ++ " world!" "Hello"
error
```

- secțiunile lui <-> sunt (<-> e) și (e <->), unde

```
Prelude> let x <-> y = x-y+1  -- definit de utilizator
Prelude> :t (<-> 3)
(<-> 3) :: Num a => a -> a
Prelude> (<-> 3) 4
2
```

Secțiuni

- secțiunile operatorului (:)

Secțiuni

- secțiunile operatorului (:)

```
Prelude> (2:)[1,2]
[2,1,2]
Prelude> (:[1,2]) 3
[3,1,2]
```

Secțiuni

- secțiunile operatorului (:)

```
Prelude> (2:)[1,2]
[2,1,2]
Prelude> (:[1,2]) 3
[3,1,2]
```

Secțiunile sunt afectate de asociativitatea și precedența operatorilor.

```
Prelude> :t (+ 3 * 4)
(+ 3 * 4) :: Num a => a -> a
Prelude> :t (* 3 + 4)
error -- + are precedenta mai mica decat *
Prelude> :t (* 3 * 4)
error  -- * este asociativa la stanga
Prelude> :t (3 * 4 *)
(3 * 4 *) :: Num a => a -> a
```

Compunerea funcțiilor — operatorul .

Matematic. Date fiind $f : A \to B$ și $g : B \to C$, compunerea lor, notată $g \circ f : A \to C$, este dată de formula

$$(g \circ f)(x) = g(f(x))$$

În Haskell.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)
```

Exemplu

```
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> :t take
take :: Int -> [a] -> [a]
Prelude> :t take 5 . reverse
take 5 . reverse :: [a] -> [a]

Prelude> (take 5 . reverse) [1..10]
[10,9,8,7,6]

Prelude> (head . reverse . take 5) [1..10]
5
```

Operatorul $

Operatorul ($) are precedența 0.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

```
Prelude> (head . reverse . take 5) [1..10]
5

Prelude> head . reverse . take 5 $ [1..10]
5
```

Operatorul ($) este asociativ la dreapta.

```
Prelude> head $ reverse $ take 5 $ [1..10]
5
```

Funcții de nivel înalt

Funcții anonime

Funcții anonime = lambda expresii

\x1 x2 $\cdots$ xn -> expresie

Funcții anonime

Funcții anonime = lambda expresii

\x1 x2 $\cdots$ **xn -> expresie**

```
Prelude> (\x -> x + 1) 3
4
Prelude> inc = \x -> x + 1
Prelude> add = \x y -> x + y
Prelude> aplic = \f x -> f x

Prelude> map (\x -> x+1) [1,2,3,4]
[2,3,4,5]
```

Funcțiile sunt valori (*first-class citizens*).

Funcțiile pot fi folosite ca argumente pentru alte funcții.

Funcțiile sunt valori

Exemplu:

```haskell
flip :: (a -> b -> c) -> (b -> a -> c)
```

Funcțiile sunt valori

Exemplu:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

- definiția cu lambda expresii

  ```
  flip f = \x y -> f y x
  ```

- definiția folosind șabloane

  ```
  flip f x y = f y x
  ```

- **flip** ca valoare de tip funcție

  ```
  flip = \f x y -> f y x
  ```

Funcții de ordin înalt

```haskell
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

Funcții de ordin înalt

```
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Funcții de ordin înalt

```haskell
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

```haskell
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Un exemplu mai complicat:

```haskell
Prelude> map ($ 3) [(4 +), (10 *), (^ 2), sqrt]
```

Funcții de ordin înalt

```
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Un exemplu mai complicat:

```
Prelude> map ($ 3) [(4 +), (10 *), (^ 2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

Funcții de ordin înalt

```
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Un exemplu mai complicat:

```
Prelude> map ($ 3) [(4 +), (10 *), (^ 2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

În acest caz:

- primul argument este o sectiune a operatorului ($)
- al doilea argument este o lista de functii

  map ($ x) [$f_1, \ldots, f_n$] == [f_1 x, ..., f_n x]

Funcții de ordin înalt

```
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [x | x <- l, p x]

Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Funcții de ordin înalt

```
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [x | x <- l, p x]
```

```
Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Compunere și aplicare

```
Prelude> let f l = map (* 3) (filter (>= 2) l)
Prelude> f [1,3,4]
```

Funcții de ordin înalt

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [x | x <- l, p x]
```

```haskell
Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Compunere și aplicare

```haskell
Prelude> let f l = map (* 3) (filter (>= 2) l)
Prelude> f [1,3,4]
[9, 12]                 -- [ x * 3 | x <- [1,3,4], x >=2 ]
```

Funcții de ordin înalt

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [x | x <- l, p x]
```

```haskell
Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Compunere și aplicare

```haskell
Prelude> let f l = map (* 3) (filter (>= 2) l)
Prelude> f [1,3,4]
[9, 12]                -- [ x * 3 | x <- [1,3,4], x >=2 ]
```

Definiția compozițională *(pointfree style)*:

```haskell
f = map (* 3) . filter (>=2)
```

Quiz time!

Seria 23: `https://www.questionpro.com/t/AT4qgZpJkQ`

Seria 24: `https://www.questionpro.com/t/AT4NiZpJGy`

Seria 25: `https://www.questionpro.com/t/AT4qgZpJkT`

Pe săptămâna viitoare!