

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C02 - Tipuri de date

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Tipuri de date

Tipuri de date. Sistemul tipurilor

"There are three interesting aspects to types in Haskell: they are strong, they are static, and they can be automatically inferred."

<http://book.realworldhaskell.org/read/types-and-functions.html>

tare – garantează absența anumitor erori

static – tipul fiecărei valori este calculat la compilare

dedus automat – compilatorul deduce automat tipul fiecărei expresii

```
Prelude> :t [( 'a' , 1 , "abc" ) ]  
[( 'a' , 1 , "abc" ) ] :: Num b => [( Char , b , [ Char ] )]
```

Tipurile de bază: Int, Integer, Float, Double, Bool, Char, String

Sistemul tipurilor

Tipurile de bază: Int, Integer, Float, Double, Bool, Char, String

Tipuri compuse: tupluri și liste

```
Prelude> :t ('a', True)  
('a', True) :: (Char, Bool)
```

```
Prelude> :t ["ana", "ion"]  
["ana", "ion"] :: [[Char]]
```

Sistemul tipurilor

Tipurile de bază: Int, Integer, Float, Double, Bool, Char, String

Tipuri compuse: tupluri și liste

```
Prelude> :t ('a', True)  
('a', True) :: (Char, Bool)
```

```
Prelude> :t ["ana", "ion"]  
["ana", "ion"] :: [[Char]]
```

Tipuri noi definite de utilizator:

```
data RGB = Red | Green | Blue  
data Point a = Pt a a      -- tip parametrizat  
                           -- a este variabila de tip
```

Tipuri de date

Integer: 4, 0, -5

```
Prelude> 4 + 3
```

```
Prelude> (+) 4 3
```

```
Prelude> mod 4 3
```

```
Prelude> 4 'mod' 3
```

Float: 3.14

```
Prelude> truncate 3.14
```

```
Prelude> sqrt 4
```

```
Prelude> let x = 4 :: Int
```

```
Prelude> sqrt (fromIntegral x)
```

Char: 'a','A','\n'

```
Prelude> import Data.Char
```

```
Prelude Data.Char> chr 65
```

```
Prelude Data.Char> ord 'A'
```

```
Prelude Data.Char> toUpper 'a'
```

```
Prelude Data.Char> digitToInt '4'
```

Tipuri de date

Bool: True, False

data Bool = True | False

Prelude> True && False || True

Prelude> not True

Prelude> 1 /= 2

Prelude> 1 == 2

String: "prog\ndec"

type String = [Char] -- sinonim pentru tip

Prelude> "aa"++"bb"

"aabb"

Prelude> "aabb" !! 2

'b'

Prelude> lines "prog\ndec"

["prog","dec"]

Prelude> words "pr og\nde cl"

["pr","og","de","cl"]

Tipul listă

```
Prelude>:t [True, False, True]  
[True, False, True] :: [Bool]
```

Tipuri de date compuse

Tipul listă

```
Prelude> :t [True, False, True]  
[True, False, True] :: [Bool]
```

Tipul tuplu – secvențe de tipuri deja existente

```
Prelude> :t (1 :: Int, 'a', "ab")  
(1 :: Int, 'a', "ab") :: (Int, Char, [Char])
```

```
Prelude> fst (1, 'a') -- numai pentru perechi  
Prelude> snd (1, 'a')
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim dacă introducem comanda?

```
Prelude> :t 1
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim dacă introducem comanda?

```
Prelude> :t 1
```

Răspunsul primit este:

```
1 :: Num a => a
```

Semnificația este următoarea:

- Num este o clasă de tipuri
- a este un parametru de tip
- 1 este o valoare de tipul a din clasa Num

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim dacă introducem comanda?

```
Prelude> :t 1
```

Răspunsul primit este:

```
1 :: Num a => a
```

Semnificația este următoarea:

- Num este o clasă de tipuri
- a este un parametru de tip
- 1 este o valoare de tipul a din clasa Num

```
Prelude> :t [1,2,3]
```

```
[1,2,3] :: Num t => [t]
```

Funcții în Haskell. Terminologie

Prototipul funcției

- numele funcției
- semnătura funcției

`double :: Integer -> Integer`

Definiția funcției

- numele funcției
- parametrul formal
- corpul funcției

`double elem = elem + elem`

Aplicarea funcției

- numele funcției
- parametrul actual (argumentul)

`double 5`

Exemplu: funcție cu două argumente

Prototipul funcției

- numele funcției
- semnătura funcției

`add :: Integer -> Integer -> Integer`

Definiția funcției

- numele funcției
- parametrii formali
- corpul funcției

`add elem1 elem2 = elem1 + elem2`

Aplicarea funcției

- numele funcției
- argumentele

`add 3 7`

Exemplu: funcție cu un argument de tip tuplu

Prototipul funcției

dist :: (Integer, Integer) -> Integer

- numele funcției
- semnatura funcției

Definiția funcției

dist (elem1, elem2) = abs (elem1 - elem2)

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției

dist (5, 7)

- numele funcției
- argumentul

Tipuri de funcții

Prelude> :t abs

abs :: Num a => a -> a

Prelude> :t div

div :: Integral a => a -> a -> a

Prelude> :t (:)

(:) :: a -> [a] -> [a]

Prelude> :t (++)

(++) :: [a] -> [a] -> [a]

Prelude> :t zip

zip :: [a] -> [b] -> [(a, b)]

Definirea funcțiilor

`fact :: Integer -> Integer`

- Definiție folosind **if**

```
fact n = if n == 0 then 1
        else n * fact(n-1)
```

- Definiție folosind ecuații

```
fact 0 = 1
fact n = n * fact(n-1)
```

- Definiție folosind cazuri

```
fact n
| n == 0    = 1
| otherwise = n * fact(n-1)
```

Definirea funcțiilor folosind șabloane și ecuații

```
fact :: Integer -> Integer
```

```
fact 0 = 1
```

```
fact n = n * fact(n-1)
```

- variabilele și valorile din partea stângă a semnului = sunt *șabloane*
- când funcția este apelată se încearcă potrivirea parametrilor actuali cu șabloanele, ecuațiile fiind încercate *în ordinea scrierii*
- în definiția factorialului, 0 și n sunt șabloane:
0 se va potrivi numai cu el însuși,
iar n se va potrivi cu orice valoare de tip Integer

Definirea funcțiilor folosind șabloane și ecuații

În Haskell, ordinea ecuațiilor este importantă.

Să presupunem că schimbăm ordinea ecuațiilor din definiția factorialului:

```
fact :: Integer -> Integer
fact n = n * fact(n-1)
fact 0 = 1
```

Ce se întâmplă?

Definirea funcțiilor folosind șabloane și ecuații

În Haskell, ordinea ecuațiilor este importantă.

Să presupunem că schimbăm ordinea ecuațiilor din definiția factorialului:

```
fact :: Integer -> Integer
fact n = n * fact (n-1)
fact 0 = 1
```

Ce se întâmplă?

Deoarece n este un pattern care se potrivește cu orice valoare, inclusiv cu 0, orice apel al funcției va alege prima ecuație.

Astfel, funcția **nu își va încheia execuția**.

Definirea funcțiilor folosind șabloane și ecuații

Tipul Bool este definit în Haskell astfel:

```
data Bool = True | False
```

Putem defini operația || astfel

```
(||) :: Bool -> Bool -> Bool
```

```
False || x = x
```

```
True  || _ = True
```

În acest exemplu șabloanele sunt `_`, `x`, `True` și `False`.

Observăm că `True` și `False` sunt constructori de date și se vor potrivi numai cu ei înșiși.

Șablonul `_` se numește *wild-card pattern*; el se potrivește cu orice valoare.

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 semnatura 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]

Liste

Orice listă poate fi scrisă folosind doar constructorul `(:)` și lista vidă `[]`.

- $[1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []$
- $"abc" == ['a','b','c'] == 'a' : ('b' : ('c' : [])) == 'a' : 'b' : 'c' : []$

Definiție recursivă. O **listă** este

- **vidă**, notată `[]`, sau
- **compusă**, notată $x : xs$, dintr-un element x numit **capul listei** (*head*) și o listă xs numită **coada listei** (*tail*).

Definirea listelor. Operații

Intervale și progresii

<code>interval = ['c'..'e']</code>	-- <code>['c','d','e']</code>
<code>progresie = [20,17..1]</code>	-- <code>[20,17,14,11,8,5,2]</code>
<code>progresie' = [2.0,2.5..4.0]</code>	-- <code>[2.0,2.5,3.0,3.5,4.0]</code>

Operații

Prelude> import Data.List

Prelude> [1,2,3] !! 2

3

Prelude> "abcd" !! 0

'a'

Prelude> [1,2] ++ [3]

[1,2,3]

Definiția prin selecție $\{x \mid P(x)\}$

$[E(x) \mid x \leftarrow [x_1, \dots, x_n], P(x)]$

```
Prelude> xs = [0..10]
```

```
Prelude> [x | x <- xs, even x]  
[0,2,4,6,8,10]
```

```
Prelude> xs = [0..6]
```

```
Prelude> [(x,y) | x <- xs, y <- xs, x + y == 10]  
[(4,6),(5,5),(6,4)]
```

Definiția prin selecție $\{x \mid P(x)\}$

$$[E(x) \mid x \leftarrow [x_1, \dots, x_n], P(x)]$$

Putem folosi **let** pentru domeniu de vizibilitate local.

```
Prelude> [(i, j) | i <- [1..2],  
                let k = 2 * i, j <- [1..k]]  
[(1, 1), (1, 2),  
 (2, 1), (2, 2), (2, 3), (2, 4)]
```

```
Prelude> :t zip
```

```
zip :: [a] -> [b] -> [(a, b)]
```

```
Prelude> ys = ['A'.. 'E']
```

```
Prelude> zip [1..] ys
```

```
[(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), (5, 'E')]
```

```
Prelude> xs = ['A'.. 'Z']
```

```
Prelude> [x | (i, x) <- [1..] 'zip' xs, even i]
```

```
"BDFHJLNPRTVXZ"
```

Observați diferența!

```
Prelude> zip [1..3] ['A'..'D']  
[(1, 'A'), (2, 'B'), (3, 'C')]
```

```
Prelude> [(x,y) | x <- [1..3], y <- ['A'..'D']]  
[(1, 'A'), (1, 'B'), (1, 'C'), (1, 'D'),  
 (2, 'A'), (2, 'B'), (2, 'C'), (2, 'D'),  
 (3, 'A'), (3, 'B'), (3, 'C'), (3, 'D')]
```

Lazy: argumentele sunt evaluate doar când e necesar și doar cât e necesar.

```
Prelude> head []  
*** Exception: Prelude.head: empty list  
Prelude> x = head []  
Prelude> f a = 5  
Prelude> f x  
5  
Prelude> [1, head [], 3] !! 0  
1  
Prelude> [1, head [], 3] !! 1  
*** Exception: Prelude.head: empty list
```


Liste infinite

Drept consecință a **evaluării leneșe**, se pot defini liste infinite.

```
Prelude> natural = [0..]
```

```
Prelude> take 5 natural  
[0,1,2,3,4]
```

```
Prelude> evenNat = [0,2..] -- progresie infinita
```

```
Prelude> take 7 evenNat  
[0,2,4,6,8,10,12]
```

```
Prelude> ones = [1,1..]
```

```
Prelude> zeros = [0,0..]
```

```
Prelude> both = zip ones zeros
```

```
Prelude> take 5 both  
[(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

`[1,2,3] == 1:[2,3] == 1:2:[3] == 1:2:3:[]`

Prelude> `x:y = [1,2,3]`

Prelude> `x`

1

Prelude> `y`

[2,3]

Ce s-a întâmplat?

- `x : y` este un șablon pentru liste
- potrivirea dintre `x : y` și `[1,2,3]` a avut ca efect:
 - "deconstrucția" valorii `[1,2,3]` în `1 : [2,3]`
 - legarea lui `x` la 1 și a lui `y` la `[2,3]`

Șabloane (patterns) pentru liste

Definiții folosind șabloane

```
reverse [] = []
```

```
reverse (x:xs) = (reverse xs) ++ [x]
```

x : xs se potrivește cu liste nevide.

Șabloanele sunt definite folosind constructori.

De exemplu, operația de concatenare pe liste este

```
(++) :: [a] -> [a] -> [a],
```

dar `[x] ++ [1] = [2,1]` **nu** va avea ca efect legarea lui `x` la `2`.

```
Prelude> [x] ++ [1] = [2,1]
```

```
Prelude> x
```

error: ...

Șabloanele sunt liniare

Șabloanele sunt **liniare**, adică o variabilă apare cel mult o dată.

Șabloanele în care o variabilă apare de mai multe ori generează mesaje de eroare. De exemplu:

- `x:x:[1] = [2,2,1]`
- `ttail (x:x:t) = t`
- `foo x x = x^2`

error: Conflicting definitions for x

O soluție este folosirea gărzilor:

```
ttail (x:y:t) | (x==y) = t
           | otherwise = ...
```

```
foo x y | (x == y) = x^2
         | otherwise = ...
```

Quiz time!



<https://tinyurl.com/PF2023-C02-Quiz1>

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

- Operatori predefiniți

`(||) :: Bool -> Bool -> Bool`

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

`(+) :: Num a => a -> a -> a`

- Operatori definiți de utilizator

`(&&&) :: Bool -> Bool -> Bool -- atentie la paranteze`

`True &&& b = b`

`False &&& _ = False`

Funcții ca operatori

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`

Prelude> mod 5 2

1

Prelude> 5 `mod` 2

1

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**
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'

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

$$(++) :: [a] \rightarrow [a] \rightarrow [a]$$

$$[] ++ ys = ys$$

$$(x:xs) ++ ys = x:(xs ++ ys)$$

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

Secțiuni (operator sections)

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

Secțiunile lui **(++) sunt **(++ e)** și **(e ++)****

```
Prelude> :t (++)
```

```
(++) :: [a] -> [a] -> [a]
```

```
Prelude> :t (++ " world!")
```

```
(++ " world!") :: [Char] -> [Char]
```

```
Prelude> (++ " world!") "Hello "
```

```
"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> x <-> y = x-y+1  -- definit de noi
```

```
Prelude> :t (<-> 3)
```

```
(<-> 3) :: Num a => a -> a
```

```
Prelude> (<-> 3) 4
```

```
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
```

Pe săptămâna viitoare!