

Programare funcțională

Date structurate.

Ioana Leuștean
Traian Florin Șerbănuță

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

1 Tipuri de date algebrice

2 Parțialitate - tipul Maybe

3 Ori-ori – tipul Either

Tipuri de date algebrice

Tipuri sumă

- În Haskell tipul **Bool** este definit astfel:

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

Bool este constructor de tip

False și **True** sunt constructori de date

Tipuri sumă

- În Haskell tipul **Bool** este definit astfel:

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

Bool este constructor de tip

False și **True** sunt constructori de date

- În mod similar putem defini

```
data Season = Spring | Summer  
             | Autumn | Winter
```

Season este constructor de tip

Spring, Summer, Autumn și Winter sunt constructori de date

Tipuri sumă

- În Haskell tipul **Bool** este definit astfel:

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

Bool este constructor de tip

False și **True** sunt constructori de date

- În mod similar putem defini

```
data Season = Spring | Summer  
             | Autumn | Winter
```

Season este constructor de tip

Spring, Summer, Autumn și Winter sunt constructori de date

Bool și Season sunt tipuri de date **sumă**, adică sunt definite prin enumerarea alternativelor.

Tipuri sumă

data Bool = False | True

- Operațiile se definesc prin "pattern matching":

not :: Bool -> Bool

not False = True

not True = False

(&&), (||) :: Bool -> Bool -> Bool

False && q = False

True && q = q

False || q = q

True || q = True

Tip sumă: anotimpuri

```
data Season = Spring | Summer  
           | Autumn | Winter
```

```
sucesor Spring = Summer  
sucesor Summer = Autumn  
sucesor Autumn = Winter  
sucesor Winter = Spring
```

```
showSeason Spring = "Primavara"  
showSeason Summer = "Vara"  
showSeason Autumn = "Toamna"  
showSeason Winter = "Iarna"
```


Tipuri produs

- Să definim un tip de date care să aibă ca valori "punctele" cu două coordonate de tipuri oarecare:

data Point a b = Pt a b

Point este constructor de tip

Pt este constructor de date

- Pentru a accesa componentele, definim **proiecțiile**:

pr1 : Point a b -> a

pr1 (Pt x _) = x

pr2 : Point a b -> b

pr2 (Pt _ y) = y

Point este un tip de date **produs**, definit prin **combinarea** tipurilor a și b.

Tipuri produs

```
data Point a b = Pt a b
```

```
Prelude> :t (Pt 1 "c")  
(Pt 1 "c") :: Num a => Point a [Char]
```

```
Prelude> :t Pt  
Pt :: a -> b -> Point a b  
-- constructorul de date este operatie
```

```
Prelude> :t (Pt 1)  
(Pt 1) :: Num a => b -> Point a b
```

Tipuri produs

```
data Point a b = Pt a b
```

```
Prelude> :t (Pt 1 "c")  
(Pt 1 "c") :: Num a => Point a [Char]
```

```
Prelude> :t Pt  
Pt :: a -> b -> Point a b  
-- constructorul de date este operatie
```

```
Prelude> :t (Pt 1)  
(Pt 1) :: Num a => b -> Point a b
```

- Se pot defini operații:

```
pointFlip :: Point a b -> Point b a  
pointFlip Pt = flip Pt  
-- pointFlip (Pt x y) = Pt y x
```

Tipuri de date definite recursiv

- Declarația listelor ca tip de date algebric

```
data   List a  = Nil  
        | Cons a (List a)
```

Tipuri de date definite recursiv

- Declarația listelor ca tip de date algebric

```
data List a = Nil
              | Cons a (List a)
```

- Se pot defini operații

```
append :: List a -> List a -> List a
append Nil ys           = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală

$$\begin{array}{lcl}
 \text{data Typename} & = & \text{Cons}_1 \ t_{11} \dots t_{1k_1} \\
 & & | \text{Cons}_2 \ t_{21} \dots t_{2k_2} \\
 & & | \dots \\
 & & | \text{Cons}_n \ t_{n1} \dots t_{nk_n}
 \end{array}$$

unde $k_1, \dots, k_n \geq 0$

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală

$$\begin{aligned} \text{data } \textit{Typename} \quad = \quad & \textit{Cons}_1 \ t_{11} \dots t_{1k_1} \\ & | \textit{Cons}_2 \ t_{21} \dots t_{2k_2} \\ & | \dots \\ & | \textit{Cons}_n \ t_{n1} \dots t_{nk_n} \end{aligned}$$

unde $k_1, \dots, k_n \geq 0$

- Se pot folosi tipuri sumă și tipuri produs.
- Se pot defini tipuri parametrizate.
- Se pot folosi definiții recursive.

Tipuri de date algebrice

Forma generală

$$\begin{aligned}
 \text{data } \textit{Typename} \quad = \quad & \textit{Cons}_1 \ t_{11} \dots t_{1k_1} \\
 & | \textit{Cons}_2 \ t_{21} \dots t_{2k_2} \\
 & | \dots \\
 & | \textit{Cons}_n \ t_{n1} \dots t_{nk_n}
 \end{aligned}$$

unde $k_1, \dots, k_n \geq 0$

Atenție! Alternativele trebuie să conțină **constructori**.

Tipuri de date algebrice

Forma generală

$$\begin{aligned} \text{data } \textit{Typename} \quad = \quad & \textit{Cons}_1 \ t_{11} \dots t_{1k_1} \\ & | \textit{Cons}_2 \ t_{21} \dots t_{2k_2} \\ & | \dots \\ & | \textit{Cons}_n \ t_{n1} \dots t_{nk_n} \end{aligned}$$

unde $k_1, \dots, k_n \geq 0$

Atenție! Alternativele trebuie să conțină **constructori**.

data StrInt = **String** | **Int** este **greșit**

Tipuri de date algebrice

Forma generală

$$\begin{aligned} \text{data } \text{Typename} \quad = \quad & \text{Cons}_1 \ t_{11} \dots t_{1k_1} \\ & | \text{Cons}_2 \ t_{21} \dots t_{2k_2} \\ & | \dots \\ & | \text{Cons}_n \ t_{n1} \dots t_{nk_n} \end{aligned}$$

unde $k_1, \dots, k_n \geq 0$

Atenție! Alternativele trebuie să conțină **constructori**.

data StrInt = **String** | **Int** este **greșit**

data StrInt = VS **String** | VI **Int** este **corect**

[VI 1, VS "abc", VI 34, VI 0, VS "xyz"] :: [StrInt]

Tipuri de date algebrice - exemple

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

```
data Season = Winter | Spring | Summer | Fall
```

```
data Shape = Circle Float | Rectangle Float Float
```

Tipuri de date algebrice - exemple

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

```
data Season = Winter | Spring | Summer | Fall
```

```
data Shape = Circle Float | Rectangle Float Float
```

```
data Maybe a = Nothing | Just a
```

```
data Pair a b = Pair a b
```

-- constructorul de tip si cel de date pot sa coincidă

Tipuri de date algebrice - exemple

data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Maybe a = Nothing | Just a

data Pair a b = Pair a b

-- constructorul de tip si cel de date pot sa coincidă

data Nat = Zero | Succ Nat

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp

data List a = Nil | Cons a (List a)

data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)

Constructori simboluri

```
data   List a  = Nil  
        | Cons a (List a)
```

Constructori simboluri

```
data List a = Nil
           | Cons a (List a)
```

Declarație ca tip de date algebric cu simboluri

```
data List a = Nil
           | a :: List a
  deriving (Show)
```

```
infixr 5 :::
```


Liste și tupluri

- Liste

data $[a] = [] \mid a : [a]$

Constructorii listelor sunt $[]$ și $:$ unde

$[] :: [a]$

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

Liste și tupluri

- Liste

data $[a] = [] \mid a : [a]$

Constructorii listelor sunt $[]$ și $:$ unde

$[] :: [a]$

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

- Tupluri

data $(a,b) = (a,b)$

data $(a,b,c) = (a,b,c)$

...

...

Nu există o declarație generică pentru tupluri, fiecare declarație de mai sus definește tuplul de lungimea corespunzătoare, iar constructorii pentru fiecare tip în parte sunt:

$(,) :: a \rightarrow b \rightarrow (a,b)$

$(,,) :: a \rightarrow b \rightarrow c \rightarrow (a,b,c)$

...

Utilizarea **type**

Cu **type** se pot redenumi tipuri deja existente.

Utilizarea **type**

Cu **type** se pot redenumi tipuri deja existente.

```
type FirstName  = String  
type LastName  = String  
type Age        = Int  
type Height     = Float  
type Phone     = String
```

```
data Person = Person FirstName LastName Age Height Phone
```

Exemplu - date personale. Proiecții

data Person = Person FirstName LastName Age Height Phone

firstName :: Person -> **String**

firstName (Person firstname _ _ _ _) = firstname

lastName :: Person -> **String**

lastName (Person _ lastname _ _ _) = lastname

age :: Person -> **Int**

age (Person _ _ age _ _ _) = age

height :: Person -> **Float**

height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> **String**

phoneNumber (Person _ _ _ _ number _) = number

Exemplu - date personale. Utilizare

```
Main*> let ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"
```

```
Main*> firstName ionel  
"Ion"
```

```
Main*> height ionel  
175.2
```

```
Main*> phoneNumber ionel  
"0712334567"
```

Date personale ca înregistrări

```
data Person = Person { firstName :: String  
                        , lastName :: String  
                        , age :: Int  
                        , height :: Float  
                        , phoneNumber :: String  
                        }
```

Date personale ca înregistrări

- Putem folosi atât forma algebrică cât și cea de înregistrare

```
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"
```

```
gigel = Person { firstName = "Gheorghe"
                 , lastName="Georgescu"
                 , age = 30, height = 192.3
                 , phoneNumber = "0798765432"
                 }
```

- Putem folosi și pattern-matching
- Proiecțiile sunt definite automat; sintaxă specializată pentru actualizări

```
nextYear :: Person -> Person
nextYear person = person { age = age person + 1 }
```


Date personale ca înregistrări

```
data Person = Person { firstName :: String
                        , lastName :: String
                        , age :: Int
                        , height :: Float
                        , phoneNumber :: String
                        }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                        , lastName :: String
                        , age :: Int
                        , height :: Float
                        , phoneNumber :: String
                        }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

```
*Main> nextYear ionel
```

No instance for (Show Person) arising from a use of 'print'

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                        , lastName  :: String
                        , age       :: Int
                        , height    :: Float
                        , phoneNumber :: String
                        }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

```
*Main> nextYear ionel
```

No instance for (Show Person) arising from a use of 'print'

Deși toate definițiile sunt corecte, o valoare de tip `Person` nu poate fi afișată deoarece nu este instanță a clasei **Show**.

Derivare automata pentru tipuri algebrice

Am definit tipuri de date noi:

```
data Season = Spring | Summer | Autumn | Winter  
           deriving (Eq, Ord, Show)
```

```
data Point a b = Pt a b  
           deriving (Eq, Ord, Show)
```

Cum putem să le facem instanțe ale claselor **Eq**, **Ord**, **Show**?

Putem să le facem explicit sau să folosim derivarea automată.

Atenție!

Derivarea automată poate fi folosită numai pentru unele clase predefinite.

Derivare automata vs Instanțiere explicită

- O clasă de tipuri este determinată de o mulțime de funcții.

```
class  Eq a  where
  (==)  :: a -> a -> Bool
  (/=)  :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
```

- Tipurile care aparțin clasei sunt instanțe ale clasei.
- Instanțierea prin derivare automată:

```
data Point a b = Pt a b
               deriving Eq
```

- Instanțiere explicită:

```
instance Eq a => Eq (Point a b) where
  (==) (Pt x1 y1) (Pt x2 y2) = (x == x1)
```

Derivare automata pentru tipuri algebrice

```
data Point a b = Pt a b
           deriving (Eq, Ord, Show)
```

Egalitatea, relația de ordine și modalitatea de afișare sunt definite implicit dacă este posibil:

```
*Main> Pt 2 3 < Pt 5 6
True
```

```
*Main> Pt 2 "b" < Pt 2 "a"
False
```

```
*Main Data.Char> Pt (+2) 3 < Pt (+5) 6
```

No instance for (Ord (Integer -> Integer)) arising from a use of '<'

Instanțiere explicită - exemplu

```
data Season = Spring | Summer | Autumn | Winter
```

```
Instance Eq Season where
  Spring == Spring = True
  Summer == Summer = True
  Autumn == Autumn = True
  Winter == Winter = True
  _      == _      = False
```

```
Instance Show Season where
  show Spring = "Primavara"
  show Summer = "Vara"
  show Autumn = "Toamna"
  show Winter = "Iarna"
```

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Declarație ca tip de date algebric folosind șabloane

```
data    Nat    =    Zero    |    Succ Nat
```

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Declarație ca tip de date algebric folosind șabloane

```
data    Nat    =    Zero    |    Succ Nat
```

- Putem să definim operații

```
(^^^) :: Float -> Nat -> Float
```

```
x ^^^ Zero      = 1.0
```

```
x ^^^ (Succ n) = x * x ^^^ n
```

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Declarație ca tip de date algebric folosind șabloane

```
data    Nat    =    Zero    |    Succ Nat
```

- Putem să definim operații

```
(^^^) :: Float -> Nat -> Float
x ^^^ Zero      = 1.0
x ^^^ (Succ n) = x * x ^^^ n
```

Comparați cu versiunea folosind notația predefinită

```
(^^) :: Float -> Int -> Float
x ^^ 0 = 1.0
x ^^ n = x * (x ^^ (n-1))
```

Exemplu: adunare și înmulțire pe Nat

Definiție pe tipul de date algebric

```

(+++) :: Nat -> Nat -> Nat
m +++ Zero      = m
m +++ (Succ n)  = Succ (m +++ n)

(*** ) :: Nat -> Nat -> Nat
m *** Zero      = Zero
m *** (Succ n)  = (m *** n) +++ m

```

Comparați cu versiunea folosind notația predefinită

```

(+) :: Int -> Int -> Int
m + 0 = m
m + n = (m + (n-1)) + 1

(*) :: Int -> Int -> Int
m * 0 = 0
m * n = (m * (n-1)) + m

```

Exemplu: liste

```
data List a = Nil  
           | a ::: List a  
           deriving (Show, Eq, Ord)  
  
infixr 5 :::
```

Exemplu: liste

```
data List a = Nil
           | a ::: List a
deriving (Show, Eq, Ord)
```

```
infixr 5 :::
```

- Putem defini operații:

```
(+++) :: List a -> List a -> List a
infixr 5 +++
Nil +++ ys          = ys
(x ::: xs) +++ ys = x ::: (xs +++ ys)
```

Exemplu: liste

```
data List a = Nil
           | a ::: List a
deriving (Show, Eq, Ord)
```

```
infixr 5 :::
```

- Putem defini operații:

```
(+++) :: List a -> List a -> List a
infixr 5 +++
Nil +++ ys      = ys
(x ::: xs) +++ ys = x ::: (xs +++ ys)
```

Comparați cu versiunea folosind notația predefinită

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

Parțialitate - tipul Maybe

Tipul Maybe (opțiune)

```
data Maybe a = Nothing | Just a
```

Tipul Maybe (opțiune)

```
data Maybe a = Nothing | Just a
```

Argumente opționale

```
power :: Maybe Int -> Int -> Int  
power Nothing n    = 2 ^ n  
power (Just m) n = m ^ n
```

Tipul Maybe (opțiune)

```
data Maybe a = Nothing | Just a
```

Argumente opționale

```
power :: Maybe Int -> Int -> Int
power Nothing n    = 2 ^ n
power (Just m) n = m ^ n
```

Rezultate opționale

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n 'div' m)
```

Maybe - folosirea unui rezultat opțional

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)
```

-- *utilizare gresita*

```
wrong :: Int -> Int -> Int
wrong n m = divide n m + 3
```

-- *utilizare corecta*

```
right :: Int -> Int -> Int
right n m = case divide n m of
    Nothing -> 3
    Just r -> r + 3
```

Maybe - folosirea unui rezultat opțional

Variantă descriptivă, cu pattern-uri

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n 'div' m)
```

-- utilizare cu descrieri de liste

```
quotients :: [Int] -> Int -> [Int]
quotients ns m = [q | Just q <- map ('divide' m) ns]
```

-- utilizare cu functii din biblioteci

```
quotients' :: [Int] -> Int -> [Int]
quotients' ns m = mapMaybe ('divide' m) ns
```

Ori-ori – tipul Either

Either A B (A sau B)

```
data Either a b = Left a | Right b
```

```
mylist :: [Either Int String]
```

```
mylist = [Left 4, Left 1, Right "hello", Left 2,  
         Right " ", Right "world", Left 17]
```

Definiți o funcție care calculează suma elementelor întregi.

```
addints    :: [Either Int String] -> Int
```

Either A B (A sau B)

```
data Either a b = Left a | Right b
```

```
mylist :: [Either Int String]
```

```
mylist = [Left 4, Left 1, Right "hello", Left 2,  
         Right " ", Right "world", Left 17]
```

Definiți o funcție care calculează suma elementelor întregi.

```
addints    :: [Either Int String] -> Int
```

```
addints    [] = 0
```

```
addints    (Left n : xs) = n + addints xs
```

```
addints    (Right s : xs) = addints xs
```

```
addints'   :: [Either Int String] -> Int
```

```
addints'   xs = sum [n | Left n <- xs]
```


A sau B

```
data Either a b = Left a | Right b
```

```
mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care întoarce concatenarea elementelor de tip **String**.

```
addstrs    :: [Either Int String] -> String
addstrs    []                        = ""
addstrs    (Left n : xs)            = addstrs xs
addstrs    (Right s : xs)           = s ++ addstrs xs
```

```
addstrs'   :: [Either Int String] -> String
addstrs'   xs = concat [s | Right s <- xs]
```

Pe săptămâna viitoare!