Programare funcțională

Introducere în programarea funcțională folosind Haskell
C05

Claudia Chiriță
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

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

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**.

Tip sumă: Bool

```
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: Season

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

succesor :: Season -> Season
succesor  Spring = Summer
succesor  Summer = Autumn
succesor  Autumn = Winter
succesor  Winter = Spring

showSeason :: Season -> String
showSeason Spring = "Primavara"
showSeason Summer = "Vara"
showSeason Autumn = "Toamna"
showSeason Winter = "Iarna"
```

Tipuri produs

Problemă. Să definim un tip de date care să aibă ca valori "puncte" 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
```

Se pot defini operații:

```
pointFlip :: Point a b -> Point b a
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)
```

- **List** este constructor de tip
- Nil și Cons sunt constructori de date

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.

Forma generală:

$$\begin{aligned} data\ \ Typename \quad = \quad & Cons_1 \ \ t_{11} \ldots t_{1k_1} \\ & \mid Cons_2 \ \ t_{21} \ldots t_{2k_2} \\ & \mid \ldots \\ & \mid Cons_n \ \ t_{n1} \ldots t_{nk_n} \end{aligned}$$

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

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

Tipuri de date algebrice

Forma generală:

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

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

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

data Pair a b = Pair a b
-- constructorul de tip și cel de date pot să coincidă

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)
```

Quiz time!

https://tinyurl.com/PF2023-C06-Q1

Liste cu simboluri

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

```
data [a] = [] | a : [a]
```

Constructorii listelor sunt [] și : unde

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

Tupluri cu simboluri

```
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 -> b -> (a,b)
(,,) :: a -> b -> c -> (a,b,c)
  ...
```

Exemplu: numerele naturale (Peano)

Declarație ca tip de date algebric folosind șabloane

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

Adunarea pe tipul de date algebric:

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

Comparați cu versiunea folosind notația predefinită:

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

Exemplu: numerele naturale (Peano)

Declarație ca tip de date algebric folosind șabloane

```
data Nat = Zero | Succ Nat

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

Comparați cu versiunea folosind notația predefinită:

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

Tipul Maybe (opțiune)

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

Rezultate opționale

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

Argumente opționale

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

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

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

Tipul Either (variante)

```
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]
```

Tipul Either (variante)

```
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!