Programare declarativă[1]

Tipuri de date algebrice

Traian Florin Șerbănuță

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

4 noiembrie 2016

[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Ce este un tip de date algebric?

Ce este un tip de date algebric?

Orice!

```
data Bool = False | True
data Season = Winter | Spring | Summer | Fall
data Shape = Circle Float | Rectangle Float Float
data List a = Nil | Cons a (List a)
data Nat = Zero | Succ Nat
data Exp = Lit Int | Add Exp Exp | Mul Exp Exp
data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
data Maybe a = Nothing | Just a
data Pair a b = Pair a b
data Either a b = Left a | Right b
```

Tipul de date Boolean

```
data Bool = False | True

not :: Bool -> Bool
not False   =  True
not True    =  False

(&&) :: Bool -> Bool -> Bool
False && q   =  False
True  && q   =  q

(||) :: Bool -> Bool -> Bool
False || q   =  q
True  || q   =  True
```

Definirea egalității și a reprezentării

Eq și Show

Eq

```
eqBool :: Bool -> Bool -> Bool
```

Show

```
showBool :: Bool -> String
```

Definirea egalității și a reprezentării

Eq și Show

Eq

```
eqBool :: Bool -> Bool -> Bool
eqBool False False = True
eqBool True  True  = True
eqBool _     _     = False
```

Show

```
showBool :: Bool -> String
showBool False = "False"
showBool True  = "True"
```

Anotimpuri

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

next :: Season -> Season
next Spring = Summer
next Summer = Autumn
next Autumn = Winter
next Winter = Spring
```

Definirea egalității și a reprezentării

Eq și Show

```
eqSeason :: Season -> Season -> Bool
eqSeason Spring Spring = True
eqSeason Summer Summer = True
eqSeason Autumn Autumn = True
eqSeason Winter Winter = True
eqSeason _      _      = False
```

```
showSeason :: Season -> String
showSeason Spring = "Spring"
showSeason Summer = "Summer"
showSeason Autumn = "Autumn"
showSeason Winter = "Winter"
```

Enumerări și indici

```
data Season = Winter | Spring | Summer | Fall
toInt :: Season -> Int
toInt Winter = 0
toInt Spring = 1
toInt Summer = 2
toInt Fall   = 3

fromInt :: Int -> Season
fromInt 0 = Winter
fromInt 1 = Spring
fromInt 2 = Summer
fromInt 3 = Fall

next :: Season -> Season


eqSeason :: Season -> Season -> Bool
```

Enumerări și indici

```
data Season = Winter | Spring | Summer | Fall
toInt :: Season -> Int
toInt Winter = 0
toInt Spring = 1
toInt Summer = 2
toInt Fall   = 3

fromInt :: Int -> Season
fromInt 0 = Winter
fromInt 1 = Spring
fromInt 2 = Summer
fromInt 3 = Fall

next :: Season -> Season
next x = fromInt ((toInt x + 1) `mod` 4)

eqSeason :: Season -> Season -> Bool
eqSeason x y = (toInt x == toInt y)
```

Cercuri și dreptunghiuri

```
type   Radius   =   Float
type   Width    =   Float
type   Height   =   Float

data   Shape    =   Circle Radius
                |   Rect Width Height

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w * h
```

Definirea egalității și a reprezentării

Eq și Show

```
eqShape :: Shape -> Shape -> Bool
```

```
showShape :: Shape -> String
```

Definirea egalității și a reprezentării

Eq și Show

```
eqShape :: Shape -> Shape -> Bool
eqShape (Circle r) (Circle r')  = (r == r')
eqShape (Rect w h) (Rect w' h') = (w == w') && (h == h')
eqShape _          _            = False

showShape :: Shape -> String
showShape (Circle r) = "Circle " ++ showF r
showShape (Rect w h) = "Rect " ++ showF w
    ++ " " ++ showF h

showF :: Float -> String
showF x | x >= 0    = show x
        | otherwise = "(" ++ show x ++ ")"
```

Teste și operatori de proiecție

```
isCircle :: Shape -> Bool
isCircle (Circle r) = True
isCircle _          = False

isRect :: Shape -> Bool
isRect (Rect w h) = True
isRect _          = False

radius :: Shape -> Float
radius (Circle r) = r

width :: Shape -> Float
width (Rect w h) = w

height :: Shape -> Float
height (Rect w h) = h
```

Pattern-matching

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w * h

area :: Shape -> Float
area s =
  if isCircle s then
    let
        r = radius s
    in
        pi * r^2
  else if isRect s then
    let
        w = width s
        h = height s
    in
        w * h
  else error "impossible"
```

Pattern-matching

Declarație ca tip de date algebric

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

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

Constructori simboluri

Declarație ca tip de date algebric cu simboluri

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

(+++) :: 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)
```

Definirea egalității și a reprezentării

Eq și Show

```
eqList :: Eq a => List a -> List a -> Bool
eqList Nil Nil  = True
eqList (x ::: xs)  (y:::ys) = x == y && eqList xs ys
eqList _           _          = False

showList :: Show a => List a -> String
showList Nil = "Nil"
showList (x ::: xs) = show x ++ " ::: " ++ showList xs
```

Numerele Naturale (Peano)

Declarație ca tip de date algebric

```
data    Nat  =    Zero
             |    Succ  Nat

(^^^) :: 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))
```

Adunare și înmulțire

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
```

Date personale

```
type  FirstName   =  String
type  LastName    =  String
type  Age         =  Int
type  Height      =  Float
type  PhoneNumber  =  String
type  Flavor       =  String

data  Person  =  Person  FirstName  LastName  Age  Height
    PhoneNumber  Flavor
```

Proiecții

```
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

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Utilizare

```
Main*> let ionel = Person "Ion" "Ionescu" 20 175.2
    "0712334567" "Caramel"
Main*> firstName ionel
"Ion"
Main*> height ionel
175.2
Main*> flavor ionel
"Caramel"
```

Date personale ca înregistrări

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

Utilizare

- Putem folosi atât forma algebrică cât și cea de înregistrare

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

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

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

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

De ce algebric?