Laboratorul 11 - Programare Logică și Funcțională

Seria 36

Mai 2024

1 Exerciţiul 1

Scrieţi o funcţie `natOrdered :: [Int] -> Bool` care verifică dacă o listă de elemente este ordonata crescător şi întoarce `True` dacă da, respectiv `False` în caz contrar. Implementaţi funcţia recursiv, apoi prin `foldr`, utilizând proprietatea de universalitate.

Implementaţi funcţia în caz general, cu relaţia de ordine primită ca argument: `ordered :: [a] -> (a -> a -> Bool) -> Bool`. Primul argument este lista de elemente, al doilea este un comparator pentru tipul de date. Puteţi implementa utilizând `foldr`?

2 Exerciţiul 2

Fie următoarele tipuri de date algebrice, care reprezintă matrici cu linii de lungime diferită.

```
data Line = L [Int]
data Matrix = M [Line]
```

a. Scrieţi o funcţie `linesN :: Matrix -> Int -> [Line]` care primeşte o matrice, un număr întreg `n` şi returnează o listă de linii de lungime `n`.

```
> linesN (M [L [1,2,3], L [4,5], L [2,3,6,8], L [8, 5, 3]]) 3
[L [1,2,3], L [8, 5, 3]]
```

b. Scrieţi o funcţie `onlyPosElems :: Matrix -> Int -> Bool` care verifică dacă toate liniile de lungime `n` au elemente strict pozitive.

```
> onlyPosElems (M [L [1, 2, 3], L [4, 5], L [2, 3, 6, 8], L [8, 5, 3]]) 3
True

> onlyPosElems (M [L[1, 2, 3], L [4, 5], L [2, 3, 6, 8], L [8, 5, 3]]) 3
False
```

3 Exerciţiul 3

Fie monoidul `(IO(), (>>), done)`, unde

```
(>>) :: IO () -> IO () -> IO ()
done :: IO ()
```

(dacă `done` nu funcţionează, îl puteţi înlocui cu `return ()`)

Structura de mai sus este monoid, deoarece respectă:

```
m >> done = m
done >> m = m
(m >> n) >> o = m >> (n >> o)
```

(este asociativă şi admite `done` drept element neutru).

Prin intermediul acestor funcţii, putem executa operaţii la STDIN, respectiv STDOUT.

De exemplu, fie funcţia predefinită

```
putChar :: Char -> IO ()
```

care primeşte un caracter şi îl afişează la *standard output*.

Putem defini extinderea naturală `putStr'` (pentru a nu ambiguiza) astfel:

```
putStr' :: String -> IO ()
putStr' [] = done
putStr' (x:xs) = putChar x >> putStr' xs
```

Tipul `Maybe` este un tip de date cu doi constructori, `Nothing` şi `Just`.

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

Exerciţiu. Scrieţi o funcţie `putStrLnVowel` care să afişeze, după fiecare vocală, litera **p** şi vocala identificată. De exemplu, dacă intrarea este *haskell*, se va afisa la stdout *hapaskepell*. Utilizaţi şi tipul Maybe.

```
> putStrLnVowel "haskell"
"hapaskepell"

> putStrLnVowel "plf"
"plf"
```

4 Exerciţiul 4

Fie următoarele tipuri de date:

```
type Name = String

data Prop
   = Var Name
   | F
   | T
   | Not Prop
   | Prop :|: Prop
   | Prop :&: Prop
   deriving Eq
infixr 2 :|:
infixr 3 :&:
```

Tipul `Prop` este o reprezentare a formulelor propoziţionale. Variabilele precum `p` şi `q` pot fi reprezentate ca `Var "p"`, respectiv `Var "q"`. În plus, constantele booleene `F` şi `T` reprezintă falsul şi adevărul din logică, operatorul unar `Not` este negaţia, iar operatorii (infix) binari `:|:` şi `:&:` reprezintă disjuncţia, respectiv conjuncţia.

```
p1 :: Prop
p1 = (Var "P" :|: Var "Q") :&: (Var "P" :&: Var "Q")

p2 :: Prop
p2 = (Var "P" :|: Var "Q") :&: (Not (Var "P") :&: Not (Var "Q"))
```

Subpunctul a. Scrieţi implementarea formulei $p_3 := (P \wedge (Q \vee R)) \wedge ((\neg P \vee \neg Q) \wedge (\neg P \vee \neg R))$

```
p3 :: Prop
p3 = undefined
```

Subpunctul b. Faceţi tipul `Prop` instanţă a clasei de tipuri `Show`, înlocuind `Not`, `:|:` şi `:&:` cu , `|`, `&` şi folosind direct numele variabilelor în loc de construcţia `Var "nume"`.

```
instance Show Prop where
    show = undefined
```

Subpunctul c. Evaluarea expresiilor. Pentru a evalua o expresie, vom considera un *environment* de evaluare, care asociază valori `Bool` variabilelor propoziţionale.

```haskell
type Env = [(Name, Bool)]
```

Pentru a obţine valoarea asociată unui `Name` în `Env`, utilizaţi funcţia predefinită

```haskell
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Pentru a simplifica, mergând pe ipoteza că mereu vom găsi numele căutat, scrieţi o variantă a funcţiei `lookup`, care să nu mai folosească tipul `Maybe`.

```haskell
impureLookup :: Eq a => a -> [(a, b)] -> b
impureLookup = undefined
```

Ţineţi cont că, pentru a extrage informaţia `Just` dintr-un `Maybe`, putem utiliza `fromJust`.

```haskell
fromJust :: Maybe a -> a
```

Exerciţiu. Definiţi funcţia `eval` cu signatura:

```haskell
eval :: Prop -> Env -> Bool
eval = undefined
```

Evaluaţi cele trei formule definite în diferite evaluări.

5 Exerciţiul 5

Fie următoarele tipuri de date reprezentând expresii şi arbori de expresii.

```haskell
data Expr = Const Int
          | Expr :+: Expr
          | Expr :*: Expr
          deriving Eq

data Operation = Add | Mult deriving (Eq, Show)

data Tree = Lf Int
          | Node Operation Tree Tree
          deriving (Eq, Show)
```

- Instanţiaţi clasa `Show` pentru tipul `Expr`, astfel încât să se afişeze mai simplu expresiile.
- Scrieţi o funcţie `evalExp :: Expr -> Int` care să evalueze o expresie, determinând valoarea acesteia.
- Scrieţi o funcţie `evalArb :: Tree -> Int` care evaluează o expresie, determinând valoarea acesteia.
- Scrieţi o funcţie `expToArb :: Expr -> Tree` care transformă o expresie în arborele corespunzător.