Laboratorul 6: Tipuri de date

I Încălzire

Exercițiul 1

Vom începe prin a scrie câteva funcții definite folosind tipul de date `Fruct`:

```
data Fruct
  = Mar String Bool
  | Portocala String Int
```

O expresie de tipul `Fruct` este fie un `Mar String Bool` sau o `Portocala String Int`. Vom folosi un `String` pentru a indica soiul de mere sau portocale, un `Bool` pentru a indica dacă mărul are viermi și un `Int` pentru a exprima numărul de felii dintr-o portocală. De exemplu:

```
ionatanFaraVierme = Mar "Ionatan" False
goldenCuVierme = Mar "Golden Delicious" True
portocalaSicilia10 = Portocala "Sanguinello" 10
listaFructe = [Mar "Ionatan" False,
               Portocala "Sanguinello" 10,
               Portocala "Valencia" 22,
               Mar "Golden Delicious" True,
               Portocala "Sanguinello" 15,
               Portocala "Moro" 12,
               Portocala "Tarocco" 3,
               Portocala "Moro" 12,
               Portocala "Valencia" 2,
               Mar "Golden Delicious" False,
               Mar "Golden" False,
               Mar "Golden" True]
```

a) Scrieți o funcție

```
ePortocalaDeSicilia :: Fruct -> Bool
ePortocalaDeSicilia = undefined
```

care indică dacă un fruct este o portocală de Sicilia sau nu. Soiurile de portocale din Sicilia sunt `Tarocco`, `Moro` și `Sanguinello`. De exemplu,

```
test_ePortocalaDeSicilia1 =
    ePortocalaDeSicilia (Portocala "Moro" 12) == True
test_ePortocalaDeSicilia2 =
    ePortocalaDeSicilia (Mar "Ionatan" True) == False
```

b) Scrieți o funcție

```
nrFeliiSicilia :: [Fruct] -> Int
nrFeliiSicilia = undefined

test_nrFeliiSicilia = nrFeliiSicilia listaFructe == 52
```

care calculează numărul total de felii ale portocalelor de Sicilia dintr-o listă de fructe.

c) Scrieți o funcție

```
nrMereViermi :: [Fruct] -> Int
nrMereViermi = undefined

test_nrMereViermi = nrMereViermi listaFructe == 2
```

care calcuelază numărul de mere care au viermi dintr-o lista de fructe.

Exercițiul 2

```
type NumeA = String
type Rasa = String
data Animal = Pisica NumeA | Caine Nume Rasa
```

a) Scrieți o funcție

```
vorbeste :: Animal -> String
vorbeste = undefined
```

care întoarce `"Meow!"` pentru pisică și `"Woof!"` pentru câine.

b) Vă reamintiți tipul de date predefinit `Maybe`

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

scrieți o funcție

```
rasa :: Animal -> Maybe String
rasa = undefined
```

care întoarce rasa unui câine dat ca parametru sau `Nothing` dacă parametrul este o pisică.

II Logică propozițională

În restul acestui laborator vom implementa funcții pentru a lucra cu logică propozițională în Haskell. Fie dată următoarea definiție:

```haskell
type Nume = String
data Prop
  = Var Nume
  | F
  | T
  | Not Prop
  | Prop :|: Prop
  | Prop :&: Prop
  deriving (Eq, Read)
infixr 2 :|:
infixr 3 :&:
```

Tipul `Prop` este o reprezentare a formulelor propoziționale. Variabilele propoziționale, precum `p` și `q` pot fi reprezentate va `Var "p"` și `Var "q"`. În plus, constantele booleene `F` și `T` reprezintă `false` și `true`, operatorul unar `Not` reprezintă negația ($\neg$; a nu se confunda cu funcția `not :: Bool -> Bool`) și operatorii (infix) binari `:|:` și `:&:`reprezintă disjuncția ($\vee$) și conjuncția ($\wedge$).

Exercițiul 1

Scrieți următoarele formule ca expresii de tip `Prop`, denumindu-le `p1`, `p2`, `p3`.

1. $(P \vee Q) \wedge (P \wedge Q)$

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

2. $(P \vee Q) \wedge (\neg P \wedge \neg Q)$

```haskell
p2 :: Prop
p2 = undefined
```

3. $(P \wedge (Q \vee R)) \wedge ((\neg P \vee \neg Q) \wedge (\neg P \vee \neg R))$

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

Exercițiul 2

Faceți tipul `Prop` instanță a clasei de tipuri `Show`, înlocuind conectivele `Not`, `:|:` și `:&:` cu `~`, `|` și `&` și folosind direct numele variabilelor în loc de construcția `Var nume`.

```haskell
instance Show Prop where
  show = undefined

test_ShowProp :: Bool
test_ShowProp =
    show (Not (Var "P") :&: Var "Q") == "((~P)&Q)"
```

Exercițiul 2' (opțional)

Schimbați definiția lui `show` astfel încât parantezele să fie puse doar atunci când sunt strict necesare. Pentru aceasta, obsevați că o subexpresie a unui operator trebuie pusă în paranteze doar dacă precedența sa este mai mică decât cea a operatorului. Asfel, întrucât precedența lui `Not` este cea a aplicației iar precedențele lui `:&:` și `:|:` sunt 3 și respectiv 2: 1. Expresia de sub `Not` trebuie pusă în paranteze doar dacă are la vârf `:|:` sau `:&:` 2. O subexpresie a lui `:&:` trebuie pusă în paranteze doar dacă are la vârf `:|:`

Evaluarea expresiilor logice

Pentru a putea evalua o expresie logică vom considera un mediu de evaluare care asociază valori `Bool` variabilelor propoziționale:

```
type Env = [(Nume, Bool)]
```

Tipul `Env` este o listă de atribuiri de valori de adevăr pentru (numele) variabilelor propoziționale.

Pentru a obține valoarea asociată unui `Nume` în `Env`, putem folosi funcția predefinită `lookup :: Eq a => a -> [(a,b)] -> Maybe b`.

Deși nu foarte elegant, pentru a simplifica exercițiile de mai jos, vom definit o variantă a funcției `lookup` care generează o eroare dacă valoarea nu este găsită.

```
impureLookup :: Eq a => a -> [(a,b)] -> b
impureLookup a = fromJust . lookup a
```

O soluție mai elegantă ar fi să reprezentăm toate funcțiile ca fiind parțiale (rezultat de tip `Maybe`) și să folosim faptul că `Maybe` este monadă.

Exercițiul 3

Definiți o funcție `eval` care dat fiind o expresie logică și un mediu de evaluare, calculează valoarea de adevăr a expresiei.

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

test_eval = eval  (Var "P" :|: Var "Q") [("P", True), ("Q", False)] == True
```

Satisfiabilitate

O formulă în logica propozițională este *satisfiabilă* dacă există o atribuire de valori de adevăr pentru variabilele propoziționale din formulă pentru care aceasta se evaluează la `True`.

Pentru a verifica dacă o formulă este satisfiabilă vom genera toate atribuirile posibile de valori de adevăr și vom testa dacă formula se evaluează la `True` pentru vreuna dintre ele.

Exercițiul 4

Definiți o funcție `variabile` care colectează lista tuturor variabilelor dintr-o formulă. *Indicație*: folosiți funcția `nub`.

```
variabile :: Prop -> [Nume]
variabile = undefined

test_variabile =
  variabile (Not (Var "P") :&: Var "Q") == ["P", "Q"]
```

Exercițiul 5

Dată fiind o listă de nume, definiți toate atribuirile de valori de adevăr posibile pentru ea.

```
envs :: [Nume] -> [Env]
envs = undefined

test_envs =
    envs ["P", "Q"]
    ==
    [ [ ("P",False)
      , ("Q",False)
      ]
    , [ ("P",False)
      , ("Q",True)
      ]
    , [ ("P",True)
      , ("Q",False)
      ]
    , [ ("P",True)
      , ("Q",True)
      ]
    ]
```

Exercițiul 6

Definiți o funcție `satisfiabila` care dată fiind o Propoziție verifică dacă aceasta este satisfiabilă. Puteți folosi rezultatele de la exercițiile 4 și 5.

```
satisfiabila :: Prop -> Bool
satisfiabila = undefined
```

```
test_satisfiabila1 = satisfiabila (Not (Var "P") :&: Var "Q") == True
test_satisfiabila2 = satisfiabila (Not (Var "P") :&: Var "P") == False
```

Exercițiul 7

O propoziție este validă dacă se evaluează la `True` pentru orice interpretare a varibilelor. O forumare echivalenta este aceea că o propoziție este validă dacă negația ei este nesatisfiabilă. Definiți o funcție `valida` care verifică dacă o propoziție este validă.

```
valida :: Prop -> Bool
valida = undefined

test_valida1 = valida (Not (Var "P") :&: Var "Q") == False
test_valida2 = valida (Not (Var "P") :|: Var "P") == True
```

Implicație și echivalență

Exercițiul 8

Extindeți tipul de date `Prop` și funcțiile definite până acum pentru a include conectivele logice `->` (implicația) și `<->` (echivalența), folosind constructorii `:->:` și `:<->:`.