

Laboratorul Monade

I. Monade. Monada Maybe

Lucrați în fișierul `mMaybe.hs`, care conține definiția monadei `Maybe`. Definiția este comentată deoarece monada `Maybe` este definită în `GHC.Base`

0. Înțelegeți funcționarea operațiilor monadice (`>=`) și `return`

```
return 3 :: Maybe Int
Just 3
(Just 3) >= (\ x -> if (x>0) then Just (x*x) else Nothing)
Just 9
```

Uneori vom folosi operația derivată (`>>`)

```
ma >> mb = ma >= \_ -> mb
```

```
(Just 3) >> Nothing
Nothing
(Just 3) >> (Just 6)
Just 6
```

1. Definiți operatorul de compunere a funcțiilor îmbogățite

```
(<=<) :: (a -> Maybe b) -> (c -> Maybe a) -> c -> Maybe b
f <=< g = (\ x -> g x >= f)
```

1.1 Creați singuri exemple prin care să înțelegeți funcționarea acestui operator.

1.2 Definiți proprietatea

```
asoc :: (Int -> Maybe Int) -> (Int -> Maybe Int) -> (Int -> Maybe Int) -> Int -> Bool
```

care pentru trei funcții date verifică asociativitatea operației (`<=<`):

```
h <=< (g <=< f) $ x = (h <=< g) <=< f $ x
```

Verificați proprietatea pentru funcții particulare folosind `QuickCheck`.

2. Definim

```
pos :: Int -> Bool
pos x = if (x>=0) then True else False

foo :: Maybe Int -> Maybe Bool
foo mx = mx >= (\x -> Just (pos x))
```

2.1 Înțelegeți ce face funcția `foo`.

2.2 Definiți funcția `foo` folosind notația `do`.

3. Vrem să definim o funcție care adună două valori de tip `Maybe Int`

```
addM :: Maybe Int -> Maybe Int -> Maybe Int
addM mx my = undefined
```

Exemplu de funcționare:

```
addM (Just 4) (Just 3)
Just 7
addM (Just 4) Nothing
Nothing
addM Nothing Nothing
Nothing
```

3.1 Definiți `addM` prin orice metodă (de exemplu, folosind șabloane).

3.2 Definiți `addM` folosind operații monadice și notația `do`.

Notația `do` și secvențiere

4. Să se treacă în notația `do` următoarele funcții:

```
cartesian_product xs ys = xs >=> ( \x -> (ys >=> \y-> return (x,y)))

prod f xs ys = [f x y | x <- xs, y<-ys]

myGetLine :: IO String
myGetLine = getChar >=> \x ->
    if x == '\n' then
        return []
    else
        myGetLine >=> \xs -> return (x:xs)
```

5. Să se treacă în notația cu secvențiere următoarea funcție:

```
prelNo noin = sqrt noin
ioNumber = do
    noin <- readLn :: IO Float
    putStrLn $ "Intrare\n" ++ (show noin)
    let noout = prelNo noin
    putStrLn $ "Iesire"
    print noout
```

II. Monada `Writer log`

Pentru următoarele exerciții lucrați cu fișierul `mWriter.hs`.

1. Fișierul `mWriter.hs` conține o definiție a monadei `Writer String` (puțin modificată pentru a compila fără opțiuni suplimentare):

```
newtype WriterS a = Writer { runWriter :: (a, String) }
```

1.1 Definiți funcțiile `logIncrement` și `logIncrement2` din curs și testați funcționarea lor.

1.2 Definiți funcția `logIncrementN`, care generalizează `logIncrement2`, astfel:

```
logIncrementN :: Int -> Int -> WriterS Int
logIncrement x n = undefined
```

Exemplu de funcționare:

```
runWriter $ logIncrementN 2 4
(6,"increment:2\nincrement:3\nincrement:4\nincrement:5\n")
```

2 Modificați definiția monadei `WriterS` astfel încât să producă lista mesajelor logate și nu concatenarea lor. Pentru a evita posibile confuzii, lucrați în fișierul `mWriterL.hs`. Definiți funcția `logIncrementN` în acest context.

```
newtype WriterLS a = Writer {runWriter :: (a, [String])}
```

Exemplu de funcționare:

```
runWriter $ logIncrementN 2 4
(6,["increment:2","increment:3","increment:4","increment:5"])
```

Funcția map în context monadic

Vom lucra tot în fișierul mWriterL.hs

3. În mod uzual, primul argument al funcției `map` este o funcție `f :: a -> b`, de exemplu:

```
map (\x -> if (x>=0) then True else False) [1,-2,3]
[True,False,True]
```

În context monadic, funcția `f` este îmbogățită, adică întoarce o valoare monadică:

```
isPos :: Int -> WriterLS Bool
isPos x = if (x>= 0) then (Writer (True, ["poz"])) else (Writer (False, ["neg"]))
```

Ce se întâmplă când aplicăm `map`?

```
map isPos [1,2,3]
```

Obțineți un mesaj de eroare! Funcția `map` a întors o listă de rezultate monadice, care pot fi vizualizate astfel:

```
map runWriter $ map isPos [1,-2,3]
[(True,["poz"]), (False,["neg"]), (True,["poz"])]
```

Problemă: cum procedăm dacă dorim ca efectele să fie înlănțuite, iar rezultatul final să fie lista rezultatelor?

4. Definiți o funcție care se comportă similar cu `map`, dar efectul final este înlănțuirea efectelor. Signatura acestei funcții este:

```
mapWriterLS :: (a -> WriterLS b) -> [a] -> WriterLS [b]
```

Exemplu de funcționare:

```
runWriter $ mapWriterLS isPos [1,-2,3]
[(True,False,True),["poz","neg","poz"]]
```

5. Definiți funcții asemănătoare cu `mapWriterLS` pentru monadele `WriterS` și `Maybe` din exercițiile anterioare.

III. Monada Reader

În continuare, vom exersa monada `Reader`.

Monada `Reader` este definită în fișierul mReader.hs

Citiți și înțelegeți exemplul de la curs.

1. Definim tipul de date

```
data Person = Person { name :: String, age :: Int }
```

1.1 Definiți funcțiile

```
showPersonN :: Person -> String
showPersonA :: Person -> String
```

care afișează “frumos” numele și vârsta unei persoane, după modelul

```
showPersonN $ Person "ada" 20
"NAME:ada"
```

```
showPersonA $ Person "ada" 20
"AGE:20"
```

1.2 Combinând funcțiile definite la punctul 1.1, definiți funcția

```
showPerson :: Person -> String
```

care afișează “frumos” toate datele unei persoane, după modelul

```
showPerson $ Person "ada" 20
"(NAME:ada,AGE:20)"
```

1.3 Folosind monada `Reader`, definiți variante monadice pentru cele trei funcții definite anterior. Variantele monadice vor avea tipul

```
mshowPersonN :: Reader Person String
mshowPersonA :: Reader Person String
mshowPerson  :: Reader Person String
```

Exemplu de funcționare:

```
runReader mshowPersonN $ Person "ada" 20
"NAME:ada"

runReader mshowPersonA $ Person "ada" 20
"AGE:20"

runReader mshowPerson  $ Person "ada" 20
"(NAME:ada,AGE:20)"
```