Exemplu subiect Programare declarativa 2019

Exercițiu rezolvat

- definirea unui interpretor folosind o combinație

  între monadele `Writer` și `Maybe`

Definirea unui interpretor

Limbajul

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
          | Out Term
  deriving (Show)
```

Exemplu de program

```
pgm :: Term

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Out (Con 10)) :+:  (Out (Con 11)))
```

Monada `Writer`

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

instance  Monad Writer  where
  return a = Writer (a, "")
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
```

```
              in  Writer (b, log1 ++ log2)

instance Applicative Writer  where
  pure = return
  mf <*> ma = do { f <- mf; a <- ma; return (f a) }

instance Functor Writer  where
fmap f ma = pure f <*> ma
```

Interpretorul folosind monada `Writer`

```
type M a = Writer a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ "\nValue: " ++ show a
  where (a, w) = runWriter ma

data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

type Environment = [(Name, Value)]

instance Show Value where
  show (Num x) = show x
  show (Fun _) = "<function>"
  show Wrong   = "<wrong>"


interp :: Term -> Environment -> M Value
interp (Var x) env = get x env
interp (Con i) _ = return $ Num i
interp (t1 :+: t2) env = do
  v1 <- interp t1 env
  v2 <- interp t2 env
  add v1 v2

get :: Name -> Environment -> M Value
get x env = case [v | (y,v) <- env , x == y] of
  (v:_) -> return v
  _     -> return Wrong

add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = return Wrong
```

```haskell
interp (Lam x e) env =
  return $ Fun $ \ v -> interp e ((x,v):env)
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
  apply f v

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = return Wrong



interp (Out t) env = do
  v <- interp t env
  tell (show v ++ "; ")
  return v

tell :: log -> Writer log ()
tell log = Writer ((), log)
```

Exemplu

```haskell
test :: Term -> String
test t = showM $ interp t []

pgm, pgmW :: Term
pgm = App
    (Lam "x" ((Var "x") :+: (Var "x")))
    ((Out (Con 10)) :+:  (Out (Con 11)))

> test pgm
"Output: 10; 11; \nValue: 42"

pgmW4 = App (Var "y") (Lam "y" (Out (Con 3)))

> test pgmW
"Output: \nValue: <wrong>"
```

Problemă

În continuare vom modifica programul astfel încât, în cazul apariției unei erori, se va întoarce rezultatul `Nothing` fără a afișa output-ul acumulat până în acel moment.

Pentru aceasta vom înlocui monada `Writer` cu o nouă monadă care combină `Writer` cu `Maybe`.

Definim

```
newtype MaybeWriter a = MW {getvalue :: Maybe (a,String)}
```

Exercițiul 1

- Faceți `MaybeWriter` instanță a clasei `Monad`, astfel încât cazurile de eroare să întoarcă numai valoarea `Nothing`, ignorand output-ul acumulat.

```
newtype MaybeWriter a = MW {getvalue :: Maybe (a,String)}

instance  Monad (MaybeWriter ) where
  return x = MW $ Just (x, "")
  ma >>= f =
      case a of
        Nothing -> MW Nothing
        Just (x,w) ->
            case getvalue (f x) of
              Nothing -> MW Nothing
              Just (y,v) -> MW $ Just (y, w++v)
    where a = getvalue ma
```

Exercițiul 2

- În interpretor modificăm următoarele definiții:

```
type M a = MaybeWriter a

data Value = Num Integer
           | Fun (Value -> M Value)
```

Precizați ce modificări trebuie făcute pentru a obține un interpretor cu valori în `MaybeWriter` astfel încât toate cazurile de eroare să întoarcă `Nothing`.

Exercițiul 2

```
showM :: Show a => M a -> String
showM ma =
    case a of
      Nothing -> "Nothing"
      Just (x,w) ->
          "Output: " ++ w ++ "\nValue: " ++ show x
  where a = getvalue ma
```

Exercițiul 2

```
get :: Name -> Environment -> M Value
get x env =
    case [v | (y,v) <- env , x == y] of
      (v:_) -> return v
      _     -> MW Nothing

add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = MW Nothing

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = MW Nothing
```

Exercițiul 2

```
tellMW :: String -> MaybeWriter ()
tellMW  ceva = MW $ Just ( ( ) , ceva)


interp (Out t) env
  = do
    v <- interp t env
    tellMW (show v ++ "; ")
    return v
```

Exemplu

```
pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Out (Con 10)) :+:  (Out (Con 11)))
> test pgm
"Output: 10; 11; \nValue: 42"


pgmW = App (Lam "y" (Out (Con 3))) (Var "y")

> test pgmW
"Nothing"
```

Exercițiul 3

Modificăm tipul de date `Term` prin adăugarea operației `:/:` care va fi interpretată ca `div`.

```
data Term = ... | Term :/: Term
```

În modulul definit la Exercițiul 2, în care interpretarea termenilor se face în monada `MaybeWriter Value`, completati definiția funcției `interp` adăugând semantica operației `:/:`, considerând ca eroare împărțirea la `0`.

Exercițiul 3

```
data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Term :/: Term
          | Lam Name Term
          | App Term Term
          | Out Term
  deriving (Show)
```

Exercițiul 3

```
interp (t1 :/: t2) env
  = do
    v1 <- interp t1 env
    v2 <- interp t2 env
    imparte v1 v2

imparte :: Value -> Value -> M Value
imparte (Num i) (Num j)
  | j == 0    = MW Nothing
  | otherwise = return (Num $ i `div` j)
imparte _ _ = MW Nothing
```

Exemplu

```
pgm2 = App
    (Lam "x" ((Var "x") :+: (Var "x")))
    ((Con 10) :/:  (Out (Con 2)))


> test pgm2
"Output: 2; \nValue: 10"

pgmW2 = App
    (Lam "x" ((Var "x") :+: (Var "x")))
    ((Con 10) :/:  (Out (Con 0)))

> test pgmW2
"Nothing"
```

Succes la examen!