Laboratorul 13

Acest laborator are două părți. În prima parte vom relua interpretarea monadică a programelor, variind interpretarea prin folosirea tuturor monadelor standard studiate. În a doua parte vom folosi monada `State` pentru a defini propria monadă `IO`.

I. Interpretarea monadică a programelor

În continuare vom începe să explorăm folosirea monadelor pentru structurarea programelor funcționale.

Vom începe cu un interpretor simplu pentru lambda calcul, o variantă simplificată a interpretorului MicroHaskell din cursul 7 și laboratorul 10, și continuare a laboratorului 12.

Sintaxa abstractă

Un termen este o variabilă, o constantă, o sumă, o funcție anonimă sau o aplicare de funcție.

```
type Name = String

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

Valori

O valoare este `Wrong`, un număr sau o funcție. Valoarea `Wrong` indică o eroare precum o variabilă nedefinită, încercarea de a aduna valori ne-numerice, sau încercarea de a aplica o valoare non-funcțională.

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

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

Variații monadice

Pentru fiecare din exercițiile (variațiile) de mai jos copiați fișierul `var0Identity.hs` ca `varM.hs` și modificați-l conform cerințelor exercițiului.

Exercițiu: evaluare parțială (variația 1)

În loc de a folosi `Wrong` pentru a înregistra evaluările eșuate, definiți `M` ca fiind `Maybe` și folosiți `Nothing` pentru evaluările care eșuează.

Eliminați `Wrong` și toate aparițiile sale din definiția interpretorului.

Monada `Maybe` este predefinită așa că nu trebuie să o definiți în fișierul soluție.

Pentru verificare puteți consulta slide-urile 14–15 din cursul 11-12.

Exercițiu: mesaje de eroare (variația 2)

Pentru a îmbunătăți mesajele de eroare, folosiți monada (predefinită) `Either`.

Pentru a modifica interpretorul definiți `M` ca fiind `Either String` și înlocuiți fiecare apariție a lui `return Wrong` cu o expresie `Left` corespunzătoare.

posibile mesaje de eroare:

- unbound variable: < name >
- should be numbers: < v1 >, < v2 >
- should be function: < v1 >

Evaluarea `interp term0 []` ar trebui să fie `Right 42`; evaluarea `interp (App (Con 7) (Con 2)) []` ar trebui să fie `Left "should be function: 7"`.

Într-un limbaj impur această modificare ar fi putut fi făcută prin intermediul excepțiilor.

Pentru verificare puteți consulta slide-urile 16–18 din cursul 11-12.

Exercițiu: alegere nedeterministă (variația 3)

Vom modifica acum interpretorul pentru a modela un limbaj nedeterminist pentru care evaluarea întoarece lista răspunsurilor posibile.

Pentru a face acest lucru vom folosi monada (pedefinită) asociata tipului listă:

```haskell
type M a = [a]
```

Extindeți limbajul cu doi constructori noi de expresii: `Fail` și `Amb Term Term`.

Evaluarea lui `Fail` ar trebui să nu întoarcă nici o valoare, în timp ce evaluarea lui `Amb u v` ar trebui să întoarcă toate valorile întoarse de `u` sau de `v`,

Extindeți `interp` pentru a obține această semantică.

De exemplu, evaluarea lui `interp (App (Lam "x" (Var "x" :+: Var "x")) (Amb (Con 1) (Con 2))) []` ar trebui să fie `[2,4]`.

Această schimbare este mai greu de gândit într-un limbaj impur.

Pentru verificare puteți consulta slide-urile 19-20 din cursul 11-12.

Exercițiu: Afișare de rezultate intermediare (variația 4)

În acest exercițiu vom modifica interpretorul pentru a afișa.

Am putea folosi monada `Stare`, dar nu este cea mai bună alegere, deoarece acumularea rezultatelor într-o stare finală implică că starea nu va putea fi afișată până la sfârșitul computației.

În loc de asta, vom folosi monada `Writer`. Pentru a nu complica prezentarea vom instanția tipul canalului de ieșire la `String`.

```
newtype StringWriter a = StringWriter { runStringWriter :: (a, String) }
```

- Faceți `StringWriter` instanță a clasei `Show` astfel încât să afișeze șirul de ieșire, urmat de valoarea rezultat.
- Faceți `StringWriter` instanța a clasei `Monad`. Monada `StringWriter` se comportă astfel:
  - Fiecare valoare este împerecheată cu șirul de ieșire produs în timpul calculării acelei valori
  - Funcția `return` întoarce valoarea dată și nu produce nimic la ieșire.
  - Funcția `>>=` efectuează o aplicație și concatenează ieșirea primului argument și ieșirea produsă de aplicație.
- Definiți o funcție `tell :: String -> StringWriter ()` care afișează valoarea dată ca argument.
- Extindeți limbajul cu o operație de afișare, adăugând termenul `Out Term`.

  Evaluarea lui `Out u` afișează valoarea lui `u`, urmată de `;` și întoarce acea valoare.

  De exemplu `interp (Out (Con 41) :+: Out (Con 1)) []` ar trebui să afișeze `"Output: 41; 1; Value: 42"`.

Într-un limbaj impur, această modificare ar putea fi făcuta folosind afișarea ca efect lateral.

Pentru verificare puteți consulta slide-urile 21–23 din cursul 11-12.

Exercițiu: Stare (variația 5)

Pentru a ilustra manipularea stării, vom modifica interpretorul pentru a calcula numărul de pași necesari pentru calcularea rezultatului.

Aceeași tehnică poate fi folosită pentru a da semantică și altor construcții care necesită stare precum pointeri și heap.

Monada transformărilor de stare este monada `State`.

O transformare de stare este o funcție care ia o stare inițială și întoarce o pereche dintre o valoare și starea cea nouă. Pentru a nu complica lucrurile, vom instanția starea la tipul `Integer` necesar în acest exercițiu:

```
newtype IntState a = IntState { runIntState :: Integer -> (a, Integer) }
```

- Faceți `IntState a` instanță a clasei `Show` afișând valoarea și starea finală obținute prin execuția transformării de stare în starea inițială `0`.
- Faceți `IntState` instanță a clasei `Monad`

  Funcția `return` întoarce valoarea dată și propagă starea neschimbată.

  Funcția »= ia o transformare de stare ma :: IntState a și o funcție (continuare) k :: a -> IntState b. Rezultatul ei încapsulează o transformare de stare care:

  - trimite starea inițială transformării de stare ma; obține astfel o valoare și o stare intermediară.
  - aplica funcția `k` valorii, obținând o nouă transformare de stare;
  - această nouă transformare de stare primește ca stare inițială starea intermediară obținută în urma evaluării lui `ma`; aceasta întoarce rezultatul și starea finală.

Evaluarea lui `interp term0 []` ar trebui să întoarcă `"Value: 42; Count: 3"`.

Pentru a obține acest lucru,

- definiți funcția `modify :: (Integer -> Integer) -> IntState ()` care modifică starea internă a monadei conform funcției date ca argument.
- definiți o computație care crește contorul: `tickS :: IntState ()` și modificați evaluările adunării și aplicației folosind `tickS` pentru a crește contorul pentru fiecare apel al lor.

Putem extinde limbajul pentru a permite accesul la valoarea curentă a contorului de execuție.

- Definiți computația `get :: IntState Integer` care obține ca valoare valoarea curentă a contorului.
- Extindeți tipul `Term` cu un nou constructor `Count`.

- Definiți `interp` pentru `Count` cu semantica de a obține numărul de pași executați până acum și a îl întoarce ca valoarea `Num` corespunzătoare termenului.

De exemplu, `interp ((Con 1 :+: Con 2) :+: Count) []` ar trebui să afișeze `"Value: 4; Count: 2"`, deoarece doar o adunare are loc înainte de evaluarea lui `Count`.

Într-un limbaj impur, aceste modificări ar putea fi făcute folosind o variabilă globală / locație de memorie pentru a ține contorul.

Pentru verificare puteți consulta slide-urile 29–32 din cursul 11-12.

II. Definirea monadei `IO` folosind monada `State`

Înainte de a rezolva acest exercițiu ar fi util sa va reamintiți modul în care am definit propria monada IO în cursul 10:

- ne-am definit propria monadă `MyIO`

```
type Input = String
type Output = String

newtype MyIO a =
        MyIO { runMyIO :: Input -> (a, Input, Output)}

instance Monad MyIO where
   return x = MyIO (\input -> (x, input, ""))
   m >>= k  = MyIO f
            where f input =
              let (x, inputx, outputx) = runMyIO m input
                  (y, inputy, outputy) = runMyIO (k x) inputx
              in  (y, inputy, outputx ++ outputy)

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

instance Functor MyIO where
   fmap f ma = do { a <- ma; return (f a) }
```

- am definit operațiile de bază

```
myPutChar :: Char -> MyIO ()
myPutChar c = MyIO (\input -> ((), input, [c]))

myGetChar :: MyIO Char
myGetChar = MyIO (\ (c:input) -> (c, input, ""))
```

```
runIO :: MyIO () -> String -> String
runIO command input = third (runMyIO command input)
                      where third (_, _, x) = x
```

- folosind operațiile de bază am definit mai multe operații derivate: `myPutStr`, `myPutStrLn`, `myGetLine`, `echo`
- am făcut legătura cu monada `IO` folosind funcția

```
convert :: MyIO () -> IO ()
convert = interact . runIO
```

Exercițiu: `MyIOState`

Definiți propria monada `IO` folosind monada `State`, plecând de la următoarele definiții:

```
type InOutWorld = (Input, Output)
type MyIOState = State InOutWorld
```

Atenție! acestă definiție trebuie scrisă într-un fișier care conține definiția monadei `State`.

- înțelegeți cum arată o valoare de tipul `MyIOState`
- definiți operațiile de bază `myGetChar` , `myPutChar` și `runIO`
- definiți operații derivate utile
- testați definițiile (puteți folosi exemplele din cursul 10)