Laboratorul 12: Monada Reader. Interpretorul monadic de baza

Pentru început, vom exersa monada `Reader`, introdusă în cursul 10.

Monada `Reader` este definită în fișierul mReader.hs

Exercițiul 0

Citiți și înțelegeți exemplul de la curs.

Exercițiul 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

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

Exemplu de funcționare:

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

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

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

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.

Sintaxa abstractă

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

```haskell
type Name = String

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

Limbajul este mic în scop ilustrativ. Poate fi extins cu ușurință cu mai multe valori (precum booleeni, perechi și liste) și mai multe feluri de expresii, precum expresii condiționale și operator de punct fix (recursie).

Vom folosi următoarea expresie ca test:

```haskell
term0 = (App (Lam "x" (Var "x" :+: Var "x")) (Con 10 :+: Con 11))
```

În notația convențională (Haskell) acesta ar fi scris ca `((\ x -> x + x) (10 + 11))`

Valoarea corespunzătoare evaluării termenului `term0` este `42`.

Ca parte a acestui laborator, un interpretor simplu este modificat pentru a oferi suport pentru diferite efecte laterale definite cu ajutorul monadelor.

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

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

Vom lucra cu o monadă definită generic `M` pe care o vom instanția pe rând cu diverse monade pentru a obține efectul dorit.

În acest laborator vom înlocui M cu monada Identity și cu monada Reader, urmând ca în laboratoarele viitoare să folosim și alte monade.

Pentru început lucrați în fișierul var0Identity.hs

Exercițiul 2: Monada `Identity`

Vom începe cu monada trivială, care nu are nici un efect lateral.

```
newtype Identity a = Identity { runIdentity :: a }
```

- Faceți `Identity` instanță a clasei `Show`.

  `show` extrage valoarea și o afișează.

- Faceți `Identity` instanță a clasei `Monad`

  `Identity` încapsulează funcția identitate pe tipuri, `return` fiind funcția identitate, iar `>>=` este operatorul de aplicare in formă postfixată.

Exercițiul 3: Interpretorul monadic general

Ideea de bază pentru a converti un program în forma sa monadică este următoarea: o funcție de tipul `a -> b` este convertită la una de tipul `a -> M b`.

De aceea, in definiția tipului `Value`, funcțiile au tipul `Value -> M Value` în loc de `Value -> Value`, și deci și funcția interpretor va avea tipul `Term -> Environment -> M Value`.

Așa cum tipul `Value` reprezintă o valoare, tipul `M Value` poate fi gândit ca o computație care produce o valoare și un efect.

Cerință: Având drept inspirație interpretorul pentru MicroHaskell definit în cursul 7 și laboratorul 10, definiți un interpretor monadic pentru limbajul de mai sus.

```
type Environment = [(Name, Value)]

interp :: Term -> Environment -> M Value
```

Funcția identitate are tipul `a -> a`. Funcția corespunzătoare în formă monadică este `return`, care are tipul `a -> M a`. `return` transformă o valoare în reprezentarea corespunzătoare ei din monadă.

De exemplu definiția lui `interp` pentru constante este:

```
interp (Con i) _ = return (Num i)
```

Expresia `(Num i)` are tipul `Value`, deci aplicându-i `return` obținem o valoare de tip `M Value` corespunzătoare tipului rezultat al lui `interp`.

Pentru cazurile mai interesante vom folosi notația `do`. De exemplu, cazul pentru operatorul de adunare:

```
interp (t1 :+: t2) env = do
   v1 <- interp t1 env
   v2 <- interp t2 env
   add v1 v2
```

Acesta poate fi citit astfel: evaluează `t1`, pune rezultatul în `v1`; evaluează `t2`, pune rezultatul în `v2`; aduna `v1` cu `v2`.

Pentru verificare puteți consulta slide-urile 8–10 din cursul 8.

Pentru a putea testa interpretorul, definiți `M` ca fiind `Identity`:

```
type M = Identity
```

Pentru această variantă a interpretorului, evaluând (și afișând) `interp term0 []` vom obține `"42"` așa cum era de așteptat.

Exercițiul 4: Interpretare în monada `Reader`

În continuare, copiați conținutul fișierului `var0Identity.hs` în alt fișier, numit `var1Reader.hs` și modificați conform cerințelor.

Putem gândi mediul de evaluare (`Environment`) ca o stare care este citită atunci când avem nevoie de valorile variabilelor, dar nu este modificată.

Monada stărilor imuabile este monada `Reader`.

Spre deosebire de transformarea de stare, în acest caz starea nu se modifică; deci nu mai avem nevoie de valoarea ei după execuția computației. Așadar, computația cu stare imuabilă va fi reprezentată de o funcție care dată fiind o stare produce o valoare corespunzătoare acelei stări.

Pentru a nu complica lucrurile, vom instanția monada `Reader` pentru tipul `Environment` al mediilor de evaluare:

```
newtype EnvReader a = { runEnvReader :: Environment -> a }
```

- Faceți `EnvReader a` instanță a clasei `Show` afișând valoarea obținută prin execuția computației în mediul de evaluare inițial `[]`.
- Faceți `EnvReader` instanță a clasei `Monad`

  Funcția `return` întoarce valoarea dată pentru orice stare inițială

  Funcția »= ia ca argumente o computație ma :: EnvReader a și o funcție (continuare) k :: a -> EnvReader b. Rezultatul ei încapsulează o funcție de la `Environment` la tipul `b` care

  - trimite starea inițială transformării de stare ma; obține astfel o valoare.
  - aplică funcția `k` valorii, obținând o nouă computație
  - această nouă computație primește ca stare aceeași stare inițială și întoarce rezultatul evaluării

Modificați interpretorul pentru a evalua in monada `EnvReader`. Pentru aceasta:

- Eliminați argumentul `Environment` de la `interp` și de la toate celelalte funcții ajutătoare
- Definiți o computație `ask :: EnvReader Environment` care întoarce ca valoare starea curentă
- Folosiți `ask` pentru a defini semantica variabilelor
- Definiți o funcție `local :: (Environment -> Environment) -> EnvReader a -> EnvReader a` cu următoarea semantică:

  `local f ma` ia o transformare de stare `f` și o computație `ma` și produce o nouă computație care se va executa în starea curentă modificată folosind funcția `f`.

- Folosiți funcția `local` pentru a defini semantica lui `Lam`, extinzând local mediul de evaluare pentru a asocia variabilei valoarea dată.

Evaluarea lui `interp term0` ar trebui să întoarcă `"42"`.

Referințe:

https://davesquared.net/2012/08/reader-monad.html

P. Wadler, Monads for functional programming,

https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf