

Laboratorul 4

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.

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)
```

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:

```
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>"
```

Variații monadice

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

Exercițiu: Interpretorul monadic general

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 în formă postfixată.

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, în 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`, 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 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.

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.

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.

Exercițiu: alegere nedeterministă (variația 3)

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

Pentru a face acest lucru vom folosi monada (predefinită) asociată tipului listă:

```
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.

Exercițiu: Interpretare în monada Reader (variația 4)

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 = Reader { 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 în monada `EnvReader`. Pentru aceasta:

- Eliminați argumentul `Environment` de la `interp` și de la toate celelalte funcții ajutoare
- 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"`.

Exercițiu: Afișare de rezultate intermediare (variația 5)

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 făcută folosind afișarea ca efect lateral.

Exercițiu: Stare (variația 6)

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.