Interpretoare monadice

după Philip Wadler

Laboratorul 9

Acest laborator explorează folosirea monadelor pentru structurarea programelor funcționale.

Monadele cresc ușurința cu care pot fi modificate programele. Ele pot imita efectele impure precum excepțiile și starea globală; pot oferi, de asemeni și efecte care nu pot fi obținute ușor de programele impure tradiționale. *Tipul* unui program identifică efectele laterale pe care acesta le poate produce.

Ca parte a acestui laborator, un interpretor simplu este modificat pentru a oferi suport pentru diferite efecte laterale: mesaje de eroare, stare, I/O și nedeterminism.

Introducere: Să fiu pur sau impur?

Limbajele funcționale pure, precum Haskell, oferă puterea evaluării lenește și simplicitatea raționamentelor în logica ecuațională. Pe de alta parte, limbajele funcționale impure, precum Ocaml, Scheme, Scala, oferă acces direct la facilități cum ar fi stare globală și tratarea excepțiilor.

Un factor care ar trebui să ne ghideze în această alegere este ușurința cu care un program poate fi modificat. Limbajele pure ușurează sarcina modificării deoarece datele de care depinde fiecare argument sunt explicite. Dar, uneori, o schimbare aparent minoră poate duce la restructurarea masivă a unui program într-un limbaj pur, pe când același efect ar fi putut fi obținut printr-o schimbare de câteva linii dacă foloseam o facilitate impură a limbajului.

Să luăm ca exemplu un interpretor într-un limbaj funcțional pur.

- Pentru a îi adăuga o facilitate de propagare și tratare a erorilor, trebuie să modific tipul rezultat pentru a include valori eroare, și la fiecare apel recursiv să propag și/sau să tratez erorile în mod corespunzător.

Dacă foloseam un limbaj impur cu excepții, n-aș fi avut nevoie de nici o restructurare.

- Pentru a adăuga un contor al execuției, trebuie să modific tipul rezultat pentru a include un astfel de contor și să modific fiecare apel recursiv

pentru a propaga acest contor în mod corespunzător.

Dacă foloseam un limbaj impur cu o variabilă globală care putea fi incrementată nu ar mai fi fost necesară o astfel de restructurare.

- Pentru a adăuga o instrucțiune de afișare, trebuie să modific tipul rezultat pentru a include o listă de mesaje și să modific fiecare apel recursiv pentru a propaga lista în mod corespunzător.

Dacă foloseam un limbaj impur în care afișarea e un efect lateral, nu aveam nevoie de nici o restructurare

Sau... aș putea folosi o monadă

Acest laborator vă prezintă o modalitate de a folosi monadele pentru a structura un interpretor astfel încât schimbările menționate mai sus să fie ușor de efectuat. Pentru fiecare dintre cazuri, este nevoie doar să redefinim monada și să facem câteva schimbări locale. Acest stil de programare recuperează o parte din flexibilitatea oferită de diverse facilități ale limbajelor impure. De asemeni, această strategie poate fi aplicată și în lipsa efectelor laterale.

Interpretorul (monadic) de bază

Vom începe cu un interpretor simplu pentru lambda calcul, o variantă simplificată a interpretorului Mini-Haskell din cursul 6 (slide-urile 10–21)

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`.

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>"
```

Ce este o monadă

În Haskell, o monadă este un triplet `(M,return,>>=)`, unde `M` este un constructor de tipuri instanță a clasei `Monad`, care este definită de o pereche de funcții polimorfice

```
return :: a -> M a
(>>=) :: M a -> (a -> M b) -> M b
```

Gândind `M a` ca tipul computațiilor (cu efecte laterale) care produc rezultate de tip `a`, `return a` este computația (trivială) care produce `a` fără efecte laterale, iar `>>=` este operația de secvențiere a computațiilor: dată fiind o computație `ma :: M a` care produce elemente de tip `a` și o funcție (continuare) `k :: a -> M b` care pentru orice element de tip `a` dă o computație care produce elemente de tip `b`, `ma >>= k :: M b` reprezintă computația care produce elemente de tip `b` astfel: atunci cand se execută, execută mai întâi `ma`, apoi în funcție de valoarea obținută alege o computație de tip `M b` folosind `k`, pe care o execută cu propagarea efectelor laterale de la `ma`.

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

Observație: în Haskell, un constructor de tipuri nu poate deveni instanță a clasei `Monad` decât dacă este instanță a lui `Functor` și `Applicative`. Mai jos va trebui să faceți mai multe tipuri instanțe ale clasei `Monad`. Puteți (ca un bun exercițiu) să vă gândiți cum fi făcute instanțe ale lui `Functor` și `Applicative`, dar puteți folosi și faptul că `Functor` și `Applicative` pot fi definiți cu `return` și `>>=`, fapt descris și în cursul 7, slide-ul 46.

Exercițiul 0: 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.

Pentru verificare puteți consulta slide-ul 13 din cursul 8.

Exercițiul 1: Interpretorul monadic

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. Astfel, scopul lui `return` este să transforme o valoare într-o computație iar scopul lui `>>=` este să evalueze o computație, obținând o valoare.

Informal, `return` ne introduce într-o monadă și `>>=` ne ajută să compunem computații într-o monadă.

Cerință: Având drept inspirație interpretorul pentru Mini-Haskell definit în cursul 6, definiți un interpretor monadic pentru limbajul de mai sus.

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

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

Funcția identiate 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.

Variații monadice

Pentru fiecare din exercițiile (variațiile) de mai jos copiați fișierul `var0.hs` ca `varn.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 17–18 din cursul 8.

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 19–22 din cursul 8.

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:

```
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 30–31 din cursul 8.

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 27–29 din cursul 8.

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 23–26 din cursul 8.

Exercițiu: stare imuabilă (varația 6)

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"`.

Pentru verificare puteți consulta slide-urile 32–36 din cursul 8.