Μονάδες

Γιάννης Κασσιός

Στις σημειώσεις αυτές, κάνουμε μια μικρή εισαγωγή στον προγραμματισμό με *μονάδες (monads)*. Οι μονάδες είναι ένας μαθηματικός μηχανισμός που χρησιμοποιείται ευρέως στην κοινότητα της Haskell. Η κύρια χρήση των μονάδων στη Haskell είναι η παροχή υπηρεσιών εισόδου/εξόδου στα προγράμματα. Παρ' όλα αυτά, οι μονάδες έχουν και άλλες πολλές και ετερόκλητες εφαρμογές και έχουν εμπνεύσει ένα ολόκληρο στυλ προγραμματισμού ονομαζόμενο *προγραμματισμός βασισμένος σε μονάδες (monadic programming)*. Ο όρος «μονάδα» προέρχεται από τον αντίστοιχο όρο της *θεωρίας κατηγοριών (category theory)*, μίας πολύ αφαιρετικής μαθηματικής θεωρίας με τεράστιο πεδίο εφαρμογών.

Στην παρουσίασή μας, θα αφιερώσουμε τον περισσότερο χώρο στην ιδέα πίσω από τη χρήση μονάδων στον προγραμματισμό και στη χρησιμότητά τους. Στη συνέχεια θα εξηγήσουμε πώς ακριβώς υλοποιείται η λειτουργικότητα μονάδων στη Haskell και θα δώσουμε τη σχετική σύνταξη και παραδείγματα. Τέλος θα αναφέρθουμε στο μηχανισμό εισόδου/εξόδου της Haskell ως μία από τις εφαρμογές των μονάδων. Ο χαρακτήρας των σημειώσεων αυτών θα κρατηθεί σε εισαγωγικό επίπεδο.

1 Εισαγωγή στις Μονάδες

Η σειριακή εκτέλεση στο συναρτησιακό προγραμματισμό δίνεται από τη σύνθεση συναρτήσεων:

$$f_{n-1}(f_{n-2}(...f_0\ x))...$$

Στο παραπάνω έχουμε γράψει τις συναρτήσεις σε ανάποδη σειρά από αυτήν με την οποία αποτιμούνται. Για να φαίνεται η σωστή σειρά αποτίμησης, ορίζουμε:

```
app :: a->(a->b)->b
app x f = f x
```

οπότε η παραπάνω αποτίμηση μπορεί να γραφεί ως εξής:

$$x\ \text{`app`}\ f_0\ \text{`app`}\ ...\ \text{`app`}\ f_{n-1}$$

Έστω τώρα ότι θέλουμε να «εμπλουτίσουμε» τις συναρτήσεις με επιπλέον λειτουργικότητα, κοινή για όλες, αλλά ανεξάρτητη από τη λειτουργία κάθε μίας από αυτές. Για παράδειγμα, θα θέλαμε ένα τρόπο όλες οι συναρτήσεις μας να μπορούν να αναφέρουν σφάλμα (όπως με τη `Maybe`) ή θα θέλαμε να μπορούν να γράφουν σε ένα log σχετικά με τη λειτουργία τους. Η έξτρα λειτουργικότητα των

συναρτήσεων αλλάζει τον τύπο εξόδου τους από T_i σε $m\ T_i$, όπου m μία συνάρτηση τύπων.

Τώρα δε μπορούμε πλέον να χρησιμοποιήσουμε την `app` για να γράφουμε σειριακούς υπολογισμούς όπως ο παραπάνω. Χρειαζόμαστε μία συνάρτηση που να λαμβάνει υπόψην της ότι οι νέες συναρτήσεις f_i' είναι τύπου T_i`->`$m\ T_{i+1}$. Η συνάρτηση αυτή θα θεωρεί ότι το αποτέλεσμα «στα αριστερά της» έχει ήδη μετατραπεί σε τύπο m `a` και το όρισμα «στα δεξιά της» θα είναι μία συνάρτηση που παίρνει τύπο `a` και επιστρέφει τύπο m `b`. Το τελικό αποτέλεσμα θα είναι τύπου m `b`. Ας ονομάσουμε τη συνάρτηση αυτή `appm`:

```
appm ::  m a->(a->m b)->m b
```

Για να ξεκινήσει ο υπολογισμός, χρειάζεται μία συνάρτηση η οποία μετατρέπει ένα όρισμα από `a` σε m `a`. Ας την πούμε `liftm`:

```
liftm ::  a->m a
```

1.1 Πρώτο Παράδειγμα: Χειρισμός Σφαλμάτων με τη `Maybe`

Γνωρίζουμε ότι ο κατασκευαστής τύπων `Maybe` προσθέτει έναν απλό χειρισμό υπολογιστικών σφαλμάτων στις συναρτήσεις, δίνοντάς τους τη δυνατότητα να επιστρέψουν `Nothing` σε περίπτωση τέτοιου σφάλματος. Στο παράδειγμα αυτό, θεωρούμε ότι θέλουμε να προσθέσουμε αυτή τη λειτουργικότητα στις συναρτήσεις μας (δηλαδή το m μας θα είναι ο κατασκευαστής `Maybe`).

Ένας τρόπος θα ήταν να μετατρέψουμε όλες τις συναρτήσεις μας από τύπο `a->b` σε τύπο `Maybe a->Maybe b`. Ας υποθέσουμε τώρα ότι ορίζουμε μία συνάρτηση `f` η οποία λειτουργεί κανονικά αν πάρει κανονικό όρισμα, ενώ διοχετεύει το σφάλμα σε περίπτωση που πάρει όρισμα `Nothing`. Ο ορισμός της `f` είναι:

```
f (Just x) = ...
f Nothing = Nothing
```

Παρατηρήστε ότι η λειτουργία της συνάρτησης βρίσκεται στο σημείο που είναι οι τρεις τελείες. Όλο το υπόλοιπο αφιερώνεται στην καινούρια λειτουργικότητα χειρισμού σφαλμάτων. Από εδώ και στο εξής, αυτή η καινούρια λειτουργικότητα θα επιφέρει αυτήν την περιπλοκή στον ορισμό κάθε συνάρτησης.

Το πρόβλημα γίνεται χειρότερο αν σκεφτούμε τη χρήση ανώνυμων συναρτήσεων, οι οποίες θα πρέπει και αυτές να υποστηρίζουν τέτοια λειτουργικότητα. Οι συναρτήσεις αυτές θα γράφονται κάπως έτσι:

```
\mbx->case mbx of Just x ->...
                  Nothing->Nothing
```

Το μόνο χρήσιμο κομμάτι είναι εκεί που βρίσκονται οι τελείες. Το υπόλοιπο έχει να κάνει με τη νέα λειτουργικότητα, αλλά δεν προσφέρει κάτι στον υπολογισμό.

Αν θεωρήσουμε τώρα αυτήν τη συντακτική περιπλοκή μέσα σε παραδείγματα που περιέχουν πολλούς υπολογισμούς, το πρόβλημα αρχίζει να εκδηλώνεται ακόμα πιο έντονα:

```
case val of
  Nothing -> Nothing
  Just x -> case f x of
              Nothing -> Nothing
              Just y -> case g y of
                          Nothing -> Nothing
                          Just z -> Just z
```

Το πρόβλημά μας είναι ότι η λειτουργικότητα εντοπισμού σφαλμάτων δε γράφεται ανεξάρτητα από τον υπολογισμό, αλλά αντιθέτως μπλέκεται μέσα σε αυτόν και αυξάνει κατά πολύ την πολυπλοκότητα του κώδικα. Θέλουμε έναν τρόπο να ξεχωρίσουμε τον «πραγματικό» υπολογισμό από το χειρισμό σφαλμάτων.

Ας επιστρέψουμε στην ιδέα της εισαγωγής: οι συναρτήσεις μας θα είναι τύπου `a->Maybe b`. Θα εφεύρουμε δύο συναρτήσεις:

```
appMaybe :: Maybe a -> (a->Maybe b) -> Maybe b
liftMaybe :: a->Maybe a
```

οι οποίες θα εμπεριέχουν μέσα τους τη λειτουργικότητα που στα πιο πάνω παραδείγματα γράφαμε από την αρχή κάθε φορά που χρησιμοποιούσαμε μία συνάρτηση.

Η συνάρτηση `liftMaybe` απλά μετατρέπει ένα όρισμα `x` στην τιμή που λέει «ο υπολογισμός δεν έχει σφάλματα και το αποτέλεσμα είναι `x`». Δηλαδή, η `liftMaybe` μετατρέπει το `x` σε `Just x`. Δηλάδή:

```
liftMaybe = Just
```

Η συνάρτηση `appMaybe` θα ελέγχει αν το αριστερό της όρισμα είναι αναφορά λάθους. Σε αυτήν την περίπτωση επιστρέφει αναφορά λάθους, αλλιώς περνάει αποτέλεσμα του σωστού υπολογισμού στα αριστερά του στη συνάρτηση που είναι στα δεξιά του:

```
appMaybe Nothing _ = Nothing
appMaybe (Just x) f = f x
```

Παρατηρούμε ότι η `appMaybe` περιγράφει τη λειτουργικότητα χειρισμού σφαλμάτων όπως τη γράφαμε συνεχώς στα πιο πάνω παραδείγματα. Τώρα, μπορούμε να ξεφορτωθούμε αυτή τη λειτουργικότητα από τον κώδικα που περιγράφει τον υπολογισμό. Ο ορισμός της `f` γίνεται:

```
f x = ...
```

(ο τύπος της είναι τώρα `a->Maybe b`). Μία ανώνυμη συνάρτηση γράφεται:

```
\x->...
```

και το παράδειγμα υπολογισμού που αναφέραμε πιο πάνω γράφεται:

```
liftMaybe val `appMaybe` f `appMaybe` g
```

Ας δούμε τώρα ένα άλλο παράδειγμα. Έστω η παρακάτω συνάρτηση ασφαλούς αντιστροφής ενός πραγματικού αριθμού:

```
saferev x = if x==0 then Nothing else Just (1/x)
```

Θέλουμε να χρησιμοποιήσουμε τη `saferev` για να υπολογίσουμε τον αριθμό `1/(x*z)`. Ο παρακάτω κώδικας χρησιμοποιεί τη `saferev` δύο φορές, και πολλαπλασιάζει τα αποτελέσματα των δύο εφαρμογών:

```
liftMaybe x ‘appMaybe‘ saferev
            ‘appMaybe‘ \rx->liftMaybe z
                          ‘appMaybe‘ saferev
                          ‘appMaybe‘ \rz->liftMaybe(rx*rz)
```

Ο κώδικας αυτός δεν έχει την πολυπλοκότητα του χειρισμού των λαθών, αλλά δεν είναι πολύ ευανάγνωστος. Δεν είναι εύκολο κανείς να καταλάβει από τον κώδικα τη σειριακή εκτέλεση των εφαρμογών, τι όρισμα παίρνει κάθε συνάρτηση και τι αποτέλεσμα παράγει. Ένας άλλος τρόπος που μπορούμε να γράψουμε τον ίδιο κώδικα είναι πολύ πιο ενδεικτικός του τι συμβαίνει:

```
liftMaybe x ‘appMaybe‘ \x->
saferev ‘appMaybe‘ \rx->
liftMaybe z ‘appMaybe‘ \z->
saferev ‘appMaybe‘ \rz->
liftMaybe(rx*rz)
```

Ο κώδικας αυτός εξηγεί με πολύ περισσότερη σαφήνεια τον υπολογισμό μας με σειριακό τρόπο:

- παίρνουμε μία τιμή `x`
- καλούμε τη `saferev` στο `x`. Έστω `rx` το αποτέλεσμα
- παίρνουμε μία τιμή `z`
- καλούμε τη `saferev` στο `z`. Έστω `rz` το αποτέλεσμα
- επιστρέφουμε την τιμή `rx*rz`

1.2 Δεύτερο Παράδειγμα: Logging

Στο δεύτερο παράδειγμά μας, θέλουμε να εμπλουτίσουμε τις συναρτήσεις μας με δυνατότητα logging, δηλαδή αναφοράς σε μία έξοδο στοιχείων για τη λειτουργία των συναρτήσεων. Το logging βοηθάει στην αποσφαλμάτωση των προγραμμάτων αλλά και τον εντοπισμό προβληματικών συνθηκών μέσα στις οποίες εκτελούνται τα προγράμματα. Πάλι, θέλουμε να ξεχωρίσουμε τη λειτουργία μίας συνάρτησης από τη λειτουργία του logging.

Ο τρόπος με τον οποίο θα εμπλουτίσουμε τις συναρτήσεις μας με logging είναι η αλλαγή του τύπου εξόδου `b` σε `(b,String)`. Το δεύτερο στοιχείο της εξόδου είναι μία συμβολοσειρά που περιέχει όλα το logging που έχει κάνει η συνάρτηση

κατά τη διάρκεια της αποτίμησής της. Για παράδειγμα, μία συνάρτηση που το μόνο logging που θέλει να κάνει είναι η αναφορά ότι αποτιμήθηκε μπορεί να γραφτεί κάπως έτσι:

```
f x = (..., "f was called")
```

Ακολουθούμε την ίδια ιδέα με πριν. Ορίζουμε τον τύπο `Log a`:

```
type Log a = (a,String)
```

και δύο συναρτήσεις, τη `liftLog` και την `appLog`. Η `liftLog` παίρνει το όρισμά της και του επισυνάπτει μία κενή συμβολοσειρά (κενό logging):

```
liftLog r = (r,"")
```

Η `appLog` πρέπει να συνενώσει το logging που έχει μέχρι τώρα συμβεί στα αριστερά της με το logging που πρόκειται να δώσει η συνάρτηση στα δεξιά της. Έτσι ο ορισμός της είναι:

```
appLog (r,s) f = (r',s++s') where (r',s') = f r
```

Τώρα θα μπορούσαμε να ορίσουμε συναρτήσεις που να αναφέρουν ότι κληθήκανε, ως εξής:

```
f x = (2*x, "f(" ++ show x ++ ") was called. ")
g x = (x+1, "g(" ++ show x ++ ") was called. ")
```

Πιο κομψά, θα μπορούσαμε να ορίσουμε μία συνάρτηση που να κάνει μόνο logging και τίποτε άλλο. Ας την πούμε `logg`[1]:

```
logg str = (undef,str)
```

Χρησιμοποιούμε την τιμή `undef` που ορίσαμε σε άλλες σημειώσεις με `undef=undef`. Στην πραγματικότητα δεν έχει σημασία ποια τιμή χρησιμοποιείται σε εκείνο το σημείο. Η συνάρτηση `logg` δεν επιστρέφει κανένα χρήσιμο αποτέλεσμα, αλλά προσθέτει κάτι στο log του υπολογισμού.

Με τη `logg`, οι `f` και `g` που ορίσαμε πιο πάνω μπορούν να γραφτούν:

```
f x =
  logg "f(" ++ show x ++ ") was called. " `appLog` \_->
  liftlog (2*x)
g x =
  logg "g(" ++ show x ++ ") was called. " `appLog` \_->
  liftlog (x+1)
```

Ο τρόπος αυτός διαχωρίζει τις δύο λειτουργίες και επεξηγεί το αποτέλεσμα καλύτερα. Ένας πιο πολύπλοκος υπολογισμός που χρησιμοποιεί τις `f` και `g` είναι ο παρακάτω:

[1] Δεν την ονομάζουμε `log` για να μην υπάρχει σύγκρουση ονομάτων με τη συνάρτηση λογαρίθμου

```
liftLog 10 ‘appLog‘ \x->
f x ‘appLog‘ \y->
g y ‘appLog‘ \z-> liftLog(x+y+z)
```

του οποίου το αποτέλεσμα είναι:

```
(51,"f(10) was called. g(20) was called. ")
```

Σε ό,τι έχουμε κάνει μέχρι τώρα, η αναδρομή χρησιμοποιείται χωρίς πρόβλημα. Ας δούμε για παράδειγμα τον υπολογισμό του παραγοντικού με logging:

```
factorial 0 =
  logg ("factorial(0) was called. ") ‘appLog‘ \_->
  liftLog 1
factorial n =
  logg("factorial(" ++ show n ++ ") was called. ")
       ‘appLog‘ \_->
  factorial(n-1) ‘appLog‘ \x->
  liftLog (x*n)
```

1.3 Ορισμός της Μονάδας

Στα παραπάνω παραδείγματα είδαμε ότι για να προσθέσουμε μία νέα λειτουργικότητα m σε μία ομάδα συναρτήσεων χρειαζόμαστε:

- Μία μετατροπή τύπων m
- Μία συνάρτηση `liftm` και μία συνάρτηση `appm` με τύπους:
  `liftm :: a->`m `a`
  `appm :: `m `a->(a->`m `b)->`m `b`

  Οι συναρτήσεις τύπων θα πρέπει να ικανοποιούν ακόμη τις εξής ιδιότητες:

- Η ύψωση ενός δεδομένου σε λειτουργικότητα m `a` και το πέρασμά του με την `appm` σε συνάρτηση `f :: a->`m `b`, θα πρέπει να λειτουργεί ακριβώς όπως η συνάρτηση `f`. Δηλαδή, οι `liftm` και `appm` δεν αλλάζουν τη λειτουργία των συναρτήσεων τις οποίες συνδέουν. Τυπικά, η ιδιότητα γράφεται ως εξής:

  ```
  liftm x ‘appm‘ f = f x
  ```

- Ομοίως, το πέρασμα ενός ορίσματος `mx` μέσω της `appm` στην `liftm` θα πρέπει να επιστρέφει το `mx`. Η συνάρτηση `liftm` λειτουργεί σαν ένα είδος «ταυτότητας» και δεν εισάγει επιπλέον υπολογισμούς:

  ```
  mx ‘appm‘ liftm = mx
  ```

- Δεν έχει σημασία με ποια σειρά θα εφαρμοστούν περισσότερα από ένα στιγμιότυπα της `appm` σε ένα σειριακό υπολογισμό. Η `appm` λειτουργεί σαν σύνθεση συναρτήσεων, πράξη προσεταιριστική:

```
mx ‘appm‘ (\x ->f x‘appm‘g) = (mx‘appm‘f)‘appm‘g
```

Μία τριάδα $(m, \texttt{liftm}, \texttt{appm})$ που ικανοποιεί τα παραπάνω ονομάζεται *μονάδα (monad)*.

2 Μονάδες στη Haskell

2.1 Υπερφόρτωση Κατασκευαστών Τύπων στη Haskell

Ένας κατασκευαστής τύπων είναι ένας πολυμορφικός αλγεβρικός τύπος. Παραδείγματα είναι οι `[]` και `Maybe` της Haskell οι οποίοι παίρνουν ως όρισμα ένα τύπο. Άλλοι κατασκευαστές, όπως ο `Either` παιρνούν παραπάνω από έναν τύπους.

Ο μηχανισμός της υπερφόρτωσης και των κλάσεων ισχύει όχι μόνο για τύπους, αλλά γενικότερα για κατασκευαστές τύπων. Στην πραγματικότητα, η υπερφόρτωση για τύπους μπορεί να θεωρηθεί υποπερίπτωση, αφού ένας τύπος είναι και ένας κατασκευαστής τύπων με μηδέν ορίσματα.

Στο παρακάτω παράδειγμα, ορίζουμε μία κλάση `SomeClass` για κατασκευαστές τύπων με ένα όρισμα και μία συνάρτηση `somefunction` για την κλάση αυτή. Μετά ορίζουμε τους κατασκευαστές `Maybe` και `[]` ως στιγμιότυπα της κλάσης:

```
class SomeClass cons where
  somefunction :: cons a -> String

instance SomeClass Maybe where
  somefunction (Just _) = "Correct"
  somefunction Nothing = "Error"

instance SomeClass [] where
  somefunction [] = "Empty"
  somefunction _ = "Non-empty"
```

Αυτό που λέει στη Haskell ότι πρόκειται για κλάση κατασκευαστών με ένα όρισμα είναι η δήλωση `somefunction :: cons a -> String` που χρησιμοποιεί τον τύπο-στιγμιότυπο της κλάσης `cons` με ένα όρισμα.

2.2 Η Κλάση `Monad`

Η κλάση `Monad` για κατασκευαστές τύπων με ένα όρισμα, εισάγει μονάδες. Τα ονόματα των συναρτήσεων που χρησιμοποιεί η κλάση είναι `return` (αυτό που εμείς λέγαμε `liftm` στην εισαγωγή) και `>>=` (αυτό που εμείς λέγαμε `‘appm‘`

στην εισαγωγή). Υποστηρίζεται και τελεστής `>>` που δρα όπως και ο `>>=`, αλλά αγνοώντας το αποτέλεσμα της παράστασης στα αριστερά του. Η δήλωση της κλάσης είναι:

```
class Monad m where
 (>>=) :: m a->(a->m b)->m b
 return :: a->m a
 (>>) :: m a->m b->m b
 m >> k  =  m >>= \_->k
```

Φυσικά, οι ιδιότητες της μονάδας δε μπορούν να επαληθευτούν από τη Haskell (εκτός από τους τύπους των συναρτήσεων) η οποία επαφίεται στον προγραμματιστή για τη σωστή υλοποίηση μίας μονάδας.

Ο κατασκευαστής `Maybe` έχει ήδη δηλωθεί ως μονάδα στη Haskell. Η δήλωση είναι ακριβώς όπως την περιγράψαμε στην εισαγωγή:

```
instance Monad Maybe where
 (Just x) >>= k  =  k x
 Nothing >>= k  =  Nothing
 return = Just
```

Το παράδειγμα του υπολογισμού του `1/(x*z)` μπορεί να γραφτεί ως εξής:

```
return x >>= \x->
saferev x >>= \rx->
return z >>= \z->
saferev z >>= \rz->
return (rx*rz)
```

Για τη μονάδα που υλοποιεί logging, δε μπορούμε να χρησιμοποιήσουμε τον ορισμό του τύπου `Log` όπως έχει δοθεί, γιατί δεν είναι αλγεβρικός. Μετατρέπουμε τον ορισμό σε αλγεβρικό τύπο, χρησιμοποιώντας κατασκευαστή `Log` (επιτρέπεται και δεν είναι σπάνιο ένας κατασκευαστής αλγεβρικού τύπου να έχει ίδιο όνομα με τον τύπο):

```
data Log a = Log (a, String)
instance Monad Log where
 return r = Log (r,"")
 (Log (r,s)) >>= k = Log (r',s++s') where Log (r',s')=k r
logg str = Log (undef,str)
```

Είναι εύκολο να ξαναγράψουμε τα παραδείγματά μας με τη χρήση της σύνταξης της Haskell για μονάδες:

```
f n = logg("f(" ++ show n ++ ") was called. ") >>
      return (2*n)
g n = logg("g(" ++ show n ++ ") was called. ") >>
      return (n+1)
exLog = return 10 >>= \x->
```

```
        f x >>= \y->
        g y >>= \z->
        return(x+y+z)
factorial 0 =
 logg"factorial(0) was called. " >>
 return 1
factorial n =
 logg("factorial(" ++ show n ++ ") was called. ") >>
 factorial(n-1) >>= \x->
 return(x*n)
```

2.3 Σύνταξη do

Οι μονάδες μας βοηθάνε να καθαρίσουμε τον κώδικά μας από μία κοινή λειτουργικότητα των συναρτήσεων που θέλουμε να κάνουμε «αόρατη». Ο κώδικάς μας όμως γεμίζει από εφαρμογές των τελεστών >>= και >>. Η Haskell παρέχει τη λέξη κλειδί do για να εισάγει μία σύνταξη που μας απαλλάσει από αυτόν το συντακτικό θόρυβο και να κάνει τον κώδικα ακόμα πιο ευανάγνωστο.

Η λέξη do ακολουθείται από μία στοιχισμένη αλληλουχία γραμμών, στις οποίες οι τελεστές >>= και >> δε γράφονται. Μία γραμμή
κώδικας>>
αντικαθίσταται από:
κώδικας
Μία γραμμή
κώδικας >>= *μεταβλητή*->
αντικαθίσταται από:
μεταβλητή<-*κώδικας*

Ας δούμε πώς μετατρέπονται τα παραδείγματα που έχουμε χρησιμοποιήσει μέχρι τώρα, χρησιμοποιώντας τη σύνταξη do:

```
revmult x z = do rx<-saferev x
                 rz<-saferev z
                 return (rx*rz)
f n = do logg("f(" ++ show n ++ ") was called. ")
         return (2*n)
g n = do logg("g(" ++ show n ++ ") was called. ")
         return (n+1)
exLog = do x<-return 10
           y<-f x
           z<- g y
           return(x+y+z)
factorial 0 =
 do logg"factorial(0) was called. "
    return 1
factorial n =
 do logg("factorial(" ++ show n ++ ") was called. ")
```

```
    x<-factorial(n-1)
    return(x*n)
```

Στη σύνταξη `do` μπορούμε να βάλουμε και γραμμές `let` ως εξής:
`do let` v `=` E
κώδικας
που ισοδυναμεί με `(\`v`->do` *κώδικας*$)E$
Για παράδειγμα, το `exLog` που γράψαμε πιο πάνω, μπορεί να γραφεί ως εξής:

```
exLog = do let x = 10
           y<-f x
           z<- g y
           return(x+y+z)
```

2.4 Μοναδική Ανάθεση

Η σύνταξη που δίνει το `do` μοιάζει πολύ με προστακτικό προγραμματισμό. Έχουμε σειριακότητα και εντολές που μοιάζουν με αναθέσεις (όπως οι `let` και οι εκφράσεις με `<-`). Δεν πρέπει να ξεχνάμε όμως ότι *δεν* είμαστε σε προστακτικό προγραμματισμό. Η σειρά της αποτίμησης πράγματι δίνεται ξεκάθαρα από τη σύνταξη, αλλά πίσω από την αποτίμηση δεν υπάρχει κατάσταση. Οι μεταβλητές που χρησιμοποιούνται είναι καθαρές μαθηματικές μεταβλητές, οι οποίες δε αντιστοιχούν σε θέσεις μνήμης και δε μπορούν να αλλάξουν τιμή, όπως στον προστακτικό προγραμματισμό.

Έστω το παρακάτω αυτοσχέδιο `while`:

```
while test action =
 do c<-test
    if c then do action
                 while test action
    else return 0
```

Φαίνεται σωστό, αλλά δε δουλεύει αν το χρησιμοποιήσουμε όπως στον προστακτικό προγραμματισμό:

```
badfact n =
 do let s = 1
    while (return (n>0))
     (do let s = s*n
         let n = n-1
         logg "X"
     )
     return s
```

Η αποτίμηση του `badfact 2` δε θα τερματίσει ποτέ. Το λάθος είναι ότι χρησιμοποιούμε τη μεταβλητή `n` σα θέση μνήμης.

Στην πραγματικότητα κάθε εντολή που μοιάζει με ανάθεση, εισάγει μια ολοκαίνουρια μεταβλητή (δείτε και τον κώδικα του οποίου το `do` είναι συντομογραφία). Έτσι, η `n` που εισάγει η έκφραση `let n = n-1` δεν έχει καμία σχέση με τη `n` που εισάγει η γραμμή `badfact n =` και που ελέγχεται στην έκφραση `n>0`. Ο κώδικας αυτός θα επαναλαμβάνει επ' άπειρον τον έλεγχο `n>0` με την ίδια τιμή για τη μεταβλητή `n`.

Αν θέλουμε να καταλαβαίνουμε τις εκφράσεις `do` ως προστακτικό προγραμματισμό, θα πρέπει να συνηθίσουμε και τη σημασιολογία της *μοναδικής ανάθεσης (single assignment)*. Σε μία γλώσσα με μοναδική ανάθεση, μία μεταβλητή παίρνει τιμή μόνο μία φορά. Κάθε νέα ανάθεση εισάγει καινούρια μεταβλητή.

2.5 Οι Λίστες ως Μονάδες

Ο κατασκευαστής `[]` έχει δηλωθεί ως μονάδα στη Haskell για να αναπαριστά *μη ντετερμινιστικό υπολογισμό*. Η μονάδα αυτή επιτρέπει δηλαδή στις συναρτήσεις να επιστρέφουν παραπάνω από ένα αποτέλεσμα (ή και κανένα αποτέλεσμα). Η σύνθεση δύο συναρτήσεων επιστρέφει όλες τις πιθανές περιπτώσεις αποτελεσμάτων αν πάρουμε οποιοδήποτε έγκυρο αποτέλεσμα από την πρώτη, το τροφοδοτήσουμε στη δεύτερη και πάρουμε οποιοδήποτε έγκυρο αποτέλεσμα από τη δεύτερη.

Για παράδειγμα, έστω οι συναρτήσεις:

```
f x = [x,x+1]
g x = [x,2*x]
```

Η `f` συμβολίζει ένα μη ντετερμινιστικό υπολογισμό που παίρνει μία τιμή και ή την επιστρέφει ως έχει ή την επιστρέφει αυξημένη κατά ένα. Η `g` είναι ένας μη ντετερμινιστικός υπολογισμός που είτε επιστρέφει το όρισμά της ως έχει είτε πολλαπλασιασμένο επί δύο.

Αν πάρουμε τον αριθμό `5` και τον περάσουμε από τη «σύνθεση» των δύο συναρτήσεων (με πρώτη την `f` και δεύτερη την `g`), τα πιθανά αποτελέσματα είναι `5, 6, 10` και `12`. Άρα θέλουμε η σύνθεση των `f` και `g` με όρισμα `5` να επιστρέφει `[5,6,10,12]`. Όντως, ο παρακάτω κώδικας Haskell περνάει αυτήν την τιμή στη μεταβλητή `s`:

```
s = do x<-return 5
       y<-f x
       z<-g y
       return z
```

Για να επιτευχθεί αυτό, η δήλωση της μονάδας `[]` έχει ως εξής: (α) η `return` μετατρέπει το όρισμα `x` σε `[x]` (ντετερμινιστικός υπολογισμός) και (β) η `>>=` παίρνει τα αποτελέσμα που έχει υπολογιστεί στα αριστερά της, το περνάει τιμή-τιμή στη συνάρτηση στα δεξιά της και μετά συνενώνει τα αποτελέσματα:

```
instance Monad [] where
  return x = [x]
  m >>= k = concat (map k m)
```

2.6 Η Τετριμμένη Μονάδα

Θα κλείσουμε την ενότητα για τις μονάδες με την τετριμμένη μονάδα `Id` η οποία δεν προσφέρει επιπλέον λειτουργικότητα στις συναρτήσεις. Ο λόγος που ασχολούμαστε με την `Id` είναι για να δείξουμε ότι και η απλή σύνθεση συναρτήσεων, χωρίς καμία επιπλέον λειτουργικότητα, μπορεί να περιγραφεί σα μία μονάδα.

Η μονάδα μας λέγεται `Id` και ορίζεται ως εξής:

```
data Id a = Id a
instance Monad Id where
  return = Id
  (Id x) >>= k = k x
```

3 Είσοδος/έξοδος στη Haskell

Μέχρι τώρα, όλα τα προγράμματά μας ήταν αποτιμήσεις εκφράσεων, οι οποίες εκκινούνταν μεν από το χρήστη, αλλά δεν αλληλεπιδρούσαν με αυτόν ή με οποιοδήποτε άλλο κομμάτι του περιβάλλοντός τους, π.χ. κάποιο αρχείο, μέχρι τον τερματισμό τους. Όπως καταλαβαίνουμε, αυτό το είδος υπολογισμού δε μας επιτρέπει να γράψουμε πολλά προγράμματα που έχουν ανάγκη αυτήν την αλληλεπίδραση. Καθώς αυτά τα προγράμματα είναι και η συντριπτική πλειοψηφία των προγραμμάτων που τρέχουν στον υπολογιστή μας, η εισαγωγή ενός μηχανισμού εισόδου / εξόδου στη Haskell είναι επιτακτική.

Στην ενότητα αυτή, θα κάνουμε μια μικρή εισαγωγή στο μηχανισμό εισόδου / εξόδου της Haskell, ο οποίος χρησιμοποιεί μονάδες. Ο στόχος δεν είναι μια εξαντλητική μελέτη της σχετικής βιβλιοθήκης, αλλά η επεξήγηση των βασικών αρχών, καθώς και των λόγων που οδήγησαν στη χρήση μονάδων για το μηχανισμό αυτό.

3.1 Το Πρόβλημα της Εισόδου/εξόδου στις Συναρτησιακές Γλώσσες

Η είσοδος/έξοδος (I/O) είναι το μεγάλο αγκάθι του συναρτησιακού προγραμματισμού. Το πρόβλημα είναι ότι η αλληλεπίδραση με τον «έξω κόσμο» χαλάει το «αγνό» και ελεύθερο παρενεργειών μαθηματικό μοντέλο πάνω στο οποίο βασίζεται το συναρτησιακό παράδειγμα.

Ας υποθέσουμε ότι έχουμε μία «συνάρτηση» `read` η οποία διαβάζει έναν αριθμό από την κονσόλα και τον επιστρέφει στο περιβάλλον της. Είναι εμφανές ότι η `read` δεν είναι πραγματική συνάρτηση, γιατί κάθε αποτίμησή της μπορεί να επιστρέψει και διαφορετική τιμή. Αυτό χαλάει το συναρτησιακό μοντέλο και τα καθαρά μαθηματικά που μέχρι τώρα χρησιμοποιούσαμε για την απόδειξη ιδιοτήτων των προγραμμάτων μας. Για παράδειγμα, η έκφραση `read-read` δεν αποτιμάται σε `0`, όπως θα ήταν μαθηματικά σωστό. Επίσης, το αποτέλεσμα της έκφρασης εξαρτάται από τη σειρά αποτίμησης των υποεκφράσεων του `-`.

Ακόμα χειρότερα, η `read` μπορεί να χρησιμοποιηθεί σε άλλους ορισμούς, «μολύνοντας» και συναρτήσεις στις οποίες η χρήση της `read` δεν είναι εμφανής:

```
f = read + 42
```

Στο παράδειγμα αυτό, η `f` επίσης δε συμπεριφέρεται σα συνάρτηση, αλλά αυτό είναι ακόμα λιγότερο εμφανές στο χρήστη της, ο οποίος μπορεί να μη γνωρίζει ότι έχει οριστεί με βάση το `read`.

3.2 Μερικές Προσεγγίσεις στο Πρόβλημα

Στο πρόβλημα εισόδου/εξόδου των συναρτησιακών γλωσσών έχουν προταθεί διαφορετικές προσεγγίσεις. Η απλούστερη είναι αυτή που ακολουθούν η ML και οι διάλεκτοι της Lisp: οι γλώσσες αυτές αποδέχονται τις παρενέργειες ως αναπόσπαστο κομμάτι της αλληλεπίδρασης με τον έξω κόσμο.

Ένας τρόπος να διατηρήσουμε τον αγνό, χωρίς παρενέργειες, συναρτησιακό χαρακτήρα της γλώσσας, είναι να θεωρήσουμε την είσοδο και την έξοδο ως άπειρες λίστες δεδομένων, όπως γίνεται στις παλιές εκδόσεις της Haskell. Ούτε η είσοδος ούτε η έξοδος είναι στην πραγματικότητα άπειρες, αλλά αυτό δεν έχει σημασία. Η μοντελοποίηση της εισόδου με μια άπειρη λίστα `input`, που καθίσταται δυνατή χάρη στην οκνηρή αποτίμηση, μας επιτρέπει να διαβάζουμε κάθε φορά ένα στοιχείο δεδομένων, ζητώντας «το επόμενο στοιχείο» της `input`.

Το πρόβλημα με αυτή τη μοντελοποίηση είναι ότι δεν επιτρέπει τη σειριακή εκτέλεση εντολών εισόδου/εξόδου που μπορεί να είναι σημαντική σε ένα πρόγραμμα. Ο χρήστης μπορεί να έχει δόσει είσοδο στην κονσόλα, πριν να του έχει δωθεί ένα μήνυμα εξόδου που να περιγράφει τι είσοδο χρειάζεται να δώσει. Χρειαζόμαστε έναν καλύτερο μηχανισμό που να να μπορεί να προσδιορίζει αν χρειαστεί τη σειρά με την οποία εκτελούνται οι διαδικασίες εισόδου/εξόδου.

3.3 Μονάδες για Είσοδο/έξοδο

Σκεφτόμενοι καθαρά μαθηματικά, η ανάγνωση δεδομένων μπορεί να θεωρηθεί ως μία συνάρτηση που παίρνει ως παράμετρο τον «κόσμο» όπως είναι εκείνη τη στιγμή:

```
read ::  World->(InputType,World)
```

και επιστρέφει τον «κόσμο» όπως είναι μετά την ανάγνωση. Ομοίως, η εγγραφή δεδομένων, μπορεί να θεωρηθεί ως συνάρτηση που παίρνει σαν παράμετρο τον κόσμο και τον αλλάζει:

```
write ::  (OutputType,World)->World
```

Αυτός ο σχεδιασμός λύνει το πρόβλημα της σειριακότητας. Για παράδειγμα, στο παρακάτω, η ανάγνωση γίνεται μετά την εγγραφή:

```
read w' where w' = write ("Give me your name",w)
```

Φυσικά, αυτό μπορούμε να το κάνουμε και με μια μονάδα. Η σύνταξη τότε θα μοιάζει κάπως έτσι:

```
do write ("Give me your name")
   x <- read
```

όπου το πέρασμα των «κόσμων» γίνεται αυτόματα από τη μονάδα και δε μας απασχολεί.

Υπάρχει ένα ακόμα πρόβλημα που πρέπει να λυθεί. Η χρήση των κόσμων μπορεί να γίνει με ένα τρόπο μαθηματικά αποδεκτό, αλλά πρακτικά αδύνατο. Για παράδειγμα, μπορεί κάποιος να ζητήσει να επιδράσει με ένα κόσμο που υπήρχε πριν από πολλούς υπολογισμούς και που τον έχει κρατήσει σε μία μεταβλητή. Δεν είναι πολύ πρακτικό να αποθηκεύεται ολόκληρος ο κόσμος σε μία μεταβλητή: το τέχνασμα που χρησιμοποιούμε με τους κόσμους είναι μία καθαρά μαθηματική αφαίρεση.

Για να λύσουν το πρόβλημα οι σχεδιαστές της Haskell αποφάσισαν να κάνουν τη μονάδα εισόδου/εξόδου έναν ΑΤΔ. Η χρήση του ΑΤΔ επιτρέπεται μόνο μέσω των συναρτήσεων που παρέχονται από τη Haskell για την είσοδο και την έξοδο. Η ανεπιθύμητη πρόσβαση σε παράξενες μαθηματικές αφαιρέσεις όπως οι κόσμοι, αλλά και στα πραγματικά δεδομένα και την υλοποίηση της εισόδου/εξόδου είναι αδύνατη.

Η μονάδα εισόδου/εξόδου της Haskell ονομάζεται `IO` και ορίζεται στο τμήμα `IO` που παρέχεται από τη διανομή της Haskell. Πρέπει επομένως να κάνουμε

```
import IO
```

για να τη χρησιμοποιήσουμε.

Ο τύπος `IO a` έχει ως τιμή οποιονδήποτε κώδικα αλληλεπιδρά με το χρήστη και επιστρέφει τιμή `a`. Στις περιπτώσεις που δε θέλουμε επιστροφή δεδομένων (π.χ. εγγραφή δεδομένων), ως τύπο `a` δηλώνουμε τον τύπο *πλειάδας μηδεν στοιχείων* `()`. Αυτός ο τύπος έχει μόνο ένα στοιχείο, που γράφεται επίσης `()`. Χρησιμεύει ακριβώς όπως και ο τύπος `void` στις C/C++/Java.

Δε θα απαριθμήσουμε εδώ τις εντολές εισόδου/εξόδου της Haskell. Θα δώσουμε μόνο μερικές για παράδειγμα:

```
getChar ::  IO Char
getLine ::  IO String
putChar ::  Char->IO()
putStr ::  String->IO()
isEOF ::  IO Bool
```

Οι πρώτες δύο είναι είσοδος χαρακτήρα και γραμμής από την κονσόλα αντίστοιχα. Οι δύο επόμενες είναι έξοδοι χαρακτήρα και συμβολοσειράς. Η τελευταία ελέγχει αν η κονσόλα δίνει τέλος αρχείου.

Σαν παράδειγμα χρήσης των παραπάνω εντολών, το παρακάτω πρόγραμμα `copy` αντιγράφει την είσοδό του στην έξοδό του:

```
import IO
copy =
 do e<-isEOF
    if e then do return ()
    else do c<-getChar
            putChar c
            copy
```

Η βιβλιοθήκη της Haskell έχει επίσης συναρτήσεις για είσοδο και έξοδο σε αρχεία, όπως π.χ.

```
readFile ::  FilePath -> IO String
writeFile ::  FilePath -> String -> IO ()
appendFile ::  FilePath -> String -> IO ()
```

όπου `FilePath` είναι συνώνυμο του `String`. Υπάρχουν και συναρτήσεις για γραφική διαπροσωπεία (GUI). Δε θα ασχοληθούμε περισσότερο με τις δυνατότητες εισόδου/εξόδου της Haskell.

4 Επισκόπηση

Στις σημειώσεις αυτές είδαμε τα εξής:

- Χρησιμοποιούμε μονάδες για να εμπλουτίσουμε τις συναρτήσεις μας με καινούργια λειτουργικότητα διαχωρισμένη από τη βασική τους λειτουργία
- Μία μονάδα είναι ένας μοναδιαίος κατασκευαστής τύπου `m` συνοδευόμενος από συναρτήσεις `return` και `>>=` που ικανοποιούν συγκεκριμένες ιδιότητες
- Η κλάση μοναδιαίων κατασκευαστών `Monad` της Haskell υπερφορτώνει αυτές τις συναρτήσεις
- Η σύνταξη `do` για τις μονάδες απλοποιεί σημαντικά τον κώδικα
- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Δεν υπάρχει κατάσταση και μία μεταβλητή παίρνει τιμή μόνο μία φορά
- Η βασική χρήση των μονάδων στη Haskell είναι η είσοδος/έξοδος που γίνεται με τη βοήθεια του ΑΤΔ/μονάδας `IO`