Αλγεβρικοί Τύποι

Γιάννης Κασσιός

Στις σημειώσεις αυτές παρουσιάζουμε τους *αλγεβρικούς τύπους (algebraic types)* όπως υποστηρίζονται στη Haskell. Οι αλγεβρικοί τύποι και κυρίως οι *αναδρομικοί* αλγεβρικοί τύποι είναι ένα από τα σημαντικότερα χαρακτηριστικά που εισήγαγε ο συναρτησιακός προγραμματισμός στη θεωρία τύπων. Θα δούμε πώς οι αλγεβρικοί τύποι υποστηρίζονται στη Haskell, τι ρόλους παίζουν, πώς συνυπάρχουν με την υπερφόρτωση και τον πολυμορφισμό, πώς υποστηρίζουν την κατασκευή αναδρομικών δομών ορισμένων από το χρήστη και πώς ένας γενικευμένος κανόνας επαγωγής μπορεί να μας βοηθήσει στις αποδείξεις ορθότητας παρουσία αυτών των τύπων.

1 Εισαγωγή στους Αλγεβρικούς Τύπους

Στους τύπους που έχουμε δει μέχρι τώρα στη Haskell υπάρχουν τα εξής μειονεκτήματα:

- Δεν υπάρχει τρόπος να ορίζουμε δικούς μας τύπους απαρίθμησης (το αντίστοιχο της `enum` σε γλώσσες προγραμματισμού όπως η C/C++).
- Δεν υπάρχει τύπος-*άθροισμα* άλλων τύπων, δηλαδή τύπος του οποίου τα στοιχεία να ανήκουν σε δύο ή περισσότερους άλλους τύπους.
- Ο τρόπος μοντελοποίησης αντικειμένων με λίστες και πλειάδες, όπως τον παρουσιάσαμε στις Σημ. 2, αφήνει έκθετη τη δομή τους.
- Δεν υπάρχει δυνατότητα να ορίσουμε τις *δικές μας αναδρομικές δομές δεδομένων*.

Η λύση σε όλα αυτά τα προβλήματα είναι οι *αλγεβρικοί τύποι (algebraic types)* που θα παρουσιάσουμε εδώ. Ας δούμε λίγο αναλυτικότερα τα δύο τελευταία και πιο σπουδαία προβλήματα.

1.1 Έκθετη Δομή

Η αναπαράσταση δεδομένων με "γυμνές" λίστες ή πλειάδες έχει ένα σημαντικό μειονέκτημα: δεν ξεκαθαρίζει τι αντικείμενο αναπαριστά η κάθε δομή δεδομένων. Έτσι, αν ορίσουμε τον τύπο `Student` ως ένα τύπο πλειάδας:

```
type Student = (String,Int,Bool)
```

όπως κάναμε στις Σημ. 2, τότε δε μπορούμε να ξεχωρίσουμε αν μία πλειάδα αυτού του τύπου `(s,i,b)` αναπαριστά ένα φοιτητή ή ένα άλλο αντικείμενο που τυχαίνει να έχει την ίδια δομή. Θα μπορούσαμε δηλαδή να χρησιμοποιήσουμε κατά λάθος ως φοιτητή ένα αντικείμενο με την ίδια δομή που δεν αναπαριστά φοιτητή ή, αντίστροφα, να χρησιμοποιήσουμε κατά λάθος ένα αντικείμενο που αναπαριστά φοιτητή ως κάποιο άλλο αντικείμενο.

Μια υποπερίπτωση του προβλήματος της έκθετης δομής είναι η προσπάθεια υλοποίησης του ίδιου τύπου με δύο διαφορετικές αναπαραστάσεις, που όμως τυχαίνει να έχουν τον ίδιο τύπο. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να αναπαραστήσουμε τους μιγαδικούς αριθμούς με δύο διαφορετικούς τρόπους, είτε με το πραγματικό και φανταστικό τους μέρος ($x + yj$), είτε με την απόλυτη τιμή τους και τη γωνία τους ($\rho e^{j\theta}$). Το πρόβλημα είναι ότι και οι δύο αναπαραστάσεις έχουν τον ίδιο τύπο:

```
type Complex1 = (Float,Float)
type Complex2 = (Float,Float)
```

που σημαίνει ότι αυτή η αναπαράσταση δε μας επιτρέπει να ξεχωρίσουμε τη μία υλοποίηση από την άλλη.

1.2 Αναδρομικές Δομές

Η Haskell προβλέπει χρήση αναδρομικών τύπων όπως οι φυσικοί αριθμοί και οι λίστες. Ο αναδρομικός ορισμός που κρύβεται πίσω από αυτούς τους τύπους μπορεί να γενικευτεί και να χρησιμοποιηθεί για τον ορισμό καινούριων αναδρομικών τύπων. Αν και έχουμε δει πολλά παραδείγματα αναδρομής στις τιμές, προς το παρόν δεν έχουμε δει κάποιο τέτοιο μηχανισμό στο σύστημα τύπων της Haskell.

2 Κατασκευή Αλγεβρικών Τύπων

Για την κατασκευή ενός αλγεβρικού τύπου χρησιμοποιούμε τη λέξη κλειδί `data` ως εξής:

`data` *όνομα_τύπου* `=` *ορισμός_τύπου*

Το όνομα ενός αλγεβρικού τύπου ξεκινάει από κεφαλαίο γράμμα. Στην Ενότ. 7 θα δούμε ότι μετά το όνομα του τύπου μπορούμε να έχουμε και μία σειρά από μεταβλητές τύπων. Στα παρακάτω θα δούμε τι μπορεί να περιέχεται στον ορισμό του τύπου.

3 Τύποι Απαρίθμησης

Οι απλούστεροι αλγεβρικοί τύποι είναι οι *τύποι απαρίθμησης (enumeration types)*. Στους τύπους απαρίθμησης, δίνουμε όλες τις τιμές που μπορεί να πάρει ο τύπος. Οι τιμές αυτές ξεκινάνε με *κεφαλαίο γράμμα* και διαχωρίζονται με τον τελεστή `|`. Για λόγους που θα φανούν καλύτερα αργότερα, ονομάζονται *κατασκευαστές (constructors)* του τύπου.

Ένας τέτοιος τύπος που ήδη γνωρίζουμε, είναι ο `Bool`:

```
data Bool = True | False
```

Θα μπορούσαμε να ορίσουμε ένα δικό μας τύπο `Weekday` που να απαριθμεί τις μέρες της εβδομάδας ως εξής:

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Αυτούς τους τύπους μπορούμε να τους χρησιμοποιήσουμε τώρα όπως οποιονδήποτε άλλο τύπο. Για παράδειγμα, η παρακάτω συνάρτηση μας λέει ότι είμαστε χαρούμενοι μόνο το σαββατοκύριακο:

```
happy :: Weekday -> Bool
happy Sat = True
happy Sun = True
happy _ = False
```

4 Παράμετροι στους Κατασκευαστές

Οι τύποι απαρίθμησης δεν έχουν ιδιαίτερα ενδιαφέρουσα δομή. Ένας κατασκευαστής αλγεβρικού τύπου μπορεί όμως να δεχτεί και παραμέτρους. Οι τύποι των παραμέτρων που μπορεί να δεχτεί ο κατασκευαστής δηλώνονται αμέσως μετά τον κατασκευαστή στη δήλωση του αλγεβρικού τύπου. Ένας κατασκευαστής που παίρνει 0 παραμέτρους, όπως οι `True`, `False` και οι κατασκευαστές `Mon`, `Tue`,... που ορίσαμε πιο πάνω, ονομάζεται *μηδενιαίος (nullary)*. Με μία παράμετρο έχουμε *μοναδιαίους (unary)* κατασκευαστές, με δύο *δυαδικούς (binary)*, με *τρεις τριαδικούς (ternary)* κ.ο.κ.

Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να φτιάξουμε τον τύπο `Student` που δηλώσαμε στις Σημ. 2 ως αλγεβρικό τύπο. Μπορούμε να ορίσουμε έναν κατασκευαστή `St` που να παίρνει τρεις παραμέτρους: το όνομα, την ηλικία και το φύλο του κάθε σπουδαστή:

```
data Student = St String Int Bool
```

Τώρα, για να "κατασκευάσουμε" μία τιμή τύπου `Student`, πρέπει να δώσουμε στον κατασκευαστή `St` τις τρεις παραμέτρους που ζητάει:

```
St "Nikos" 20 False
```

Ας δούμε ένα παράδειγμα μίας συνάρτησης που παίρνει ως παράμετρο ένα αντικείμενο τύπου `Student` και επιστρέφει μία συμβολοσειρά καλωσορίσματος ως εξής: "Meet (όνομα). He/She is (ηλικία) years old". Για να το κάνουμε αυτό, χρησιμοποιούμε το μηχανισμό ταιριάσματος μοτίβων της Haskell *που επιδέχεται κατασκευαστές αλγεβρικών τύπων*:

```
introduce :: Student -> String
introduce (St name age sex)
 =    "Meet " ++ name ++ ".  " ++ (if sex then "She" else "He")
   ++ " is " ++ show (age) ++ " years old."
```

Για παράδειγμα, η παρακάτω ερώτηση (που δημιουργεί μία τιμή `Student` και την περνάει στην `introduce`):

```
introduce (St "Georgia" 25 True)
```

θα έχει ως αποτέλεσμα:

```
"Meet Georgia.  She is 25 years old."
```

Βλέπουμε λοιπόν ότι μπορούμε να φτιάξουμε δεδομένα σχετικά με μία οντότητα και να χρησιμοποιήσουμε ένα κατασκευαστή για να τα πακετάρουμε σε μία τιμή αλγεβρικού τύπου. Ο κατασκευαστής πλέον λειτουργεί ως μία "ετικέττα" που μας δείχνει τι ακριβώς πληροφορία αναπαριστούν τα δεδομένα. Έτσι δεν είναι δυνατό να χειριστούμε ένα αντικείμενο `Student` κατά λάθος ως μία τυχαία πλειάδα τύπου `(String, Int, Bool)` ή ως ένα άλλο αντικείμενο που τυχαίνει να έχει την ίδια δομή. Ούτε είναι δυνατό να χειριστούμε ένα τυχαίο αντικείμενο που έχει αυτή τη δομή ως αντικείμενο `Student`. Για παράδειγμα, η συνάρτηση `introduce` που ορίσαμε πιο πάνω δεν πρόκειται να χρησιμοποιηθεί για τιμές που δεν ανήκουν στον τύπο `Student`. Αυτό σημαίνει ότι οι αλγεβρικοί τύποι και οι κατασκευαστές λύνουν το πρόβλημα της έκθετης δομής που παρουσιάσαμε στην Ενότ. 1.1.

Το άλλο σημαντικό που πρέπει να κρατήσουμε από το παράδειγμά μας είναι ότι ο μηχανισμός ταιριάσματος μοτίβων (pattern matching) της Haskell επιδέχεται κατασκευαστές, έτσι ώστε να μπορούμε να "ξεπακετάρουμε" τα δεδομένα ενός αντικειμένου αλγεβρικού τύπου, όπως στον ορισμό της `introduce` παραπάνω.

Το θέμα με την έκθετη δομή επαναλαμβάνεται και μέσα στους αλγεβρικούς τύπους. Θα μπορούσαμε για παράδειγμα να αντικαταστήσουμε τον γενικό τύπο `Bool` με ένα τύπο πιο εξειδικευμένο στην περίπτωση:

```
data Gender = Male | Female
data Student = St String Int Gender
```

ή, ακόμα περισσότερο,

```
data Gender = Male | Female
data Age = Yrs Int
data Name = Nm String
data Student = St Name Age Gender
```

αν και ένας τέτοιος σχεδιασμός μπορεί να χαρακτηριστεί υπερβολικός.

5 Εναλλακτικοί Κατασκευαστές

Οι αλγεβρικοί τύποι μπορούν να χρησιμοποιήσουν κατασκευαστές με παραμέτρους και τον τελεστή `|` για να δημιουργήσουν εναλλακτικές περιπτώσεις για τον οριζόμενο αλγεβρικό τύπο. Για παράδειγμα, έστω ότι θέλουμε να φτιάξουμε έναν απλό τύπο για

γεωμετρικά σχήματα. Αυτός ο τύπος μπορεί να οριστεί ως εξής: ένα σχήμα μπορεί να είναι ένα ορθογώνιο ή ένας κύκλος. Στην περίπτωση του ορθογωνίου, θα πρέπει να δώσουμε τις δύο διαστάσεις του, ενώ για τον κύκλο πρέπει να δώσουμε την ακτίνα του. Ο ορισμός είναι:

```
data Shape = Rectangle Float Float | Circle Float
```

Μία συνάρτηση που υπολογίζει το εμβαδό ενός σχήματος είναι:

```
area :: Shape -> Float
area (Rectangle a b) = a*b
area (Circle r) = pi*r^2
```

Βλέπουμε πώς χρησιμοποιούμε το μηχανισμό ταιριάσματος μοτίβων για να ξεχωρίσουμε τις δύο περιπτώσεις.

Η περίπτωση της διαφορετικής υλοποίησης μιγαδικών αριθμών που αναφέραμε στην Ενότ. 1.1, μπορεί να λυθεί ως εξής:

```
data Complex = CompXY Float Float | CompRhTh Float Float
```

Τώρα, κάθε ζεύγος πραγματικών αριθμών που χρησιμοποιούμε για την αναπαράσταση μιγαδικών, φέρει και τον κατασκευαστή του, που χρησιμεύει ως "ετικέττα" για να ξεχωρίσουμε ποια αναπαράσταση χρησιμοποιείται. Για παράδειγμα, η συνάρτηση νόρμας μπορεί να γραφτεί, ξεχωρίζοντας τις δύο περιπτώσεις, ως εξής:

```
norm :: Complex -> Float
norm (CompXY x y) = sqrt(x^2 + y^2)
norm (CompRhTh rho _) = rho
```

6 Αλγεβρικοί Τύποι ως Στιγμιότυπα Κλάσεων

Μπορούμε να ορίσουμε τους αλγεβρικούς τύπους ως στιγμιότυπα κλάσεων, δικών μας ή της Haskell.

Για παράδειγμα, ένας τρόπος να ορίσουμε συνάρτηση ισότητας στον τύπο `Shape` είναι να θεωρήσουμε ότι δύο σχήματα είναι ίσα αν και μόνον αν είναι και τα δύο ορθογώνια με τις ίδιες διαστάσεις ή είναι και τα δύο κύκλοι με την ίδια ακτίνα:

```
instance Eq Shape where
  Rectangle a b == Rectangle c d = (a==c && b==d) || (a==d && b==c)
  Circle r == Circle q = r==q
  _ == _ = False
```

Φυσικά, δεν είμαστε υποχρεωμένοι να δώσουμε ένα τέτοιο ορισμό. Για παράδειγμα, θα μπορούσαμε να θεωρήσουμε ίσα δύο σχήματα αν και μόνον αν έχουν το ίδιο εμβαδόν:

```
instance Eq Shape where s == t = area s == area t
```

Σε αυτήν την περίπτωση, η σύγκριση πχ.

```
Rectangle 4 pi == Circle 2
```

θα επιστρέψει `True`.

Ο πιο προφανής τρόπος να ορίζουμε ισότητα δύο αλγεβρικών τύπων είναι να απαιτούμε ο κατασκευαστής τους και τα ορίσματά του να είναι ίσα. Παίρνοντας το παράδειγμα τύπων απαρίθμησης, ο ορισμός ισότητας στον τύπο `Bool` δίνεται από:

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
```

Όμοια, μπορούμε να ορίσουμε και εμείς τον τελεστή ισότητας για τον τύπο `Weekday` ως εξής:

```
instance Eq Weekday where
  Mon == Mon = True
  Tue == Tue = True
  Wed == Wed = True
  Thu == Thu = True
  Fri == Fri = True
  Sat == Sat = True
  Sun == Sun = True
  _ == _ = False
```

Δε χρειάζεται να τονίσουμε πόσο άχαρο είναι να γράφουμε τη συνάρτηση ισότητας με αυτόν τον τρόπο. Αναγκαζόμαστε να γράψουμε $n + 1$ ορισμούς (όπου n ο αριθμός των κατασκευαστών του αλγεβρικού τύπου) για να ορίσουμε τελικά τον πιο προφανή τύπο ισότητας που θα μπορούσαμε να έχουμε. Είναι καλό μία γλώσσα να παρέχει σύνταξη που να αποφεύγει τη συγγραφή τόσο μεγάλης ποσότητας κώδικα για τέτοιου είδους κοινοτοπίες.

Στη Haskell αυτό γίνεται με τη λέξη κλειδί `deriving`. Η `deriving` τοποθετείται στον ορισμό του αλγεβρικού τύπου και ακολουθείται από ένα σύνολο κλάσεων της Haskell (οι κλάσεις αυτές χωρίζονται με κόμμα και μπαίνουν μέσα σε παρένθεση αν είναι περισσότερες από μία). Ο αλγεβρικός τύπος είναι στιγμιότυπο όλων των κλάσεων που υπάρχουν μέσα στη δήλωση `deriving`. Η υλοποίηση των υπερφορτωμένων συναρτήσεων αυτών των κλάσεων είναι η προφανέστερη δυνατή.

Για παράδειγμα, μπορούμε να γράψουμε τον ορισμό της `Weekday` ως εξής:

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving Eq
```

που είναι ισοδύναμος με αυτόν που είχαμε γράψει πιο πάνω. Η δήλωση `deriving Eq` σημαίνει ότι κάθε κατασκευαστής της `Weekday` ισούται μόνο με τον εαυτό του.

Ομοίως θα μπορούσαμε να είχαμε δώσει τον εξής ορισμό:

```
data Shape = Rectangle Float Float | Circle Float
  deriving Eq
```

ο οποίος θα όριζε την `(==)` να επιστρέφει `True` αν και μόνον αν τα δύο ορίσματά της προέρχονται από τον ίδιο κατασκευαστή και τα ίδια ορίσματα, δηλαδή σα να είχαμε γράψει

```
instance Eq Shape where
  Rectangle a b == Rectangle c d = a==b && c==d
  Circle r == Circle q = r==q
  _ == _ = False
```

Παρατηρήστε ότι ο ορισμός της `(==)` που δίνει η `deriving Eq` δεν είναι ισοδύναμος με κανέναν από τους δύο εναλλακτικούς ορισμούς που δώσαμε πιο πάνω (π.χ. ο πρώτος ορισμός θεωρεί ότι τα `Rectangle 1 2` και `Rectangle 2 1` είναι ίσα). Αυτό μας δείχνει ότι δεν είμαστε υποχρεωμένοι ούτε να χρησιμοποιήσουμε τη `deriving`, ούτε να ορίσουμε ισότητα που να έχει απαραίτητα κάποια σχέση με αυτή που θα όριζε η `deriving`.

Εκός από την κλάση `Eq` μπορούμε να βάλουμε μέσα στη δήλωση `deriving` ακόμα τις εξής κλάσεις:

- Κλάση `Ord` (προϋποτείθεται ο τύπος να είναι στιγμιότυπο και της `Eq`): Σε αυτήν την περίπτωση δίνονται τελεστές σύγκρισης στον τύπο. Οι κατασκευαστές θεωρούνται ότι έχουν δηλωθεί από το μικρότερο στο μεγαλύτερο, π.χ. στη δήλωση:

  ```
  data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving (Eq,Ord)
  ```

  οι εκφράσεις `Mon<Tue` και `Fri>=Wed` επιστρέφουν `True`.

- Κλάση `Show`: Η προκαθορισμένη συνάρτηση `show` τυπώνει τον κατασκευαστή του ορίσματός της και μετά καλεί τη `show` των ορισμάτων του. Για παράδειγμα, στη δήλωση:

  ```
  data Shape = Rectangle Float Float | Circle Float
    deriving Show
  ```

  αν ζητήσουμε από το διερμηνέα να αποτιμήσει την έκφραση `Circle 2`, θα μας απαντήσει `Circle 2.0` (τυπώνει το όνομα του κατασκευαστή και καλεί την `(show::(Float->String))2` για να τυπώσει `2.0`)

- Κλάση `Read`: Υλοποιεί την αναμενόμενη αντίστροφη της `show` που ορίζει η `Show`.
- Κλάση `Enum`: Επιτρέπει τη χρήση του τελεστή `[x .. y]` για να αναπαραστήσει λίστες του τύπου. Η σειρά των κατασκευαστών είναι πάλι, όπως και στην `Ord`, αυτή με την οποία έχουν δηλωθεί. Για παράδειγμα, στη δήλωση:

  ```
  data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving (Eq,Ord,Show,Enum)
  ```

  η αποτίμηση της έκφρασης `[Mon .. Fri]` θα δώσει

  ```
  [Mon,Tue,Wed,Thu,Fri,Sat,Sun]
  ```

7 Πολυμορφισμός

Ένας αλγεβρικός τύπος μπορεί να είναι πολυμορφικός. Σε αυτήν την περίπτωση, ο τύπος επιδέχεται μεταβλητές τύπων ως τυπικά ορίσματα. Αυτά τα τυπικά ορίσματα μπορούν να εμφανίζονται ως ορίσματα στη δήλωση των κατασκευαστών του τύπου. Θα δούμε τώρα δύο απλά παραδείγματα χρήσης μη αναδρομικών πολυμορφικών αλγεβρικών τύπων.

7.1 Χειρισμός Λαθών

Ένας υπολογισμός που τερματίζει μπορεί να μην επιστρέφει τιμή λόγω λάθους. Για παράδειγμα, μπορεί να προκαλείται διαίρεση με το μηδέν ή εφαρμογή μίας συνάρτησης της οποίας κανένα μοτίβο δεν ταιριάζει στην πραγματική παράμετρο. Θα μπορούσαμε να αφήσουμε τη Haskell να χειριστεί το λάθος τερματίζοντας τον υπολογισμό. Παρ'όλα αυτά μπορεί να θέλουμε να συνεχιστεί ο υπολογισμός, χειριζόμενοι μόνοι μας το λάθος.

Ο αλγεβρικός τύπος `Maybe` κάνει ακριβώς αυτό, προσφέροντας ένα τρόπο χειρισμού λαθών μέσα σε ένα υπολογισμό, όμοιο με το μηχανισμό *εξαιρέσεων (exceptions)*. Ο ορισμός του είναι:

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Read, Show)
```

Βλέπουμε ότι ο `Maybe` είναι πολυμορφικός: μπορεί να χρησιμοποιηθεί σε υπολογισμο οποιουδήποτε τύπου `a`. Η τιμή `Nothing` αντιστοιχεί στην περίπτωση λάθους, ενώ η τιμή `Just x` σημαίνει ότι ο υπολογισμός δεν έχει λάθος και το αποτέλεσμα είναι `x`.

Μία συνάρτηση που μπορεί να προκαλέσει λάθος είναι η διαίρεση ακεραίων. Μπορούμε να τη γράψουμε χρησιμοποιώντας `Maybe` ως εξής:

```
mydiv :: Int -> Int -> Maybe Int
mydiv x 0 = Nothing
mydiv x y = Just (x`div`y)
```

Ο καθορισμός τύπου δηλώνει ότι η `mydiv` *ίσως (maybe)* επιστρέψει ακέραιο, αλλά υπάρχει και η πιθανότητα λάθους. Αυτό είναι αντίστοιχο με τη δήλωση `throws` της Java, η οποία ενημερώνει τους χρήστες μίας μεθόδου ότι η μέθοδος αυτή μπορεί να ρίξει εξαίρεση. Σε περίπτωση διαίρεσης με το 0, η `mydiv` επιστρέφει `Nothing`. Αυτό είναι αντίστοιχο με τη ρίψη εξαίρεσης, που στη Java γίνεται με την εντολή `throw`.

Η χρήση της `mydiv` μπορεί να προκαλέσει λάθος. Μία συνάρτηση που χρησιμοποιεί την `mydiv` μπορεί να διοχετεύσει αυτό το λάθος στο αποτέλεσμά της ώστε να το χειριστεί κάποια άλλη συνάρτηση:

```
myfunc :: Int -> Int -> Maybe Int
myfunc x y =
  if division == Nothing then Nothing
  else Just (z + 1)
  where division = x`mydiv`y
        Just z = division
```

Επειδή αυτός ο τρόπος διοχέτευσης του λάθους είναι μάλλον άκομψος, μπορούμε να φτιάξουμε μία συνάρτηση `liftMaybe` της οποίας ο ρόλος είναι να πάρει μία συνάρτηση `f` τύπου `a->b` και να τη μετατρέψει σε συνάρτηση `Maybe a->Maybe b`, συμπεριφερόμενη όπως η `f` όταν δεν υπάρχει λάθος και διοχετεύοντας το λάθος όταν αυτό υπάρχει:

```
liftMaybe :: (a->b) -> Maybe a -> Maybe b
liftMaybe g Nothing = Nothing
liftMaybe g (Just x) = Just (g x)
```

Η `myfunc` μπορεί να γραφτεί τώρα ως εξής:

```
myfunc :: Int -> Int -> Maybe Int
myfunc x y = liftMaybe (\z->z+1) (x`mydiv`y)
```

Τέλος, μπορούμε να "παγιδεύσουμε" το λάθος, όπως κάνουμε στη Java με τις δομές `try/catch`. Η Haskell μας παρέχει τη συνάρτηση `maybe` τύπου `a -> (b -> a) -> Maybe b -> a` που κάνει ακριβώς αυτό: υπολογίζει μία έκφραση τύπου `Maybe b`. Σε περίπτωση που δεν υπάρξει λάθος, το αποτέλεσμα διοχετεύεται σε μία συνάρτηση τύπου `b->a` για περαιτέρω υπολογισμό. Σε περίπτωση λάθους μία έκφραση τύπου `a` αποτιμάται και επιστρέφεται:

```
maybe :: a -> (b->a) -> Maybe b -> a
maybe n f Nothing = n
maybe n f (Just x) = f x
```

Τώρα μπορούμε να χρησιμοποιήσουμε τη `mydiv` μέσα σε μία άλλη συνάρτηση που παγιδεύει το λάθος. Σε περίπτωση λάθους, η συνάρτησή μας επιστρέφει π.χ. `-1`, αλλιώς επιστρέφει το αποτέλεσμα της ακέραιας διαίρεσης:

```
mydivcatching :: Int->Int->Int
mydivcatching x y = maybe (-1) id (x`mydiv`y)
```

7.2 Εναλλακτικοί Τύποι

Για κάθε ζεύγος τύπων `a,b` μπορούμε να χρησιμοποιήσουμε έναν τύπο στοιχείων που αντιστοιχούν είτε στον `a` είτε στον `b`. Αυτό γίνεται με τον αλγεβρικό τύπο `Either` της Haskell που ορίζεται ως εξής:

```
data Either a b = Left a | Right b
  deriving (Eq,Ord,Read,Show)
```

Ο ρόλος της `Either` είναι όμοιος με αυτόν της δομής `union` στις C/C++.

Σαν ένα παράδειγμα, θα χρησιμοποιήσουμε την `Either` για να δώσουμε δύο διαφορετικές υλοποιήσεις στη δομή "σύνολα φυσικών αριθμών". Στην πρώτη υλοποίηση, ένα σύνολο αναπαρίσταται από μία λίστα που περιέχει τους φυσικούς αριθμούς που περιέχει το σύνολο. Στη δεύτερη υλοποίηση, ένα σύνολο αναπαρίσταται από μία λίστα αληθοτιμών `l` έτσι ώστε `l!!i` αν και μόνον αν ο `i` είναι μέσα στο σύνολο. Θα υλοποιήσουμε μόνο τη συνάρτηση `element` που επιστρέφει `True` αν και μόνον αν ένα στοιχείο είναι στο σύνολο (για ευκολία χρησιμοποιούμε τη `lSearch` που έχουμε ορίσει πολλές φορές στο παρελθόν -- οποιοσδήποτε από τους ορισμούς μας κάνει):

```
type SetofNats = Either [Int] [Bool]
element :: Int->SetofNats->Bool
element x (Left li) = lSearch x li
element x (Right lb) = lb!!x
```

Ιδού μερικές ερωτήσεις που μπορούμε να κάνουμε στο διερμηνέα, μαζί με τις απαντήσεις τους (οι ερωτήσεις αρχίζουν με > για να ξεχωρίζουν):

```
> element 3 (Left[2,1,2])
False
> element 3 (Left[2,1,2,3])
True
> element 3 (Right[True,False,False,False,True])
False
> element 3 (Right[True,False,False,True])
True
```

8 Αναδρομικοί Τύποι

Ένας *αναδρομικός τύπος (recursive type)* είναι ένας τύπος που εμφανίζεται στον ίδιο τον ορισμό του. Στη Haskell, οι αλγεβρικοί τύποι μπορούν να οριστούν αναδρομικά,

αφού στα ορίσματα ενός κατασκευαστή μπορεί να περιέχεται ο ίδιος ο υπό ορισμό τύπος (φυσικά, υπάρχει και η δυνατότητα αμοιβαίας αναδρομής).

Δύο αναδρομικοί τύποι που ήδη υποστηρίζονται από τη Haskell είναι οι *φυσικοί αριθμοί* και οι *λίστες*. Οι φυσικοί αριθμοί, ικανοποιούν την εξής αναδρομική εξίσωση:

$$\mathbb{N} = \{0\} \cup \{x + 1 | x \in \mathbb{N}\}$$

οπότε και στη Haskell θα μπορούσαμε να τους είχαμε ορίσει, με δύο κατασκευαστές, ως εξής (ασχέτως αν στη Haskell τελικά δεν υλοποιούνται με αυτόν τον τρόπο):

```
data Nat = Zero | Succ Nat
```

Ο αριθμός 2 π.χ. αναπαρίσταται σε αυτόν τον τύπο ως `Succ(Succ Zero)`.

Οι λίστες θα μπορούσαν να αναπαρασταθούν, επίσης με δύο κατασκευαστές, ως εξής:

```
data List a = EmptyList | Cons a (List a)
```

Στη Haskell οι λίστες συμπεριφέρονται ακριβώς έτσι, αλλά με διαφορετική σύνταξη. Ο τύπος `List a` γράφεται `[a]`, ο κατασκευαστής `EmptyList` γράφεται `[]` και ο κατασκευαστής `Cons` γράφεται `(:)`.

Μέσω των αλγεβρικών τύπων, μπορούμε να ορίσουμε δικούς μας αναδρομικούς τύπους. Στη συνέχεια θα δούμε δύο χαρακτηριστικά παραδείγματα και θα συζητήσουμε τη δυνατότητα άπειρων δομών που μας προσφέρει η αναδρομή στους τύπους.

8.1 Δυαδικά Δέντρα

Το κλασικό παράδειγμα αναδρομικής δομής, μετά από τα απλούστερα παραδείγματα των φυσικών αριθμών και των λιστών, είναι τα *δυαδικά δέντρα (binary trees)*. Ένας ορισμός του συνόλου των δυαδικών δέντρων έχει ως εξής. Ένα δυαδικό δέντρο είναι:

- το *κενό δέντρο* (ένα δέντρο χωρίς κόμβους), είτε
- ένας *κόμβος* με μία πληροφορία (κάποιου τύπου) και με δύο δυαδικά δέντρα που ονομάζονται *παιδιά* του κόμβου αυτού.

Σε κείμενα πιο προσανατολισμένα στα μαθηματικά, μπορεί να δείτε ορισμούς που να μην περιλαμβάνουν το "κενό δέντρο" ως δέντρο. Αυτό δυσχεραίνει τον ορισμό της δομής των δυαδικών δέντρων και αποκλείει την περίπτωση που ένα δέντρο δεν περιέχει καθόλου πληροφορία. Είναι καλύτερα να συμπεριλάβουμε στους αλγεβρικούς τύπους μας τετριμμένες περιπτώσεις όπως το μηδέν, την κενή λίστα και το κενό δέντρο.

Στη Haskell, ο παραπάνω αναδρομικός ορισμός γράφεται (μαζί με κάποιες κλάσεις στις οποίες θέλουμε ο τύπος να ανήκει):

```
data BTree a = EmptyBTree | BNode a (BTree a) (BTree a)
  deriving (Eq, Show)
```

Μία συνάρτηση `mapBTree` που κάνει τη δουλειά της `map` σε δυαδικά δέντρα, μπορεί να γραφεί ως εξής:

```
mapBTree :: (a->b) -> BTree a -> BTree b
mapBTree f EmptyBTree = EmptyBTree
mapBTree f (BNode root left right)
  = BNode (f root) (mapBTree f left) (mapBTree f right)
```

8.2 Ορισμός Δομών μίας Γλώσσας Προγραμματισμού

Άλλο ένα κλασικό παράδειγμα αναδρομικών ορισμών είναι ο ορισμός σύνταξης μιας γλώσσας προγραμματισμού ή άλλης τυπικής γλώσσας, όπως αυτή χρησιμοποιείται για μεταγλωττιστές. Για παράδειγμα, οι εκφράσεις μιας απλής γλώσσας προγραμματισμού, η οποία υποστηρίζει μόνο ακεραίους και μερικούς μοναδιαίους και δυαδικούς τελεστές, μπορεί να αναπαρασταθούν στη Haskell από τον εξής αλγεβρικό τύπο:

```
data Expr =  Constant Int
           | Variable String
           | UnaryExp UnOp Expr
           | BinaryExp BinOp Expr Expr

data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div
```

Η έκφραση

```
-b*(c+a/5)*2
```

(με βάση τους κανόνες προτεραιότητας και προσεταιριστικότητας της Haskell), αναπαρίσταται ως εξής:

```
BinaryExp Mult
          (BinaryExp Mult
                     (UnaryExp UMinus (Variable "b"))
                     (BinaryExp Plus
                                (Variable "c")
                                (BinaryExp Div
                                           (Variable "a")
                                           (Constant 5)
                                )
                     )
          )
          (Constant 2)
```

Ένας διερμηνέας για αυτήν τη γλώσσα `eval` θα παίρνει μία λίστα συσχετίσεων μεταβλητών με τιμές και μία έκφραση τύπου `Expr` και θα παράγει μία τιμή τύπου `Int`. Η λίστα συσχετίσεων ονομάζεται *περιβάλλον* και θα μπορούσε να έχει για παράδειγμα τύπο `[(String,Int)]`. Ένα κομμάτι της υλοποίησης του διερμηνέα έχει ως εξής:

```
eval env (Constant x) = x
...
eval env (UnaryExp UMinus e) = - eval env e
...
eval env (BinaryExp Plus e1 e2) = eval env e1 + eval env e2
...
```

Η υλοποίηση ενός πλήρους διερμηνέα αφήνεται ως άσκηση.

8.3 Άπειρες Δομές

Όπως και με τις λίστες, όλοι οι αναδρομικοί τύποι μπορούν να χρησιμοποιηθούν για ορισμό άπειρων δομών, χρησιμοποιώντας αναδρομή τιμών. Για παράδειγμα, το παρακάτω δυαδικό δέντρο είναι άπειρο:

```
inftree = inftreestart 0
  where
  inftreeaux n = BTree n (inftreeaux (2*n+1)) (inftreeaux (2*n+2))
```

Οι κανόνες οκνηρής αποτίμησης και η σχετική μεθοδολογία που είδαμε στις λίστες ισχύουν και εδώ.

9 Αποδείξεις Ορθότητας για Αλγεβρικούς Τύπους

Οι περιπτώσεις επαγωγής που έχουμε δει μέχρι στιγμής, μπορούν να αναχθούν σε ένα γενικότερο κανόνα που ταιριάζει σε όλους τους αναδρομικά οριζόμενους τύπους. Η γενικευμένη αυτή επαγωγή οφείλεται στους Knaster [Kna28] και Tarski [Tar55].

Έστω σύνολο **U** και συνάρτηση

$$F \in \mathbf{U} \to \mathbf{U}$$

που είναι *μονότονη*, δηλ. για κάθε υποσύνολα X, Y του **U** είναι[1]

$$\mathrm{X} \sqsubseteq Y \;\Rightarrow\; F X \sqsubseteq F\ Y$$

Το θεώρημα Knaster-Tarski μας λέει ότι:

- Υπάρχουν λύσεις στην εξίσωση $X = F X$ (ως προς X). Οι λύσεις αυτές λέγονται *σταθερά σημεία (fixed points)* της F.
- Υπάρχει *ελάχιστο σταθερό σημείο (least fixed point)* της F, δηλαδή ένα σύνολο X_0 που περιέχεται σε όλα τα υπόλοιπα σταθερά σημεία:

$$S = F\ S \;\Rightarrow\; X_0 \sqsubseteq S$$

[1] Για κάποιο λόγο, το πρόγραμμα στοιχειοθέτησης που χρησιμοποιήθηκε για αυτές τις σημειώσεις αρνήθηκε πεισματικά να τυπώσει το συνηθισμένο σύμβολο του υποσυνόλου. Μέχρι να διορθωθεί το πρόβλημα, θα χρησιμοποιούμε το σύμβολο $\sqsubseteq$.

Το ελάχιστο σταθερό σημείο X_0 γράφεται και μF. Αν η F δίνεται με τη μορφή λ-έκφρασης, τότε το ελάχιστο σταθερό σημείο της γράφεται αντικαθιστώντας το λ με το μ στην έκφραση αυτή.

- Το ελάχιστο σταθερό σημείο της F είναι και το ελάχιστο σύνολο X ώστε $F\ X \sqsubseteq X$

Ας αναπαράγουμε τώρα τον κανόνα επαγωγής για φυσικούς αριθμούς, χρησιμοποιώντας το θεώρημα Knaster-Tarski. Έστω ότι $\mathbf{U} = \mathbb{R}$. Οι φυσικοί αριθμοί, όπως είδαμε, ικανοποιούν την εξίσωση

$$\mathbb{N} = F\ \mathbb{N}$$

όπου

$$F = \lambda X.\{0\} \cup \{x + 1 \mid x \in X\}$$

Το θεώρημα Knaster-Tarski μας λέει ότι υπάρχουν λύσεις στην αναδρομική εξίσωση, καθώς η F είναι μονότονη. Το ελάχιστο σταθερό σημείο της F ορίζεται να είναι οι φυσικοί αριθμοί:

$$\mathbb{N} = \mu X.\{0\} \cup \{x + 1 \mid x \in X\}$$

ή, πιο σύντομα,

$$\mathbb{N} = \mu F$$

Παρατηρήστε ότι το γεγονός ότι το $\mathbb{N}$ είναι το ελάχιστο σταθερό σημείο της F είναι πολύ σημαντικό. Υπάρχουν πολλά σταθερά σημεία της F, πχ. το σύνολο των ακεραίων αριθμών $\mathbb{Z}$ ή το σύνολο των μη αρνητικών πραγματικών αριθμών κτλ.

Ας αναπαραστήσουμε τώρα μία ιδιότητα αριθμών, ως ένα υποσύνολο P του $\mathbf{U}$ όλων των αριθμών που ικανοποιούν αυτήν την ιδιότητα. Για να αποδείξουμε ότι *όλοι οι φυσικοί αριθμοί ικανοποιούν την ιδιότητα* θα πρέπει να δείξουμε ότι $\mathbb{N} \sqsubseteq P$. Το θεώρημα Knaster-Tarski μας λέει ότι αρκεί να αποδείξουμε ότι το P είναι σταθερό σημείο της F, δηλαδή:

$$F\ P \sqsubseteq P$$

Αυτό μας δίνει, από τον ορισμό της F:

$$\forall y \in \{0\}.\ y \in P$$

$$\forall y \in \{x + 1 \mid x \in P\}.\ y \in P$$

ή ισοδύναμα

$$0 \in P$$

$$\forall x \in P.\ x + 1 \in P$$

που είναι και τα αξιώματα της επαγωγής φυσικών αριθμών: για να αποδείξουμε ότι μία ιδιότητα ισχύει για όλους τους φυσικούς αριθμούς, αποδεικνύουμε ότι ισχύει για το 0 και μετά, υποθέτοντας ότι ισχύει για κάποιο x, αποδεικνύουμε ότι ισχύει για $x + 1$.

Υποδεικνύουμε για μία ακόμα φορά ότι η επαγωγή ισχύει μόνο για το *ελάχιστο σταθερό σημείο* $\mathbb{N}$ και όχι για άλλα σταθερά σημεία π.χ. $\mathbb{N} \cup \{\text{-}1\}$.

Στις λίστες, η κατάσταση είναι η ίδια: αν $\mathbf{U} = \mathbb{Z}^\omega$ (δηλ. το σύνολο των πεπερασμένων και άπειρων λιστών ακεραίων), τότε ο αναδρομικός ορισμός των *πεπερασμένων* λιστών ακεραίων είναι

$$\mathbb{Z}^\star = \mu L.\, \{[]\} \cup \{h : t \mid h \in \mathbb{Z}, t \in L\}$$

που δίνει την επαγωγή πεπερασμένων λιστών. Η επαγωγή δεν ισχύει για τις άπειρες λίστες, γιατί το σύνολο άπειρων και πεπερασμένων λιστών $\mathbb{Z}^\omega$ *δεν* είναι το ελάχιστο σταθερό σημείο του παραπάνω ορισμού.

Για να βρούμε έναν νόμο επαγωγής για οποιονδήποτε αλγεβρικό τύπο κοιτάμε τους κατασκευαστές του. Για κάθε κατασκευαστή χωρίς αναδρομή, έχουμε μία συνθήκη *βάσης* επαγωγής, ενώ για κάθε κατασκευαστή με αναδρομή, έχουμε μία συνθήκη *βήματος* επαγωγής.

Στους φυσικούς, οι κατασκευαστές είναι ο μηδενιαίος `0` και ο μοναδιαίος `(+1)` ο οποίος έχει αναδρομή (δηλαδή παίρνει σαν όρισμα φυσικό αριθμό). Επομένως έχουμε μία συνθήκη βάσης

```
p 0
```

και μία συνθήκη βήματος

$$\texttt{p x} \Rightarrow \texttt{p(x+1)}$$

Στις λίστες, έχουμε ομοίως μία βάση (από τον κατασκευαστή `[]`) και ένα βήμα (από τον κατασκευαστή `(:)`). Η βάση είναι

```
p[]
```

και το βήμα

$$\texttt{p t} \Rightarrow \texttt{p(h:t)}$$

Σε αλγεβρικούς τύπους χωρίς αναδρομή, η επαγωγή γίνεται τετριμμένη (απόδειξη για κάθε περίπτωση ξεχωριστά). Για παράδειγμα, ο τύπος `Maybe Int` επιδέχεται τετριμμένο κανόνα επαγωγής με δύο βάσεις:

```
p(Just x)
```

και

```
p Nothing
```

Στον τύπο των δυαδικών δέντρων, έχουμε μία βάση:

```
p EmptyBTree
```

και ένα βήμα, αλλά με δύο επαγωγικές υποθέσεις, καθώς η αναδρομή του κατασκευαστή `BNode` είναι διπλή:

$$\texttt{p l \&\& p r} \Rightarrow \texttt{p(BNode x l r)}$$

Και πάλι θα επαναλάβουμε ότι η επαγωγή *δεν ισχύει* για τα άπειρα δέντρα. Όπως και στις λίστες, θα χρησιμοποιούμε τα $_{\texttt{d}}$ και $_{\texttt{f}}$ για ορισμένα και πεπερασμένα δέντρα αντίστοιχα.

9.1 Κόμβοι και Ύψος Δυαδικού Δέντρου

Για επίδειξη της επαγωγής σε δυαδικά δέντρα, θα αποδείξουμε ένα πολύ απλό θεώρημα, ότι ο αριθμός κόμβων ενός δυαδικού δέντρου είναι αυστηρά άνω φραγμένος από την έκφραση `2^h` όπου `h` το ύψος του δέντρου. Το παράδειγμα χρησιμοποιεί τις συναρτήσεις:

```
height :: BTree a -> Int
height EmptyBTree = 0
height (BNode _ l r) = max(height l)(height r) + 1

nodes :: BTree a -> Int
nodes EmptyBTree = 0
nodes (BNode _ l r) = nodes l + nodes r + 1
```

Θα αποδείξουμε την έκφραση

```
nodes t < 2^(height t)
```

για κάθε `t::(BTree a)`$_{\texttt{df}}$ με επαγωγή.

Βάση επαγωγής:

```
nodes EmptyBTree < 2^(height EmptyBTree) = 0<1 = True
```

Βήμα επαγωγής. Οι επαγωγικές υποθέσεις είναι:

```
nodes l < 2^(height l)
nodes r < 2^(height r)
```

Τώρα το βήμα αποδεικνύεται ως εξής:

```
    2^(height (BNode _ l r))
=   2^(max(height l)(height r)+1)
=   2*2^(max(height l)(height r))
=   2^(max(height l)(height r)) + 2^(max(height l)(height r))
>=  2^(height l) + 2^(height r)                                  επαγ.υπόθ.
>=  nodes l + 1 + 2^(height r)                                   επαγ.υπόθ.
>   nodes l + 1 + nodes r
=   nodes(BNode _ l r)
```

10 Επισκόπηση

Σε αυτές τις σημειώσεις είδαμε τα εξής:

- Οι αλγεβρικοί τύποι της Haskell:

  - δίνουν τη δυνατότητα για δημιουργία τύπων απαρίθμησης και αθροισμάτων τύπων.
  - λύνουν το πρόβλημα της *έκθετης δομής* που έχουν οι αναπαραστάσεις με "γυμνές" λίστες και πλειάδες. Με αυτόν τον όρο εννοούμε το ότι τέτοιες αναπαραστάσεις δεν ξεκαθαρίζουν το αντικείμενο το οποίο αναπαριστούν τα γυμνά δεδομένα.
  - επιτρέπουν την εισαγωγή από το χρήστη νέων *αναδρομικών τύπων*.

- Οι αλγεβρικοί τύποι ορίζονται με τη λέξη-κλειδί `data` και με τη βοήθεια ενός ή περισσότερων *κατασκευαστών* οι οποίοι παίρνουν έναν αριθμό παραμέτρων.
- Οι *τύποι απαρίθμησης* είναι αλγεβρικοί τύποι των οποίων οι κατασκευαστές παίρνουν 0 παραμέτρους.
- Οι κατασκευαστές λύνουν το πρόβλημα της έκθετης δομής.
- Τα μοτίβα της Haskell επιδέχονται κατασκευαστές.
- Η λέξη-κλειδί `deriving` διευκολύνει τη δήλωση αλγεβρικών τύπων ως στιγμιότυπα συγκεκριμένων κλάσεων της Haskell.
- Οι αλγεβρικοί τύποι μπορεί να είναι πολυμορφικοί. Είδαμε τους εξής αλγεβρικούς τύπους που υποστηρίζει η Haskell:
  - ο τύπος `Maybe` χρησιμοποιείται για το χειρισμό λαθών υπολογισμού.
  - ο τύπος `Either` χρησιμοποιείται για άθροισμα τυπών.
- Οι αλγεβρικοί τύποι επιδέχονται *αναδρομή*. Οι φυσικοί αριθμοί και οι λίστες είναι αναδρομικοί τύποι που ήδη υποστηρίζονται. Εμείς είδαμε την κατασκευή
  - δυαδικών δέντρων
  - δέντρων δομών σύνταξης γλωσσών προγραμματισμού
- Η εισαγωγή τετριμμένων περιπτώσεων (όπως το μηδέν, η κενή λίστα και το κενό δέντρο) είναι σημαντική στους αναδρομικούς τύπους αλλά και στο σχεδιασμό τυπικών θεωριών και γλωσσών γενικότερα:
  - επιτρέπει την αναπαράσταση μηδενικής πληροφορίας.
  - διευκολύνει τον αναδρομικό ορισμό.
- Όπως και στις λίστες, μπορούμε να κατασκευάσουμε άπειρες δομές κάθε αναδρομικού τύπου.
- Οι πεπερασμένες δομές κάθε αναδρομικού τύπου επιδέχονται επαγωγή. Όλες οι επαγωγές είναι συνέπεια του θεωρήματος Knaster-Tarski.
- Για να βρούμε την επαγωγή κάθε αναδρομικού τύπου που κατασκευάζουμε, φτιάχνουμε:
  - μία βάση επαγωγής για κάθε μη αναδρομικό κατασκευαστή
  - ένα βήμα επαγωγής για κάθε αναδρομικό κατασκευαστή

Αναφορές

[Kna28] B. Knaster. Un théorème sur les fonctions d'ensembles. *Ann. Soc. Polon. Math*, 6:133--134, 1928.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285--309, 1955.