Οκνηρή Αποτίμηση και Αποδείξεις Ορθότητας

Γιάννης Κασσιός

Στις σημειώσεις αυτές, έχουμε δύο διαφορετικά θέματα, την *οκνηρή αποτίμηση* και τις *αποδείξεις ορθότητας*.

Στην Ενότ. 1 ασχολούμαστε με την *οκνηρή αποτίμηση* της Haskell. Εξηγούμε γιατί η οκνηρή αποτίμηση είναι σημαντικό χαρακτηριστικό, δίνουμε τους κανόνες οκνηρής αποτίμησης που ακολουθεί η Haskell, εξετάζουμε την ιδέα του *υπολογισμού οδηγούμενου από τα δεδομένα*, τις άπειρες λίστες και την ιδέα του *dovetailing* για το χειρισμό περισσότερων από μίας άπειρων λιστών.

Στην Ενότ. 2 κάνουμε μία εισαγωγική αναφορά στην *απόδειξη ορθότητας* των προγραμμάτων και πώς αυτή θα μπορούσε να εφαρμοστεί στη Haskell. Συζητάμε για τη διαφορά των αποδείξεων από το testing, εισάγουμε συμβολισμό για να χειριστούμε την περίπτωση υπολογισμών που δεν τερματίζουν, εισάγουμε τις τεχνικές απόδειξης με επαγωγή για πεπερασμένους αριθμούς και για πεπερασμένες λίστες και δίνουμε σχετικά παραδείγματα. Έμφαση δίνεται στην ιδέα της *ισχυροποίησης* του θεωρήματος που προσπαθούμε να αποδείξουμε με επαγωγή, αλλά και στην ακαταλληλότητα της επαγωγής για άπειρες δομές.

1 Οκνηρή Αποτίμηση

Στην *οκνηρή αποτίμηση (lazy evaluation)* μία δομή δεδομένων δεν κατασκευάζεται ολόκληρη όταν ορίζεται ή όταν περνάει σαν όρισμα σε μία συνάρτηση. Αντίθετα, μέρη της δομής αυτής κατασκευάζονται μόνο όταν χρειάζονται σε μία αποτίμηση. Με αυτόν τον τρόπο αποφεύγεται η κατασκευή κομματιών δομών που δε χρειάζονται. Το αποτέλεσμα είναι ότι ο ορισμός μίας δομής δεδομένων να μπορεί να γίνει εντελώς ανεξάρτητα από τις χρήσεις της. Ο κατασκευαστής της δομής δεν ανησυχεί για το μέγεθος της δομής (δομές άπειρου μεγέθους είναι δυνατές χάρη στην οκνηρή αποτίμηση), ενώ ο χρήστης δε χρειάζεται να κατασκευάσει τα κομμάτια της δομής που χρησιμοποιούνται: η δομή μπορεί να χρησιμοποιηθεί σα να έχει ήδη κατασκευαστεί ολόκληρη.

1.1 Κανόνες Οκνηρής Αποτίμησης στη Haskell

Σε γενικές γραμμές, η αποτίμηση στη Haskell γίνεται ως εξής: η εφαρμογή μίας συνάρτησης `f x` οδηγεί στην αποτίμηση του `f` και στο ταίριασμα της έκφρασης `x` με τα μοτίβα των ορισμών της `f`. Το ταίριασμα μοτίβων μπορεί να προκαλέσει (μερική) αποτίμηση της `x`. Η αποτίμηση αυτή γίνεται έως ότου η Haskell καταλάβει αν το `x` ταιριάζει στο μοτίβο.

Για παράδειγμα, ας υποθέσουμε ότι έχουμε τους εξής ορισμούς:

```
head (x:_) = x
l = [n^2 | n<-[1..100]]
```

Η αποτίμηση της `head l` θα οδηγήσει στη μερική αποτίμηση της `l`, ώστε να ταιριάξει με το μοτίβο `x:_`. Η αποτίμηση της `l` δηλαδή θα εξασφαλίσει ότι η λίστα δεν είναι κενή, υπολογίζοντας μόνο το πρώτο στοιχείο της:

```
l = 1:[n^2 | n<-[2..100]]
```

Στη συνέχεια, η μεταβλητή `x` θα πάρει την τιμή `1`, η οποία και θα επιστραφεί, χωρίς περαιτέρω κατασκευή της `l`.

Μετά το ταίριασμα μοτίβων, η αποτίμηση συνεχίζεται έχοντας αντιστοιχίσει τις τυπικές παραμέτρους μιας συνάρτησης με εκφράσεις και όχι κατ' ανάγκη υπολογισμένες τιμές. Αυτές οι εκφράσεις θα υπολογιστούν αν και μόνον αν χρειαστούν στον υπολογισμό της τελικής τιμής. Για παράδειγμα, θεωρείστε τη συνάρτηση:

```
c b t e = if b then t else e
```

Η αποτίμηση της `c True (1+2) (3+4)` θα οδηγήσει μόνο στον υπολογισμό της παραμέτρου `(1+2)` και όχι στον υπολογισμό της `(3+4)`. Αντίστοιχα, η αποτίμηση της `c False (1+2) (3+4)` θα οδηγήσει μόνο στον υπολογισμό της `(1+2)`.

Το πέρασμα εκφράσεων αντί για υπολογισμένες τιμές είναι ασύμφορο, σε περιπτώσεις όπως αυτή:

```
f x = x+x
```

Εδώ, έχουμε `f (5+5) = (5+5)+(5+5)`, που σημαίνει ότι η οκνηρή αποτίμηση απαιτεί τον υπολογισμό της έκφρασης `(5+5)` δύο φορές. Όμως αυτό δε συμβαίνει. Η Haskell εκμεταλλεύεται το γεγονός ότι οι δύο εμφανίσεις της `x` αναφέρονται στο ίδιο στοιχείο δεδομένων, και εξασφαλίζει ότι αυτό θα υπολογιστεί μία φορά.

Αυτή η ενδιάμεση αποθήκευση συμβαίνει για οποιοδήποτε επώνυμο δεδομένο και όχι μόνο για τις τυπικές παραμέτρους. Στις προηγούμενες σημειώσεις είδαμε ότι ο ορισμός

```
fibpair n = ( fst (fibpair (n-1)) + snd (fibpair (n-1))
            , fst (fibpair (n-1))  )
```

ήταν απελπιστικά ασύμφορος, γιατί η έκφραση `fibpair (n-1)` αποτιμάται τρεις φορές. Είδαμε ότι η λύση είναι να δώσουμε ένα όνομα σε αυτήν την έκφραση, κάτι που κάνει τη Haskell να καταλάβει ότι μία αποτίμηση είναι αρκετή. Χρησιμοποιήσαμε τη `where` για την εισαγωγή αυτού του ονόματος:

```
fibpair n = (fst p + snd p , fst p) where p = fibpair (n-1)
```

Τέλος, ένα ακόμα χαρακτηριστικό της οκνηρής αποτίμησης, που συναντάμε και σε πολλές άλλες γλώσσες προγραμματισμού, είναι η αποτίμηση εκφράσεων όπως

```
True || x
False && x
```

Η πρώτη επιστρέφει `True` χωρίς αποτίμηση του `x`. Ομοίως η δεύτερη επιστρέφει αμέσως `False`. Προσοχή όμως ότι *δε συμβαίνει το αντίστοιχο* με τις εκφράσεις

```
x || True
x && False
```

Στη Haskell γενικά η αποτίμηση γίνεται από τα αριστερά προς τα δεξιά.

Στα παρακάτω θα δούμε μερικά παραδείγματα χρήσης της οκνηρής αποτίμησης.

1.2 Υπολογισμός Οδηγούμενος από τα Δεδομένα

Η οκνηρή αποτίμηση μας επιτρέπει ένα στυλ προγραμματισμού που ονομάζεται *υπολογισμός οδηγούμενος από δεδομένα (data-driven computation)*. Αυτό σημαίνει ότι εκφράζουμε έναν υπολογισμό ορίζοντας δεδομένα, πιθανώς μεγάλης πολυπλοκότητας, που ποτέ όμως δε θα κατασκευαστούν πραγματικά. Ξαναθυμόμαστε το παράδειγμα γραμμικής αναζήτησης των προηγούμενων σημειώσεων:

```
lSearch x l = foldr (||) False (map (x==) l)
```

όπου στην πραγματικότητα η ενδιάμεση λίστα `map (x==) l` δεν κατασκευάζεται. Ιδού η αποτίμηση που κάνει η Haskell στην έκφραση `lSearch 2 [1,2,3]`:

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

Οδηγούμενοι από τα δεδομένα, έχουμε αρκετή εκφραστική δύναμη στη διάθεσή μας χωρίς μείωση της αποδοτικότητας των προγραμμάτων μας. Ένα εντυπωσιακό παράδειγμα, είναι η εύρεση του ελάχιστου στοιχείου μίας λίστας ως εξής:

```
minList = head.iSort
```

όπου `iSort` η συνάρτηση για Insertion Sort που ορίσαμε στις προηγούμενες σημειώσεις. Ο ορισμός μας λέει ότι για να πάρουμε το ελάχιστο στοιχείο μίας λίστας, μπορούμε να την ταξινομήσουμε και να πάρουμε το πρώτο στοιχείο της. Μαθηματικά αυτός ο ορισμός είναι σωστός, αλλά υπολογιστικά θα ήταν μη αποδοτικός, αν η λίστα πράγματι ταξινομούνταν. Όμως αυτό δε συμβαίνει: στην αποτίμηση της `minList l`, μόλις το ελάχιστο στοιχείο εμφανιστεί στην αρχή της λίστας `iSort l`, η αποτίμηση σταματάει και το επιστρέφει, χωρίς φυσικά να συνεχίσει με την ταξινόμηση της λίστας:

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

1.3 Άπειρες Δομές

Η οκνηρή αποτίμηση δίνει την πιθανότητα ορισμού άπειρων δομών, χρησιμοποιώντας αναδρομή. Εδώ θα δούμε άπειρες λίστες, αν και μπορούν να δημιουργηθούν πολλές άλλες άπειρες δομές στη Haskell. Τα παραδείγματα που έχουμε δει μέχρι τώρα είναι η λίστα που έχει άπειρους άσσους

```
listOfOnes = 1:listOfOnes
```

και η λίστα όλων των φυσικών αριθμών

```
nat = 0: [n+1 | n<-nat]
```

Η Haskell υποστηρίζει τις άπειρες λίστες `[n..]` και `[n, m..]` για `n` και `m` ακέραιους. Η λίστα `[n..]` είναι όλοι οι ακέραιοι από `n` και πάνω:

```
[n..]=n:[n+1..]
```

ενώ η λίστα `[n,m..]` είναι όλοι οι ακέραιοι ξεκινώντας από `n` και πηγαίνοντας με βήμα `m-n`:

```
[n,m..]=n:[m,2*m-n..]
```

Ας δούμε δύο παραδείγματα στα οποία χρησιμοποιούμε άπειρες λίστες.

1.3.1 Το Κόσκινο του Ερατοσθένη

Στο πρώτο παράδειγμα έχουμε τον αλγόριθμο του Ερατοσθένη για τον υπολογισμό της λίστας όλων των πρώτων αριθμών. Φυσικά, η λίστα αυτή είναι άπειρη. Η ιδέα είναι απλή: ξεκινάμε από τη λίστα `[2..]`. Η κεφαλή της λίστας είναι πρώτος. Εξαιρούμε από τη λίστα όλους τους αριθμούς που είναι πολλαπλάσια του 2. Η κεφαλή της λίστας που μένει (το 3) είναι πρώτος. Συνεχίζουμε με το 3, αποκλείοντας από τη λίστα όλους τους αριθμούς που είναι πολλαπλάσιά του. Με αυτόν τον τρόπο, εξαιρώντας για κάθε πρώτο που βρίσκουμε κάθε φορά όλα τα πολλαπλάσιά του, σχηματίζουμε τη λίστα όλων των πρώτων αριθμών.

Για να το κάνουμε αυτό, μπορούμε να προγραμματίσουμε οδηγούμενοι από τα δεδομένα. Δοθείσης μίας λίστας `l`, είναι εύκολο να κατασκευάσουμε τη λίστα όλων των στοιχείων που δεν είναι πολλαπλάσια της κεφαλής της λίστας, ακόμα και αν η `l` είναι άπειρη:

```
f :: [Int]->[Int]
f (x:xs) = [y | y<-xs , y`mod`x /= 0]
```

Εκτός από το φιλτράρισμα, θέλουμε: (α) να κρατήσουμε το `x` στη λίστα (οι πρώτοι αριθμοί μένουν στη λίστα)

```
f :: [Int]->[Int]
f (x:xs) = x:[y | y<-xs , y`mod`x /= 0]
```

και (β) να εφαρμόσουμε την `f` αναδρομικά στην ουρά της λίστας. Ο λόγος είναι ότι θέλουμε να φιλτράρουμε τη λίστα και με βάση το επόμενο πρώτο στοιχείο και το επόμενο κτλ. Θα ονομάσουμε τη συνάρτησή μας `sieve` (κόσκινο):

```
sieve :: [Int]->[Int]
sieve (x:xs) = x: sieve[y | y<-xs , y`mod`x /= 0]
```

Προσέξτε ότι κάθε εφαρμογή της `sieve` προσθέτει νέες συνθήκες "κοκκινίσματος".

Η `sieve` παράγει τους πρώτους αριθμούς αν εφαρμοστεί στην `[2..]`:

```
primes :: [Int]
primes = sieve [2..]
```

Για να ελέγξουμε αν ένας αριθμός είναι πρώτος, πρέπει να ελέγξουμε αν ανήκει στην `primes`. Επειδή η `primes` είναι άπειρη, για να αποφύγουμε την περίπτωση μη τερματισμού σε περίπτωση που ο ελεγχόμενος αριθμός δεν είναι πρώτος, θα πρέπει να εκμεταλλευτούμε το γεγονός ότι η `primes` είναι ταξινομημένη. Η λύση μας απορρίπτει όλα τα στοιχεία της `primes` που είναι μικρότερα από `n` και ελέγχει αν η κεφαλή της λίστας που προκύπτει είναι ίση με `n`:

```
isPrime :: Int->Bool
isPrime n = head (dropWhile (<n) primes) == n
```

1.3.2 Πυθαγόρειες Τριάδες

Σε αυτό το παράδειγμα παράγουμε τη λίστα *πυθαγόρειων τριάδων*, δηλαδή τριάδων θετικών ακεραίων αριθμών x, y, z έτσι ώστε $x^2 + y^2 = z^2$. Πρόκειται για μία άπειρη λίστα, που μπορούμε να ορίσουμε χρησιμοποιώντας μια έκφραση διαχωρισμού. Μία πρώτη, απρόσεκτη, προσέγγιση είναι η εξής:

```
pythagorean = [(x,y,z) | x<-[1..], y<-[1..], z<-[1..], x^2+y^2==z^2]
```

Δυστυχώς, η έκφραση αυτή δεν παράγει τίποτα. Αν δοκιμάσουμε την ερώτηση

```
pythagorean!!0
```

η αποτίμηση δε θα τερματίσει ποτέ. Ο λόγος είναι ότι η έκφραση διαχωρισμού θα δώσει τιμή `1` στο `x` και μετά θα προσπαθήσει να το συνδυάσει με όλες τις άλλες τιμές των `y` και `z` πριν να προσπαθήσει την επόμενη τιμή του `x`. Έτσι, και αφού η εξίσωση `x^2+y^2=z^2` δεν έχει λύση για `x==1`, δε θα παραχθεί κανένα αποτέλεσμα, και η αποτίμηση θα τρέχει για πάντα.

Βλέπουμε ότι σε περιπτώσεις που αναμιγνύουμε πολλές άπειρες λίστες μπορεί να αντιμέτωπίσουμε πρόβλημα. Ο τρόπος με τον οποίον παράγονται οι συνδυασμοί στοιχείων από τις μπορεί να μην τους καλύπτει όλους. Για αυτήν την περίπτωση, θα πρέπει να βρεθεί ένας τρόπος να παραχθεί μία και μόνο άπειρη λίστα που να έχει όλους τους συνδυασμούς.

Η ιδέα που θα χρησιμοποιήσουμε για να το κάνουμε αυτό είναι η εξής: προχωρούμε σε βήματα, ξεκινώντας από το βήμα `0`. Σε κάθε βήμα `n` παίρνουμε από τις άπειρες λίστες συνδυασμούς στοιχείων των οποίων οι δείκτες έχουν άθροισμα `n`. Για παράδειγμα, έστω ότι έχαμε δύο άπειρες λίστες

```
[2,..]
[-1,-2,..]
```

- Στο βήμα `0`, θα πάρουμε όλους τους συνδυασμούς στοιχείων, των οποίων οι δείκτες έχουν άθροισμα `0`. Πρόκειται για ένα μόνο συνδυασμό: το στοιχείο `0` της πρώτης λίστας και το στοιχείο `0` της δεύτερης λίστας. Αυτά μας δίνουν το ζεύγος `(2,-1)`.

- Στο βήμα `1`, θα πάρουμε τους συνδυασμούς στοιχείων, των οποίων οι δείκτες έχουν άθροισμα `1`. Πρόκειται για 2 συνδυασμούς. Ο πρώτος είναι το στοιχείο `0` της πρώτης λίστας και το στοιχείο `1` της δεύτερης και μας δίνει το ζεύγος `(2,-2)`. Ο δεύτερος είναι το στοιχείο `1` της πρώτης και το `0` της δεύτερης και μας δίνει `(3,-1)`.

- Στο βημα `2`, έχουμε τους συνδυασμούς στοιχείων με άθροισμα δεικτών `2`. Είναι τρεις συνδυασμοί δεικτών: `(0,2)`,`(1,1)`,`(2,0)` και δίνουν αντιστοίχως τα ζεύγη `(2,-3)`,`(3,-2)`,`(4,-1)`.

- Συνεχίζοντας έτσι, παράγονται όλοι οι άπειροι συνδυασμοί ζευγών από μία και μόνη άπειρη λίστα.

Η τεχνική αυτή ονομάζεται *dovetailing*.

Στην περίπτωσή μας, θέλουμε να παράγουμε μία λίστα που να έχει όλους τους δυνατούς συνδυασμούς από τρεις άπειρες λίστες. Ας δούμε πώς μπορεί να γίνει αυτό στη Haskell. Θα ορίσουμε μία συνάρτηση με όνομα `dovetail3` ώστε να φαίνεται ότι καλύπτουμε την περίπτωση τριών λιστών Έστω οι λίστες `l1`, `l2` και `l3`. Η λίστα των τριάδων των στοιχείων των `l1`, `l2` και `l3` με άθροισμα δεικτών `n` δίνεται από την εξής έκφραση διαχωρισμού:

```
[ (l1!!i1), (l2!!i2), (l3!!i3)
| i1<-[0..n], i2<-[0..n], i3<-[0..n], i1+i2+i3 == n ]
```

Για να παράγουμε μία άπειρη λίστα, ξεκινάμε συνήθως από ένα βήμα `n` και εφαρμόζουμε αναδρομικά τη οριζόμενη συνάρτηση στο σημείο `n+1`. Θα χρησιμοποιήσουμε για αυτό μία βοηθητική συνάρτηση `dovetail3aux` που θα παίρνει έναν ακέραιο `n` και θα μας επιστρέφει την άπειρη λίστα των συνδυασμών από το βήμα `n` και πάνω:

```
dovetail3aux n l1 l2 l3 =
    [ ((l1!!i1), (l2!!i2), (l3!!i3))
    | i1<-[0..n], i2<-[0..n], i3<-[0..n], i1+i2+i3 == n ]
 ++ dovetail3aux (n+1) l1 l2 l3
```

Η συνάρτηση `dovetail3` ξεκινάει την αναδρομή από βήμα `0`:

```
dovetail3 = dovetail3aux 0
 where
 dovetail3aux n l1 l2 l3 =
     [ ((l1!!i1), (l2!!i2), (l3!!i3))
     | i1<-[0..n], i2<-[0..n], i3<-[0..n], i1+i2+i3 == n ]
  ++ dovetail3aux (n+1) l1 l2 l3
```

Οι τρεις άπειρες λίστες που θέλουμε να συνδυάσουμε σε μία είναι και οι τρεις ίσες με την `[1..]`. Έτσι θα πρέπει να εφαρμόσουμε την `dovetail3 [1..] [1..] [1..]` στον υπολογισμό μας. Ο ορισμός της λίστας των πυθαγόρειων τριάδων γίνεται τώρα:

```
pythagorean
 = [(x,y,z) | (x,y,z)<-dovetail3 [1..] [1..] [1..], x^2+y^2==z^2]
```

Η λίστα αυτή περιέχει όλες τις πυθαγόρειες τριάδες. Η ερώτηση `take 10 pythagorean` δίνει:

```
[(3,4,5),(4,3,5),(6,8,10),(8,6,10),(5,12,13),(12,5,13),(9,12,15),
(12,9,15),(8,15,17),(15,8,17)]
```

2 Αποδείξεις Ορθότητας

Μία *απόδειξη ορθότητας (proof of correctness)* ενός προγράμματος είναι ένα μαθηματικό επιχείρημα που εξασφαλίζει ότι ένα πρόγραμμα συμπεριφέρεται με ένα συγκεκριμένο τρόπο για *κάθε* είσοδο που θα του δωθεί (ή τουλάχιστον, για κάθε είσοδο που να ικανοποιεί κάποιες συνθήκες). Ο επιθυμητός τρόπος συμπεριφοράς του προγράμματος, που εκφράζεται επίσης με μαθηματικό τρόπο, ονομάζεται *προδιαγραφή (specification)* του προγράμματος.

Οι αποδείξεις ορθότητας διαφέρουν ριζικά από το testing, διαδικασία στην οποία ψάχνουμε να βρούμε σφάλματα ελέγχοντας απ' ευθείας τη συμπεριφορά του προγράμματος με κάποιες εισόδους. Σε αντίθεση με το testing που είναι μια σχεδόν αυτοματοποιημένη διαδικασία, οι αποδείξεις ορθότητας απαιτούν προσπάθεια εκ μέρους του προγραμματιστή. Επίσης, σε αντίθεση με το testing, μια απόδειξη εξασφαλίζει ότι το πρόγραμμα λειτουργεί με βάση την προδιαγραφή του για *κάθε* δεδομένο εισόδου.

Στο συναρτησιακό προγραμματισμό, οι αποδείξεις ορθότητας τείνουν να είναι πιο εύκολες και πιο κατανοητές απ' ό,τι στο προστακτικό μοντέλο και τις επεκτάσεις του. Στην ενότητα αυτή θα δούμε μερικές από τις βασικές ιδέες πίσω από τις αποδείξεις ορθότητας όπως αυτές γίνονται στο συναρτησιακό προγραμματισμό.

2.1 Μη τερματισμός

Ένα στοιχείο που περιπλέκει τα μαθηματικά που έχουν να κάνουν με αποδείξεις ορθότητας είναι η δυνατότητα μη τερματισμού ενός υπολογισμού. Για να μπορούμε να χρησιμοποιήσουμε προγράμματα που πιθανώς δεν τερματίζουν στα μαθηματικά μας, θα πρέπει να δώσουμε μία τεχνητή τιμή σε κάθε υπολογισμό που δεν τερματίζει. Αυτή η τιμή λέγεται *μη ορισμένο (undefined)*. Οποιοσδήποτε υπολογισμός δεν τερματίζει έχει αυτήν την τιμή, η οποία μπορεί να οριστεί ως εξής:

```
undef = undef
```

(είναι φανερό ότι η αποτίμηση του `undef` δε θα σταματήσει ποτέ).

Επειδή μη τερματισμός μπορεί να συμβεί σε κάθε τύπο, η τιμή `undef` ανήκει σε κάθε τύπο της Haskell. Οι βασικές τιμές που δεν ισούνται με `undef` ονομάζονται *ορισμένες*.

Σε δομημένα δεδομένα, όπως οι λίστες, τα πράγματα είναι πιο περίπλοκα. Η `undef` ανήκει σε κάθε τύπο λίστας, οπότε η `undef` είναι λίστα. Γι' αυτό το λόγο και επειδή κάθε μη κενή λίστα κατασκευάζεται ενώνοντας ένα στοιχείο με μία λίστα με τον τελεστή `(:)`, αυτό σημαίνει ότι και τα παρακάτω είναι επίσης λίστες:

```
2:undef
2:3:undef
```

Τέλος, επειδή και τα στοιχεία της λίστας μπορεί να είναι `undef`, και τα παρακάτω είναι λίστες:

```
undef:[2,3]
[1,undef,3]
undef:undef
```

Κάθε μία από τις παραπάνω λίστες είναι διάφορη της `undef`.

Θα λέμε ότι μία λίστα είναι *ορισμένη* αν κάθε στοιχείο της είναι ορισμένο. Αυτός ο ορισμός είναι αναδρομικός, πχ. οι λίστες

```
[0,[1,undef],2]
[0,[1]:undef,2]
```

δεν είναι ορισμένες, επειδή το δεύτερο στοιχείο τους δεν είναι ορισμένο. Ομοίως θα λέμε ότι μία πλειάδα είναι ορισμένη αν κάθε στοιχείο της είναι ορισμένο.

Μία λίστα `l` είναι *πεπερασμένη* αν `length l /= undef` και κάθε στοιχείο της είναι πεπερασμένο.

Για διευκόλυνση θα εισάγουμε τους παρακάτω συμβολισμούς. Αν `T` είναι ένας βασικός τύπος, θα συμβολίζουμε με $\texttt{T}_\texttt{d}$ όλα τα ορισμένα στοιχεία του `T`, δηλ. τα στοιχεία του `T` εκτός από το `undef`. Για κάθε τύπο `T`, γράφουμε $\texttt{[T]}_\texttt{f}$ για τις πεπερασμένες λίστες τύπου `[T]` και $\texttt{[T]}_\texttt{df}$ για τις ορισμένες και πεπερασμένες λίστες τύπου `[T]`.

2.2 Επαγωγή σε Ακεραίους

Ο κανόνας της *επαγωγής (induction)* είναι πρωταρχικής σημασίας στην απόδειξη ορθότητας των προγραμμάτων. Η κλασική μορφή της επαγωγής εφαρμόζεται στους *φυσικούς αριθμούς*, δηλ. στους ορισμένους ακεραίους που είναι μεγαλύτεροι ή ίσοι με το `0` και χρησιμεύει στο να αποδείξουμε ότι μία ιδιότητα είναι αληθής για κάθε ακέραιο.

Θα συμβολίσουμε τις ιδιότητες των φυσικών με συναρτήσεις που παίρνουν φυσικούς και επιστρέφουν ορισμένες αληθοτιμές. Μία συνάρτηση που επιστρέφει ορισμένες αληθοτιμές ονομάζεται *κατηγόρημα (predicate)*. Ένα κατηγόρημα που παίρνει παραμέτρους από ένα σύνολο S θα ονομάζεται *κατηγόρημα στο* S.

Έστω κατηγόρημα `p` στους φυσικούς αριθμούς[1]. Ο κανόνας της επαγωγής στους φυσικούς αριθμούς είναι ο εξής:

- **Αν:**
  - `p 0`, και
  - Για κάθε φυσικό `n`, ισχύει `p n` $\Rightarrow$ `p(n+1)`
- **Τότε:**
  - Για κάθε φυσικό `n`, ισχύει `p n`

[1] Το κατηγόρημα δεν είναι απαραίτητο να γράφεται στη Haskell. Οι συναρτήσεις της Haskell είναι μαθηματικές συναρτήσεις, όμως υπάρχουν και μαθηματικές συναρτήσεις που δε μπορούν να εκφραστούν στη Haskell.

Η πρώτη υπόθεση λέγεται *βάση της επαγωγής* ενώ η δεύτερη *βήμα της επαγωγής*.

Μπορούμε να γενικεύσουμε άμεσα την επαγωγή σε ολόκληρο το σύνολο των ορισμένων ακεραίων, συμπεριλαμβάνοντας και την αφαίρεση στο βήμα της επαγωγής:

- **Αν:**
  - `p 0`, και
  - Για κάθε ορισμένο ακέραιο `n`, ισχύει `p n` $\Rightarrow$ `p(n+1)`, και
  - Για κάθε ορισμένο ακέραιο `n`, ισχύει `p n` $\Rightarrow$ `p(n-1)`
- **Τότε:**
  - Για κάθε ορισμένο ακέραιο `n`, ισχύει `p n`

Τέλος, όλοι οι ακέραιοι είναι οι ορισμένοι ακέραιοι μαζί με τον `undef`. Επομένως, ο κανόνας της επαγωγής για όλους τους ακέραιους της Haskell, συμπεριλαμβάνει και τον `undef` στη βάση:

- **Αν:**
  - `p 0`, και
  - `p undef`, και
  - Για κάθε ορισμένο ακέραιο `n`, ισχύει `p n` $\Rightarrow$ `p(n+1)`, και
  - Για κάθε ορισμένο ακέραιο `n`, ισχύει `p n` $\Rightarrow$ `p(n-1)`
- **Τότε:**
  - Για κάθε ακέραιο `n`, ισχύει `p n`

2.2.1 Παραγοντικό

Η επαγωγή παίζει σημαντικό ρόλο στις αποδείξεις για προγράμματα που γράφονται με πρωτρχική αναδρομή. Η δομή της απόδειξης είναι ήδη έτοιμη από τη δομή του προγράμματος: χρησιμοποιούμε τον ορισμό για την αρχική τιμή, π.χ. `0` για να αποδείξουμε τη βάση της επαγωγής και τον ορισμό για τις επόμενες τιμές για να αποδείξουμε το βήμα της επαγωγής.

Ο ορισμός της συνάρτησης του παραγοντικού είναι μια κλασσική πρωταρχική αναδρομή:

```
factorial 0 = 1
factorial n = n*factorial(n-1)
```

Θα αποδείξουμε, με επαγωγή στους φυσικούς, ότι για κάθε φυσικό αριθμό `n`, ο αριθμός `factorial n` είναι ακέραιο πολλαπλάσιο όλων των θετικών ακεραίων `k` ώστε `k<=n`. Σε τυπικά μαθηματικά:

```
p n == ∀k<-[1..n].  factorial n `mod` k == 0
```

Όπου η σύνταξη `∀k<-[1..n].`, δανεισμένη από τις γεννήτριες των εκφράσεων διαχωρισμού, σημαίνει "για κάθε ακέραιο `k` από `1` έως και `n`".

Η απόδειξη της βάσης της επαγωγής είναι:

```
    p 0
=   (∀k<-[1..0].  factorial n `mod` k == 0)
=   (∀k<-[].  factorial n `mod` k == 0)
=   True
```

Εδώ κάνουμε χρήση του θεωρήματος του καθολικού ποσοδείκτη:

$$\forall x \in \emptyset . S$$

δηλαδή, ο καθολικός ποσοδείκτης πάνω σε κενό σύνολο επιστρέφει πάντα `True`.

Για να αποδείξουμε το βήμα της επαγωγής, αρκεί να υποθέσουμε `p n` και να αποδείξουμε `p(n+1)`. Η υπόθεση `p n` ονομάζεται *επαγωγική υπόθεση (induction hypothesis)*. Η απόδειξη της `p(n+1)` (δοθείσης της επαγωγικής υπόθεσης) είναι:

```
    p(n+1)
=   (∀k<-[1..n+1]. factorial(n+1) `mod` k == 0)
=   (   factorial(n+1) `mod` (n+1) == 0
    &&  ∀k<-[1..n]. factorial(n+1) `mod` k == 0)
=   (   ((n+1)*factorial n) `mod` (n+1) == 0
    &&  ∀k<-[1..n]. factorial(n+1) `mod` k == 0)
=   (True  && ∀k<-[1..n]. factorial(n+1) `mod` k == 0)
=   (∀k<-[1..n]. factorial(n+1) `mod` k == 0)
=   (∀k<-[1..n]. (n+1)*factorial n `mod` k == 0) επαγ. υπόθ.
=   (∀k<-[1..n]. True)
=   True
```

Αφού αποδείξαμε τη βάση και το βήμα της επαγωγής για τους φυσικούς, συμπεραίνουμε ότι η ιδιότητα `p n` ισχύει για κάθε φυσικό `n`:

```
∀n<-[0..],k<-[1..n].  factorial n `mod` k  ==  0
```

2.3 Επαγωγή σε Λίστες

Όπως και οι φυσικοί και οι ακέραιοι αριθμοί, οι λίστες είναι ένα μαθηματικό σύνολο που κατασκευάζεται από έναν αναδρομικό ορισμό. Επομένως, μπορούμε να ορίσουμε επαγωγικούς κανόνες και για τις λίστες. Οι κανόνες αυτοί, όπως θα δούμε, έχουν να κάνουν *μόνο* με πεπερασμένες λίστες (όπως άλλωστε και ο κανόνας επαγωγής για ακεραίους δεν επεκτείνεται στο ∞).

Ο κανόνας επαγωγής για πεπερασμένες και ορισμένες λίστες τύπου `[T]` είναι ο εξής:

- **Αν:**
  - `p[]`, και
  - Για κάθε ορισμένο πεπερασμένο `x::T` και ορισμένη πεπερασμένη λίστα `xs::[T]`, ισχύει `p xs` $\Rightarrow$ `p(x:xs)`
- **Τότε:**
  - Για κάθε ορισμένη πεπερασμένη λίστα `l::[T]`, ισχύει `p l`

Για πεπερασμένες λίστες γενικότερα, θα πρέπει να συμπεριλάβουμε τη λίστα `undef` καθώς και το στοιχείο `undef`:

- **Αν:**
  - `p[]`, και
  - `p undef`, και
  - Για κάθε `x::T` και πεπερασμένη λίστα `xs::[T]`, ισχύει `p xs` $\Rightarrow$ `p(x:xs)`
- **Τότε:**
  - Για κάθε πεπερασμένη λίστα `l::[T]`, ισχύει `p l`

2.3.1 Η συνάρτηση `reverse`

Έστω ο παρακάτω ορισμός της `reverse`:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Θέλουμε να αποδείξουμε την ορθότητα του παραπάνω ορισμού. Η ορθότητα δίνεται από την εξής μη τυπική πρόταση: "η `reverse` αναστρέφει μία ορισμένη και πεπερασμένη λίστα οποιουδήποτε τύπου". Με άλλα λόγια, αν `l` είναι ορισμένη και πεπερασμένη, τότε η `reverse l` είναι μία λίστα ίδιου μήκους με την `l`, έτσι ώστε για κάθε δείκτη `i` της `l` είναι:

```
(reverse l)!!i == l!!(length l-1-i)
```

Η τυπική προδιαγραφή είναι (για κάθε τύπο `T` και λίστα $\texttt{l::[T]}_{\texttt{df}}$):

```
   length l == length(reverse l)
&& ∀i<-[0..length l-1].  (reverse l)!!i == l!!(length l-1-i)
```

Μας βολεύει να σπάσουμε τη προδιαγραφή στους δύο τελεσταίους της σύζευξης και να τους αποδείξουμε ξεχωριστά. Κάθε περίπτωση αποδεικνύεται με επαγωγή σε πεπερασμένες-ορισμένες λίστες.

Πρώτο κομμάτι: `length l == length(reverse l)`.

Βάση επαγωγής: τετριμμένη, από ορισμό του `reverse[]`.

Βήμα επαγωγής: Υποθέτουμε `length xs == length (reverse xs)` (επαγωγική υπόθεση)

```
    length(reverse(x:xs))
=   length(reverse xs++[x])   ιδιότ. του length
=   length(reverse xs) + 1          επαγ. υποθ.
=   length xs + 1             ιδιότ. του length
=   length(x:xs)
```

Δεύτερο κομμάτι:

```
∀i<-[0..length l-1].  (reverse l)!!i == l!!(length l-1-i)
```

Βάση επαγωγής: αληθής από την ιδιότητα του ∀ για κενά σύνολα.
Βήμα επαγωγής: υποθέτουμε ότι

```
∀i<-[0..length xs-1].  (reverse xs)!!i == xs!!(length xs-1-i)
```

(επαγωγική υπόθεση) και ότι `i<-[0..length(x:xs)-1]` και θέλουμε να αποδείξουμε ότι `reverse(x:xs)!!i == (x:xs)!!(length(x:xs)-1-i)`.

Πρώτη περίπτωση: `i == length xs` (υποθ.1). Τότε έχουμε:

```
    reverse(x:xs)!!i                                  υποθ. 1
=   (reverse xs++[x])!!(length xs)          από πρώτο κομμάτι
=   (reverse xs++[x])!!(length(reverse xs))
=   x
```

και

```
    (x:xs)!!(length(x:xs)-1-i)   υποθ. 1
=   (x:xs)!!0
=   x
```

Δεύτερη περίπτωση: `i<-[0..length xs-1]` (υποθ. 2). Τότε έχουμε:

```
    (reverse(x:xs))!!i
=   (reverse xs ++ [x])!!i       υποθ.2 και πρώτο κομμάτι
=   (reverse xs)!!i                          επαγ. υποθ.
=   xs!!(length xs-1-i)
=   (x:xs)!!(length xs-1-i+1)
=   (x:xs)!!(length(x:xs)-1-i)
```

Η προδιαγραφή αποδείχτηκε με δύο επαγωγές σε ορισμένες πεπερασμένες λίστες.

2.3.2 Η συνάρτηση `foldr`

Η συνάρτηση `foldr` ορίζεται ως εξής:

```
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

Έστω ότι υπάρχει ένας τύπος `T` έτσι ώστε:

- `f::T->T->T` και για κάθε $\texttt{x::T}_\texttt{d}$,$\texttt{y::T}_\texttt{d}$, ισχύει $\texttt{x`f`y::T}_\texttt{d}$
- Η `f` είναι *προσεταιριστική*, δηλ. για κάθε $\texttt{a::T}_\texttt{d}$, $\texttt{b::T}_\texttt{d}$, $\texttt{c::T}_\texttt{d}$, ισχύει

  ```
  (a`f`b)`f`c == a`f`(b`f`c)
  ```

  Θα γράφουμε λοιπόν `a`f`b`f`c` για κάθε έναν από τους δύο εναλλακτικούς τρόπους να βάζουμε παρενθέσεις.

- $\texttt{z::T}_\texttt{d}$ και το `z` είναι *ουδέτερο στοιχείο* της `f`, δηλ. για κάθε $\texttt{x::T}_\texttt{d}$, ισχύει

  ```
  z`f`x == x  && x`f`z == x
  ```

Θέλουμε να δείξουμε ότι για κάθε $\texttt{xl::[T]}_\texttt{df}$, $\texttt{yl::[T]}_\texttt{df}$ ισχύει

```
foldr f z (xl++yl) == (foldr f z xl)`f`(foldr f z yl)
```

Αυτό σημαίνει ότι αν τα `f` και `z` έχουν τις απαιτούμενες προδιαγραφές, μία πεπερασμένη και ορισμένη λίστα `l` μπορεί να "σπάσει" σε οποιοδήποτε σημείο της σε δύο λίστες `xl` και `yl`, και το αποτέλεσμα της `foldr f z l` να περιγραφεί ως η εφαρμογή της `f` στις `foldr f z xl` με την `foldr f z yl`.

Για παράδειγμα, έστω:

```
sum = foldr (+) 0
mult = foldr (*) 1
or = foldr (||) False
```

Το θεώρημα που θέλουμε να αποδείξουμε μας δίνει:

```
sum(xl++yl) == sum xl + sum yl
mult(xl++yl) == mult xl * mult yl
or(xl++yl) == or xl || or yl
```

κτλ.

Για διευκόλυνση θα ορίσουμε:

```
ff = foldr f z
```

Τώρα το θεώρημα που θέλουμε να αποδείξουμε γίνεται:

```
ff(xl++yl) == (ff xl) `f` (ff yl)
```

Η απόδειξη θα γίνει με επαγωγή στη λίστα `xl`.
Βάση επαγωγής:

```
    (ff[])`f`(ff y)          ορισμός της ff
=   z`f`(ff y)       z ουδέτερο στοιχείο της f
=   ff y
=   ff([]++y)
```

Βήμα επαγωγής. Έστω ότι `ff(xs++yl) == (ff xs) `f` (ff yl)` (επαγωγική υπόθεση). Η απόδειξη του βήματος επαγωγής είναι τώρα:

```
    ff((x:xs)++yl)
=   ff(x:(xs++yl))
=   x`f`(ff(xs++yl))        επαγ. υπόθ. και προσεταιριστικότητα της f
=   x`f`(ff xs)`f`(ff yl)                              ορισμός της ff
=   (ff(x:xs)) `f` (ff yl)
```

Η προδιαγραφή αποδείχτηκε με επαγωγή σε ορισμένες πεπερασμένες λίστες.

2.3.3 Άπειρες Λίστες

Όπως η επαγωγή στους φυσικούς δε συμπεριλαμβάνει το ∞, έτσι και η επαγωγή στις λίστες δεν περιλαμβάνει τις άπειρες λίστες. Ο λόγος είναι ότι χρησιμοποιώντας τους *κατασκευαστές* λιστών `[]` και `(:)` δε μπορούμε να κατασκευάσουμε μία άπειρη λίστα σε πεπερασμένο αριθμό βημάτων (μία απόδειξη απαιτεί πεπερασμένο αριθμό βημάτων). Ομοίως, στους φυσικούς αριθμούς, δε μπορούμε να χρησιμοποιήσουμε τους κατασκευαστές `0` και `(+1)` ώστε να κατασκευάσουμε το ∞ σε πεπερασμένο αριθμό βημάτων.

Ένα παράδειγμα μη εφαρμογής της επαγωγής λιστών για άπειρες λίστες έχει ως εξής: Ο ορισμός της `length` είναι:

```
length [] = 0
length (_:xs) = 1+length xs
```

Μπορούμε εύκολα να αποδείξουμε ότι για όλες τις ορισμένες πεπερασμένες `l` ισχύει `length l :: Int`$_\mathtt{d}$. Αλλά, σε μία άπειρη λίστα, η εφαρμογή της `length` δε θα σταματήσει ποτέ. Επομένως το συμπέρασμα δεν ισχύει για άπειρες λίστες, αφού, αν η `l` είναι άπειρη τότε, `length l = undef`.

Για να αποδείξουμε πράγματα σχετικά με μία άπειρη λίστα, συνήθως διατυπώνουμε μία πρόταση για την `take n l` ή για την `l!!n` και χρησιμοποιούμε επαγωγή φυσικών στο `n`. Δε θα ασχοληθούμε στη συνέχεια με απόδειξη ιδιοτήτων άπειρων λιστών.

2.3.4 Ισχυροποίηση της Θεωρήματος στην Επαγωγή

Σε αυτό το παράδειγμα, θα δούμε πως οι αποδείξεις με επαγωγή χρειάζονται πολλές φορές *ισχυροποίηση* του θεωρήματος το οποίο αποδεικνύουμε. Δηλαδή, για να λύσουμε ένα πρόβλημα με επαγωγή, πολλές φορές χρειάζεται να το *κάνουμε πιο δύσκολο πρώτα*. Αυτό φαίνεται να αντίκειται στην κοινή λογική, αλλά στην πραγματικότητα έχει νόημα. Ο λόγος είναι ότι, ενώ αυτό που θέλουμε να αποδείξουμε είναι ισχυρότερο, *και η επαγωγική υπόθεση που χρησιμοποιούμε είναι ισχυρότερη*. Όσο ισχυρότερες είναι οι υποθέσεις μας, τόσο πιο εύκολα μπορούμε να αποδείξουμε καινούρια πράγματα. Επομένως, αν η ισχυροποίηση είναι σωστά επιλεγμένη, τότε ενώ φαίνεται ότι δυσκολεύουμε το πρόβλημα, στην πραγματικότητα το κάνουμε πιο εύκολο.

Ας δημιουργήσουμε μία νέα συνάρτηση για τον υπολογισμό του αθροίσματος των στοιχείων μίας λίστας. Η συνάρτηση, σε αντίθεση με τη `sum`, που είδαμε πιο πάνω, χρησιμοποιεί *αναδρομή ουράς*:

```
mysum :: [Int]->Int
mysum = mysumaux 0
 where mysumaux s [] = s
       mysumaux s (x:xs) = mysumaux (s+x) xs
```

Θέλουμε να δείξουμε ότι οι `sum` και `mysum` κάνουν το ίδιο πράγμα. Δηλαδή, για κάθε $\texttt{l::[Int]}_{\texttt{df}}$ ισχύει:

```
mysum l = sum l
```

Ας αποπειραθούμε να το κάνουμε αυτό με επαγωγή. Φυσικά η βάση της επαγωγής είναι πολύ απλή:

```
   mysum []
=  mysumaux 0 []
=  0
```

και

```
   sum []
=  foldr (+) 0 []
=  0
```

Για το βήμα της επαγωγής, υποθέτουμε ότι `mysum xs = sum xs` και πάμε να αποδείξουμε ότι `mysum(x:xs) = sum(x:xs)`. Είναι εύκολο να χειριστούμε τη `sum`:

```
   sum (x:xs)
=  foldr (+) 0 (x:xs)
=  x+foldr (+) 0 xs
=  x+sum xs              επαγωγική υπόθεση
=  x+mysum xs
```

Ας δούμε τώρα τι γίνεται με τη `mysum`:

```
   mysum(x:xs)
=  mysumaux 0 (x:xs)
=  mysumaux (0+x) xs
=  mysumaux x xs
```

Εδώ σταματάμε, γιατί δε μπορούμε να χρησιμοποιήσουμε την επαγωγική υπόθεση για να εξισώσουμε το `mysumaux x xs` με το `x+mysum xs`. Η επαγωγική υπόθεση μιλάει μόνο για το `mysum`, δηλαδή `mysumaux 0`. Χρειαζόμαστε ένα *ισχυρότερο θεώρημα*, που να λέει κάτι για τη `mysumaux x` για *οποιοδήποτε* `x`.

Ισχυροποιούμε (δυσκολεύουμε) το θεώρημα που θέλουμε να αποδείξουμε, ώστε να μιλάει για κάθε παράμετρο της `mysumaux`. Το θεώρημα είναι τώρα: για κάθε $\texttt{l::[Int]}_{\texttt{df}}$

$$\forall \texttt{s::Int}_{\texttt{d}}\texttt{. mysumaux s l = s+sum l}$$

Προσέξτε πώς το θεώρημα που θέλαμε να αποδείξουμε είναι μία υποπερίπτωση του θεωρήματος που θέλουμε να αποδείξουμε τώρα (για `s=0`).

Η ισχυροποίηση αυτή, κάνει ισχυρότερη και την επαγωγική υπόθεση, καθιστώντας δυνατή την απόδειξη με επαγωγικό τρόπο του θεωρήματος.

Βάση επαγωγής: `mysumaux s [] == s` και

```
   s + sum []
=  s + foldr (+) 0 []
=  s + 0
=  s
```

Βήμα επαγωγής: Υποθέτουμε
$\forall \texttt{t::Int}_{\texttt{d}}\texttt{. mysumaux t xs = t+sum xs}$
(επαγωγική υπόθεση) και έχουμε:

```
   mysumaux s (x:xs)
=  mysumaux (s+x) xs   επαγ. υπόθ. για t=s+x
=  (s+x) + sum xs
=  s + x + sum xs
```

και

```
   s + sum (x:xs)
=  s + foldr (+) 0 (x:xs)
=  s + x + foldr (+) 0 xs
=  s + x + sum xs
```

Ισχυροποιώντας το θεώρημα που θέλαμε να αποδείξουμε, η επαγωγή σε πεπερασμένες ορισμένες λίστες πέτυχε. Το θεώρημα που αποδείξαμε συνεπάγεται το ασθενέστερο θεώρημα που θέλαμε να αποδείξουμε.

3 Επισκόπηση

Σε αυτές τις σημειώσεις είδαμε δύο ξεχωριστά θέματα, την οκνηρή αποτίμηση και τις αποδείξεις ορθότητας στη Haskell.

Στην οκνηρή αποτίμηση είδαμε τα εξής:

- Ο ρόλος της οκνηρής αποτίμησης είναι ο *διαχωρισμός της κατασκευής από τη χρήση* μίας πιθανώς περίπλοκης δομής δεδομένων.
- Η Haskell υποστηρίζει οκνηρή αποτίμηση σε όλες τις δομές της. Οι κανόνες της Haskell εξασφαλίζουν ότι *επώνυμα δεδομένα θα αποτιμηθούν το πολύ μία φορά*.
- Η οκνηρή αποτίμηση διευκολύνει ένα στυλ προγραμματισμού που λέγεται *υπολογισμός οδηγούμενος από τα δεδομένα*. Ο υπολογισμός σε αυτό το στυλ στηρίζεται στον ορισμό περίπλοκων δομών, οι οποίες δεν κατασκευάζονται πλήρως.
- Η οκνηρή αποτίμηση δίνει επίσης τη δυνατότητα ορισμού άπειρων δομών.
- Η χρήση περισσότερων της μίας άπειρων δομών μπορεί να προκαλέσει προβλήματα. Μία τεχνική για να συνδυάσουμε πολλές άπειρες δομές σε μία είναι το *dovetailing*.

Στις αποδείξεις ορθότητας είδαμε τα εξής:

- Ο ρόλος των αποδείξεων ορθότητας είναι να εξασφαλίσουν ότι μία προδιαγραφή ικανοποιείται από το πρόγραμμα για *όλες* τις πιθανές εισόδους. Σε αντίθεση, το testing ελέγχει μερικές μόνο δυνατές εισόδους.
- Οι αποδείξεις ορθότητας απαιτούν ανθρώπινη προσπάθεια, ενώ το testing όχι.
- Μία μαθηματική θεωρία ορθότητας χρειάζεται ένα τρόπο χειρισμού του υπολογισμού που δεν τερματίζει. Εμείς εξισώνουμε το μη τερματίζοντα υπολογισμό με το στοιχείο `undef` που ανήκει σε όλους τους τύπους.
- Η *μαθηματική επαγωγή* είναι ο βασικός τρόπος απόδειξης στο συναρτησιακό προγραμματισμό. Εισάγαμε πέντε κανόνες μαθηματικής επαγωγής που καλύπτουν τους ακεραίους και τις πεπερασμένες λίστες.
- Η επαγωγή δεν είναι κατάλληλος τρόπος αποδείξεως για άπειρες δομές.
- Πολλές φορές χρειάζεται να *ισχυροποιούμε* το θεώρημα που θέλουμε να αποδείξουμε για να είναι δυνατή η απόδειξή του με επαγωγή.