ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

# ΕΠΑΛΗΘΕΥΣΗ ΙΔΙΟΤΗΤΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΕ LIQUID HASKELL

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΠΕΤΡΟΥ ΓΕΩΡΓΙΟΣ**

**Επιβλέπων:** Παπασπύρου Νικόλαος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

# ΕΠΑΛΗΘΕΥΣΗ ΙΔΙΟΤΗΤΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΕ LIQUID HASKELL

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΠΕΤΡΟΥ ΓΕΩΡΓΙΟΣ**

**Επιβλέπων** : Παπασπύρου Νικόλαος
Αναπληρωτής Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Ιουλίου 2018

| ..... | ..... | ..... |
|---|---|---|
| Παπασπύρου Νικόλαος<br>Αν Καθηγητής Ε.Μ.Π | Βάζου Νίκη<br>Μεταδιδακτορική Ερευνήτρια<br>University of Maryland | Παγουρτζής Αριστείδης<br>Αν. Καθηγητής Ε.Μ.Π |

Αθήνα, Ιούλιος 2018

.....

**Πέτρου Γεώργιος**

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

## Περίληψη

Στη Διπλωματική εργασία αυτή γίνεται μελέτη των τεχνικών που παρέχονται από τη Liquid Haskell για την απόδειξη θεωρημάτων και η χρησιμοποίησή τους για την επαλήθευση ιδιοτήτων διαφόρων αλγορίθμων.

Η Liquid Haskell αποτελεί ένα σύστημα που μας δίνει τη δυνατότητα να εκφράζουμε ιδιότητες για προγράμματα γραμμένα σε Haskell με τη χρήση refinement types οι οποίες επαληθεύονται αυτόματα με τη χρήση SMT solver. Επιπλέον η Liquid Haskell με την εισαγωγή του reflection μας επιτρέπει την διατύπωση και απόδειξη θεωρημάτων εκφρασμένων ως συναρτήσεις σε Haskell, προσφέρωντας ειδικές βιβλιοθήκες που παρέχουν ένα πλήρες περιβάλλον για την ανάπτυξη αρκτεκά εξελιγμένων αποδείξεων χρησιμοποιώντας βασικά refinements και ορισμούς συναρτήσεων .

Στο πλαίσιο της εργασίας , αρχικά γίνεται μια ανασκόπηση σχετικά με τις ένοιες που θα συναντήσουμε στη Liquid Haskell, όπως refinement types, την έννοια του reflection, μεθόδους αποδείξεων, ανάλυση της βιβλιοθήκης που μας δίνει τη δυνατότητα για συγγραφή αποδείξεων κλπ.. και στη συνέχεια προχωράμε στην ανάλυση ιδιοτήτων ορισμένων διαδέδομένων αλγορίθμων και δομών δεδομένων που αποτελεί και το βασικό μέρος της διπλωματικής. Οι αλγόριθμοι που αναλύουμε προέχρονται από το 3$^{ο}$ τόμο του βιβλίου Software Foundations που περιέχει διατυπώσεις θεωρημάτων και ιδιοτήτων σε γλώσσα Coq. Στη παρούσα διπλωματική θα αναδιατυπώσουμε τα θεωρήματα και ιδιότητες σε γλώσσα Haskell και θα τα επαληθευσουμε γράφωντας αποδείξεις σε Liquid Haskell παρουσιάζοντας έτσι το τρόπο με τον οποίο μπορούμε να διατυπώνουμε θεωρήματα και αναδεικνύωντας τη Liquid Haskell ως ένα πλήρες εργαλείο για απόδειξη θεωρημάτων.



## Λέξεις κλειδιά

Liquid Haskell, refinement types, απόδειξη θεωρημάτων, επαλήθευση, SMT solver, reflection, proof by logical evaluation.

## Abstract

The purpose of this diploma thesis is to study the techniques provided by Liquid Haskell for theoretical proofs and to use them to verify the properties of various algorithms.

Liquid Haskell is a system that enables us to express properties for Haskell-written programs by using refinement types that are automatically verified using SMT solver. In addition Liquid Haskell with the introduction of reflection allows us to formulate and demonstrate theorems expressed as functions in Haskell, by offering special libraries that provide a complete environment for the development of sophisticated proofs using basic refinements and function definitions.

In the context of the diploma thesis, we initially review the concepts we will encounter in Liquid Haskell, such as refinement types, the concept of reflection, methods of evidence, analysis of the library that enables us to write proofs, etc., and then proceed in the analysis of properties of certain well known algorithms and data structures, which is also the main part of this diploma thesis. The algorithms we analyze are from the 3rd volume of the Software Foundations book that contains theorems and properties expressed in Coq. In this diploma thesis we will reformulate the theorems and properties in Haskell and we will verify them by writing proofs with Liquid Haskell, presenting how we can formulate theorems and highlight Liquid Haskell as a complete tool for theorem proving.

## Key Words

Liquid Haskell, refinement types, theorem proving, verification, SMT solver, reflection, proof by logical evaluation.

## Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή της διπλωματικής μου εργασίας κ. Παπασπύρου Νικόλαο για την πολύτιμη καθοδήγηση, την υποστήριξη και το χρόνο που αφιέρωσε για την επιτυχή διεκπεραίωση της παρούσας εργασίας. Επιπλέον θα ήθελα να ευχαριστήσω την μεταδιδακτορική ερευνήτρια Βάζου Νίκη (University of Maryland) και τον διδακτορικό φοιτητή Λαμπρόπουλο Λεωνίδα (University of Pennsylvania) για την πολύτιμη συνεισφορά τους και την συνεχή καθοδήγηση που μου παρείχαν για την επιτυχή διεκπεραίωση της διπλωματικής εργασίας. Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου για όλη τη στήριξη που μου παρείχε σε όλη τη διάρκεια των σπουδών μου.

Πέτρου Γεώργιος,
Αθήνα, 16 Ιουλίου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-7-18, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2018.

URL: http://www.softlab.ntua.gr/techrep/
FTP: ftp://ftp.softlab.ntua.gr/pub/techrep/

# Περιεχόμενα

# 1 Εισαγωγή

## 1.1 Σκοπός της εργασίας

Σκοπός της εργασίας είναι η μελέτη των διαφόρων τεχνικών για την απόδειξη θεωρημάτων που προσφέρει η Liquid Haskell. Η Liquid Haskell αποτελεί ένα SMT-based deductive verifier δηλαδή ένα σύστημα με το οποίο μπορούμε να κάνουμε ανάλυση ιδιοτήτων προγραμμάτων που είναι γραμμένα σε γλώσσα Haskell με τη χρήση SMT solver. Επιπλέον εκτός από την ανάλυση των ιδιοτήτων, με την εισαγωγή του refinement reflection το οποίο περιγράφεται στη συνέχεια, μπορεί να χρησιμοποιηθεί και για την απόδειξη ιδιοτήτων και θεωρημάτων εκφρασμένων ως συναρτήσεων σε γλώσσα Haskell.

Βασικό στόχο της παρούσας διπλωματικής αποτελεί η ανάλυση και απόδειξη ιδιοτήτων ορισμένων βασικών αλγορίθμων. Η ανάλυση και απόδειξη των ιδιοτήτων των αλγορίθμων αυτών έχει γίνει χρησιμοποιώντας to deductive verifier Coq στο 3$^{ο}$ τόμο της σειράς Software Foundations. Έτσι λοιπόν στο πλαίσιο της εργασίας μεταφράζουμε τις ιδιότητες και τις αποδείξεις θεωρημάτων που είχαν γίνει στο Coq, σε Liquid Haskell. Με αυτό το τρόπο επιχειρούμε να δείξουμε ότι οι αποδείξεις που υπήρχαν στο Coq μπορούν να γίνου σε liquid Haskell, να παρουσιάσουμε το τρόπο με τον οποίο γίνεται η μετάφραση και τέλος να εξάγουμε τα συμπεράσματα που προέκυψαν από την παραραπάνω διαδικασία.

Στο εισαγωγικό κομμάτι της εργασίας κάνουμε μια αναδρομή στους Deductive verifiers και στην αντίστοιχη θεωρία που σχετίζεται με την απόδειξη θεωρημάτων. Στη συνέχεια αναλύουμε βασικές έννοιες όπως τα refinement types (εκλεπτυσμένοι τύποι), ανάλυση ιδιοτήτων και επαλήθευση προγραμμάτων καθώς επίσης εισάγουμε και αναλύουμε την ένοια του refinement reflection που είναι ένα νέο πλαίσιο για την κατασκευή SMT-based deductive verifiers που χρησιμοποιείται στη Liquid Haskell. Πριν προχωρήσουμε στο βασικό μέρος της εργασίας που είναι η ανάλυση και επαλήθευση αλγορίθμων επεξηγούμε παραθέτοντας κατάλληλα παραδείγματα όπου είναι εφικτό, πως η Liquid Haskell εκτελεί συλλογιστική στα προγράμματα και πως αυτοματοποιείται η διαδικασία της συλλογιστικής. Αναλύουμε επίσης πως η Liquid Haskell ελέγχει το totality και το termination των συναρτήσεων και ποια η σημασία των δύο αυτών ελέγχων. Μετά ακολουθεί το βασικό μέρος της εργασίας όπου γίνεται η ανάλυση και απόδειξη των ιδιοτήτων βασικών αλγορίθμων σε Liquid Haskell που αποτελεί το βασικό σκοπό της διπλωματικής εργασίας. Τέλος θα αναφέρουμε κλείνοντας την εργασία προτάσεις για μελλοντική έρευνα καθώς επίσης και συγκριτικά σχόλια και παρατηρήσεις μεταξύ των δύο verifiers που μελετάμε (Liquid Haskell και Coq) που προέκυψαν κατά την εκπόνηση και συγγραφή τη διπλωματικής.

## 1.2 Εισαγωγή στους Deductive Verifiers

Οι deductive verifiers διαχωρίζονται κυρίως σε δύο κατηγορίες [1]:

Στη πρώτη κατηγορία ανήκουν οι verifiers που βασίζονται στο Satisfiability Modulo Theory (SMT) και χρησιμοποιούν διαδικασίες λήψης γρήγορων αποφάσεων για την αυτοματοποίηση της επαλήθευσης προγραμμάτων που απαιτούν μόνο συλλογιστική πάνω σε ένα σταθερό

σύνολο θεωριών όπως γραμμική αριθμητική (linear arithmetic), συμβολοσειρές, θεωρία συνόλων και λειτουργίες σε bitvector. Αυτοί οι verifiers, ωστόσο, κωδικοποιούν τη σημασιολογία των καθορισμένων από το χρήστη συναρτήσεων με αξιώματα με καθολικούς ποσοδείκτες και χρησιμοποιούν ελλιπή (αν και αποτελεσματικά) ευρετικά για να δομήσουν αυτά τα αξιώματα. Αυτά τα ευρετικά στοιχεία καθιστούν δύσκολο τον χαρακτηρισμό των ειδών αποδείξεων που μπορούν να αυτοματοποιηθούν και, ως εκ τούτου, να εξηγήσουν γιατί αποτυγχάνει μια δεδομένη απόδειξη.

Στη δευτερη κατηγορία ανήκουν οι verifiers που βασίζονται στη θεωρία τύπων (Type Theory) (π.χ Coq και Agda) που χρησιμοποιούν υπολογισμούς σε επίπεδο τύπων (normalization) προκειμένου να κάνουν ευκολότερη τη συλλογιστική σχετικά με το τερματισμό συναρτήσεων που ορίζει ο χρήστης, αλλά απαιτούν από τον χρήστη να παρέχει λήμματα ή να ξαναγράψει στοιχεία που βοηθούν στην απόρριψη αποδείξεων πάνω σε αποκρίσιμες θεωρίες.

Στη συγκεκριμένη διπλωματική εργασία οι verifiers που θα συναντήσουμε είναι:

- **Coq**: Το Coq είναι ένας deductive verifier που κατατάσσεται στη δεύτερη κατηγορία δηλαδή βασίζεται στη θεωρία τύπων. Το Coq μας προσφέρει μια καθαρά συναρτησιακή γλώσσα προγραμματισμού με την οποία μπορούμε να γράψουμε και να επαληθευσουμε ιδιότητες προγραμμάτων. Θα συναντήσουμε το Coq σε ορισμένα κεφάλαια καθώς χρησιμοποιείται στο βιβλίο Software Foundations, αλλά δεν θα ασχοληθούμε πολύ με το Coq.

- **Liquid Haskell**: Η Liquid Haskell κατατάσσεται στην πρώτη κατηγορία καθώς εισάγει την έννοια του Refinement Reflection [1], ένα νέο πλαίσιο για την κατασκευή deductive verifiers που βασίζονται στην SMT θεωρία, το οποίο επιτρέπει την ανάλυση και επαλήθευση ιδιοτήτων προγραμμάτων. Θα συναντήσουμε και θα αναλύσουμε την Liquid Haskell και το τρόπο με το οποίο εκτελεί συλλογιστική στα προγράμματα σε όλη την έκταση αυτής της διπλωματικής εργασίας.

## 1.3 Σύνοψη της εργασίας

Η διπλωματική εργασία έχει την ακόλουθη δομή:

**Κεφάλαιο 2$^{ο}$ :** Εισαγωγή σε βασικές έννοιες όπως refinement types, Specification και Verification. Αναλύουμε επίσης την έννοια του refinement reflection, ως την βασική τεχνική που μας επιτρέπει να εκτελούμε συλλογιστική και να αποδεικνύουμε ιδιότητες προγραμμάτων Haskell στη Liquid Haskell.

**Κεφάλαιο 3$^{ο}$ :** Στο 3$^{ο}$ κεφάλαιο αναλύουμε πως η Liquid Haskell κάνει συλλογιστική στα προγράμματα και κατηγοριοποιούμε τα είδη της συλλογιστικής σε Lightweight και Deep reasoning. Επιπλέον επισημαίνουμε πως η Liquid Haskell αυτοματοποιεί τις αποδείξεις χρησιμοποιώντας Proof by Logic Evaluation. Τέλος στο κεφάλαιο αυτό αναλύουμε τις έννοιες του totality και termination στις συναρτήσεις και τον τόπο που αυτές ελέγχονται στη Liquid Haskell.

**Κεφάλαιο 4$^{ο}$:** Το κεφάλαιο αυτό αποτελεί το πιο βασικό κεφάλαιο της διπλωματικής εργασίας. Στο 4$^{ο}$ κεφάλαιο αναλύουμε και επαληθεύουμε ιδιότητες αλγορίθμων και δομών δεδομένων με την εξής σειρά:

4.1 Λίστες
4.2 Permutations
4.3 Insertion Sort
4.4 Selection Sort
4.5 Binary Search Trees
4.6 Abstract Data Types
4.7 Priority Queues
4.8 Trie: Number representation and efficient lookup tables
4.9 Red Black Trees
4.10 Binomial Queues

**Κεφάλαιο 5$^{ο}$** : Συμπεράσματα και προτάσεις μελλοντικής έρευνας.

# 2 Βασικές έννοιες στη Liquid Haskell

## 2.1 Refinement Types και Specifications

Refinement types (εκλέπτυνση τύπων) είναι οι τύποι του προγράμματος του πηγαίου κώδικα (στη συγκεκριμένη περίπτωση πρόγραμμα γραμμένο σε γλώσσα προγραμματισμου Haskell) που έχουν εμπλουτιστεί με λογικά κατηγορήματα που προέρχονται από μια αποκρίσιμη SMT λογική. Χρησιμοποιούμε λογική χωρίς ποσοδείκτες, με ισότητα (quantifier-free logic of equality), ανερμήνευτες συναρτήσεις (uninterpreted functions ) και γραμμική αριθμητική (linear arithmetic) (QF-EUFLIA). Για παράδειγμα, ορίζουμε ως Nat το σύνολο των τιμών Integer v που ικανοποιούν το κατηγόρημα $0 \leq v$, [2]:

**type Nat = { v:Integer | 0 ≤ v }**

Για τη σύνταξη των refinement types η Liquid Haskell επεκτείνει τη σύνταξη της Haskell ερμηνεύοντας τα σχόλια της μορφής {-@ ... @-} ως δηλώσεις για refinements. Τα refinement types [3] μας επιτρέπουν να εμπλουτίσουμε το σύστημα τύπων της Haskell με κατηγορήματα που περιγράφουν με ακρίβεια τα σύνολα των έγκυρων εισόδων και εξόδων των συναρτήσεων. Αυτά τα κατηγορήματα προέρχονται από ειδικές λογικές για τις οποίες υπάρχουν γρήγορες διαδικασίες λήψης αποφάσεων που ονομάζονται SMT solvers. Επομένως μας επιτρέπουν να συντάξουμε προδιαγραφές (Specifications) για τα προγράμματα μας, τα οποία θα ελεγχθούν ως προς την εγκυρότητα τους από τον SMT solver. Συνδυάζοντας τύπους και κατηγορήματα επιτυγχάνεται ο καθορισμός συμβολαίων (contracts) που περιγράφουν τις έγκυρες εισόδους και εξόδους λειτουργιών. Έτσι το refinement type system εγγυάται κατά τον χρόνο μεταγλώττισης ότι οι συναρτήσεις τηρούν αυτά τα συμβόλαια και κατά αυτόν το τρόπο να αποφευχθούν σφάλματα κατά την εκτέλεση. Η Liquid Haskell αποτελεί έναν ελεγκτή των refinement type για τη Haskell.

## 2.2 Verification

Η Liquid Haskell λειτουργεί χωρίς να εκτελεί τα προγράμματα. Αντίθετα ελέγχει αν το πρόγραμμά πληροί τις προδιαγραφές ουσιαστικά σε δύο στάδια.

- Πρώτον, η Liquid Haskell συνδυάζει τον κώδικα και τους τύπους σε ένα σύνολο από verification conditions (VC), τα οποία είναι έγκυρα μόνο αν το πρόγραμμά ικανοποιεί μια δεδομένη ιδιότητα.

- Στη συνέχεια, η LH ρωτάει τον SMT solver για να προσδιορίσει εάν αυτά τα VCs είναι έγκυρα. Εάν ναι, λέει ότι το πρόγραμμά είναι ασφαλές, διαφορετικά απορρίπτει το πρόγραμμά.

Ο SMT solver αποφασίζει αν ένα predicate (VC) είναι έγκυρο χωρίς να απαριθμήσει και να αξιολογήσει όλες τις δυνατές αναθέσεις. Πράγματι, είναι αδύνατο να το πράξουμε, καθώς συνήθως υπάρχουν άπειρες πολλές αναθέσεις όταν οι πρόβες αναφέρονται σε ακέραιους ή λίστες και ούτω καθεξής. Αντίθετα, ο διαλυτής SMT χρησιμοποιεί μια ποικιλία περίπλοκων συμβολικών αλγορίθμων για να συναγάγει αν ένα κατηγόρημα είναι έγκυρο ή όχι.

Περιορίζουμε τη λογική για να διασφαλίσουμε ότι όλα τα κατηγορήματα VC μας είναι αποκρίσιμα, δηλαδή μπορούμε να αποφανθούμε αν τερματίζουν και είναι έγκυρα η όχι. Αυτό καθιστά τη Liquid Haskell εξαιρετικά αυτόματη - δεν υπάρχει σαφής χειρισμός των αποδείξεων, μόνο οι προδιαγραφές των ιδιοτήτων μέσω τύπων και φυσικά η εφαρμογή μέσω κώδικα Haskell. Ωστόσο αυτή η αυτοματοποίηση επιβάλλει ότι όλες οι προδιαγραφές πρέπει να ανήκουν στη παραπάνω λογική (QF-EUFLIA) [4].

Ως παράδειγμα μπορούμε να γράψουμε τη συνάρτηση fibonacci που υπολογίζει τους αριθμού Fibonacci:

```
{-@ fib :: Nat → Nat -@}
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Στη παραπάνω συνάρτηση έχουμε προσθέσει σαν specification ότι η είσοδος και η έξοδος της συνάρτησης είναι φυσικοί αριθμοί.

Για να διασφαλιστεί ο τερματισμός, η προδιαγραφή του τύπου εισόδου καθορίζει μια προϋπόθεση(pre-condition) ότι η παράμετρος πρέπει να είναι φυσικός αριθμός (Nat). Η προδιαγραφή του τύπου εξόδου καθορίζει μια μετα-προϋπόθεση (post-condition) ότι το αποτέλεσμα είναι επίσης φυσικός αριθμός. Ο έλεγχος των refinement types εισόδου και εξόδου ελέγχει αυτόματα ότι εάν η συνάρτηση κληθεί με ένα μη αρνητικό ακέραιο, τότε τερματίζει και επιστρέφει ένα μη αρνητικό ακέραιο [2].

## 2.3 Refinement Reflection

Στο συγκεκριμένο κεφάλαιο αναλύουμε την έννοια του Refinement Reflection [2][18] που αποτελεί μία μέθοδο με την οποία επεκτάθηκε η Liquid Haskell ώστε να λειτουργεί ως theorem prover. Τα refinement types όπως παρουσιάζονται μέχρι τώρα, προσφέρουν ήδη μια μορφή προγραμματισμού με αποδείξεις. Για να εξηγήσουμε την έννοια του refinement reflection θα εξετάσουμε ένα παράδειγμα ενός θεωρήματος που θα θέλαμε να αποδείξουμε:

Έστω λοιπόν ότι θέλουμε να αποδείξουμε κάποιες ιδιότητες για τη συνάρτηση **fib** που ορίσαμε στο προηγούμενο κεφάλαιο, για παράδειγμα έστω ότι θα επιθυμούσαμε να αποδείξουμε ότι:

**{ fib 2 = 1}**

Χρησιμοποιώντας την καθιερωμένη διαδικασία ελέγχου των refinement types όπως περιγράφηκε στο προηγούμενο κεφάλαιο θα συναντούσαμε δύο προβλήματα:

1. Πρώτον, για να διατηρήσουμε την αποκρισιμότητα και την ορθότητα, οι αυθαίρετες συναρτήσεις που ορίζονται από το χρήστη δεν μπορούν να ανήκουν στη refinement λογική. Δηλαδή δεν μπορούμε να αναφερθούμε στη συνάρτηση fib μέσα σε κάποιο refinement.

2. Δεύτερον, η μόνη προδιαγραφή που υπάρχει και μπορεί να χρησιμοποιηθεί κατά τον έλεγχο του refinement type της συνάρτησης fib είναι ότι η συνάρτηση έχει τύπο $Nat \longrightarrow Nat$, κάτι που είναι πολύ ασθενές για να αποδείξουμε το ζητούμενο δηλαδή ότι $\{\text{fib } 2 = 1\}$.

Για να αντιμετωπίσουμε και τα δύο προβλήματα, αντανακλάμε (reflect) τη συνάρτηση fib στην λογική, πράγμα που ενεργοποιεί τα παρακάτω τρία βήματα του refinement reflection:

Βήμα 1ο: Ορισμός: Ο ορισμός της συνάρτησης fib δημιουργεί μία ανερμήνευτη συνάρτηση $fib :: Int \longrightarrow Int$ στη refinement λογική. Με τον όρο ανερμήνευτη (uninterpreted) εννοούμε ότι η fib που ορίστηκε στη λογική δεν συνδέεται με τη συνάρτηση fib που έχουμε ορίσει στο πρόγραμμά μας. Στη λογική, για τη fib το μόνο που γνωρίζουμε είναι ότι ικανοποιεί το αξίωμα της ισότητας (congruence axiom): $\forall m, n.\ n = m \implies fib\ n = fib\ m$. Έτσι από μόνη της μια ανερμήνευτη συνάρτηση όπως η fib δεν είναι εξαιρετικά χρήσιμη καθώς για να αποδείξει ο SMT solver ότι $\{\text{fib } 2 = 1\}$ απαιτείται συλλογιστική στον ορισμό της συνάρτησης fib.

Βήμα 2ο :Reflection : Στο επόμενο βασικό βήμα αντικατοπτρίζουμε τον ορισμό της συνάρτησης fib στο refinement type της συνάρτησης ενισχύοντας τον τύπο που είχε ορίσει ο χρήστης στον εξής τύπο:

$$\text{fib} :: \text{n: Nat} \rightarrow \{\text{v: Nat} \mid \text{v} = \text{fib n} \ \&\& \ \text{fibP n}\}$$

όπου fibP αναφέρεται σε ένα refinement που έχει προκύψει από τον ορισμό της συνάρτησης fib, δηλαδή ουσιαστικά ο ορισμός της συνάρτησης fib έχει μεταφερθεί στη refinement λογική με το όνομα fibP.

$$\begin{aligned}\text{fibP n} = \ & \text{n} == 0 \ \Rightarrow \text{fib n} = 0 \\ & \wedge \ \text{n} == 1 \ \Rightarrow \ \text{fib n} = 1 \\ & \wedge \text{n} > 1 \quad \Rightarrow \text{fibn} = \text{fib}(\text{n}-1) + \text{fib}(\text{n}-2)\end{aligned}$$

Βήμα 3ο: Εφαρμογή: Χρησιμοποιώντας το reflected refinement , κάθε εφαρμογή της συνάρτησης fib στον κώδικα αυτόματα ξεδιπλώνει κατά ένα βήμα τον ορισμό του fib στη λογική. Έτσι μπορούμε να αποδείξουμε ότι $\{\text{fib } 2 = 1\}$ με το εξής:

```
pf_fib2 :: { fib 2 = 1 }
pf_fib2 = let { t0 = fib 0; t1 = fib 1; t2 = fib 2 } in ()
```

Γράφουμε με έντονους χαρακτήρες, fib, για να επισημάνουμε τα σημεία όπου είναι σημαντικό το ξεδίπλωμα του ορισμού της f. Τα παραπάνω αποδίδουν το ακόλουθο Verification Condition που αποδεικνύεται από τον SMT, παρόλο που η fib δεν έχει ερμηνευτεί:

$$(\text{fibP } 0 \wedge \text{fibP } 1 \wedge \text{fibP } 2) \Rightarrow (\text{fib } 2 = 1)$$

Η επαλήθευση του pf_fib2 εξαρτάται απλώς από το γεγονός ότι η fib εφαρμόζεται (δηλ. ξεδιπλώνεται) στα 0, 1 και 2 και έτσι ο SMT solver συνδυάζει αυτόματα τα γεγονότα, μόλις βρεθούν στη προηγούμενη μορφή. Επομένως, και τα παρακάτω ομοίως επιβεβαιώνονται:

```
pf_fib2' :: {v:[Nat] | fib 2 = 1 }
pf_fib2'  = [ fib 0, fib 1, fib 2 ]
```

# 3 Reasoning about Programs

Η Liquid Haskell αποσκοπεί στο να μεταφέρει τη συλλογιστική για τα προγράμματα σε Haskell στα ίδια τα προγράμματα και να αυτοματοποιήσει αυτή τη διαδικασία όσο το δυνατόν περισσότερο. Αυτό επιτυγχάνεται μέσω των refinement types τα οποία ελέγχονται από έναν SMT solver. Η συλλογιστική των προγραμμάτων διαχωρίζεται στις παρακάτω κατηγορίες [5]:

## 3.1 Κατηγορίες Reasoning

### 3.1.1 Lightweight Reasoning

Ιδιότητες οι οποίες μπορούν να επαληθευτούν στη Liquid Haskell αυτόματα από τον SMT solver, χωρίς ο χρήστης να παρέχει κάποια απόδειξη ονομάζονται Lightweight ιδιότητες του προγράμματος. Τέτοιες ιδιότητες βασίζονται αποκλειστικά στη γνώση και συλλογιστική που έχει ο SMT solver και εμπίπτουν στις παρακάτω υποκατηγορίες.

**Linear Arithmetic:** Πολλές ιδιότητες προγραμμάτων που αφορούν αριθμητική μπορούν να αποδειχθούν αυτόματα από τον SMT solver. Για παράδειγμα έστω η παρακάτω συνάρτηση υπολογισμού του μήκους μιας λίστας:

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

Έστω ότι για την παραπάνω συνάρτηση θεωρούμε χρήσιμο να προσδιορίσουμε ότι το μήκος μιας λίστας είναι πάντα μη αρνητικός αριθμός. Εκφράζουμε αυτή την ιδιότητα με ένα refinement type ως εξής:

```
{-@ length:: [a] → {v:Int | 0 <= v } @-}
```

Η Liquid Haskell είναι ικανή να επαλήθευση αυτή την ιδιότητα αυτόματα χρησιμοποιώντας τη γνώση που έχει ο SMT solver στην αριθμητική:

- Στην πρώτη εξίσωση στον ορισμό της συνάρτησης length, η τιμή v είναι 0, έτσι ο SMT solver μπορεί να αποφανθεί ότι $0 \leq v$.

- Στη δεύτερη εξίσωση, η τιμή v είναι 1 + v ', όπου v' είναι το αποτέλεσμα της αναδρομικής κλήσης στο μήκος xs. Από το refinement type της length, η Liquid Haskell γνωρίζει ότι $0 \leq v$ 'και ο SMT solver μπορεί να συμπεράνει ότι $0 \leq v$.

Είδαμε λοιπόν ότι η απόδειξη ότι το μήκος μιας λίστας είναι πάντα μη αρνητικός αριθμός γίνεται αυτόματα από τον SMT solver καθώς μπορεί να αποφανθεί αποτελεσματικά για ερωτήματα σχετικά με αριθμητική. Επίσης να τονίσουμε ότι στο refinement type της length δεν γίνεται αναφορά στην αναδρομική συνάρτηση length.

**Measures**: Προκειμένου να γίνεται αναφορά σε Haskell συναρτήσεις μέσα στα refinement

types θα πρέπει να τις αναπαραστήσουμε στο επίπεδο των refinement types. Η Liquid Haskell προσφέρει ένα μηχανισμό με τον οποίο μπορούμε να πετύχουμε αυτή την αναπαράσταση ορισμένων ειδικών κατηγοριών συναρτήσεων που ονομάζονται measures. Ως measure ορίζουμε την κατηγορία συναρτήσεων στην οποία ανήκουν οι συναρτήσεις που:

1. Δέχονται μόνο μία παράμετρο, η οποία πρέπει να είναι ένας αλγεβρικός τύπος δεδομένων.
2. Έχουν οριστεί από μία μόνο εξίσωση για κάθε constructor.
3. Στο σώμα της συνάρτησης καλούνται μόνο αριθμητικές συναρτήσεις και συναρτήσεις που είναι measures.

Για την παραπάνω κλάση συναρτήσεων τα refinement type μπορούν να ελεγχθούν αυτόματα.

Για παράδειγμα η συνάρτηση length όπως ορίστηκε παραπάνω είναι measure καθώς παίρνει μόνο μία παράμετρο, έχει μόνο μία εξίσωση για κάθε constructor της παραμέτρου (μία εξίσωση για τη κενή λίστα και μία εξίσωση για την λίστα με ένα η περισσότερα στοιχεία) και στο σώμα της συναρτήσεις καλούμε μόνο την ίδια τη συνάρτηση και τον αριθμητικό τελεστή (+). Για να αναφερθούμε στη συνάρτηση length μέσα σε κάποιο refinement, τη δηλώνουμε ως measure μέσω της παρακάτω δήλωσης:

```
{-@ measure length @-}
```

Για παράδειγμα μπορούμε τώρα χρησιμοποιήσουμε τη συνάρτηση length μέσα σε refinement type για να δηλώσουμε ότι το μήκος της λίστας που προκύπτει από τη συνένωση δύο λιστών ισούται με το άθροισμα των μηκών των δύο λιστών:

```
{-@ (++) :: xs:[a] → ys:[a] →
     {zs:[a] | length zs == length xs + length ys}
@-}

(++) :: [a] → [a] → [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Η Liquid Haskell ελέγχει το παραπάνω παράδειγμα σε δύο βήματα:

- Στην πρώτη εξίσωση του ορισμού της συνάρτηση (++) η λίστα xs είναι κενή οπότε έχει μήκος 0, και ο SMT solver μπορεί να επίλυση αυτή περίπτωση χρησιμοποιώντας γραμμική αριθμητική.

- Στη περίπτωση της δεύτερης εξίσωσης η λίστα εισόδου έχει τη μορφή x:xs οπότε το μήκος της είναι 1+ length xs , ενώ η αναδρομική κλήση επίσης υποδηλώνει ότι length (xs++ys) = length xs + length ys , οπότε ο solver μπορεί να επιλύσει και αυτή τη περίπτωση χρησιμοποιώντας αριθιτική.

### 3.1.2 Deep Reasoning

Στη προηγούμενη ενότητα του κεφαλαίου είδαμε ότι συναρτήσεις που μπορούν να οριστούν ως measures μπορούν να χρησιμοποιηθούν στη refinement λογική και να διατηρώντας εντελώς

αυτοματοποιημένη συλλογιστική. Ωστόσο αυτό δεν ισχύει και για τις συναρτήσεις που δεν ανήκουν στην κλάση των measures (για παράδειγμα η συνάρτηση (++) που εξετάσαμε προηγουμένως). [5]

Η Liquid Haskell εξακολουθεί να επιτρέπει τη συλλογιστική σχετικά με τέτοιες συναρτήσεις, αλλά αυτός ο περιορισμός σημαίνει ότι ο χρήστης μπορεί να χρειαστεί να παράσχει τις αποδείξεις. Ως deep reasoning ιδιότητες ορίζουμε τις ιδιότητες αυτές που δεν μπορεί να επαληθεύσει αυτόματα ο SMT solver.

Σαν παράδειγμα θα εξετάσουμε τη συνάρτηση reverse που δέχεται σαν είσοδο μία λίστα και επιστρέφει σαν έξοδο την αντίστροφη λίστα:

```
{-@ reverse :: is:[a] → {os:[a] | length is == length os} @-}

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

Η συνάρτηση δεν είναι measure καθώς στο σώμα της έχουμε κλήση της συνάρτησης (++) που δεν είναι measure, οπότε η συλλογιστική σχετικά με τη συνάρτηση reverse δεν μπορεί να είναι πλήρως αυτόματη. Για να χρησιμοποιήσουμε συναρτήσεις που δεν είναι measures, σε επίπεδο refinement χρησιμοποιούμε την έννοια του reflection όπως έχει περιγραφεί στο προηγούμενο κεφάλαιο. Το reflection παρέχει στον SMT solver μόνο τη τιμή της συνάρτησης για τα ορίσματα με τα οποία καλείται. Έτσι περιορίζοντας τις πληροφορίες που παρέχουμε στον solver κατά αυτόν το τρόπο ο έλεγχος των refinement types παραμένει αποκρίσιμος.

Για να δούμε τα παραπάνω στη πράξη θα αποδείξουμε ότι η αντίστροφη λίστα της λίστας που περιέχει μόνο ένα στοιχείο είναι η ίδια η λίστα δηλαδή: reverse [x] == x. Αρχικά ορίζουμε τις συναρτήσεις reverse και ++ ως reflected:

```
{-@ reflect reverse @-}
{-@ reflect ++      @-}
```

Στη συνέχεια εισάγουμε μία νέα συνάρτηση singletonP της οποίας το refinement type εκφράζει τη πρόταση που θέλουμε να αποδείξουμε και το σώμα της συνάρτησης παρέχει την απόδειξη σε μορφή equational reasoning:

```
{-@ singletonP :: x:a → {reverse [x] == [x]} @-}

singletonP :: a → Proof
singletonP x = reverse [x]
                  -- εφαρμόζουμε reverse στο [x]
             ==. reverse [] ++ [x]
                  -- εφαρμόζουμε reverse στο []
             ==. [] ++ [x]
                 -- εφαρμόζουμε ++ στο [] και [x]
             ==. [x] *** QED
```

Μπορούμε να σκεφτούμε την παραπάνω συνάρτηση σαν μια απεικόνιση που απεικονίζει μια τιμή x σε μία απόδειξη: reverse [x] = [x]. Ο τύπος Proof που συμβολίζει την απόδειξη είναι απλά ο τύπος μονάδας της Haskell (τον συμβολίζουμε με ( ) ) και το {reverse [x] == [x]} είναι

μια απλοποίηση του {v:() | reverse [x] == [x]}, δηλαδή ενός refinement για το τύπο μονάδας. Αυτή η σύνταξη κρύβει τη τιμή v (τύπου μονάδας) που δεν σχετίζεται με το θεώρημα.

Εδώ να σημειώσουμε ότι το σώμα της συνάρτησης singletonP μοιάζει πολύ με ένα μια τυπική απόδειξη που θα γράφαμε στο χαρτί. Η αντιστοιχία είναι τόσο κοντά που ισχυριζόμαστε ότι το να αποδείξουμε μια ιδιότητα στη liquid Haskell μπορεί να είναι εξίσου εύκολο όσο αποδεικνύεται σε χαρτί σε equational reasoning μορφή, αλλά η απόδειξη στο Liquid Haskell ελέγχεται μηχανικά. Η Liquid Haskell χρησιμοποιεί τον SMT solver για να ελέγξει την παραπάνω απόδειξη. Επειδή το σώμα της συνάρτηση singletonP περιέχει τους όρους reverse [x], reverse [] και []++[x] η Liquid Haskell περνάει τις αντίστοιχες εξισώσεις στον solver:

```
reverse [x] = reverse [] ++ [x]
reverse [] = []
[] ++ [x] = [x]
```

Με τις παραπάνω εξισώσεις ο solver μπορεί να συμπεράνει το ζητούμενο, δηλαδή ότι reverse [x] = [x].

Εδώ να σημειώσουμε ότι ο εξής ορισμός θα ήταν επίσης μια ορθή απόδειξη:

```
singletonP x = const () (reverse [x], reverse [], [] ++ [x])
```

Ωστόσο είναι προφανές ότι μια τέτοια συμπιεσμένη απόδειξη δεν είναι ούτε εύκολη να τη σκεφτεί ο χρήστης άμεσα, ούτε είναι ευανάγνωστη. Επομένως στη Liquid Haskell χρησιμοποιούμε proof combinators για να συντάξουμε ευανάγνωστες αποδείξεις σε equational μορφή, όπου κάθε ενδιάμεσο βήμα ελέγχεται ως προς την ορθότητα του.

Στη συνέχεια θα αναλύσουμε τα αποδεικτικά στοιχεία (proof combinators) που παρέχονται από τη Liquid Haskell, τα οποία είναι απλές Haskell συναρτήσεις που εξυπηρετούν το χρήστη στη συγγραφή αποδείξεων. Είδαμε παραπάνω ότι ο τύπος μονάδας της Haskell αναπαριστά το τύπο μιας απόδειξης Proof δηλαδή:

```
type Proof = ()
```

Ο τύπος () είναι επαρκής καθώς ένα θεώρημα εκφράζεται ως refinement στα ορίσματα μιας συνάρτησης και η τιμή που επιστρέφει ένα θεώρημα δεν έχει καμία σημασία.

Η πιο βασική συνάρτηση που ορίζεται στο Proof combinators module είναι η *** η οποία δέχεται ένα όρισμα και επιστρέφει πάντα το τύπο Proof αγνοώντας την τιμή του ορίσματος:

```
data QED = QED
(***) :: a → QED → Proof
_ *** QED = ()
infixl 2 ***
```

Ο τύπος δεδομένων QED που ορίστηκε παραπάνω χρησιμοποιείται μόνο για αισθητικούς λόγους και το χρησιμοποιούμε για να δηλώσουμε το τέλος μιας απόδειξης.

Ο σημαντικότερος τελεστής που παρέχει η Liquid Haskell για τη συγγραφή equational αποδείξεων είναι ο τελεστής **==.** ο οποίος δέχεται δύο ορίσματα και το refinement type του

τελεστή αυτού διασφαλίζει ότι τα ορίσματα είναι ίσα και επιστρέφει το δεύτερο όρισμα. Κατά αυτόν το τρόπο είναι εύκολο πολλαπλές χρήσης του τελεστή ==. να συνδεθούν μεταξύ τους:

```
{-@ (==.) :: x:a → y:{a | x == y} → {o:a | o == y && o == x} @-}
(==.) :: a → a → a
_ ==. x = x
```

Εξίσου σημαντικός είναι και ο τελεστής **(?)** με τον οποίο μπορούμε να αναφερθούμε σε άλλα θεωρήματα που χρησιμοποιούμε στην απόδειξη μας. Επειδή τα θεωρήματα είναι συναρτήσεις Haskell το μόνο που χρειάζεται είναι ένας τελεστής που δέχεται ένα όρισμα τύπου Proof και ορίζεται ως εξής:

```
(?) :: a → Proof → a
x?_=x
```

Ως παράδειγμα έστω ότι θέλαμε να αποδείξουμε ότι ισχύει reverse [1] == [1] :

```
{-@ singleton1P :: { reverse [1] == [1] } @-}
singleton1P
    = reverse [1]
    ==. [1]    ? singletonP 1
    *** QED
```

Στο παραπάνω χρησιμοποιούμε το τελεστή ? για να χρησιμοποιήσουμε το θεώρημα singletonP με είσοδο τη τιμή 1.

Τέλος ο τελευταίος τελεστής που θα αναλύσουμε καθώς θα τον χρησιμοποιήσουμε αρκετά στο κεφάλαιο 4 είναι ο τελεστής **&&&** με τον οποίο μπορούμε να αναφερθούμε σε περισσότερα θεωρήματα που χρειάζονται για την απόδειξή μας. Τα θεωρήματα συνδέονται με τη χρήση του τελεστή αυτού. Όπως και στον τελεστή **==. ,** πολλαπλές χρήσης του τελεστή **&&&** μπορούν να συνδεθούν μεταξύ τους.

```
(&&&) :: Proof -> Proof -> Proof
x &&& _ = x
```

### 3.1.3 Επαγωγή σε λίστες

Η δομική επαγωγή είναι μια θεμελιώδης τεχνική για την απόδειξη ιδιοτήτων των συναρτησιακών προγραμμάτων. Για τον τύπο λίστας στη Haskell, η αρχή της επαγωγής δηλώνει ότι για να αποδειχθεί ότι μια ιδιότητα ισχύει για όλες τις (πεπερασμένες) λίστες, αρκεί να δείξουμε ότι [5]:

- Ισχύει για τη κενή λίστα (βάση της επαγωγής)
- Ισχύει για κάθε μη-κενή λίστα x:xs υποθέτοντας ότι ισχύει για την ουρά xs της λίστας (επαγωγικό βήμα)

Η επαγωγή δεν απαιτεί να εισαχθεί νέος proof combinator. Αντιθέτως, οι αποδείξεις με επαγωγή μπορούν να εκφραστούν ως αναδρομικές συναρτήσεις στη Liquid Haskell. Για

παράδειγμα, θα δείξουμε ότι για μία λίστα xs ισχύει (xs ++ []) == xs. Εκφράζουμε αυτήν την ιδιότητα ως τον τύπο της συνάρτησης appendNilId, του οποίου το σώμα αποτελεί την απόδειξη:

```
{-@ appendNilId ::  xs:_ -> { xs ++ [] = xs } @-}
appendNilId     :: [a] -> Proof
appendNilId []
  =   [] ++ []
  ==. []
  *** QED
appendNilId (x:xs)
   =   (x:xs) ++ []
   ==. x:(xs ++ xs)
   ==. (x:xs)        ? appendNilId xs
*** QED
```

Επειδή η συνάρτηση appendNilId είναι αναδρομική μπορούμε να χρησιμοποιήσουμε επαγωγή:

- Στην βάση της επαγωγής, επειδή το σώμα της συνάρτησης περιέχει τους όρους []++[] και [], οι αντίστοιχες εξισώσεις μεταφέρονται στον SMT solver, ο οποίος αποδεικνύει ότι []++[] == [].

- Στο επαγωγικό βήμα πρέπει να δείξουμε ότι (x:xs) ++ [] == (x:xs) πράγμα που επιτυγχάνεται σε αρκετά βήματα. Η εγκυρότητα κάθε βήματος ελέγχεται από την Liquid Haskell κατά την επαλήθευση ότι ικανοποιείται το refinement type του τελεστή (==.). Ορισμένα από τα βήματα προκύπτουν άμεσα από τους ορισμούς ενώ στο τελευταίο βήμα επικαλούμαστε την επαγωγική υπόθεση μέσω του τελεστή ?.

## 3.2 Automating Equational Reasoning

Στις προηγούμενες αποδείξεις που εξετάσαμε μέχρι τώρα, γράψαμε ρητά κάθε βήμα της απόδειξης αποτιμώντας και αναπτύσσοντας ορισμούς συναρτήσεων. Γράφοντας αποδείξεις ρητά με αυτόν τον τρόπο είναι συχνά αρκετά χρήσιμο αλλά μπορεί να γίνει αρκετά κουραστικό.

Για να απλοποιήσει τις αποδείξεις, η Liquid Haskell χρησιμοποιεί την πλήρη τεχνική απόδειξης Proof By (Logical) Evaluation (PLE) [1][5][18], και επιπλέον η Liquid Haskell εγγυάται τον τερματισμό της τεχνικής PLE. Εννοιολογικά, το PLE εκτελεί τις συναρτήσεις για όσα βήματα χρειάζεται και παρέχει αυτόματα όλες τις εξισώσεις που προκύπτουν στον SMT solver.

Παραπάνω αναλύσαμε την απόδειξη του θεωρήματος appendNilId χωρίς PLE. Παρακάτω παρουσιάζουμε την απόδειξη του θεωρήματος χρησιμοποιώντας PLE:

```
{-@ appendNilId:: xs:[a] → { xs ++ [] == xs } @-}

{-@ ple appendNilId @-}
 appendNilId:: [a] → Proof
 appendNilId [] = ()
```

```
appendNilId (_:xs) = appendNilId xs
```

Για να ενεργοποιήσουμε το PLE είτε γράφουμε το όνομα της συνάρτησης μέσα σε: {-@ ple appendNilId @-} είτε προσθέτουμε στο αρχείο με το κώδικα το flag {-@ LIQUID "--ple" @}.

Στη παραπάνω απόδειξη στη βάση της επαγωγής για την κενή λίστα η απόδειξη γίνεται πλήρως αυτοματοποιημένα. Για το επαγωγικό βήμα το μόνο που γράφουμε είναι είναι μια αναδρομική κλήση για να επικαλεστούμε την επαγωγική υπόθεση αλλά η υπόλοιπη διαδικασία γίνεται αυτόματα από τη τεχνική PLE.

Το PLE είναι ένα ισχυρό εργαλείο που κάνει τις αποδείξεις πιο σύντομες και ευκολότερες στην εγγραφή. Ωστόσο, οι αποδείξεις που χρησιμοποιούν αυτή την τεχνική είναι συνήθως πιο δύσκολο να διαβαστούν, καθώς κρύβουν τις λεπτομέρειες της επέκτασης της συνάρτησης.

## 3.3 Έλεγχος Totality και Termination

Σε αυτό το σημείο να τονίσουμε ότι η χρήση μιας αναδρομικής συνάρτησης για να μοντελοποιήσει μια απόδειξη με επαγωγή δεν είναι ορθή εάν η αναδρομική συνάρτηση είναι μερική ή δεν τερματίζει. Ωστόσο, η Liquid Haskell παρέχει επίσης έναν ισχυρό έλεγχο προκειμένου να αποφανθεί αν μία συνάρτηση είναι ολική (total) και τερματίζει, και απορρίπτει οποιαδήποτε συνάρτηση για τον οποία δεν μπορεί να αποδειχθεί ότι είναι ολική και τερματίζει [1][5].

### 3.3.1 Έλεγχος για Totality

Η Liquid Haskell χρησιμοποιεί το μηχανισμό του GHC για να συμπληρώσει τα πρότυπα των συναρτήσεων που δεν είναι ολικές. Για παράδειγμα αν η συνάρτηση appendNilId είχε οριστεί μόνο για την κενή λίστα δηλαδή:

```
appendNilId:: [a] → Proof
appendNilId [] = ()
```

τότε δοκιμάζοντας τη παραπάνω συνάρτηση στη Liquid Haskell, το μήνυμα λάθους που θα εμφανιστεί είναι:

*Your function isn't total:*
*some patterns aren't defined*.

Για να επιτευχθεί αυτός ο έλεγχος για totality συναρτήσεων, ο GCH συμπληρώνει τον ορισμό της συνάρτησης appendNilId χρησιμοποιώντας τη συνάρτηση **patError**:

```
appendNilId [] = ()
appendNilId _ = patError "function involutionP"
```

Η Liquid Haskell επιτυγχάνει τον έλεγχο για totality, ορίζοντας τον εξής refinement τύπο για τη συνάρτηση patError**:**

{-@ patError :: { i:String | false } → a @-}

Στο παραπάνω refinement type η Liquid Haskell προσθέτει μία υπόθεση που είναι πάντα ψευδής. Αυτό διότι δεν υπάρχει όρισμα που να ικανοποιεί το false, οπότε κάθε κλήση στη συνάρτηση patError δεν μπορεί να αποδειχθεί ότι είναι «νεκρός» κώδικας, δηλαδή κώδικας που δεν χρησιμοποιείται ποτέ από το πρόγραμμά μας, και η Liquid Haskell τυπώνει σφάλμα ότι η συνάρτηση που κάλεσε την patError δεν είναι ολική.

### 3.3.2 Έλεγχος για Τερματισμό

Η Liquid Haskell ελέγχει ότι όλες οι συναρτήσεις τερματίζουν χρησιμοποιώντας είτε δομικό είτε σημασιολογικό έλεγχο για τερματισμό [5]:

#### 3.3.2.1 Δομικός έλεγχος τερματισμού

Ο δομικός έλεγχος τερματισμού (Structural termination**)** είναι πλήρως αυτόματος και ανιχνεύει τα κοινά μοτίβα αναδρομής όπου το όρισμα της αναδρομικής κλήσης είναι ένα (άμεσο ή έμμεσο) υπο-όρος του αρχικού ορίσματος [5].

#### 3.3.2.2 Σημασιολογικός έλεγχος τερματισμού

Όταν ο δομικός έλεγχος τερματισμού αποτύχει, η Liquid Haskell προσπαθεί να αποδείξει τον τερματισμό χρησιμοποιώντας ένα σημασιολογικό επιχείρημα, πράγμα που απαιτεί ένα ρητό όρισμα τερματισμού: μια έκφραση που υπολογίζει έναν φυσικό αριθμό από το όρισμα της συνάρτησης και που μειώνεται σε κάθε αναδρομική κλήση [5].

Μπορούμε να χρησιμοποιήσουμε αυτόν τον έλεγχο τερματισμού για την απόδειξη του θεωρήματος appendNilId, χρησιμοποιώντας τη σύνταξη / [length xs]:

```
{-@ appendNilId:: xs:[a] → { xs ++ [] == xs } / [length xs] @-}
```

Στη γενική μορφή ένα όρισμα που χρησιμοποιείται για τον έλεγχο τερματισμού είναι της μορφής: / $[e_1, e_2, \ldots, e_n]$ , όπου οι εκφράσεις $e_i$ εξαρτώνται από τα ορίσματα της συνάρτησης και παράγουν φυσικούς αριθμούς. Θα πρέπει να μειώνονται λεξικογραφικά σε κάθε αναδρομική κλήση. Αυτές οι προϋποθέσεις ελέγχονται από τον SMT solver κατά τον έλεγχο του refinement type της συνάρτησης.

Εάν ο χρήστης δεν καθορίσει μια έκφραση τερματισμού και ο έλεγχος δομικής τερματισμού αποτυγχάνει, η Liquid Haskell προσπαθεί να μαντέψει μια έκφραση τερματισμού όπου μειώνεται το πρώτο όρισμα που δεν είναι συνάρτηση.

Ο σημασιολογικός τερματισμός (Semantic termination) έχει δύο βασικά πλεονεκτήματα έναντι του δομικού τερματισμού:

- Πρώτον, δεν είναι κάθε συνάρτηση δομικά αναδρομική (και καθιστώντας την δομικά αναδρομική με αναδιάρθρωση της δομής της με την προσθήκη πρόσθετων παραμέτρων μπορεί να έχει ως αποτέλεσμα να είναι δυσνόητη).

- Και δεύτερον, από τη στιγμή που ο τερματισμός ελέγχεται από τον SMT solver, μπορεί να χρησιμοποιήσει τις ιδιότητες των εισόδων της συνάρτησης που δηλώνονται στο refinement type.

Ωστόσο, ο σημασιολογικός τερματισμός έχει επίσης δύο κύρια μειονεκτήματα:

- Πρώτον, όταν η έκφραση τερματισμού είναι τετριμμένη, τότε οι κλήσεις προς τον SMT solver επιβαρύνουν το χρόνο που χρειάζεται ο solver για κάνει τον έλεγχο των συναρτήσεων.

- Και δεύτερον, ο έλεγχος τερματισμού συχνά απαιτεί την ρητή παροχή της έκφρασης τερματισμού, όπως το μήκος μιας λίστας εισόδου.

# 4 Επαλήθευση Ιδιοτήτων Αλγορίθμων και Δομών Δεδομένων

Το κεφάλαιο αυτό αποτελεί το βασικό κεφάλαιο αυτής της διπλωματικής εργασίας, όπου παρουσιάζουμε ιδιότητες και αποδείξεις ιδιοτήτων αλγορίθμων και δομών δεδομένων προκειμένου να εξασφαλίσουμε την ορθότητα τους. Όπως έχουμε ήδη αναφέρει οι διατυπώσεις των ιδιοτήτων και θεωρημάτων υπάρχουν στο 3ο τόμο του Software Foundations και στην εργασία αυτή επιτυγχάνεται η μετάφραση των θεωρημάτων αυτών σε γλώσσα Haskell και η συγγραφή αποδείξεων για τα θεωρήματα αυτά σε liquid Haskell, προκειμένου να δείξουμε πως μπορούμε να επαληθεύσουμε ιδιότητες προγραμμάτων στη Liquid Haskell.

## 4.1 Λίστες

Στο συγκεκριμένο κεφάλαιο θα μας απασχολήσουν θεωρήματα πάνω σε ιδιότητες που έχουν οι Λίστες. Το συγκεκριμένο κεφάλαιο είναι το μόνο κεφάλαιο που περιέχεται στον 1ο τόμο του βιβλίου Software Foundations [6] και όχι στον 3ο τόμο στον οποίο περιέχονται τα υπόλοιπα υποκεφάλαια. Ο βασικός λόγος που έγινε μετάφραση των αποδείξεων από το Coq σε Liquid Haskell είναι ότι οι λίστες αποτελούν ένα ιδανικό κεφάλαιο για εισαγωγή και εισαγωγή και κατανόηση αποδείξεων σε Liquid Haskell καθώς επίσης πολλά πράγματα που θα αναφέρουμε θα φανούν χρήσιμα σε επόμενα κεφάλαια.

Στο κεφάλαιο αυτό σχεδόν όλες οι αποδείξεις είναι αρκετά απλές καθώς μπορούν να αποδειχθούν χρησιμοποιώντας απλή επαγωγή. Έτσι στις περισσότερες αποδείξεις αυτού του κεφαλαίου χρησιμοποιούμε PLE για περισσότερη αυτοματοποίηση χωρίς οι αποδείξεις να γίνονται δυσνόητες αφού γράφοντας τις αντίστοιχες equational αποδείξεις είναι αρκετά απλή διαδικασία.

Αρχικά θέτουμε τα κατάλληλα flag της Liquid Haskell (flag για ενεργοποίηση του ple και reflection) στην αρχή του αρχείου και εισάγουμε τα κατάλληλα modules:

```
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}

module Lists where
import Language.Haskell.Liquid.ProofCombinators
import Prelude hiding (length, (++), reverse, pred, (^))

{-@ infix   : @-}
```

Τώρα μπορούμε να προχωρήσουμε στην απόδειξη ορισμένων θεωρημάτων:

```
 --έχουμε ξανασυναντήσει την απόδειξη αυτή όπου την είχαμε ονομάσει
 --appendNilId
{-@ listNuetralRight :: xs:[a] -> { xs ++ [] == xs } @-}
listNuetralRight :: [a] -> Proof
listNuetralRight []     = trivial
listNuetralRight (_:xs) = listNuetralRight xs

--ορίζουμε τη συνάρτηση συνένωσης δύο λιστών
```

```
{-@ infix   ++ @-}
{-@ reflect ++ @-}
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

--αποδεικνύουμε ότι συνάρτηση ++ είναι προσεταιριστική
{-@ appendAssoc :: xs:_ -> ys:_ -> zs:_
                -> { xs ++ (ys ++ zs) = (xs ++ ys) ++ zs }
@-}
appendAssoc :: [a] -> [a] -> [a] -> Proof
appendAssoc [] _ _       = trivial
appendAssoc (_:xs) ys zs = appendAssoc xs ys zs

--με αντίστοιχο τρόπο αποδεικνύουμε την προσεταιριστικότητα του ++
--για τέσσερις λίστες
{-@ app_assoc4 ::  xs:_ -> ys:_ -> zs:_ -> ws:_ ->
    { xs ++ (ys ++ (zs ++ ws)) = ((xs ++ ys) ++ zs) ++ ws }
@-}
app_assoc4 :: [a] -> [a] -> [a] -> [a] -> Proof
app_assoc4 [] ys zs ws     = appendAssoc ys zs ws
app_assoc4 (x:xs) ys zs ws = app_assoc4 xs ys zs ws

--συνάρτηση που επιστρέφει το μήκος μιας λίστας, η length πληροί τα
--κριτήρια για να δηλωθεί ως measure
{-@ measure length @-}
{-@ length :: [a] -> Nat @-}
length :: [a] -> Int
length []       = 0
length (x:xs) = 1 + length xs

--το μήκος της συνένωσης δύο λιστών ισούται με το άθροισμα του
--μήκους των δύο λιστών
{-@ app_length ::  xs:_ -> ys:_ ->
    { length (xs ++ ys) == length xs + length ys }
@-}
app_length :: [a] -> [a] -> Proof
app_length [] ys     = trivial
app_length (x:xs) ys = app_length xs ys

--ορίζουμε τη συνάρτηση αντιστροφής μιας λίστας
{-@ reflect reverse @-}
reverse :: [a] -> [a]
reverse []       = []
reverse (x:xs) = reverse xs ++ [x]

--Η αντίστροφη της συνένωσης δύο λιστών είναι το ίδιο με τη συνένωση
--της αντίστροφης της δευτερης λίστας με την αντίστροφη της πρώτης
--λίστας
{-@ rev_app_distr ::  xs:_ -> ys:_ ->
    {reverse (xs ++ ys) = reverse (ys) ++ reverse (xs)}
@-}
rev_app_distr :: [a] -> [a] -> Proof
rev_app_distr [] ys = listNuetralRight (reverse ys)
```

```
rev_app_distr (x:xs) ys =
  [ rev_app_distr xs ys,
    appendAssoc (reverse ys) (reverse xs) [x]
  ] *** QED

--η αντίστροφη της αντίστροφης μιας λίστας είναι η ίδια η λίστα
{-@ rev_involutive ::  xs:_ -> { reverse (reverse xs) =  xs } @-}
rev_involutive :: [a] ->  Proof
rev_involutive []     = trivial
rev_involutive (x:xs) =
  [rev_app_distr (reverse xs) ([x]), rev_involutive xs] *** QED

--το μήκος της αντίστροφης μιας λίστας είναι ίσο με το μήκος της
--λίστας.
{-@ rev_length :: xs:_ ->
     { length (reverse xs) == length xs }
@-}
rev_length :: [a] -> Proof
rev_length []    = trivial
rev_length (x:xs)=
    [ app_length (reverse xs) [x],
      rev_length xs,
      app_length [x] xs
    ] *** QED

--Ορίζουμε τη συνάρτηση nonzeros που δέχεται μια λίστα ακεραίων και
--επιστρέφει την αρχική λίστα έχοντας αφαιρέσει τα μηδενικά.
{-@ reflect nonzeros @-}
nonzeros ::  [Int] -> [Int]
nonzeros []     = []
nonzeros (x:xs) =
     if (x==0) then nonzeros xs else x:(nonzeros xs)

--Τα μη μηδενικά στοιχεία της συνένωσης δύο λιστών ισούται με τη
--συνένωση των μη μηδενικών στοιχείων της πρώτης λίστας με τα μη
--μηδενικά στοιχεία της δεύτερης λίστας
{-@  nonzeros_app ::  xs:_ -> ys:_ -> { nonzeros (xs ++ ys) =
(nonzeros xs) ++ (nonzeros ys) } @-}
nonzeros_app :: [Int] -> [Int] -> Proof
nonzeros_app [] ys = trivial
nonzeros_app (0:xs) ys = nonzeros_app xs ys
nonzeros_app (x:xs) ys = nonzeros_app xs ys

--ορίζουμε τη συνάρτηση που δέχεται ένα ακέραιο επιστρέφει το
--προηγούμενο του.
{-@ reflect pred @-}
pred :: Int -> Int
pred n
  | n == 0 = 0
  | otherwise = n-1

--συνάρτηση που επιστρέφει το πρώτο στοιχείο μιας λίστας
{-@ lHd :: (Ord a) => {v: [a] | len v > 0} -> a @-}
lHd :: (Ord a) => [a] -> a
```

```
lHd (x:_) = x

--συνάρτηση που επιστρέφει την ουρά μιας λίστας
{-@ reflect lTl @-}
lTl :: (Ord a) =>  [a] -> [a]
lTl [] = []
lTl (_:xs) = xs

--το μήκος της ουράς μιας λίστας ισούται με το μήκος της λίστας
--μειωμένο κατά ένα
{-@ tl_length_pred ::  xs:_ ->
   { pred (length xs) == length (lTl xs) }
@-}
tl_length_pred :: [a] -> Proof
tl_length_pred []     = trivial
tl_length_pred (x:xs) = tl_length_pred xs

--συνάρτηση που ελέγχει αν δύο λίστες έχουν τα ίδια στοιχεία
{-@ reflect beq_list @-}
beq_list :: [Int] -> [Int] -> Bool
beq_list [] []         = True
beq_list [] _          = False
beq_list _ []          = False
beq_list (x:xs) (y:ys) = if (x==y) then (beq_list xs ys) else False

--αποδεικνύουμε ότι μια λίστα έχει τα ίδια στοιχεία με τον εαυτό
--της
{-@ beq_natlist_refl ::  xs: [Nat] -> { beq_list xs xs } @-}
beq_natlist_refl :: [Int]  -> Proof
beq_natlist_refl []     = trivial
beq_natlist_refl (x:xs) = beq_natlist_refl xs
```

Με τις παραπάνω αποδείξεις ολοκληρώνουμε το κεφάλαιο αυτό.

## 4.2 Permutation

Το συγκεκριμένο κεφάλαιο αναφέρεται στο κεφάλαιο permutations [7] του 3ου τόμου του βιβλίου Software Foundations. Στο συγκεκριμένο κεφάλαιο θα αποδείξουμε διάφορες χρήσιμες ιδιότητες και θεωρήματα τα οποία θα φανούν πολύ χρήσιμα σε μετέπειτα κεφάλαια.

Αρχικά ορίζουμε ότι δύο λίστες είναι permutation η μία της άλλης αν και μόνο αν περιέχουν τα ίδια στοιχεία με ίδια πληθικότητα (πιθανώς σε διαφορετική σειρά). Για να ορίσουμε το permutation χρησιμοποιούμε multiset που είναι ήδη υλοποιημένα στη Liquid Haskell:

```
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}

module Permutations where

import Language.Haskell.Liquid.Bag
import Language.Haskell.Liquid.ProofCombinators
import Lists
import Prelude hiding ((++))
```

```
import qualified Data.Set as S

{-@ reflect permutation @-}
permutation :: Ord a => [a] -> [a] -> Bool
permutation xs ys = fromList xs == fromList ys
```

Στη συνέχεια παραθέτουμε ορισμένα παραδείγματα με permutations καθώς επίσης και ορισμένα απλά θεωρήματα:

```
{-@ exampleBut :: { permutation (1:(2:(3:[]))) (3:(2:(1:[]))) } @-}
exampleBut :: Proof
exampleBut = trivial

{-@ exNotPermutation :: { not (permutation (1:(1:[])) (1:(2:[]))) }
@-}
exNotPermutation = trivial

{-@ exampleButterfly :: { permutation
    ("b":("u":("t":("t":("e":("r":("f":("l":("y":[])))))))))
    ("f":("l":("u":("t":("t":("e":("r":("b":("y":[])))))))))  } @-}

exampleButterfly :: Proof
exampleButterfly = trivial

{-@ thmPermProp2 :: x:_ -> y:_ -> xs:_ ->
       {permutation (x:(y:xs)) (y:(x:xs))}
@-}
thmPermProp2:: (Ord a) => a -> a -> [a] -> Proof

thmPermProp2 x y []     = trivial
thmPermProp2 x y (_:xs) = thmPermProp2 x y xs

{-@ thmPermProp1 :: xs:_ -> {permutation xs xs} @-}
thmPermProp1:: (Ord a) => [a] -> Proof

thmPermProp1 []     = trivial
thmPermProp1 (_:xs) = thmPermProp1 xs


{-@ thmPermProp3 :: x:a -> ys:[a] -> xs:{[a] | permutation xs ys} ->
      {permutation (x:xs) (x:ys)}
@-}
thmPermProp3:: (Ord a) => a -> [a] -> [a] -> Proof
thmPermProp3 x xs ys = trivial


{-@ permut_example :: a:_ -> b:_
         -> {permutation (5:(6:(a ++ b))) ((5:b) ++ ((6:a) ++ [])) }
@-}

permut_example :: [Int] -> [Int] -> Proof
```

```
permut_example as bs
  = [ appendBag as bs
    , listNuetralRight (6:as)
    , appendBag (5:bs) ((6:as) ++ [])]
    *** QED
```

Ορίζουμε τη συνάρτηση maybe_swap που δέχεται ως είσοδο μια λίστα και αν η λίστα είχε τουλάχιστον δύο στοιχεία εκ των οποίων το $1^{ο}$ είναι μικρότερο του $2^{ου}$ επιστρέφει ως έξοδο τη λίστα με αντεστραμμένα τα δύο πρώτα στοιχεία ενώ σε κάθε άλλη περίπτωση επιστρέφει την ίδια τη λίστα. Παρακάτω αποδεικνύουμε ορισμένα θεωρήματα σχετικά με permutations και τη συνάρτηση maybe_swap:

```
--Ορισμός maybe_swap
{-@ reflect maybeSwap @-}
maybeSwap :: Ord a => [a] -> [a]
maybeSwap (a:(b:ar)) = if a > b then b:(a:ar) else (a:(b:ar))
maybeSwap ar         = ar


{-@ reflect fstLeSnd @-}
fstLeSnd :: (Ord a) => [a] -> Bool
fstLeSnd (a1:(a2:as)) = a1 <= a2
fstLeSnd as = True

--Απλά παραδείγματα
{-@ testMaybeSwap123 :: { maybeSwap (1:(2:(3:[]))) = (1:(2:(3:[]))) }
@-}
testMaybeSwap123 :: Proof
testMaybeSwap123 = trivial


{-@ testMaybeSwap321 :: { maybeSwap (3:(2:(1:[]))) = (2:(3:(1:[]))) }
@-}
testMaybeSwap321 :: Proof
testMaybeSwap321 = trivial

-- Εφαρμόζοντας δύο φορές την maybe swap επιστρέφει την αρχική λίστα
{-@ maybe_swap_idempotent :: xs: [a] -> { maybeSwap (maybeSwap xs) =
    maybeSwap xs } @-}
maybe_swap_idempotent :: (Ord a) => [a] -> Proof
maybe_swap_idempotent []       = trivial
maybe_swap_idempotent [x]      = trivial
maybe_swap_idempotent (x:y:xs) = trivial


{-@ thmMaybeSwap12 :: xs:_ -> { fstLeSnd (maybeSwap xs) } @-}
thmMaybeSwap12 :: (Ord a) => [a] -> Proof
thmMaybeSwap12 []           = trivial
thmMaybeSwap12 [a1]         = trivial
thmMaybeSwap12 (a1:(a2:as))
  | a1 < a2                 = trivial
  | otherwise               = trivial
```

```
--Η maybe_swap επιστρέφει πάντα permutation της αρχικής λίστας
{-@ maybe_swap_perm :: xs:_ -> {permutation xs (maybeSwap xs)} @-}
maybe_swap_perm:: (Ord a) => [a]-> Proof
maybe_swap_perm []      = trivial
maybe_swap_perm [_]     = trivial
maybe_swap_perm (_:_:_) = trivial
```

Τέλος αποδεικνύουμε παρακάτω το θεώρημα thm_Forall_perm, δηλαδή ότι αν μια λίστα bl είναι permutation μιας λίστας al και για όλα τα στοιχεία της al ισχύει μια ιδιότητα f τότε και για όλα τα στοιχεία της bl ισχύει η f.

```
--αρχικά ορίζουμε τη συνάρτηση forall2 που δέχεται μία συνάρτηση
--f: a -> Bool και αν η f αληθεύει για όλα τα στοιχεία τη λίστας
--επιστρέφει True
{-@ reflect forall2 @-}
forall2 :: (a->Bool) -> [a] -> Bool
forall2 f []     = True
forall2 f (x:xs) = (f x) && (forall2 f xs)

--Βοηθητικό λήμμα προσθέτοντας ένα στοιχείο σε ένα bag δεν μπορεί να
--επιστρέφει το κενό
{-@ thm_emp_bag :: Ord k => x:k -> xs:Bag k
   ->  { empty /= put x xs }
@-}
thm_emp_bag :: Ord k => k -> Bag k -> ()
thm_emp_bag x xs  = get x xs  *** QED


--Βοηθητικό λήμμα όπου αποδεικνύουμε ότι αν δεν ισχύει μια ιδιότητα
--f για όλα τα στοιχεία μιας λίστας τότε θα υπάρχει ένα x που ανήκει
--στη λίστα για το οποίο f x = False
{-@ elemForAllExists :: f:(a -> Bool)
        -> xs:{[a] | not (forall2 f xs)}
        -> (x::a, {v:() | S.member x (listElts xs) && (not (f x))})
@-}
elemForAllExists :: (a -> Bool) -> [a] -> (a,())
elemForAllExists f (x:xs)
    | not (forall2 f (x:xs))
      = if f x then elemForAllExists f xs else (x,())


--Αν ένα στοιχείο ανήκει σε μία λίστα τότε το στοιχείο αυτό θα
--υπάρχει στο Bag με τα στοιχεία της λίστας
{-@ elemToBagMember :: x:a -> xs:{[a] | S.member x (listElts xs)}
        -> {1 <= get x (fromList xs) }
 @-}
elemToBagMember :: Ord a => a -> [a] -> ()
elemToBagMember x []     = ()
elemToBagMember x (y:ys)
           | x == y
                  =   get x (fromList (y:ys))
```

```
                ==. get x (put y (fromList ys))
                ==. 1 + get x (fromList ys)
                *** QED
        | x /= y
                =   get x (fromList (y:ys))
                ==. get x (put y (fromList ys))
                ==. get x (fromList ys)
                        ? elemToBagMember x ys
                *** QED

--Το παραπάνω λήμμα με ισοδυναμία
{-@ bagMember :: x:a -> xs:[a]
     -> {S.member x (listElts xs) <=> (1 <= get x (fromList xs))}
@-}
bagMember :: Ord a => a -> [a] -> ()

bagMember x []     = ()
bagMember x (y:ys)
        |  not (S.member x (S.fromList (y:ys)))
              = bagMember x ys
        |  not (1 <= get x (fromList (y:ys)))
              = elemToBagMember x (y:ys) &&& bagMember x ys
        | otherwise
              = ()

--αν δύο λίστες xs,ys είναι permutation και ένα στοιχείο x ανήκει
στη --xs θα ανήκει και στη ys
{-@ permutationsElem :: x:a -> ys:[a]
       -> xs:{[a] | permutation xs ys }
       -> {  S.member x (listElts xs) <=> S.member x (listElts ys) }
@-}
permutationsElem :: Ord a => a -> [a] -> [a] -> ()
permutationsElem x ys xs
         =   assert (permutation xs ys)
             &&& (bagMember x xs)
             &&& (bagMember x ys)


{-@ assert :: b:{Bool | b } -> { b } @-}
assert :: Bool -> ()
assert _ = ()


--Αν για ένα στοιχείο x μιας λίστας δεν ισχύει η ιδιότητα f τότε δεν
--θα ισχύει η f για όλα τα στοιχεία της λίστας
{-@ elemForAll :: f:(a -> Bool) -> x:{a | not (f x) }
           -> xs:{[a] | S.member x (listElts xs) }
           -> { not (forall2 f xs) }
@-}
elemForAll :: Ord a => (a -> Bool) -> a -> [a] -> ()
elemForAll f x []
              = S.member x (S.fromList []) *** QED
elemForAll f y (x:xs)
           | y == x
```

```
            =   forall2 f (x:xs)
            ==. (f x && forall2 f xs)
            ==. False
            *** QED
        | otherwise
            =   forall2 f (x:xs)
            ==. (f x && forall2 f xs)
                    ? elemForAll f y xs
            ==. False
            *** QED
```

Με τα παραπάνω βοηθητικά λήμματα είμαστε σε θέση να αποδείξουμε το θεώρημα thm_Forall_perm:

```
{-@ thm_Forall_perm :: f: (a -> Bool) -> al: [a]
    -> bl: {[a] | permutation al bl}
    -> {  forall2 f al = forall2 f bl  }
@-}

thm_Forall_perm :: Ord a => (a -> Bool) -> [a] -> [a] -> Proof

thm_Forall_perm f xs ys
     | forall2 f xs && not (forall2 f ys)
        = case elemForAllExists f ys of
          (y, pf) -> permutationsElem y xs ys
                     &&& elemForAll f y xs
     | not (forall2 f xs) && forall2 f ys
        = case elemForAllExists f xs of
          (x, pf) -> permutationsElem x xs ys
                     &&& elemForAll f x ys
     | otherwise = ()
```

Με το παραπάνω θεώρημα κλείνουμε το κεφάλαιο Permutations.

## 4.3 Insertion Sort

Στο συγκεκριμένο κεφάλαιο διατυπώνουμε και αποδεικνύουμε τις ιδιότητες που εξασφαλίζουν την ορθότητα του αλγορίθμου όπως παρουσιάζονται στο κεφάλαιο [8]. Ο αλγόριθμος insertion sort επιλέγει με τη σειρά ένα κάθε φορά από τα στοιχεία της λίστας εισόδου και το εισάγει στη σωστή θέση στη λίστα εξόδου επιστρέφοντας έτσι μια ταξινομημένη λίστα.

Αρχικά εισάγουμε τα κατάλληλα αρχεία και ορίζουμε τη συνάρτηση εισαγωγής στοιχείων σε λίστα:

```
{-@ LIQUID "--reflection"                              @-}
{-@ LIQUID "--ple"                                     @-}

module ISort where
import Language.Haskell.Liquid.Bag
import Language.Haskell.Liquid.ProofCombinators
```

```
import Permutations
import Prelude hiding ((++))

{-@ reflect insert2 @-}
insert2 :: (Ord a) => a -> [a] -> [a]
insert2 i []    =  [i]
insert2 i (x:xs)
       | i <= x     = (i:(x:xs))
       | otherwise  = (x:(insert2 i xs))
```

Η παραπάνω συνάρτηση δέχεται ως είσοδο ένα στοιχείο και μια λίστα και εισάγει το στοιχείο στη λίστα στη θέση όπου όλα τα στοιχεία πριν από αυτό να είναι μικρότερα του.

Στη συνέχεια ορίζουμε τη συνάρτηση ταξινόμησης:

```
{-@ reflect sort@-}
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = insert2 x (sort xs)
```

Για να αποδείξουμε την ορθότητα ενός αλγορίθμου ταξινόμησης ορίζουμε μια συνάρτηση που δέχεται σαν όρισμα μια λίστα και επιστρέφει True αν και μόνο αν η λίστα είναι ταξινομημένη

```
{-@ reflect sorted @-}
sorted :: (Ord a) => [a] -> Bool
sorted []        = True
sorted (h:t) = sorted1 h t

{-@ reflect sorted1 @-}
sorted1 :: (Ord a) => a -> [a] -> Bool
sorted1 x []     = True
sorted1 x (y:ys) = if x <= y then sorted1 y ys else False
```

Ένας αλγόριθμος ταξινόμησης για να είναι ορθός θα πρέπει να επιστρέφει ταξινομημένη λίστα αλλά αυτό δεν είναι αρκετό, θα πρέπει επίσης η λίστα εξόδου να περιέχει όλα τα στοιχεία της λίστας εισόδου, να είναι δηλαδή permutation της λίστας εισόδου.

Για να αποδείξουμε την ορθότητα του αλγορίθμου Insertion Sort ξεκινάμε πρώτα από την συνάρτηση insert για την οποία θέλουμε να δείξουμε ότι η έξοδος που παράγει είναι μια λίστα που είναι permutation της αρχικής λίστας αν προσθέσουμε στην αρχή το στοιχείο που εισάγουμε:

```
{-@ thmInsertPerm :: (Ord a) => x:a -> xs:[a] ->
      { permutation (x:xs) (insert2 x xs) }
@-}

thmInsertPerm :: (Ord a) => a -> [a] -> Proof
thmInsertPerm x [] = trivial
thmInsertPerm x (h:t)
  | x <= h    = trivial
  | otherwise = thmInsertPerm x t
```

Στη συνέχεια χρησιμοποιούμε το παραπάνω θεώρημα και επαγωγή ώστε να αποδείξουμε ότι η συνάρτηση Sort επιστρέφει permutation της λίστας εισόδου:

```
{-@ thmSortPerm :: (Ord a) => xs: [a] ->
        { permutation xs (sort xs) }
@-}
thmSortPerm :: (Ord a) => [a] -> Proof
thmSortPerm []     = trivial
thmSortPerm (x:xs) = [ thmSortPerm xs, thmInsertPerm x (sort xs) ]
                     *** QED
```

Τέλος αποδεικνύουμε ότι η εισάγοντας ένα στοιχείο σε ταξινομημένη λίστα επιστρέφει πάντα ταξινομημένη λίστα και χρησιμοποιούμε αυτό το λήμμα για να αποδείξουμε το βασικό θεώρημα που εξασφαλίζει την ορθότητα του αλγορίθμου ταξινόμησης ότι δηλαδή επιστρέφει ταξινομημένη λίστα.

```
{-@ thmInsertSorted :: x:a -> ys:{[a] | sorted ys} ->
     { sorted (insert2 x ys) }
 @-}
thmInsertSorted :: (Ord a) => a -> [a] -> Proof
thmInsertSorted x []        =   trivial
thmInsertSorted x [h]       =   trivial
thmInsertSorted x (h:(y:t))
     | x <= h               =   trivial
     | x > h && x <= y      =   trivial
     | x > h && x > y       =   sorted (insert2 x (h:(y:t)))
                            ==. sorted (h:(insert2 x (y:t)) )
                            ==. sorted (h:(y:(insert2 x t)) )
                            ==. True ? thmInsertSorted x (y:t)
                            *** QED

{-@ thmSortSorted :: xs: [a] -> { sorted (sort xs) } @-}
thmSortSorted :: (Ord a) => [a] -> Proof
thmSortSorted []         = trivial
thmSortSorted (x:xs)     = (thmSortSorted xs
                           , thmInsertSorted x (sort xs) ) *** QED
```

Το τελευταίο θεώρημα (thmSortSorted) μαζί με το θεώρημα που έχουμε αποδείξει παραπάνω, ότι ο αλγόριθμος Insertion Sort επιστρέφει permutation της αρχικής λίστας (thmSortPerm) μας εξασφαλίζουν την ορθότητα του αλγορίθμου.

## 4.4 Selection Sort

Στο κεφάλαιο αυτό θα αναλύσουμε και θα επαληθεύσουμε ιδιότητες σχετικά με τον αλγόριθμο ταξινόμησης Selection Sort. Οι ιδιότητες που θα αποδείξουμε βασίζονται στο στο κεφάλαιο Selection Sort του Software Foundations [9].

Θα αρχίσουμε το κεφάλαιο ορίζοντας τους απαραίτητος τύπου δεδομένων, συμπεριλαμβάνοντας τα κατάλληλα modules (πχ permutations), βιβλιοθήκες και θα ορίσουμε τη συνάρτηση select που βρίσκει και διαγράφει το ελάχιστο στοιχείο από μία λίστα:

```
{-@ LIQUID "--reflection"                              @-}
{-@ LIQUID "--ple"                                     @-}

module SSort where
import Language.Haskell.Liquid.Bag
import Language.Haskell.Liquid.ProofCombinators
import Permutations

{-@ infix : @-}

--Τύπος δεδομένων που συμβολίζει ζεύγος μεταβλητών
data Pair a b = Pair a b

--Συνάρτηση που δέχεται ένα Pair και επιστρέφει το δεύτερο στοιχείο
{-@ measure snd1 @-}
snd1 :: (Pair a b) -> b
snd1 (Pair _ x) = x

--Συνάρτηση που δέχεται ένα Pair και επιστρέφει το πρώτο στοιχείο
{-@ measure fst1 @-}
fst1 ::  (Pair a b) -> a
fst1 (Pair x _) = x

--Συνάρτηση που υποδηλώνει τη σχέση <=
{-@ reflect greater_eq @-}
greater_eq :: (Ord a) => a -> a -> Bool
greater_eq x y = x <= y


{-@ reflect select @-}
{-@ select :: (Ord a) => x:a -> l1: [a]
           -> p: {(Pair  a [a]) |  (len l1 == len (snd1 p))  }@-}

select :: (Ord a) => a -> [a] -> (Pair a ([a]))
select i [] = (Pair i  [])
select i (x:xs)
  | i <= x    =(Pair (fst1 (select i xs)) (x: (snd1 (select i xs))))
  | otherwise =(Pair (fst1 (select x xs)) (i:(snd1 (select x xs))))
```

Η συνάρτηση select δέχεται ως πρώτο όρισμα ένα αριθμό ως αρχικό ελάχιστο, και μία λίστα και επιστρέφει ένα ζεύγος που έχει σας πρώτο στοιχείο το ελάχιστο στοιχείο και σαν δεύτερο στοιχείο την υπόλοιπη λίστα. Στη συνάρτηση select έχουμε προσθέσει refinement type που

δηλώνει ότι η λίστα που επιστρέφει έχει ίδιο μήκος με τη λίστα εισόδου, αυτή η συνθήκη είναι απαραίτητη για τη συνάρτησης selsort που θα ορίσουμε παρακάτω προκειμένου η Liquid Haskell να μπορεί να εγγυηθεί το τερματισμό της selsort. Αν δεν παρείχαμε αυτή τη πληροφορία η Liquid Haskell δεν θα μπορούσε να εγγυηθεί το τερματισμό της selsort αφού δεν θα μπορούσε να εξάγει αυτή τη συνθήκη αυτόματα από τη select αφού η select δεν είναι δομικά αναδρομική συνάρτηση.

Ο αλγόριθμος ταξινόμησης selection sort βρίσκει το ελάχιστο στοιχείο της λίστας εισόδου, το τοποθετεί στην αρχή και συνεχίζει αναδρομικά την ίδια διαδικασία για την υπολίστα που μένει μέχρι να φτάσει στη κενή λίστα ως είσοδο. Ο συνάρτηση ταξινόμησης με επιλογή δίνεται παρακάτω:

```
{-@ reflect selSort @-}
{-@ selSort :: (Ord a) => xs:[a]  -> ys:[a]  @-}

selSort :: (Ord a) => [a]  -> [a]
selSort []     = []
selSort (x:xs) = ((fst1 (select x xs)):selSort (snd1 (select x xs)))
```

Ομοίως με την ταξινόμηση με εισαγωγή που αναλύσαμε στο προηγούμενο κεφάλαιο για να επαληθεύσουμε την ορθότητα θα δείξουμε ότι η συνάρτηση selsort επιστρέφει permutation της αρχικής λίστας και ότι η λίστα εξόδου είναι ταξινομημένη, πράγμα που όπως και στο προηγούμενο κεφάλαιο εξετάζουμε με τη συνάρτηση sorted:

```
{-@ reflect sorted @-}
sorted :: (Ord a) => [a] -> Bool
sorted []        = True
sorted (h:t) = sorted1 h t

{-@ reflect sorted1 @-}
sorted1 :: (Ord a) => a -> [a] -> Bool
sorted1 x []     = True
sorted1 x (y:ys) = if x <= y then sorted1 y ys else False
```

Εδώ θα ήταν χρήσιμο να σημειώσουμε ότι η συνάρτηση ταξινόμησης με επιλογή, λόγο ότι χρησιμοποιεί τη συνάρτηση select που δεν είναι δομικά αναδρομική είναι αρκετά πιο δύσκολο να επαληθευτούν διάφορες ιδιότητες του αλγορίθμου, πράγμα που θα γίνει εμφανές και στη συνέχεια. Αρχικά για να αποδείξουμε ότι η συνάρτηση selsort επιστρέφει permutation της λίστας εισόδου που είναι και το ένα ζητούμενο, θα πρέπει να το δείξουμε πρώτα για τη συνάρτηση selection με το παρακάτω θεώρημα:

```
 --Βοηθητικό λήμμα που θα φανεί χρήσιμο στο θεώρημα παρακάτω
{-@ thm_perm :: x: a -> y:a -> xs: [a]
      -> ys: {[a]| permutation (x:xs) (y:ys) } -> i:a
      -> { permutation (x:(i:(xs)))  (y:(i:(ys))) }
@-}
thm_perm:: a -> a -> [a] -> [a] -> a -> Proof
thm_perm x xs y ys i = trivial


{-@ thm_select_perm:: (Ord a) => x: a -> l: [a] ->
```

```
     {permutation (x:l) ((fst1 (select x l)) : (snd1 (select x l)))}
   / [len l]
@-}

thm_select_perm:: (Ord a) => a -> [a]  -> Proof
thm_select_perm x []                  =   trivial

thm_select_perm x (l:ls)  | x <= l
          = permutation (x:(l:ls)) ((fst1 (select x (l:ls))) : (snd1
            (select x (l:ls))))

          ==. permutation (x:(l:ls)) ((fst1 (select x ls)) :
             (l:(snd1 (select x ls))) )

          ==. True ? thm_select_perm x ls  &&& thm_perm x (fst1
                    (select x ls)) ls (snd1 (select x ls)) l
          *** QED

thm_select_perm x (l:ls)  | x > l
           =   permutation (x:(l:ls)) ((fst1 (select x (l:ls))) :
               (snd1 (select x (l:ls))))

           ==. permutation (x:(l:ls)) ((fst1 (select l ls)) :
               (x:(snd1 (select l ls))) )
            ==. True ? thm_select_perm l ls  &&& thm_perm l (fst1
                          (select l ls)) ls (snd1 (select l ls)) x
            *** QED
```

Χρησιμοποιώντας τα δύο παραπάνω λήμματα μπορούμε τώρα να αποδείξουμε ότι η συνάρτηση selsort επιστρέφει permutation της λίστας εισόδου:

```
 --Ένα ακόμα βοηθητικό λήμμα για το παρακάτω θεώρημα
{-@ thm_selsort_perm_aux :: xs:[a] -> ys:{[a] | permutation xs ys}
     -> zs:[a] -> { permutation xs zs <=> permutation ys zs}
@-}
thm_selsort_perm_aux :: [a] -> [a] -> [a] -> Proof
thm_selsort_perm_aux xs ys zs = trivial


{-@ thm_selsort_perm:: (Ord a) => l: [a] ->
      { permutation l (selSort l) }
@-}
thm_selsort_perm:: (Ord a) => [a]  -> Proof

thm_selsort_perm []     = trivial
thm_selsort_perm (x:xs)
    =   permutation   (x:xs) (selSort (x:xs))
    ==. permutation (x:xs) ((fst1 (select x xs)):selSort (snd1
          (select x xs)))
    ==. permutation ((fst1 (select x xs)) : (snd1 (select x xs)))
          ((fst1 (select x xs)):selSort (snd1 (select x xs)))

           ? thm_select_perm x xs &&&
```

```
            thm_selsort_perm_aux (x:xs) ((fst1 (select x xs)) :
                  (snd1 (select x xs))) ((fst1 (select x xs)):
                    selSort (snd1 (select x xs)))

      ==. True ? thm_selsort_perm  (snd1 (select x xs))
      *** QED
```

Στη συνέχεια αποδεικνύουμε σαν τελευταίο βήμα ότι η συνάρτηση selsort επιστρέφει σαν έξοδο ταξινομημένη λίστα. Για να το αποδείξουμε διατυπώνουμε και αποδεικνύουμε αρκετά ενδιάμεσα λήμματα που παρουσιάζονται στη συνέχεια:

```
-- Βοηθητικό λήμμα, αν ένα στοιχείο είναι μικρότερο από όλα τα
-- στοιχεία μιας λίστας l, τότε είναι μικρότερο και από όλα τα
-- στοιχεία της selsort l

{-@ thm_greater_eq:: (Ord a) => x:a
      -> l:{[a] | forall2 (greater_eq x) l }
      -> { forall2 (greater_eq x) (selSort l) }
@-}
thm_greater_eq :: (Ord a) => a -> [a] -> Proof
thm_greater_eq x l = [thm_selsort_perm l, thm_Forall_perm
(greater_eq x) l (selSort l)] *** QED



--Αν προσθέσουμε ένα στοιχείο στη αρχή μιας ταξινομημένης λίστας και
--το στοιχείο αυτό είναι μικρότερο ίσο από όλα τα στοιχεία της
--λίστας τότε προκύπτει ταξινομημένη λίστα.

{-@ selection_sort_sorted_aux :: (Ord a) =>
       bl :{ [a] | sorted (selSort bl) }
       -> y: {a | forall2 (greater_eq y) bl }
       -> { sorted (y: selSort bl) }
@-}
selection_sort_sorted_aux :: (Ord a) => [a] -> a -> Proof
selection_sort_sorted_aux [] y = trivial
selection_sort_sorted_aux bl y =
  (sorted (selSort bl), forall2 (greater_eq y) bl,
    thm_greater_eq y bl)
  *** QED



--Το πρώτο στοιχείο μιας λίστας είναι μικρότερο ίσο από όλα τα στοιχεία
--της λίστας συμπεριλαμβανομένου και του ίδιου αν και μόνο αν είναι
--μικρότερο ίσο από τα υπόλοιπα στοιχεία της λίστας

{-@ thm_smallest_aux :: xs:[a] -> x:a -> { forall2 (greater_eq x)
(x:xs) <=> forall2 (greater_eq x) xs } @-}
thm_smallest_aux :: [a] -> a -> Proof
thm_smallest_aux xs x = trivial
```

```
-- Αν ένα στοιχείο y είναι μικρότερο ίσο από όλα τα στοιχεία μιας
-- λίστας τότε είναι και μικρότερο από το πρώτο στοιχείο

{-@ thm_smallest_aux2 :: (Ord a) =>  x:a -> y:a
     ->  al: { [a] | forall2 (greater_eq y) (x:al) } -> {y<= x}
@-}
thm_smallest_aux2 :: (Ord a) => a -> a -> [a] -> Proof
thm_smallest_aux2 x y al = trivial


-- Το ελάχιστο στοιχείο που επιστρέφει η select είναι μικρότερο ίσο
-- από το αρχικό στοιχείο που δίνουμε ως πρώτο όρισμα

{-@ select_smallest_aux :: (Ord a) =>  x:a ->
       al:{[a] |forall2 (greater_eq (fst1 (select x al)))
       (snd1 (select x al)) } ->  { (fst1 (select x al)) <= x}
@-}
select_smallest_aux :: (Ord a) => a -> [a] -> Proof
select_smallest_aux x al =
   ( thm_select_perm x al ,
     thm_smallest_aux  ((fst1 (select x al)): (snd1 (select x al)))
      (fst1 (select x al)) ,
     thm_Forall_perm ( greater_eq (fst1 (select x al)) ) (x:al)
      ((fst1 (select x al)) :(snd1 (select x al)) ),
     forall2 (greater_eq (fst1 (select x al)))  (x:al)
    ,thm_smallest_aux2  x (fst1 (select x al)) al
  )*** QED


--Για κάθε στοιχείο x και λίστα xs το πρώτο στοιχείο που επιστρέφει
--η select x xs είναι μικρότερο ίσο από όλα τα στοιχεία της λίστας
--που επιστρέφει σαν δεύτερο στοιχείο η select

{-@ thm_select_smallest :: (Ord a) => x:a -> xs:[a] ->
      { forall2 (greater_eq (fst1 (select x xs)))
       (snd1 (select x xs)) }   / [len xs]
@-}
thm_select_smallest  :: (Ord a) => a -> [a] -> Proof
thm_select_smallest k [] = trivial
thm_select_smallest k (x:xs) | k <= x
 =   forall2 (greater_eq (fst1 (select x xs))) (snd1 (select x xs))
 ==.forall2 (greater_eq (fst1 (select k xs))) (x:snd1 (select k xs))
 ==. True ?  thm_select_smallest k xs &&& select_smallest_aux k xs
 *** QED

thm_select_smallest k (x:xs) | k > x
 = forall2 (greater_eq (fst1 (select x xs))) (snd1 (select x xs))
 ==.forall2 (greater_eq (fst1 (select x xs))) (k:snd1 (select x xs))
 ==. True ?  thm_select_smallest x xs &&& select_smallest_aux x xs
 *** QED
```

Χρησιμοποιώντας τα παραπάνω βοηθητικά λήμματα μπορούμε πλέον να αποδείξουμε το ζητούμενο:

```
{-@ selection_sort_sorted :: (Ord a) => l: [a]
      -> { sorted (selSort l) }
@-}
selection_sort_sorted :: (Ord a) => [a]  -> Proof
selection_sort_sorted []  = trivial
selection_sort_sorted (h:t)
    =   sorted (selSort (h:t))
    ==. sorted (((fst1 (select h t) ) :selSort (snd1 (select h t))))
    ==. True  ? selection_sort_sorted (snd1 (select h t)) &&&
                (thm_select_smallest h t) &&&
                selection_sort_sorted_aux (snd1 (select h t))
                      (fst1 (select h t))
    *** QED
```

Με την τελευταία απόδειξη ολοκληρώνουμε το κεφάλαιο του selection sort έχοντας αποδείξει ότι ο αλγόριθμος selection sort επιστρέφει ταξινομημένη λίστα και permutation της αρχικής λίστας οπότε είναι ένας ορθός αλγόριθμος ταξινόμησης.

## 4.5 Binary Search Trees

Στο παρόν κεφάλαιο θα αναφερθούμε στα δυαδικά δέντρα και στα δυαδικά δέντρα αναζήτησης. Θα βασιστούμε στο κεφάλαιο Binary Search Trees [10] του Software Foundations αλλά δεν θα το ακολουθήσουμε με ακρίβεια όπως κάναμε στα προηγούμενα κεφάλαια καθώς διάφορα πράγματα δεν είναι τόσο εφικτό να μεταφραστούν και να αντιστοιχιστούν από τη γλώσσα Coq στη γλώσσα Haskell.

Αρχικά εισάγουμε τις κατάλληλες βιβλιοθήκες και modules και ορίζουμε τους απαραίτητους δεδομένων. Ορίζουμε τον τύπο δεδομένων Maybe a = Just a | Nothing που θα χρειαστούμε αργότερα για αναπαραστήσουμε ότι μια συνάρτηση δεν επιστρέφει κάτι που το αναπαριστούμε με το Nothing αλλιώς επιστρέφει κάτι που είναι τύπου a δηλαδή Just a. Επιπλέον ορίζουμε το τύπο δεδομένων για τα δυαδικά δέντρα:

```
{-@ LIQUID "--reflection"                              @-}
{-@ LIQUID "--ple"                                     @-}

module BST where
import qualified Data.Set as S
import Language.Haskell.Liquid.ProofCombinators
import Prelude hiding (lookup, Maybe(..))


data Maybe a = Just a | Nothing


{-@ measure isJust @-}
isJust :: Maybe a -> Bool
isJust (Just _) = True
isJust Nothing  = False

{-@ measure fromJust @-}
{-@ fromJust :: {v:Maybe a | isJust v } -> a @-}
```

```
fromJust :: Maybe a -> a
fromJust (Just x) = x


data BST k v = Empty | Bind k v (BST k v) (BST k v)


{-@
data BST [blen] k v = Empty
  | Bind { bKey   :: k
         , bValue :: v
         , bLeft  :: BST k v
         , bRight :: BST k v }
@-}


{-@ measure blen @-}
blen :: (BST k v) -> Int
{-@ blen :: (BST k v) -> Nat @-}
blen(Empty)        = 0
blen(Bind k v l r) = 1 + (blen l) + (blen r)
```

Στη συνέχεια θα ορίσουμε διάφορες συναρτήσεις για τα δυαδικά δέντρα για τις οποίες θα επαληθεύσουμε διάφορες χρήσιμες ιδιότητες στη συνέχεια του κεφαλαίου. Αυτές οι συναρτήσεις είναι οι:

- insert: δέχεται σαν ορίσματα ένα key, ένα value και ένα binary search tree και εισάγει τα k, v στο δέντρο ως εξής:
  - αν το δέντρο είναι κενό δημιουργεί ένα δέντρο που περιέχει μόνο το key k και το value v
  - αν υπάρχει το k στη ρίζα του δέντρου τότε δεν αλλάζει το δέντρο,
  - αν το k είναι μικρότερο από το κλειδί που βρίσκεται στη ρίζα εκτελεί τη διαδικασία εισαγωγής των k,v στο αριστερό υποδέντρο αναδρομικά και τέλος
  - αν το k είναι μεγαλύτερο από το κλειδί που βρίσκεται στη ρίζα εκτελεί τη διαδικασία εισαγωγής των k,v στο δεξί υποδέντρο αναδρομικά.

- lookup: δέχεται ένα κλειδί k και ένα binary search tree και εκτελεί αναζήτηση προκειμένου να βρει το κλειδί στο δέντρο ως εξής:
  - Αν το δέντρο είναι κενό τότε επιστρέφει τη τιμή Nothing δηλώνοντας κατά αυτό το τρόπο ότι το συγκεκριμένο κλειδί δεν βρέθηκε.
  - Αν το κλειδί ισούται με το κλειδί που βρίσκεται στη ρίζα τότε επιστρέφει το value της ρίζας του δέντρου
  - Αν το κλειδί είναι μικρότερο από το κλειδί της ρίζας εκτελεί αναζήτηση στο αριστερό υποδέντρο αλλιώς
  - Αν είναι μεγαλύτερο εκτελεί αναζήτηση στο δεξί υποδέντρο.

Η υλοποίηση των συναρτήσεων insert, lookup είναι η εξής:

```
{-@ reflect insert @-}
insert :: (Eq k, Ord k) => k -> v -> BST k v -> BST k v
insert k v Empty  = Bind k v Empty Empty
insert k v (Bind k' v' l r)
  | k == k'       = Bind k v l r
```

```
  | k < k'        = Bind k' v' (insert k v l) r
  | otherwise     = Bind k' v' l (insert k v r)


{-@ reflect lookup @-}
lookup :: (Eq k, Ord k) => k -> BST k v -> Maybe v
lookup k Empty  = Nothing
lookup k (Bind k' v l r)
  | k == k'       = Just v
  | k < k'        = lookup k l
  | otherwise     = lookup k r
```

Μέχρι έχουμε αναφερθεί σε δυαδικά δέντρα αλλά δεν έχουμε εκφράσει κάποια ιδιότητα των δυαδικών δέντρων αναζήτησης. Έτσι λοιπόν ορίζουμε την ιδιότητα-συνάρτηση searchTree που δέχεται ως παράμετρο ένα δυαδικό δέντρο και επιστρέφει True αν και μόνο αν αυτό είναι διάδικό δέντρο αναζήτησης:

```
{-@ reflect searchTree @-}
{-@ searchTree :: Nat ->  t:BST Nat v -> Nat -> Bool / [blen t]@-}
searchTree :: Int ->  BST Int v -> Int -> Bool
searchTree lo (Empty)        hi = (lo<=hi)
searchTree lo (Bind k v l r) hi =
      (searchTree lo l k) && (searchTree (k+1) r hi)
```

Τώρα που έχουμε αυτή την ιδιότητα αποδεικνύουμε ορισμένα σχετικά θεωρήματα:

```
-- Αν το ισχύει searchTree lo t hi τότε για τα όρια lo,hi έχουμε lo<=hi
{-@ lemma_SearchTree_le :: lo: Nat -> hi: Nat
       -> t:{ BST Nat v | searchTree lo t hi}
       -> {lo <= hi} / [blen t]
@-}
lemma_SearchTree_le :: Int -> Int -> BST Int v -> Proof
lemma_SearchTree_le lo hi Empty        = trivial
lemma_SearchTree_le lo hi (Bind k v l r)
   = [lemma_SearchTree_le lo k l,
      lemma_SearchTree_le (k+1) hi r] *** QED

--Για κάθε lo,hi με lo<= hi ισχύει searchTree lo t hi
{-@ empty_tree_SearchTree :: lo: Nat -> hi: {Nat | lo <= hi } ->
{searchTree lo Empty hi} @-}
empty_tree_SearchTree :: Int -> Int -> Proof
empty_tree_SearchTree lo hi = trivial


--Αν εισάγουμε ένα νέο κλειδί k (και αντίστοιχο value v) για το οποίο
--ισχύει lo<=k και k<hi σε ένα δέντρο t που ισχύει η ιδιότητα
--searchTree τότε συνεχίζει να ισχύει η ιδιότητα και μετά την εισαγωγή.

{-@ theorem_insert_SearchTree :: k:Nat -> vk: v -> lo:{Int | lo<=k }
      -> hi:{Int | k<hi } -> t:{BST Nat v | searchTree lo t hi }
      -> {searchTree lo (insert k vk t) hi}   / [blen t]
@-}
```

```
theorem_insert_SearchTree :: Int -> v -> Int -> Int -> BST Int v ->
Proof
theorem_insert_SearchTree k v lo hi Empty = trivial
theorem_insert_SearchTree k v lo hi (Bind k1 v1 l r)
          | k==k1 = trivial
          | k<k1 = theorem_insert_SearchTree k v lo k1 l
          | k>k1 = theorem_insert_SearchTree k v (k1+1) hi r
```

Στο τελευταίο κομμάτι αυτής της ενότητας θα δείξουμε ότι θα binary search trees ικανοποιούν τις ιδιότητες των Total Maps. Το κεφάλαιο που περιγράφει τα Total Maps [11] ανήκει στον 1ο τόμο του Software Foundations, με τον οποίο δεν ασχολούμαστε σε αυτή τη διπλωματική, ωστόσο δεν χρειάζεται κάποια γνώση από αυτό το κεφάλαιο αφού θα περιγράψουμε τις ιδιότητες των Total Maps που θέλουμε να επαληθεύσουμε για τα binary trees.

Οι ιδιότητες των Maps που θέλουμε επαληθεύσουμε ότι ισχύουν στα Binary search trees είναι:

```
--Αν αναζητήσουμε ένα στοιχείο x σε ένα δέντρο που περιέχει μόνο το
-- x, η αναζήτηση μας επιστρέφει το στοιχείο x.

{-@ t_apply_empty :: (Eq k, Ord k, Eq v) => x :k -> v1 :v ->
       {lookup x (Bind x v1 Empty Empty) = (Just v1) }
@-}
t_apply_empty :: (Eq k, Ord k, Eq v) => k -> v -> Proof
t_apply_empty x v1 = (lookup x (Bind x v1 Empty Empty) ) *** QED


--Αν εισάγουμε το κλειδί x και το value c σε ένα bst και
--αναζητήσουμε το κλειδί x η lookup θα μας επιστρέψει το value v

{-@ t_update_eq :: (Eq k, Ord k, Eq v) => t: (BST k v) -> x:k ->
      v1 :v -> { lookup x (insert x v1 t) = (Just v1) }
@-}
t_update_eq :: (Eq k, Ord k, Eq v) => (BST k v) -> k -> v -> Proof
t_update_eq Empty x v1  =   lookup x (insert x v1 Empty)
                        ==. lookup x (Bind x v1 Empty Empty)
                        ==. (Just v1)
                        *** QED

t_update_eq (Bind k v l r) x v1
      | k == x =   lookup x (insert x v1 (Bind k v l r))
               ==. lookup x (Bind x v1 l r)
               ==. Just v1
               *** QED

      | k <  x =   lookup x (insert x v1 (Bind k v l r))
               ==. lookup x ( Bind k v l (insert x v1 r))
               ==. lookup x (insert x v1 r)
               ==. (Just v1) ? t_update_eq r x v1
               *** QED

      | k > x =   lookup x (insert x v1 (Bind k v l r))
              ==. lookup x ( Bind k v (insert x v1 l) r)
```

```
              ==. lookup x (insert x v1 l)
              ==. (Just v1) ? t_update_eq l x v1
              *** QED


--Αν εισάγουμε ένα κλειδί k2 και αναζητήσουμε μετά ένα κλειδί k1 με
--k1 /= k2, η αναζήτηση επιστρέφει το ίδιο αποτέλεσμα με το να
--αναζητήσουμε το k1 στο αρχικό δέντρο

{-@ t_update_neq :: (Eq k, Ord k, Eq v) => k1:k -> k2:{k | k1 /= k2}
        -> v2:v -> t: (BST k v)
        -> { lookup k1 (insert k2 v2 t) == lookup k1 t }
@-}
t_update_neq :: (Eq k, Ord k, Eq v)
      => k -> k -> v -> (BST k v) -> Proof
t_update_neq k1 k2 v2 Empty
  | k1 < k2             =   lookup k1 (insert k2 v2 Empty)
                        ==. lookup k1 (Bind k2 v2 Empty Empty)
                        ==. lookup k1 Empty
                        *** QED

  | otherwise           =   lookup k1 (insert k2 v2 Empty)
                        ==. lookup k1 (Bind k2 v2 Empty Empty)
                        ==. lookup k1 Empty
                        *** QED

t_update_neq k1 k2 v2 (Bind k v l r)
  | k1 <  k, k <  k2    =   lookup k1 (insert k2 v2 (Bind k v l r))
                        ==. lookup k1 (Bind k v l (insert k2 v2 r))
                        ==. lookup k1 (Bind k v l r)
                        *** QED

  | k == k2             =   lookup k1 (insert k2 v2 (Bind k v l r))
                        ==. lookup k1 (Bind k v2 l r)
                        ==. lookup k1 (Bind k v l r)
                        *** QED

  | k1 == k, k <  k2    =   lookup k1 (insert k2 v2 (Bind k v l r))
                        ==. lookup k1 (Bind k v l (insert k2 v2 r))
                        ==. lookup k1 (Bind k v l r)
                        *** QED

  | k2 <  k, k == k1    =   lookup k1 (insert k2 v2 (Bind k v l r))
                        ==. lookup k1 (Bind k v (insert k2 v2 l) r)
                        ==. lookup k1 (Bind k v l r)
                        *** QED

  | k2 <  k, k <  k1    =   lookup k1 (insert k2 v2 (Bind k v l r))
                        ==. lookup k1 (Bind k v (insert k2 v2 l) r)
                        ==. lookup k1 r
                        ==. lookup k1 (Bind k v l r)
                        *** QED

  | k1 < k, k2 < k      =   lookup k1 (insert k2 v2 (Bind k v l r))
```

```
                        ==. lookup k1 (Bind k v (insert k2 v2 l) r)
                        ==. lookup k1 (insert k2 v2 l)   ?
t_update_neq k1 k2 v2 l
                        ==. lookup k1 l
                        ==. lookup k1 (Bind k v l r)
                        *** QED

  | k < k1, k < k2      =   lookup k1 (insert k2 v2 (Bind k v l r))
                        ==. lookup k1 (Bind k v l (insert k2 v2 r))
                        ==. lookup k1 (insert k2 v2 r)   ?
t_update_neq k1 k2 v2 r
                        ==. lookup k1 r
                        ==. lookup k1 (Bind k v l r)
                        *** QED


-- Αν εισάγουμε ένα κλειδί k1 και ένα value v1 και στη συνέχεια
--ξαναεισάγουμε το k1 με value v2 (οπότε αντικαθιστά το v1), είναι το
--ίδιο με το να εισάγουμε μόνο το v2 .

{-@ thm_same_tree ::  (Eq k, Ord k, Eq v) => t:(BST k v) -> v1:v
    -> v2:v -> x:k
    -> { (insert x v2 (insert x v1 t)) = (insert x v2 t) }
@-}
thm_same_tree ::  (Eq k, Ord k, Eq v)
       => (BST k v) -> v -> v -> k -> Proof
thm_same_tree Empty v1 v2 x
                =   (insert x v2 (insert x v1 Empty))
                ==. (insert x v2  (Bind x v1 Empty Empty))
                ==. (Bind x v2 Empty Empty)
                ==. (insert x v2 Empty)
                *** QED

thm_same_tree (Bind x1 v l r) v1 v2 x

      | x == x1 =  (insert x v2 (insert x v1 (Bind x v l r)))
                ==. (insert x v2  (Bind x v1 l r))
                ==. (Bind x v2 l r)
                ==. (insert x v2 (Bind x1 v l r))
                *** QED

      | x < x1  =   (insert x v2 (insert x v1 (Bind x1 v l r)))
                ==. (insert x v2  (Bind x1 v (insert x v1 l) r))
                ==. (Bind x1 v (insert x v2 (insert x v1 l)) r)
                ==. (Bind x1 v (insert x v2 l) r)
                         ? thm_same_tree l v1 v2 x
                ==. (insert x v2 (Bind x1 v l r))
                *** QED

      | x > x1  =   (insert x v2 (insert x v1 (Bind x1 v l r)))
                ==. (insert x v2  (Bind x1 v l (insert x v1 r)))
                ==. (Bind x1 v l (insert x v2 (insert x v1 r)))
                ==. (Bind x1 v l (insert x v2 r))
                          ? thm_same_tree r v1 v2 x
```

```
                ==. (insert x v2 (Bind x1 v l r))
                *** QED


-- Ορίζουμε το τύπο Eqtree που εκφράζει ότι δύο δέντρα είναι ισοδύναμα,
--δηλαδή η αναζήτηση του ίδιου κλειδιού και στα δύο δέντρα επιστρέφει
--την ίδια τιμή

{-@ type EqTree k v T1 T2 = x:k -> { lookup x T1 == lookup x T2} @-}


-- όπως είδαμε και πριν αν εισάγουμε ένα κλειδί k1 και ένα value v1
--στο δέντρο t και στη συνέχεια ξαναεισάγουμε το k1 με value v2 (οπότε
--αντικαθιστά το v1), είναι το ίδιο με το να εισάγουμε μόνο το v2, άρα
--τα δέντρα (insert x v2 (insert x v1 t)) και (insert x v2 t) είναι
--ισοδύναμα.

{-@ t_update_shadow :: (Eq k, Ord k, Eq v) => t:(BST k v) -> v1:v
      -> v2:v -> x:k ->
       EqTree k v (insert x v2 (insert x v1 t)) (insert x v2 t)
@-}

t_update_shadow :: (Eq k, Ord k, Eq v)
       => (BST k v) -> v -> v -> k -> k -> Proof
t_update_shadow t v1 v2 x y = (thm_same_tree t v1 v2 x) *** QED


--Αν ένα κλειδί k υπάρχει στο δέντρο t, τότε αν ξαναεισάγουμε το k
--με την ίδια τιμή παίρνουμε το ίδιο δέντρο. Για την απόδειξη
--χρησιμοποιούμε το λήμμα t_update_same_aux που βρίσκεται στη συνέχεια

{-@ t_update_same :: (Eq k, Ord k, Eq v) => t:(BST k v)
     -> x: {k | isJust (lookup x t) }
     -> EqTree k v (insert x (fromJust (lookup x t)) t)  t
@-}
t_update_same :: (Eq k, Ord k, Eq v)
     => (BST k v) ->  k -> k -> Proof
t_update_same t x y = (t_update_same_aux t x)*** QED


{-@ t_update_same_aux
  :: (Eq k, Ord k, Eq v)
  => t:BST k v
  -> x: {k | isJust (lookup x t) }
  ->  {(insert x (fromJust (lookup x t)) t) == t}
@-}
t_update_same_aux
  :: (Eq k, Ord k, Eq v) => BST k v -> k -> Proof

t_update_same_aux (Bind k v l r) x
  | x == k
  =   insert x (fromJust (lookup x (Bind x v l r))) (Bind x v l r)
  ==. insert x (fromJust (Just v)) (Bind x v l r)
```

```
  ==. insert x v (Bind x v l r)
  ==. Bind x v l r
  *** QED

  | x < k
  =   insert x (fromJust (lookup x (Bind k v l r))) (Bind k v l r)
  ==. insert x (fromJust (lookup x l)) (Bind k v l r)
  ==. Bind k v (insert x (fromJust (lookup x l)) l) r
       ? t_update_same_aux l x
  ==. Bind k v l r
  *** QED

  | x > k
  =   insert x (fromJust (lookup x (Bind k v l r))) (Bind k v l r)
  ==. insert x (fromJust (lookup x r)) (Bind k v l r)
  ==. Bind k v l (insert x (fromJust (lookup x r)) r)
       ? t_update_same_aux r x
  ==. Bind k v l r
  *** QED

t_update_same_aux Empty x
  =   isJust (lookup x Empty)
  ==. isJust Nothing
  *** QED


-- Τελευταία ιδιότητα για τα binary search trees, αν εισάγουμε δύο
-- διαφορετικά κλειδιά (με αντίστοιχα values) σε ένα δέντρο, με
-- αντίστροφη σειρά το δέντρο που θα πάρουμε και στις δύο περιπτώσεις
-- μετά τις εισαγωγές θα είναι το ίδιο.

{-@ t_update_permute :: (Eq k, Ord k, Eq v) => t: (BST k v) -> x1:k
       -> x2:{k | x1 /= x2} -> y:k -> v1:v -> v2:v
       -> { lookup y (insert x2 v2 (insert x1 v1 t)) ==
            lookup y (insert x1 v1 (insert x2 v2 t)) }
@-}
t_update_permute :: (Eq k, Ord k, Eq v)
      => (BST k v) -> k -> k -> k -> v -> v -> Proof
t_update_permute t x1 x2 y v1 v2  | x1 == x2 = trivial
t_update_permute t x1 x2 y v1 v2

        |  y == x2  = [ t_update_eq (insert x1 v1 t) x2 v2 ,
                        t_update_neq y x1 v1 (insert y v2 t ) ,
                        t_update_eq t y v2 ] *** QED

        |  y == x1  = [ t_update_eq (insert x2 v2 t) x1 v1 ,
                        t_update_neq y x2 v2 (insert y v1 t ) ,
                        t_update_eq t y v1 ] *** QED

        | otherwise = [ t_update_neq y x2 v2 (insert x1 v1 t) ,
                        t_update_neq y x1 v1 (insert x2 v2 t),
                        t_update_neq y x1 v1 t ,
                        t_update_neq y x2 v2 t ] *** QED
```

Δείχνοντας ότι τα binary search trees ικανοποιούν τις παραπάνω ιδιότητες που προέρχονται από τα Maps, κλείνουμε αυτό το κεφάλαιο.

## 4.6 Abstract Data Types

Το κεφάλαιο αυτό αναφέρεται στο κεφάλαιο abstract data types [12] του Software Foundations. Το συγκεκριμένο κεφάλαιο αναφέρεται σε αφηρημένους τύπους δεδομένων, για παράδειγμα έστω ότι θέλουμε να ορίσουμε lookup tables και να υλοποιήσουμε διάφορες λειτουργίες σε αυτά. Θα μπορούσαμε να το ορίσουμε ως ένα interface που να κρύβει την πληροφορία για την υλοποίηση των διαφόρων συναρτήσεων-λειτουργιών. Έτσι στη συνέχεια θα μπορούσαμε να επιλέξουμε να υλοποιήσουμε τα lookup tables με χρήση binary search trees ή Total Maps. Αυτή είναι ή ιδέα των αφηρημένων τύπων δεδομένων, ορίζουμε κάποιο τύπο δεδομένων και διάφορες λειτουργίες σε αυτόν αλλά χωρίς να τις υλοποιούμε και έτσι μας παρέχεται η ευχέρεια να τα υλοποιήσουμε στη συνέχεια με διαφορετικούς ισοδύναμους τρόπους, όπως στο παραπάνω παράδειγμα θα μπορούσαμε να υλοποιήσουμε τα lookup tables με χρήση Maps ή Binary Trees.

Στη Haskell παρέχεται η δυνατότητα να ορίζουμε Interface με ορίζοντας ένα Typeclass. Ωστόσο χρησιμοποιώντας τη Liquid Haskell δεν παρέχεται η δυνατότητα να ορίζουμε Interface καθώς δεν μπορούμε να κάνουμε reflect συναρτήσεις που είναι instances κάποιου type class, πράγμα που είναι αναγκαίο καθώς για να χρησιμοποιήσουμε μια συνάρτηση σε refinement logic πρέπει να την κάνουμε reflect (ή measure) όπως έχει αναφερθεί σε προηγούμενα κεφάλαια. Συνεπώς δεν μπορούμε να ακολουθήσουμε πλήρως το συγκεκριμένο κεφάλαιο από το Software Foundations καθώς δεν θα μπορούμε να ορίσουμε abstract data types όπου αυτά ορίζονται στο κεφάλαιο [12]. Παρά το γεγονός αυτό θα μεταφράσουμε ορισμένες ιδιότητες που αναφέρονται στη συνάρτηση Fibonacci στο κεφάλαιο αυτό σε Liquid Haskell χωρίς να ορίσουμε αφηρημένους τύπου δεδομένων αλλά απλού τύπους δεδομένων.

Μέχρι στιγμής έχουμε ξανασυναντήσει τη συνάρτηση Fibonacci, ωστόσο σε αυτό το κεφάλαιο Θα ορίσουμε δύο νέες διαφορετικές εκδοχές της ίδιας συνάρτησης και θα αποδείξουμε ότι είναι ισοδύναμες. Αρχίζουμε με τον πρώτο ορισμό της Fibonacci:

```
{-@ LIQUID "--reflection"                              @-}
{-@ LIQUID "--ple"                                     @-}

{-@ infix : @-}

module ADT where
import Prelude hiding(abs,repeat)
import Language.Haskell.Liquid.ProofCombinators
import Language.Haskell.Liquid.Bag
import Permutations


{-@ reflect repeat @-}
{-@ repeat :: (a -> a) -> a -> Nat -> a @-}
repeat :: (a -> a) -> a -> Int -> a
repeat f x 0 = x
repeat f x n = f (repeat f x (n-1))
```

```
{-@reflect step@-}
{-@ step :: xs:[Nat] -> ys:[Nat]@-}
step :: [Int] -> [Int]
step []       = []
step [x]      = [x,x]
step (x:y:xs) = (x+y):(x:y:xs)


{-@reflect nth1@-}
{-@nth1 :: Nat  -> [a] -> a -> a@-}
nth1 :: Int -> [a] -> a -> a
nth1 0 []     d = d
nth1 0 (x:xs) d = x
nth1 n (x:xs) d = nth1 (n-1) xs d


{-@reflect firstn@-}
{-@firstn :: Nat  -> [a] -> [a] @-}
firstn :: Int -> [a] -> [a]
firstn 0 l      = []
firstn n []     = []
firstn n (x:xs) = x:firstn (n-1) xs


{-@ reflect lHd @-}
{-@ lHd :: (Ord a) => [a]  -> a @-}
lHd :: (Ord a) => [a] -> a
lHd (x:_) = x


{-@reflect fib@-}
{-@ fib :: Nat -> Nat @-}
fib :: Int -> Int
fib n = lHd ( repeat step (1:(0:(0:([])))) n)
```

Στο παραπάνω κώδικα ορίζουμε τις εξής συναρτήσεις:

- repeat: δέχεται ένα στοιχείο x, μία συνάρτηση f και ένα φυσικό αριθμό n και υπολογίζει το $f^{n}(x)$ .
- step: δέχεται μία λίστα και αν είναι κενή επιστρέφει τη κενή λίστα, αν η λίστα έχει ένα στοιχείο επιστρέφει τη λίστα που περιέχει δύο φορές αυτό το στοιχείο αλλιώς αθροίζει τα δύο πρώτα στοιχεία και το άθροισμα το εισάγει στην αρχή της λίστας.
- nth1: δέχεται ένα φυσικό αριθμό n και μία λίστα και επιστρέφει το n-οστό στοιχείο της λίστας
- firstn: δέχεται ένα φυσικό αριθμό n και μία λίστα και επιστρέφει τα n πρώτα στοιχεία της λίστας
- lHd: επιστρέφει το πρώτο στοιχείο της λίστας
- fib : Η συνάρτηση που υπολογίζει το n-οστό αριθμό της ακολουθίας Fibonacci

Στη συνέχεια ορίζουμε ένα νέο τύπο λίστας που κρατάει μόνο τα τρία τελευταία στοιχεία της λίστας:

```
data Listish = L Int Int Int

{-@reflect create@-}
create :: Int -> Int -> Int -> Listish
create x y z = L x y z

{-@reflect cons@-}
cons :: Int ->  Listish -> Listish
cons i (L x y z) = (L i x y)

{-@ reflect lHd2 @-}
{-@ lHd2 :: Listish  -> Int @-}
lHd2 :: Listish -> Int
lHd2 (L x y z) = x

{-@ reflect stepish@-}
stepish :: Listish -> Listish
stepish (L x y z) = cons (x+y) (L x y z)

{-@ reflect fibish @-}
{-@ fibish :: Nat -> Int @-}
fibish :: Int -> Int
fibish n = lHd2 ( repeat stepish (create 1 0 0) n)
```

- Στο Software Foundation η Listish ορίζεται ως abstract data type, αλλά όπως εξηγήσαμε προηγουμένως θα χρησιμοποιήσουμε απλό data type για την αναπαράσταση του τύπου δεδομένων. Ομοίως με τη συνάρτηση step ορίζουμε τη συνάρτηση stepish για το τύπο Listish και τη συνάρτηση fibish υπολογισμού του το n-οστού όρου της ακολουθίας Fibonacci χρησιμοποιώντας λίστες τύπου Listish.

Αρχικά αποδεικνύουμε το εξής λήμμα:

```
-- Το να επιλέξουμε το i-οστό στοιχείο μιας λίστας είναι το ίδιο  με
-- το να επιλέξουμε τα j πρώτα στοιχεία της λίστας με j>i και μετά να
-- επιλέξουμε το i-οστό

{-@ lemma_nth_firstn :: al:[a] -> d:a -> i: Nat -> j: {Nat| i<j} ->
      {  nth1 i (firstn j al) d = nth1 i al d  }
@-}
lemma_nth_firstn :: [a] -> a -> Int -> Int -> Proof
lemma_nth_firstn [] d i j     = trivial
lemma_nth_firstn (x:xs) d i j
        | i == 0
            =   nth1 0 (  firstn j (x:xs)) d
            ==. nth1 0 (  firstn j (x:xs)) d
            ==. nth1 0 (x:firstn (j-1) xs) d
            ==. x
            ==. nth1 0 (x:xs) d
            *** QED

        | otherwise
            =   nth1 i (firstn j (x:xs) ) d
```

```
            ==. nth1 i (x:firstn (j-1) (xs) ) d
            ==. nth1 (i-1) (firstn (j-1) (xs) ) d
            ==. nth1 (i-1) xs d  ? lemma_nth_firstn xs d (i-1) (j-1)
            ==. nth1 i (x:xs) d
            *** QED
```

Στη συνέχεια ορίζουμε μια σχέση ισοδυναμίας μεταξύ μιας κανονικής – built-in λίστας και μιας λίστας τύπου Listish:

```
{-@ reflect l_Abs @-}
l_Abs :: [Int] -> Listish -> Bool
l_Abs [] (L x y z)            = False
l_Abs [x1] (L x y z)          = False
l_Abs [x1,x2] (L x y z)       = False
l_Abs (x1:x2:x3:xs) (L x y z) = if(x==x1 && y==x2 && z==x3)
                                then True
                                else False
```

Στη συνέχεια αποδεικνύουμε ότι για δύο ισοδύναμε λίστες εφαρμόζοντας τις συναρτήσεις step και stepish στην αντίστοιχη λίστα η σχέση ισοδυναμίας ισχύει και μετά την εφαρμογή αυτών των συναρτήσεων.

```
{-@lemma_step_relate:: al:[Int] -> bl: {Listish | l_Abs al bl} ->
       {l_Abs  (step al) (stepish bl)}
@-}
lemma_step_relate:: [Int] -> Listish -> Proof
lemma_step_relate al bl = trivial
```

Ομοίως το ίδιο αποδεικνύουμε και για τη συνάρτηση repeat:

```
{-@lemma_repeat_step_relate:: n:Nat -> al:[Nat]
     -> bl: {Listish | l_Abs al bl}
     -> {l_Abs (repeat step al n) (repeat stepish bl n)}
@-}
lemma_repeat_step_relate:: Int -> [Int] -> Listish -> Proof
lemma_repeat_step_relate 0 al bl = trivial
lemma_repeat_step_relate n al bl
   = l_Abs (repeat step al n) (repeat stepish bl n)

   ==. l_Abs (step (repeat step al (n-1)))
             (stepish (repeat stepish bl (n-1)))

   ==. True ?  lemma_repeat_step_relate (n-1) al bl &&&
               lemma_step_relate (repeat step al (n-1))
                      (repeat stepish bl (n-1))
   *** QED
```

Τέλος αποδεικνύουμε ότι οι δύο συναρτήσεις υπολογισμού των αριθμών Fibonacci (fib και fibish) είναι ισοδύναμες δηλαδή για κάθε είσοδο μας επιστρέφουν την ίδια έξοδο.

```
{-@ fibish_correct :: n:Nat -> {fibish n = fib n} @-}
```

```
fibish_correct :: Int -> Proof
fibish_correct n =
   (lemma_repeat_step_relate n (1:(0:(0:([])))) (create 1 0 0))
   *** QED
```

Με την παραπάνω απόδειξη ολοκληρώνουμε το κεφάλαιο abstract data types.

## 4.7 Priority Queues

Στο κεφάλαιο Priority Queues θα αναλύσουμε και θα επαληθευσουμε ιδιότητες σχετικά με μια απλή υλοποίηση των priority queues. Για την ανάλυση των priority queues ακολουθούμε το κεφάλαιο Priority Queues [13] του Software Foundations.

Αρχικά εισάγουμε τις κατάλληλες βιβλιοθήκες και modules και ορίζουμε τους απαραίτητους τύπους δεδομένων:

```
{-@ LIQUID "--reflection"                          @-}
{-@ LIQUID "--ple"                                 @-}
{-@ LIQUID "--exactdc"                             @-}
{-@ LIQUID "--exact-data-con"                      @-}
{-@ LIQUID "--higherorder"                         @-}

{-@ infix : @-}
{-@ infix ++ @-}

module PRIQUEUE where
import Prelude hiding(abs, (++), isJust, Maybe(..))
import Language.Haskell.Liquid.ProofCombinators
import Language.Haskell.Liquid.Bag
import Permutations
import Lists

data Maybe a = Just a | Nothing  deriving (Eq)

{-@ measure isJust' @-}
isJust' :: Maybe a -> Bool
isJust' (Just _) = True
isJust' Nothing  = False

{-@ measure fromJust @-}
{-@ fromJust :: {v:Maybe a | isJust' v } -> a @-}
fromJust :: Maybe a -> a
fromJust (Just x) = x

data Pair a b = Pair a b


{-@ measure snd1 @-}
snd1 :: (Pair a b) -> b
snd1 (Pair _ x) = x
```

```
{-@ measure fst1 @-}
fst1 ::  (Pair a b) -> a
fst1 (Pair x _) = x
```

Ορίζουμε ώς priority queue τύπο δεδομένων ουσιαστικά μία λίστα από ακεραίους:

```
data PRIQUEUE = P [Int] deriving (Ord, Eq)
```

Στη συνέχεια ορίζουμε την ιδιότητα priq που δηλώνει αν μια priority queue είναι μια έγκυρη priorty queue. Στην απλή περίπτωση όμως που εξετάζουμε όπου μια priority queue είναι μια λίστα, κάθε λίστα πληροί την ιδιότητα priq:

```
{-@reflect priq@-}
{-@ priq:: l: PRIQUEUE -> {v:Bool | True} @-}
priq:: PRIQUEUE -> Bool
priq (P l) = True

--κενή priority queue
{-@ reflect p_empty@-}
p_empty = P []

-- συνάρτηση μετατροπής priority queue σε λίστα
{-@reflect toList@-}
toList :: PRIQUEUE -> [Int]
toList (P l) = l

-- συνάρτηση συνένωσης δύο priority queue
{-@reflect merge@-}
{-@merge ::  PRIQUEUE -> PRIQUEUE -> PRIQUEUE @-}
merge :: PRIQUEUE -> PRIQUEUE -> PRIQUEUE
merge (P xs) (P ys) = P (xs ++ ys)

--συνάρτηση εισαγωγής στοιχείου σε priority queue – απλά το
--τοποθετούμε στην αρχή της λίστας
{-@reflect insert@-}
{-@insert ::  Nat -> PRIQUEUE -> PRIQUEUE @-}
insert :: Int -> PRIQUEUE -> PRIQUEUE
insert x (P xs) = P (x:xs)

-- συνάρτηση για να εξάγουμε το πρώτο στοιχείο από Maybe Pair
{-@reflect first@-}
{-@ first:: p: {Maybe (Pair Int PRIQUEUE) | p/= Nothing} -> Int @-}
first:: Maybe (Pair Int PRIQUEUE) -> Int
first (Just (Pair a b)) = a

-- συνάρτηση για να εξάγουμε το δεύτερο στοιχείο από Maybe Pair
{-@reflect second@-}
{-@ second:: p: {Maybe (Pair Int PRIQUEUE) | p/= Nothing} ->
PRIQUEUE @-}
second:: Maybe (Pair Int PRIQUEUE) -> PRIQUEUE
second (Just (Pair a b)) = b
```

```
--συνάρτηση που εκφράζει τη σχέση >=
{-@ reflect less_eq @-}
less_eq :: (Ord a) => a -> a -> Bool
less_eq x h = (x >= h)


{-@ reflect select @-}
{-@ select :: (Ord a) => x:a -> l1: [a]
         -> p: {(Pair  a [a]) | permutation (x:l1) (fst1 p : snd1 p)
         && (forall2 (less_eq (fst1 p)) (x:l1) ) }
@-}
select :: (Ord a) => a -> [a] -> (Pair a ([a]))
select i [] = (Pair i  [])
select i (x:xs)
  | i >= x    = let (Pair j2 l2) = (select i xs) in (Pair j2 (x:l2))
  | otherwise = let (Pair j1 l1) = (select x xs) in (Pair j1 (i:l1))
```

Παραπάνω ορίσαμε τη συνάρτηση select που είχαμε ξανασυναντήσει στο κεφάλαιο selection sort, ωστόσο με μια μικρή τροποποίηση ώστε να επιστρέφει το μέγιστο και όχι το ελάχιστο στοιχείο. Επιπλέον εδώ θα ήταν πολύ χρήσιμο να τονίσουμε ότι η κλήση select x xs επιστρέφει Pair a b και στο κεφάλαιο selection sort είχαμε αποδείξει ότι ισχύει permutation (x:xs) (a:b). Εδώ παρουσιάζουμε μία άλλη εναλλακτική, αντί να αποδείξουμε αυτή την ιδιότητα την έχουμε εκφράσει στο refinement type της select. Όπως είχαμε τονίσει ξανά τα refinement types είναι ισοδύναμα με αποδείξεις καθώς εκφράζουμε ιδιότητες οι οποίες ελέγχονται αυτόματα από τον SMT solver. Αν ο SMT solver μπορεί να επαληθεύσει αυτές τις ιδιότητες τότε θεωρούνται έγκυρες όπως στην περίπτωση μας η ιδιότητα permutation της selection η οποία επιτυχώς επαληθεύεται από το solver.

Στη συνέχεια χρησιμοποιώντας τη συνάρτηση select για να βρούμε το μέγιστο στοιχείο ορίζουμε τη συνάρτηση delete_max που διαγράφει το μέγιστο στοιχείο:

```
{-@reflect delete_max@-}
{-@ delete_max :: p:PRIQUEUE -> p2 :{Maybe (Pair Int PRIQUEUE) |
     p2/= Nothing ==>
    (permutation (toList p) ((first p2):(toList (second p2)))
    && (forall2 (less_eq (first p2)) (toList p) ) )}
@-}
delete_max :: PRIQUEUE -> Maybe (Pair Int PRIQUEUE)
delete_max (P []) = Nothing
delete_max (P (x:xs) ) = Just (Pair (fst1 y)  (P (snd1 y)))
                                 where y = select x xs
```

Ορίζουμε τη συνάρτηση abs που δηλώνει πότε μια priority queue είναι ισοδύναμη με μια λίστα, πράγμα που ισχύει αν και μόνο αν περιέχουν τα ίδια στοιχεία:

```
{-@reflect abs@-}
{-@ abs:: l1: PRIQUEUE -> l2:[Nat] -> Bool @-}
abs:: PRIQUEUE -> [Int] -> Bool
abs (P l1) l2 = permutation l1 l2
```

```
-- λήμμα που εκφράζει την ιδιότητα ότι αν μια λίστα είναι ισοδύναμη
--με μια priority queue , και όλα τα στοιχεία της priority queue είναι
--μεγαλύτερα ίσα ενός στοιχείου k, τότε και τα στοιχεία της λίστας θα
--είναι μεγαλύτερα ίσα του k.
{-@ abs_prop :: k:Nat ->
  q:{PRIQUEUE | forall2 (less_eq k) (toList q) } ->
  ql : {[Nat] | abs q ql}  ->
  { forall2 (less_eq k) ql}
@-}
abs_prop :: Int -> PRIQUEUE -> [Int] -> Proof
abs_prop k q ql = thm_Forall_perm (less_eq k) (toList q) ql
```

Παρακάτω αποδεικνύουμε ορισμένα αρκετά απλά λήμματα σχετικά με τα priority queues:

```
{-@ lemma_abs_perm :: p:PRIQUEUE -> al:{[Nat] | abs p al} ->
    bl:{[Nat] | abs p bl} -> { (permutation al bl)}
@-}
lemma_abs_perm :: PRIQUEUE -> [Int] -> [Int] -> Proof
lemma_abs_perm p al bl = trivial

--για κάθε priority queue υπάρχει, λίστα ισοδύναμη με τη priority
--queue, η ίδια η λίστα που περιέχει η priority queue
{-@ lemma_can_relate :: p:{PRIQUEUE | priq p}
   -> { abs p (toList p)}
@-}
lemma_can_relate :: PRIQUEUE ->  Proof
lemma_can_relate p = trivial

--Η κενή priority queue είναι έγκυρη
{-@ lemma_empty_priq :: {priq p_empty}@-}
lemma_empty_priq:: Proof
lemma_empty_priq = trivial

--υπάρχει λίστα ισοδύναμη με τη p_empty, η κενή λίστα
{-@lemma_empty_relate :: {abs p_empty []}@-}
lemma_empty_relate:: Proof
lemma_empty_relate = trivial

--η συνάρτηση insert επιστρέφει έγκυρη priority queue
{-@ lemma_insert_priq:: p: {PRIQUEUE | priq p} -> k: Nat
     -> { priq (insert k p)}
@-}
lemma_insert_priq:: PRIQUEUE -> Int -> Proof
lemma_insert_priq p k = trivial

--αν μία priority queue είναι ισοδύναμη με μία λίστα, τότε προσθέτοντας
-- το ίδιο στοιχείο στη λίστα και τη priority queue διατηρείται η
-- ιδιότητα της ισοδυναμίας
{-@ lemma_insert_relate :: p: {PRIQUEUE | priq p} ->
    al: {[Nat]| abs  p al} -> k: Nat ->
    {abs (insert k p) (k:al)}
@-}
lemma_insert_relate :: PRIQUEUE -> [Int] -> Int -> Proof
```

```
lemma_insert_relate p al k = trivial

-- η συνένωση δύο priority queue επιστρεφει έγκυρη priority queue
{-@ lemma_merge_priq :: p: {PRIQUEUE | priq p} ->
    q: {PRIQUEUE | priq q} -> { priq (merge p q)}
@-}
lemma_merge_priq :: PRIQUEUE -> PRIQUEUE -> Proof
lemma_merge_priq p q = trivial

--κατά τη διαγραφεί στοιχείου η priority queue που απομένει είναι
--έγκυρη
{-@ lemma_delete_max_Some_priq :: p: {PRIQUEUE | priq p}
    -> k: Int -> q: {PRIQUEUE| (delete_max p = Just (Pair k q)) }
    -> {priq q}
@-}
lemma_delete_max_Some_priq :: PRIQUEUE -> Int -> PRIQUEUE -> Proof
lemma_delete_max_Some_priq p k q = trivial
```

Στη συνέχεια αποδεικνύουμε τρία ακόμη βασικά θεωρήματα για τα priority queue που είναι πιο απαιτητικά:

```
--Βοηθητικό λήμμα
{-@ thmElemsApp :: xs: [a] -> ys: [a] ->
      { fromList (xs ++ ys) = union (fromList xs) (fromList ys) }
@-}
thmElemsApp :: (Ord a) =>  [a] -> [a] -> Proof
thmElemsApp [] ys     = trivial
thmElemsApp (x:xs) ys = thmElemsApp xs ys


--1ο Βασικό θεώρημα που δηλώνει ότι αν η συνένωση δύο priority queue
--είναι ισοδύναμη με μία λίστα al, τότε η al πρέπει να είναι
--permutation της (pl++ql), όπου pl,ql ισοδύναμες λίστες με τις δύο
--priority queues που συνενώσαμε

{-@ lemma_merge_relate :: p: {PRIQUEUE | priq p} ->
      q: {PRIQUEUE | priq q} ->
      pl: {[Nat] | abs p pl} ->
      ql: {[Nat] | abs q ql} ->
      al: {[Nat] | abs (merge p q) al} ->
      {permutation al (pl++ql)}
@-}
lemma_merge_relate ::
    PRIQUEUE -> PRIQUEUE -> [Int] -> [Int] -> [Int] -> Proof
lemma_merge_relate p q pl ql al = [thmElemsApp pl ql, thmElemsApp
(toList p) (toList q)] *** QED


--2ο θεώρημα: μια priority queue είναι ισοδύναμη με τη κενή λίστα αν
--και μόνο αν η delete_max επιστρέψει Nothing δηλαδή η priority
--queue είναι η κενή

{-@ lemma_delete_max_None_relate :: p: {PRIQUEUE | priq p}  ->
```

```
    { abs p [] <=> delete_max p == Nothing }
@-}
lemma_delete_max_None_relate :: PRIQUEUE -> Proof
lemma_delete_max_None_relate (P [])    = trivial
lemma_delete_max_None_relate (P l)     = [thm_perm1 l] *** QED


-- Τα thm_perm2, thm_perm1 είναι βοηθιτικά λήμματα
{-@ thm_perm2 :: (Ord a, Eq a) => xs: [a] -> ys: {[a] |len xs /= len
ys}  -> {permutation xs ys == False} @-}
thm_perm2 :: (Ord a, Eq a) => [a] -> [a] -> Proof
thm_perm2 xs ys
  =   permutation xs ys
  ==. fromList xs == fromList ys
  ==. bagSize (fromList xs) == bagSize (fromList ys)
       ? thm_size xs &&& thm_size ys
  ==. length xs == length ys
  ==. False
  *** QED

{-@ thm_perm1 :: xs: {[a] | len xs >0} ->
      { permutation xs [] == False }
  @-}
thm_perm1 :: (Ord a) =>  [a]  -> Proof
thm_perm1 [x] = trivial
thm_perm1 (x:xs) = [thm_perm2 (x:xs) []]***QED


--3ο Βασικό θεώρημα το οποίο δηλώνει ουσιαστικά ότι η delete_max
--επιστρέφει τον μέγιστο στοιχείο και αν το ενώσουμε με την priority
--queue που απομένει είναι permutation της αρχικής priority queue

{-@ lemma_delete_max_Some_relate :: q:PRIQUEUE -> k:Nat ->
     p: {PRIQUEUE | priq p && delete_max p = Just (Pair k q) }
     -> pl: {[Nat] | abs p pl} -> ql: {[Nat] | abs q ql}  ->
      {permutation pl (k:ql) && (forall2 (less_eq k) pl) }
@-}
lemma_delete_max_Some_relate ::
     PRIQUEUE -> Int -> PRIQUEUE -> [Int] -> [Int] -> Proof
lemma_delete_max_Some_relate q k p pl ql =
      (delete_max p  ,permutation (toList p) (k:toList q),
        abs_prop k p pl )*** QED
```

Με τα παραπάνω θεωρήματα ολοκληρώνουμε το κεφάλαιο των priority queues.

## 4.8 Red Black Trees

Το συγκεκριμένο Στο κεφάλαιο αυτό θα μεταφράσουμε και θα αποδείξουμε με χρήση της Liquid Haskell ιδιότητες που αφορούν τη δομή δεδομένων Red Black Trees από το κεφάλαιο Red Black Trees στο Software Foundations[14] . Επιπλέον θα κάνουμε μια συκριτική μελέτη και θα σχολιάσουμε τα πλεονεκτήματα και μειονεκτήματα κάθε συστήματος.

Τα red black trees είναι μια μορφή δυαδικού δέντρου αναζήτησης (BST), αλλά με εξισορρόπηση . Το βάθος ενός κόμβου σε ένα δέντρο είναι η απόσταση από τη ρίζα στον κόμβο. Το ύψος ενός δέντρου είναι το βάθος του βαθύτερου κόμβου. Η λειτουργία εισαγωγής ή αναζήτησης του αλγορίθμου BST (στο κεφάλαιο Binary Search Trees) απαιτεί χρόνο ανάλογο προς το βάθος του κόμβου που βρίσκεται (ή εισάγεται). Για να λειτουργήσουν γρήγορα αυτές οι λειτουργίες, θέλουμε δέντρα όπου το βάθος χειρότερης περίπτωσης (ή το μέσο βάθος) είναι όσο το δυνατόν μικρότερο.

Σε ένα τέλεια ισορροπημένο δέντρο N κόμβων, κάθε κόμβος έχει βάθος μικρότερο από ή ίσο με το λογάριθμο N, χρησιμοποιώντας λογάριθμους βάσης 2. Σε ένα ισορροπημένο δέντρο, κάθε κόμβος έχει βάθος μικρότερο ή ίσο με 2 log N. Αυτό είναι αρκετά καλό για να πραγματοποιούμε εισαγωγή και αναζήτηση σε χρονικό διάστημα ανάλογο με το λογάριθμο του N. Το τέχνασμα για να το πετύχουμε αυτό, είναι να χρωματίσουμε τους κόμβους Κόκκινο και Μαύρο, και από αυτά τα σημάδια να μάθουμε πότε να εξισορροπήσουμε τοπικά το δέντρο μέσω της συνάρτησης balance όπως παρουσιάζεται παρακάτω.

### 4.8.1 Ανάλυση με Coq

Αναπαριστούμε τη δομή δεδομένων Red black Trees σε Coq [14]:

```
Require Import Perm.
Require Import Extract.
Require Import Coq.Lists.List.
Export ListNotations.

Definition key := int.

Inductive color := Red | Black.

Section TREES.
Variable V : Type.
Variable default: V.

 Inductive tree : Type :=
 | E : tree
 | T: color → tree → key → V → tree → tree.

 Definition empty_tree := E.
```

Έχοντας ορίσει τη δομή δεδομένων ορίζουμε συναρτήσεις όπως lookup για αναζήτηση στοιχείου, balance για εξισορρόπηση του δέντρου μετά από εισαγωγή στοιχείου και ins, insert για εισαγωγή στοιχείου.
Η αναζήτηση είναι ακριβώς όπως στα binary search trees, εκτός από το ότι ο constructor T φέρει ένα στοιχείο χρώματος, το οποίο μπορούμε να αγνοήσουμε εδώ:

```
Fixpoint lookup (x: key) (t : tree) : V :=
  match t with
  | E ⇒ default
  | T _ tl k v tr ⇒ if ltb x k then lookup x tl
                        else if ltb k x then lookup x tr
```

```
                          else v
  end.
```

Η συνάρτηση εξισορρόπησης (balance) είναι ακριβώς ίδια με αυτή που παρουσιάζεται στο paper του Okasaki.

```
Definition balance rb t₁ k vk t₂ :=
 match rb with Red ⇒ T Red t₁ k vk t₂
 | _ ⇒
 match t₁ with
 | T Red (T Red a x vx b) y vy c ⇒
      T Red (T Black a x vx b) y vy (T Black c k vk t₂)
 | T Red a x vx (T Red b y vy c) ⇒
      T Red (T Black a x vx b) y vy (T Black c k vk t₂)
 | a ⇒ match t₂ with
            | T Red (T Red b y vy c) z vz d ⇒
                T Red (T Black t₁ k vk b) y vy (T Black c z vz d)
            | T Red b y vy (T Red c z vz d) ⇒
                T Red (T Black t₁ k vk b) y vy (T Black c z vz d)
            | _ ⇒ T Black t₁ k vk t₂
            end
  end
 end.

Definition makeBlack t :=
  match t with
  | E ⇒ E
  | T _ a x vx b ⇒ T Black a x vx b
  end.

Fixpoint ins x vx s :=
 match s with
 | E ⇒ T Red E x vx E
 | T c a y vy b ⇒ if ltb x y then balance c (ins x vx a) y vy b
                       else if ltb y x then balance c a y vy (ins x
vx b)
                       else T c a x vx b
 end.

Definition insert x vx s := makeBlack (ins x vx s).
```

Τώρα που έχει οριστεί το πρόγραμμα, μπορούμε να προχωρήσουμε να αποδείξουμε τις ιδιότητές του. Ένα red black tree έχει δύο είδη ιδιοτήτων:

- SearchTree: Τα κλειδιά σε κάθε αριστερό υποδέντρο είναι όλα μικρότερα από το κλειδί του κόμβου και τα κλειδιά σε κάθε δεξί υποδέντρο είναι μεγαλύτερα

- Ισορροπημένο: υπάρχει ο ίδιος αριθμός μαύρων κόμβων σε κάθε διαδρομή από τη ρίζα σε κάθε φύλλο και δεν υπάρχουν ποτέ δύο κόκκινοι κόμβοι στη σειρά.

Αρχικά θα αποδείξουμε ότι κάθε δέντρο της μορφής (T c l k v r) δεν είναι κενό:

```
Lemma T_neq_E:
  ∀ c l k v r, T c l k v r ≠ E.
Proof.
intros. intro Hx. inversion Hx.
Qed.
```

Πολλές από τις αποδείξεις για red black trees απαιτούν ανάλυση αρκετών περιπτώσεων καθώς πρέπει να εξετάσουμε όλες τις περιπτώσεις της συνάρτησης balance. Αυτές οι αποδείξεις είναι πολύ κουραστικό να υλοποιηθούν "με το χέρι", αλλά είναι εύκολο να αυτοματοποιηθούν.

Ένα παράδειγμα παρουσιάζεται στη συνέχεια. Έστω λοιπόν ότι θέλουμε να αποδείξουμε το εξής λήμμα:

```
Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

Ας χρησιμοποιήσουμε destruct στην ανώτατη περίπτωση, δηλαδή στο: ltb x k. Μπορούμε να χρησιμοποιήσουμε destruct αντί για bdestruct επειδή δεν χρειάζεται να θυμόμαστε αν ισχύει x <k ή x≥k.

```
destruct (ltb x k).
(* The topmost test is match c with..., so just destruct c *)
destruct c.
apply T_neq_E.
(* The topmost test is match a1 with..., so just destruct a1 *)
destruct a1.
(* The topmost test is match s2 with..., so just destruct s2 *)
destruct s2.
intro Hx; inversion Hx.
```

Συνεχίζοντας τη παραπάνω διαδικασία θα χρειαστεί να εξετάσουμε και να αναλύσουμε πάρα πολλές περιπτώσεις οπότε προσπαθούμε να αυτοματοποιήσουμε τη διαδικασία.

Η ακόλουθη τακτική ισχύει όταν ο τρέχων στόχος μοιάζει με , match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _, και αυτό που κάνει σε αυτή την περίπτωση είναι, destruct c:

```
match goal with
| |- match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
end.
```

Η επόμενη τακτική ισχύει όποτε ο τρέχον στόχος είναι της μορφής:

```
match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _,
```

και αυτό που κάνει σε αυτή περίπτωση είναι destruct s:

```
match goal with
    | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
```

Ας εφαρμόσουμε αυτή την τακτική ξανά, και στη συνέχεια δοκιμάζουμε στους υποστόχους, αναδρομικά. Η τακτική repeat συνεχίζει να προσπαθεί να κάνει την ίδια τακτική στους υποστόχους:

```
repeat match goal with
    | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
match goal with
  | |- T _ _ _ _ _ ≠ E ⇒ apply T_neq_E
end.
```

Τώρα μπορούμε να ξαναπροσπαθήσουμε την απόδειξη του θεωρήματος:

```
Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

Εδώ αρχίζει η ανάλυση των πολλών περιπτώσεων. Αυτή τη φορά, ας συνδυάσουμε πολλές τακτικές μαζί:

```
repeat match goal with
  | |- (if ?x then _ else _) ≠ _ ⇒ destruct x
  | |- match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
  | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
```

Μας απομένουν 117 περιπτώσεις, κάθε μία εκ των οποίων επιλύεται με τη ίδια τεχνική:

```
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
(* Only 111 cases to go... *)
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
(* Only 107 cases to go... *)
Abort.
```

Σαν τελευταία προσπάθεια αυτοματοποιούμε και την απόδειξη των τελευταίων περιπτώσεων:

```
Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
This is the beginning of the big case analysis. This time, we add
one more clause to the match goal command:
repeat match goal with
  | |- (if ?x then _ else _) ≠ _ ⇒ destruct x
  | |- match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
  | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
  | |- T _ _ _ _ _ ≠ E ⇒ apply T_neq_E
 end.
Qed.
```

Στη συνέχεια θα αναφερθούμε στη ιδιότητα SearchTree η οποία είναι ακριβώς η ίδια με αυτή που είχαμε ορίσει στα δυαδικά δέντρα με τη διαφορά ότι αγνοούμε το χρώμα στα red black trees:

```
Inductive SearchTree' : Z → tree → Z → Prop :=
| ST_E : ∀ lo hi, lo ≤ hi → SearchTree' lo E hi
| ST_T: ∀ lo c l k v r hi,
    SearchTree' lo l (int2Z k) →
    SearchTree' (int2Z k + 1) r hi →
    SearchTree' lo (T c l k v r) hi.

Inductive SearchTree: tree → Prop :=
| ST_intro: ∀ t lo hi, SearchTree' lo t hi → SearchTree t.
```

θα προχωρήσουμε στην απόδειξη ενός λήμματος που χρειάζεται ξανά να αυτοματοποιήσουμε την απόδειξη λόγω των πολλών περιπτώσεων που πρέπει να αναλύσουμε στη συνάρτηση balance:

```
Lemma balance_SearchTree:
 ∀ c s1 k kv s2 lo hi,
   SearchTree' lo s1 (int2Z k) →
   SearchTree' (int2Z k + 1) s2 hi →
   SearchTree' lo (balance c s1 k kv s2) hi.
Proof.
intros.
unfold balance.
--Χρησιμοποιούμε proof automation για την ανάλυση των περιπτώσεων.

repeat match goal with
  | |- SearchTree' _ (match ?c with Red ⇒ _ | Black ⇒ _ end) _ ⇒
              destruct c
  | |- SearchTree' _ (match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) _
⇒
              destruct s
  | H: SearchTree' _ E _ |- _ ⇒ inv H
```

```
  | H: SearchTree' _ (T _ _ _ _ _) _ |- _ ⇒ inv H
  end;
 repeat (constructor; auto).
Qed.
```

### 4.8.2 Ανάλυση με Liquid Haskell

Θα αναφέρουμε δύο τρόπους με τους οποίου μπορούμε να επαληθεύσουμε ιδιότητες των red black trees στη Liquid Haskell. Στην πρώτη μας προσέγγιση θα προσπαθήσουμε να αποδείξουμε ιδιότητες γράφοντας αποδείξεις στη Liquid Haskell όπως ακριβώς κάναμε και στα προηγούμενα κεφάλαια.

Αρχικά αναπαριστούμε τη δομή δεδομένων Red black Trees και τις συναρτήσεις που θα χρησιμοποιήσουμε σε Haskell κατά αντιστοιχία με το πρόγραμμα που είχαμε ορίσει στο Coq :

```
{-@ LIQUID "--reflection"                                  @-}
{-@ LIQUID "--ple"                                         @-}
{-@ LIQUID "--noadt"                                         @-}

module Redblack where
import Language.Haskell.Liquid.ProofCombinators


{-@ type Key = Nat@-}
type Key = Int

data Color = Red | Black deriving (Eq)

data Tree  v = E | T Color (Tree v) Key v (Tree v) deriving (Eq)


{-@
data Tree [tlen]  v = E
  | T   {  color  :: Color
         , bLeft  :: Tree  v
         , bKey   :: Key
         , bValue :: v
         , bRight :: Tree v }
@-}


{-@ reflect empty_tree @-}
empty_tree = E


{-@ measure tlen @-}
{-@ tlen :: Tree v-> Nat @-}
tlen :: (Tree v) -> Int
tlen (E)                 = 0
tlen (T color l key v  r) = 1 + (tlen l) + (tlen r)
```

```
{-@ reflect lookup' @-}
lookup' :: (Ord v, Eq v) => Key -> v -> Tree v -> v
lookup' x def (E)           = def
lookup' x def (T _  l k v r) = if(x<k)      then lookup' x def l
                               else if(x>k) then lookup' x def r
                               else v


{-@ reflect balance @-}
{-@ balance :: Color -> Tree v -> Nat -> v -> Tree v -> Tree v @-}
balance :: Color -> Tree v -> Key -> v -> Tree v -> Tree v
balance Red t1 k vk t2 = (T Red t1 k vk t2)
balance Black  (T Red (T Red a x vx b) y vy c) k vk t2
      = (T Red (T Black a x vx b) y vy (T Black c k vk t2))

balance Black  (T Red a x vx (T Red b y vy c)) k vk t2
      = (T Red (T Black a x vx b) y vy (T Black c k vk t2))

balance Black t1 k vk (T Red (T Red b y vy c) z vz d)
      = (T Red (T Black t1 k vk b) y vy (T Black c z vz d))

balance Black t1 k vk (T Red b y vy (T Red c z vz d))
      = (T Red (T Black t1 k vk b) y vy (T Black c z vz d))

balance Black t1 k vk t2
      = (T Black t1 k vk t2)


{-@ reflect makeBlack @-}
makeBlack :: Tree v -> Tree v
makeBlack E              = E
makeBlack (T _ a x vx b) = (T Black a x vx b)


{-@ reflect ins@-}
{-@ ins :: Nat -> v -> Tree v -> Tree v@-}
ins :: Key -> v -> Tree v -> Tree v
ins x vx (E) = (T Red E x vx E)
ins x vx (T c a y vy b)
             = if(x<y)      then balance c (ins x vx a) y vy b
               else if(y<x) then balance c a y vy (ins x vx b)
               else (T c a x vx b)

{-@ reflect insert @-}
{-@ insert :: Nat -> v -> Tree v -> Tree v @-}
insert :: Key -> v -> Tree v -> Tree v
insert x vx s = makeBlack (ins x vx s)
```

Τώρα που έχουμε ορίσει τη δομή δεδομένων και τις αντίστοιχες συναρτήσεις θα αποδείξουμε ορισμένες ιδιότητες για τα Red Black Trees σε Liquid Haskell:

- Για κάθε χρώμα c, red black trees l,r , κλειδί k, τιμή v το red black tree (T c l k val r) είναι διάφορο του κενού red black tree:

```
{-@ lemma_T_neq_E :: (Eq v) => c:Color -> l:Tree v -> k:Key -> val:v
-> r:Tree v -> { (T c l k val r) /= E }@-}
lemma_T_neq_E :: (Eq v) => Color -> Tree v -> Key -> v -> Tree v ->
Proof
lemma_T_neq_E c l k v r = ((T c l k v r) /= E ) *** QED
```

- Για κάθε κλειδί x, τιμή vx και red black tree s η εισαγωγή του (x,vx) στο s επιστρέφει μη κενό δέντρο:

```
{-@ lemma_ins_not_E :: x:Key -> vx:v -> s:Tree v -> {ins x vx s /=
E} @-}
lemma_ins_not_E :: Key -> v -> Tree v -> Proof
lemma_ins_not_E x vx E = trivial
lemma_ins_not_E x vx (T c a y vy b)
        |(x<y)   = trivial
        |(x>y)   = trivial
        |(x==y) = trivial
```

Ορίζουμε την ιδιότητα SearchTree που δηλώνει ότι ένα red black tree είναι δυαδικό δέντρο αναζήτησης ως εξής:

```
{-@reflect searchTree @-}
searchTree :: Int ->  Tree v -> Int -> Bool
searchTree lo E hi = (lo<=hi)
searchTree lo (T _ l k v r) hi
      = (searchTree lo l k) && (searchTree (k+1) r hi)
```

Έχοντας ορίσει την ιδιότητα searchTree μπορούμε να προχωρήσουμε στην απόδειξη ορισμένων θεωρημάτων σχετικών με την ιδιότητα αυτή. Το πρώτο θεώρημα που θα αποδείξουμε είναι ότι η balance επιστρέφει searchTree. Παρά το γεγονός ότι το παρακάτω θεώρημα εμπεριέχει τη συνάρτηση balance όλες οι περιπτώσεις μπορούν να επιλυθούν αυτόματα από το solver ενώ αν γράφαμε αυτή την απόδειξη σε coq θα έπρεπε να γράψουμε κάποιο tactic που να αυτοματοποιεί την ανάλυση των περιπτώσεων της balance όπως δείξαμε σε παρόμοια θεωρήματα:

```
{-@ lemma_balance_SearchTree :: lo:Int -> hi:Int -> k:Key
     -> s1:{ Tree v| searchTree lo s1 k}
     -> s2:{ Tree v| searchTree (k+1) s2 hi} -> c:Color
     -> kv:{v | searchTree lo (T c s1 k kv s2) hi}
     -> { searchTree  lo (balance c s1 k kv s2) hi}
@-}
lemma_balance_SearchTree ::
   Int -> Int -> Key -> Tree v -> Tree v -> Color  -> v -> Proof
lemma_balance_SearchTree lo hi k s1 s2 Red vk = trivial
lemma_balance_SearchTree lo hi k (T Red (T Red a x vx b) y vy c1) s2
Black vk = trivial
lemma_balance_SearchTree lo hi k (T Red a x vx (T Red b y vy c1)) s2
Black vk = trivial
```

```
lemma_balance_SearchTree lo hi k s1 (T Red (T Red b y vy c1) z vz d)
Black vk = trivial
lemma_balance_SearchTree lo hi k s1 (T Red b y vy (T Red c1 z vz d))
Black vk = trivial
lemma_balance_SearchTree lo hi k s1 s2 Black vk = trivial


--Αν για κάποιο δέντρο s ισχύει searchTree lo s hi, τότε αν
--εισάγουμε ένα κλειδί x (και αντίστοιχα κάποιο value) με lo<=x<hi
--παίρνουμε πάλι searchtree με όρια lo,hi

{-@ lemma_ins_SearchTree :: x:Key -> vx:v -> lo:{Int | lo<=x }
     -> hi:{Int | x<hi } -> s:{Tree v | searchTree lo s hi}
     -> {searchTree lo (ins x vx s) hi}
@-}
lemma_ins_SearchTree :: Key -> v -> Int -> Int -> Tree v -> Proof
lemma_ins_SearchTree x vx lo hi E = trivial
lemma_ins_SearchTree x vx lo hi (T c l k vk r)
         | (x<k)
            = [lemma_ins_SearchTree x vx lo k l,
               lemma_balance_SearchTree lo hi k (ins x vx l) r c vk]
              *** QED

         | (x>k)
            = [lemma_ins_SearchTree x vx (k+1) hi r,
               lemma_balance_SearchTree lo hi k l (ins x vx r) c vk]
               *** QED

         | otherwise = trivial


--Το κενό δέντρο είναι searchTree

{-@ empty_tree_SearchTree :: lo:Int -> hi:{Int | lo <= hi }
     -> {searchTree lo empty_tree hi}
@-}
empty_tree_SearchTree :: Int -> Int -> Proof
empty_tree_SearchTree lo hi = trivial


--Αν ένα δέντρο είναι searchTree για κάποια lo,hi τότε μπορούμε να
--εξάγουμε ότι lo <= hi

{-@ lemma_SearchTree:: lo:Int -> hi:Int
      -> t: {Tree v| searchTree lo t hi}
      -> {lo<=hi}
@-}
lemma_SearchTree :: Int -> Int -> Tree v -> Proof
lemma_SearchTree lo hi E = trivial
lemma_SearchTree lo hi (T c l k vk r) = [lemma_SearchTree lo k l,
lemma_SearchTree (k+1) hi r] *** QED


--Απλό θεώρημα όπου αλλάζουμε τα όρια lo,hi στην ιδιότητα
```

```
--searchTree, αποδεικνύεται εύκολα με επαγωγή και το προηγούμενο
--λήμμα

{-@ lemma_expand_range_SearchTree:: lo:Int -> hi:Int
     -> s: {Tree v| searchTree lo s hi} -> lo':{Int | lo'<=lo }
     -> hi':{Int | hi<=hi' }
     -> {searchTree lo' s hi'}
@-}
lemma_expand_range_SearchTree
    :: Int -> Int -> Tree v -> Int -> Int -> Proof
lemma_expand_range_SearchTree lo hi E lo' hi' = trivial
lemma_expand_range_SearchTree lo hi (T c l k vk r) lo' hi'
       = ( lemma_SearchTree lo k l,
           lemma_SearchTree (k+1) hi r,
           lemma_expand_range_SearchTree lo k l lo' k ,
           lemma_expand_range_SearchTree (k+1) hi r (k+1) hi' )
       *** QED
```

Μέχρι στιγμής δεν έχουμε αναφερθεί σε ισορροπημένα δέντρα που είναι μια εκ των δύο ιδιοτήτων που πρέπει να έχουν τα red black trees. Ορίζουμε λοιπόν την συνάρτηση is_redblack που δέχεται ένα red black tree και ελέγχει αν το πλήθος των μαύρων κόμβων για το αριστερό και δεξί υποδέντρο είναι ίσα και δεν υπάρχουν δύο διαδοχικοί κόκκινοι κόμβοι. Η συνάρτηση δέχεται επίσης μία παράμετρο τύπου Color προκειμένου να κρατάμε το χρώμα συναντήσαμε στο προηγούμενο βήμα για δούμε τι χρώμα επιτρέπεται να υπάρχει στο επόμενο για να πληρείται η παραπάνω ιδιότητα, καθώς επίσης και ένα φυσικό αριθμό N για να μετράμε τους μαύρους κόμβους και να εγγυηθούμε ότι κάθε μονοπάτι από τη ρίζα έχει ακριβώς N Black κόμβους, οπότε έτσι εγγυόμαστε ότι όλα τα μονοπάτια από τη ρίζα έχουν ίδιο πλήθος Black κόμβων (και ίσο με N):

```
{-@ reflect is_redblack @-}
{-@ is_redblack :: Tree v -> Color -> Nat -> Bool @-}
is_redblack :: Tree v -> Color -> Int -> Bool
is_redblack  E                   c     0  = True
is_redblack (T Red   tl k kv tr) Black n  =
      is_redblack tl Red   n && is_redblack tr Red   n
is_redblack (T Black tl k kv tr) c     0  = False
is_redblack (T Black tl k kv tr) c     n  =
      is_redblack tl Black (n-1) && is_redblack tr Black (n-1)
is_redblack  _                   _     _  = False
```

Από τη παραπάνω συνάρτηση μπορούμε εύκολα να παρατηρήσουμε ότι αν δώσουμε σαν αρχικό χρώμα Red και ισχύει για κάποιο δέντρο t και φυσικό αριθμό N τότε η ιδιότητα is_redblack θα ισχύει και αν δίναμε σαν αρχικό χρώμα Black αφού επιτρέπεται να έχουμε δύο η περισσότερους διαδοχικούς μαύρους κόμβους. Αυτό το θεώρημα αποδεικνύουμε παρακάτω και η απόδειξη γίνεται αυτόματα από τον SMT solver:

```
{-@ lemma_is_redblack_toblack :: n:Nat ->
    s:{ Tree v | is_redblack s Red n} -> { is_redblack s Black n }
@-}
lemma_is_redblack_toblack :: Int -> Tree v -> Proof
```

```
lemma_is_redblack_toblack n E = trivial
lemma_is_redblack_toblack n (T Red tl k kv tr)   = trivial
lemma_is_redblack_toblack n (T Black tl k kv tr) = trivial
```

Το τελευταίο θεώρημα που θα αποδείξουμε για την ιδιότητα is_redblack είναι ότι αν ισχύει is_redblack s Black n για κάποιο red black tree s τότε αν μετατρέψουμε το κόμβο της ρίζας σε Black χρησιμοποιώντας (μπορεί να ήταν ήδη Black) θα ισχύει πάλι is_redblack (makeBlack s) Black n', όπου το n' θα ισούται με n αν ο κόμβος ήταν black αφού σε αυτή τη περίπτωση δεν αλλάξαμε κάτι, και n' = n+1 αν η ρίζα ήταν Red αφού έτσι μετατρέποντας τη σε Black έχουμε n+1 μαύρους κόμβους πλέον.

```
{-@ lemma_makeblack_fiddle:: n:Nat ->
       s:{Tree v | is_redblack s Black n } ->
      (n':: {v:Int | v>=0}, {is_redblack (makeBlack s) Red n'})
 @-}
lemma_makeblack_fiddle :: Int -> Tree v -> (Int , Proof)
lemma_makeblack_fiddle 0 E = (0, trivial )
lemma_makeblack_fiddle n E = (n+1, trivial )
lemma_makeblack_fiddle n (T Red tl k kv tr) =
                  (n+1, [lemma_is_redblack_toblack n tl,
                         lemma_is_redblack_toblack n tr]*** QED )
lemma_makeblack_fiddle n (T Black tl k kv tr) = (n, trivial )
```

Όπως έχουμε τονίσει μια προϋπόθεση ώστε ένα δέντρο για να είναι red black tree είναι ότι πρέπει να μην έχει δύο διαδοχικούς κόκκινους κόμβους. Ορίζουμε ως Almost Red Black Tree ένα δέντρο που πληροί όλες τις προϋποθέσεις των red black tree αλλά επιτρέπεται να περιέχει δύο διαδοχικού κόκκινους κόμβους μόνο στη ρίζα. Χρησιμοποιώντας τη συνάρτηση is_redblack μπορούμε εύκολα να ορίσουμε αυτή την ιδιότητα:

```
{-@ measure almost _redblack @-}
{-@ almost _redblack :: Tree v -> Nat -> Bool @-}
almost_redblack :: Tree v -> Int -> Bool
almost _redblack (T Red tl k kv tr)     n = is_redblack tl Black n && is_redblack tr Black n
almost _redblack (T Black tl k kv tr)  0 = False
almost _redblack (T Black tl k kv tr)  n =
                                    is_redblack tl Black (n-1) && is_redblack tr Black (n-1)
almost_redblack  E  n = False
```

Τώρα θα θέλαμε να αποδείξουμε ορισμένες ιδιότητες της παραπάνω συνάρτησης όπως:

- ```
  {-@ lemma_ins_is_redblack1 :: x:Key -> vx:v -> s:Tree v ->
           n:{Nat |  is_redblack s Red n}
          ->{ is_redblack (ins x vx s) Black n }
  @-}
  ```

- ```
  {-@ lemma_ins_is_redblack :: x:Key -> vx:v -> s:Tree v -> n:{Nat |
  is_redblack s Black n}
          ->{ nearly_redblack (ins x vx s) n }
  ```

```
@-}
```

Τα δύο παραπάνω θεωρήματα είναι αρκετά σύνθετα και δεν μπορούν να επαληθευτούν αυτόματα από τον SMT solver. Για να τα αποδείξουμε θα πρέπει να παρέχουμε κάποια βοηθητικά λήμματα πιθανώς και να γράψουμε κάποια equational proof, για κάθε πιθανή περίπτωση. Η δυσκολία των παραπάνω θεωρημάτων έγκειται ότι συνδυάζουν εκτός από τη συνάρτηση almost_redblack και τη συνάρτηση ins που χρησιμοποιεί τη balance που όπως έχουμε δει έχει πολλές περιπτώσεις. Έτσι τόσο σε Coq όσο και σε liquid Haskell θα έπρεπε να αναλύσουμε κάθε περίπτωση της balance και κάθε περίπτωση της ins, συνολικά 3*5=15 περιπτώσεις ή να γράψουμε κώδικα που να αυτοματοποιεί αυτή τη διαδικασία. Αντί αυτού θα δείξουμε μία διαφορετική αντιμετώπιση σε Liquid Haskell χρησιμοποιώντας refinement types.

### 4.8.3 Ανάλυση με Liquid Haskell χρησιμοποιώντας αποκλειστικά refinement types

Σε αυτή τη προσέγγιση θα προσπαθήσουμε να εκφράσουμε και να επαληθεύσουμε ιδιότητες των red black trees χρησιμοποιώντας refinement types στη Liquid Haskell.

Όπως και πριν ορίζουμε τη δομή δεδομένων:

```
{-@ LIQUID "--no-totality" @-}

module Redblack where
import Language.Haskell.Liquid.Prelude

data Color = Red | Black deriving (Eq, Show)

data RBTree a v = Leaf
                | Node { nCol   :: Color
                       , nKey   :: a
                       , nvalue :: v
                       , nLeft  :: !(RBTree a v)
                       , nRight :: !(RBTree a v)
                       }
                deriving (Show)


{-@ data RBTree a v <l :: a -> a -> Bool, r :: a -> a -> Bool>
              = Leaf
              | Node { nCol   :: Color
                     , nKey   :: a
                     , nvalue :: v
                     , nLeft  :: RBTree <l, r> (a <l nKey>) v
                     , nRight :: RBTree <l, r> (a <r nKey>) v
                    }
@-}
```

Ορίζουμε τις εξής ιδιότητες στα red black trees:

- Το μήκος των μαύρων κόμβων στο δεξί και το αριστερό υποδέντρο πρέπει να είναι ίσα:

```
--συνάρτηση που ελέγχει για την ιδιότητα που αναφέραμε
{-@ measure isBH        :: RBTree a v -> Bool
    isBH (Leaf)         = true
    isBH (Node c x v l r) = (isBH l && isBH r && bh l = bh r)
 @-}

--συνάρτηση που μετράει το πλήθος των μαύρων κόμβων
{-@ measure bh        :: RBTree a v -> Int
    bh (Leaf)         = 0
    bh (Node c x v l r) = bh l + if (c == Red) then 0 else 1
 @-}
```

- Εκφράζουμε πότε ένας κόμβος έχει μαύρο χρώμα με τις παρακάτω συναρτήσεις:

```
--συνάρτηση που επιστρέφει το χρώμα του κόμβου που είναι στη ρίζα:
{-@ measure col         :: RBTree a v-> Color
    col (Node c x v l r)  = c
    col (Leaf)            = Black
 @-}

--συνάρτηση που ελέγχει πότε ένας κόμβος έχει μαύρο χρώμα
{-@ measure isB        :: RBTree a v -> Bool
    isB (Leaf)           = false
    isB (Node c v x l r) = c == Black
@-}

--κατηγόρημα που εκφράζει ότι ένας κόμβος δεν έχει κόκκινο χρώμα
{-@ predicate IsB T = not (col T == Red) @-}
```

- Αντίστοιχα με την ιδιότητα searchTree που είχαμε ορίσει στην προηγούμενη ενότητα, διατυπώνουμε πότε ένα δέντρο είναι δέντρο αναζήτησης ως εξής:

```
--ουσιαστικά διατυπώνουμε ότι για κάθε κόμβο στο δέντρο όλοι οι
--κόμβοι που βρίσκονται στο αριστερό υποδέντρο είναι μικρότεροι και
--όλοι που βρίσκονται στο δεξί είναι μεγαλύτεροι

{-@ type ORBT a v = RBTree <{\root v -> v < root }, {\root v -> v >
root}> a v@-}
```

- Ένα δέντρο είναι red black tree όταν είναι searchtree και το πλήθος των μαύρων κόμβων σε κάθε μονοπάτι από τη ρίζα ως τα φύλλα είναι ίσο και δεν υπάρχουν διαδοχικοί κόκκινοι κόμβοι (και το αριστερό και δεξί υποδέντρο πληρούν επίσης τις ιδιότητες αυτές).

```
--συνάρτηση που μετράει και ελέγχει αν πληρείται η ιδιότητα για το
--πλήθος των μαύρων κόμβων
{-@ measure isRB        :: RBTree a v -> Bool
    isRB (Leaf)         = true
    isRB (Node c x v l r)
          = isRB l && isRB r && (c == Red => (IsB l && IsB r))
```

```
@-}

--εκφράζουμε τα red black tree χρησιμοποιώντας το refinement type
--που εκφράζει ότι είναι searchtree πληρούν την ιδιότητα με του
--μαύρους κόμβους και δεν υπάρχουν διαδοχικοί κόκκινοι κόμβοι.

{-@ type RBT a  v1  = {v: ORBT a v1 | isRB v && isBH v } @-}

--Red black tree με πλήθος μαυρων κόμβων σε κάθε μονοπάτι από τη
--ρίζα ως τα φύλλα ίσο με τη παράμετρο N
{-@ type RBTN a v1 N = {v: RBT a v1 | bh v = N } @-}
```

- Τέλος εκφράζουμε την ιδιότητα almost red black tree ως εξής:

```
--συνάρτηση που αγνοεί το χρώμα στη ρίζα και ελέγχει αν τα υποδέντρα
--πληρούν τις ιδιότητες των red black tree
{-@ measure isARB        :: (RBTree a v) -> Bool
    isARB (Leaf)           = true
    isARB (Node c x v l r) = (isRB l && isRB r)
@-}

--almost red black tree
{-@ type ARBT a v1    = {v: ORBT a v1 | isARB v && isBH v} @-}

--almost red black tree με πλήθος μαυρων κόμβων σε κάθε μονοπάτι από
--τη ρίζα ως τα φύλλα ίσο με τη παράμετρο N
{-@ type ARBTN a v1 N = {v: ARBT a v1 | bh v = N }         @-}
```

Τώρα που έχουμε εκφράσει όλες τις απαραίτητες ιδιότητες θα προχωρήσουμε να ορίσουμε τις συναρτήσεις στα red black trees που είχαμε ορίσει και στη προηγούμενη ενότητα και θα προσθέσουμε refinement types που να επαληθεύουν ιδιότητες:

- Ξεκινάμε με τη συνάρτηση lookup καθώς είναι η πιο απλή και δεν έχει ιδιαίτερο ενδιαφέρον αφού το refinement type της lookup είναι πολύ απλό:

```
{-@ lookup :: (Ord k, Eq k) => k -> v -> t:RBT k v -> v @-}
lookup x def (Leaf)         = def
lookup x def (Node _  k v l r) = if(x<k)      then lookup x def l
                                 else if(x>k) then lookup x def r
                                 else v
```

- Στη συνέχεια θα δούμε τη συνάρτηση balance που έχει και το περισσότερο ενδιαφέρον:

```
{-@ balance :: c:Color -> k:a -> t1:ORBT {k1:a | k1<k} v1 -> v1
    -> t2:{ ORBT  {k1:a | k1>k} v1 |
    if (c==Black) then ((isRB t2 && isBH t2 && (bh t2 = bh t1)
       && (isARB t1 && isBH t1) ) ||
               (isRB t1 && isBH t1 && (bh t2 = bh t1)
       && (isARB t2 && isBH t2) )  )
    else
       (isRB t2 && isBH t2 && (bh t2 = bh t1) && isRB t1 && isBH t1 )
   }
    -> t3:{ARBT a v1 |
```

```
    (if (c==Black) then (bh t3 = bh t1+1) else (bh t3 = bh t1))
     && ((c == Red && IsB t1 && IsB t2) || (c == Black) => isRB t3)
     }
@-}
balance Red t1 k vk t2 = (T Red t1 k vk t2)
balance Black  (T Red (T Red a x vx b) y vy c) k vk t2
      = (T Red (T Black a x vx b) y vy (T Black c k vk t2))

balance Black  (T Red a x vx (T Red b y vy c)) k vk t2
      = (T Red (T Black a x vx b) y vy (T Black c k vk t2))

balance Black t1 k vk (T Red (T Red b y vy c) z vz d)
      = (T Red (T Black t1 k vk b) y vy (T Black c z vz d))

balance Black t1 k vk (T Red b y vy (T Red c z vz d))
      = (T Red (T Black t1 k vk b) y vy (T Black c z vz d))

balance Black t1 k vk t2
      = (T Black t1 k vk t2)
```

Στο refinement type της balance δηλώνουμε ότι δέχεται μία παράμετρο c τύπου Color, έναν Key k, ένα red black tree t1 που έχει την ιδιότητα searchtree και επιπλέον η τιμή του κλειδιού που βρίσκεται στη ρίζα του t1 είναι μικρότερη του κλειδιού k, ένα red black tree t2 που έχει την ιδιότητα searchtree και και επιπλέον η τιμή του κλειδιού που βρίσκεται στη ρίζα του t2 είναι μεγαλύτερη του κλειδιού k και τέλος επισημαίνουμε ότι για τις παραμέτρους t1,t2 ότι:

- Αν το c είναι Black το πολύ ένα εκ των t1,t2 θα είναι almost red black tree ενώ το άλλο (ή και τα δύο) θα είναι έγκυρα red black tree (δηλαδή θα πληρούν όλες τις ιδιότητες των red black trees) και θα έχουν το ίδιο πλήθος μαύρων κόμβων (bh t2 = bh t1).

- Αν το c είναι Red, τα t1,t2 θα είναι έγκυρα red black tree και θα έχουν το ίδιο πλήθος μαύρων κόμβων.

Τα παραπάνω είναι οι συνθήκες που γράφουμε για τις εισόδους προκειμένου ο SMT να μπορέσει να αποδείξει τις ιδιότητες της εξόδου με δεδομένες τις παραπάνω ιδιότητες. Για να διατυπώσουμε ποιες είναι οι συνθήκες εισόδου έπρεπε να σκεφτούμε τι πρέπει να ισχύει ώστε η balance να επιστρέφει την επιθυμητή έξοδο, δηλαδή πάντα επιστρέφει almost red black tree και υπό ορισμένες συνθήκες red black tree.

Οι ιδιότητες που επαληθεύονται αυτόματα από τον SMT solver για την balance είναι οι εξής:

- Η balance επιστρέφει πάντα almost red black tree και αν το c είναι Black ή αν το c είναι Red και οι ρίζες των t1,t2 είναι black (IsB t1, IsB t2) τότε επιστρέφει red black tree.
- Αν το c είναι Red το δέντρο που επιστρέφει έχει σε κάθε μονοπάτι από τη ρίζα μέχρι τα φύλλα, όσους μαύρους κόμβους έχουν τα t1,t2 (τα t1,t2 έχουν πάντα ίδιο πλήθος μαύρων κόμβων) αλλιώς αν το c είναι Black έχει ένα παραπάνω από τα t1,t2. Η τελευταία ιδιότητα θα φανεί χρήσιμη στα refinement type μετέπειτα συναρτήσεων.

➢ Θα προχωρήσουμε να αναλύσουμε τη συνάρτηση makeblack:

```
{-@ makeBlack :: ARBT a v -> RBT a v @-}
```

```
makeBlack Leaf           = Leaf
makeBlack (Node _ x v l r) = Node Black x v l r
```

Η παραπάνω συνάρτηση μετατρέπει το πρώτο κόμβο ενός δέντρου που είναι almost red black tree, δηλαδή πληροί τις ιδιότητες των red black trees αλλά μπορεί να έχει δύο διαδοχικούς κόκκινους κόμβους μόνο στη ρίζα, σε μαύρο κόμβο άρα αν όντως είχε δύο κόκκινου διαδοχικού κόμβους μετατρέποντας τη ρίζα σε μαύρο γίνεται red black tree. Αυτή την ιδιότητα ότι η makeblack επιστρέφει red black tree την εξάγει αυτόματα ο solver έχοντας απλά ως δεδομένο ότι η είσοδος είναι almost red black tree. Αυτή την ιδιότητα την είχαμε αποδείξει στη προηγούμενη ενότητα στο θεώρημα: lemma_makeblack_fiddle.

- Θα εξετάσουμε τώρα τη συνάρτηση ins για να ελέγξουμε τι ιδιότητες μπορούμε να επαληθεύσουμε:

```
{-@ ins :: (Ord a) => a -> v1 -> t:RBT a v1 ->
    v:{ARBTN a v1 {bh t} | IsB t => isRB v}
@-}
ins k kx (Leaf)              = Node Red k kx Leaf Leaf
ins k kx (Node Red y vy a b) =
           if(k<y)      then  balance Red y (ins k kx a)  vy b
           else if(y<k) then  balance Red y a  vy (ins k kx b)
           else (Node Red  k kx a  b)
ins k kx (Node Black y vy a b) =
           if(k<y)      then  balance Black y (ins k kx a)  vy b
           else if(y<k) then  balance Black y a  vy (ins k kx b)
           else (Node Black  k kx a  b)
```

Η συνάρτηση δέχεται ένα key, ένα value και ένα red black tree και εισάγει το key και το αντίστοιχο value στο δέντρο και καλεί τη συνάρτηση balance για να ισορροπήσει το δέντρο ώστε να συνεχίσουν να ισχύουν οι διάφορες ιδιότητες του δέντρου και μετά την εισαγωγή. Έχουμε ήδη αποδείξει ότι η balance επιστρέφει ARBT και υπό κάποιες συνθήκες επιστρέφει Red black tree καθώς επίσης αναλόγως την παράμετρο color που δέχεται η balance είχαμε εξάγει και πληροφορίες για το πλήθος των μαύρων κόμβων σε κάθε μονοπάτι που ξεκινάει από τη ρίζα. Στη συνάρτηση ins ο solver μπορεί αυτόματα να επαληθεύσει ότι:

  - Πάντα επιστρέφει ARBT με πλήθος μαύρων κόμβων όσους είχε και το δέντρο στο οποίο εισάγει το key. Αυτό είναι το λήμμα lemma_ins_is_redblack το οποίο ήταν αρκετά πολύπλοκο για να αποδείξουμε προηγουμένως.

  - Αν το δέντρο που δέχεται σαν είσοδο έχει μαυρο κόμβο στη ρίζα τότε το δέντρο που επιστρέφει είναι red black tree (όχι δηλαδή μόνο ARBT). Αυτό είναι το θεώημα lemma_ins_is_redblack1 που επίσης ήταν αρκετά πολύπλοκο για να αποδείξουμε προηγουμένως

- Τέλος η τελευταία συνάρτηση που θα μελετήσουμε είναι η insert που το μόνο που κάνει είναι να εισάγει ένα key, value σε ένα red black tree χρησιμοποιώντας την ins, και μετατρέπει το πρώτο κόμβο του δέντρου που επιστρέφει η ins σε black μέσω της makeblack, προκειμένου να διασφαλίσει ότι το δέντρο που θα επιστραφεί είναι red black tree και όχι almost red black tree που πιθανώς να επιστρέψει η ins.

```
{-@ insert :: k -> v -> RBT k v -> RBT k v @-}
insert x vx s = makeBlack (ins x vx s)
```

Το θεώρημα που αποδεικνύει αυτόματα για την παραπάνω συνάρτηση ο solver είναι ότι η insert πάντα αν κληθεί με είσοδο red black tree επιστρέφει πάντα red black tree δηλαδή πληροί όλες τις απαραίτητες συνθήκες που έχουμε περιγράψει για τα red black trees.

### 4.8.4 Συμπεράσματα

Στις παραπάνω ενότητες εξετάσαμε και αποδείξαμε διάφορες ιδιότητες για τα red black tree χρησιμοποιώντας το Coq και δύο διαφορετικές προσεγγίσεις με τη Liquid Haskell. Από τα παραπάνω λοιπόν εξάγουμε τα εξής συμπεράσματα:

1. Χρησιμοποιώντας το Coq απλές ιδιότητες όπως το λήμμα ins_not_E χρειάζεται ο χρήστης να γράψει κάποιο κώδικα που να αυτοματοποιεί τη διαδικασία καθώς αλλιώς οι περιπτώσεις που έχει να εξετάσει είναι υπερβολικά πολλές.

2. Με τη πρώτη προσέγγιση της Liquid Haskell, γράφοντας θεωρήματα καταφέραμε να αποδείξουμε αρκετά από αυτά πλήρως αυτοματοποιημένα χωρίς να γράψουμε κώδικα που να εξετάζει αυτόματα διάφορες περιπτώσεις αφού την ανάλυση των περιπτώσεων την αναλαμβάνει επιτυχώς ο solver. Ωστόσο με αυτή τη διαδικασία δεν μπορέσαμε να αποδείξουμε ορισμένα πιο σύνθετα θεωρήματα που πλέον ο solver από μόνο τους δεν μπορεί να αποδείξει.

3. Με την δεύτερη προσέγγιση στη Liquid Haskell δηλαδή γράφοντας ιδιότητες με refinement types καταφέραμε ουσιαστικά να επαληθεύσουμε όλα τα θεωρήματα και αυτά που πριν ήταν αρκετά δύσκολα, πλήρως αυτοματοποιημένα από το solver.

4. Επιπλέον η τελευταία προσέγγιση στη Liquid Haskellμπορούμε να πούμε ότι είναι πιο σωστή και αξιόπιστη καθώς όπως αναφέραμε στην αρχή στα red black tree μας ενδιαφέρουν δύο ιδιότητες, η ιδιότητα searchtree και οι ιδιότητες που σχετίζονται με το αν το δέντρο είναι ισορροπημένο (όπως πλήθος μαύρων κόμβων κλπ..), και στη δεύτερη προσέγγιση εξετάζουμε και τα δύο επιθυμητά είδη ιδιοτήτων συγχρόνως σε κάθε συνάρτηση. Στις αποδείξεις με το Coq και στη πρώτη προσέγγιση με τη Liquid Haskell πρώτα εξετάσαμε την ιδιότητα searchtree και αποδείξαμε σχετικά θεωρήματα και μετά ασχοληθήκαμε ξεχωριστά με ιδιότητες που έχουν να κάνουν εξισορρόπηση, πράγμα που δεν είναι τόσο αξιόπιστο.

5. Τέλος είδαμε ότι η Liquid Haskell μας προσφέρει πιο πολλές τεχνικές για να επαληθεύσουμε ιδιότητες , είτε διατυπώνοντας θεωρήματα και αποδεικνύοντας τα είτε μέσω refinement types. Και με τις δύο τεχνικές είδαμε ότι υπάρχει καλύτερη αυτοματοποίηση αφού όλες οι ιδιότητες και τα θεωρήματα αποδεικνύονται αυτόματα από το solver, ακόμα και θεωρήματα που είχαν πολλές περιπτώσεις (πχ θεωρήματα που περιέχουν τη συνάρτηση balance) όπου στο Coq θα ήταν μονόδρομος να γράψουμε κάποιο tactic που να αυτοματοποιεί τη διαδικασία αποδείξεων. Εδώ να τονίσουμε ότι θα μπορούσαμε να συνενώναμε τις δύο προσεγγίσεις στη Liquid Haskell έτσι ώστε ορισμένες ιδιότητες να τις αποδεικνύαμε σαν θεωρήματα και άλλες όπου πιθανώς να

δυσκολευόμασταν (π.χ lemma_ins_is_redblack) να χρησιμοποιήσουμε refinement types.

## 4.9 Trie: Number Representations and Efficient Lookup Tables

Είναι γνωστό ότι οι καθαρά συναρτησιακές γλώσσες προγραμματισμού λόγο της έλλειψης πινάκων αρκετές φορές επιβαρύνονται ως προς την ασυμπτωτική πολυπλοκότητα κατά ένα παράγοντα log n σε σχέση με τις προστακτικές γλώσσες. Σαν παράδειγμα έχουμε το παρακάτω πρόγραμμα γραμμένο σε προστακτική γλώσσα προγραμματισμού [15] :

```
collisions=0;
  for (i=0; i<2N; i++)
     a[i]=0;
  for (j=0; j<N; j++) {
     i = input[j];
     if (a[i] != 0)
       collisions++;
     a[i]=1;
  }
  return collisions;
```

Το παραπάνω πρόγραμμα διαβάζει N φυσικούς αριθμούς που ανήκουν στο διάστημα [0, 2N] και χρησιμοποιεί πίνακα προκειμένου να υπολογίσει το πλήθος των αριθμών που εμφανίζονται τουλάχιστον δύο φορές. Η ασυμπτωτική πολυπλοκότητα του προγράμματος αυτού είναι O(N).

Σε κάποιο συναρτησιακή γλώσσα προγραμματισμού για να γράψουμε το παραπάνω πρόγραμμα θα έπρεπε να χρησιμοποιήσουμε Map αντί πινάκων και κάθε φορά αντί να θέτουμε a[i]=1 θα ενημερώνουμε το αντίστοιχο πεδίο στο Map με τη τιμή 1. Αυτή η λύση έχει μια επιβάρυνση στην ασυμπτωτική πολυπλοκότητα κατά ένα παράγοντα log N καθώς η ασυμπτωτική πολυπλοκότητα σε μια συναρτησιακή γλώσσα θα ήταν O(N log N). Αυτό ωστόσο εξαρτάται από την υλοποίηση των Maps που χρησιμοποιούμε, μια μη αποδοτική υλοποίηση όπου κάθε ενημέρωση στο map παίρνει O(N) θα οδηγούσε σε συνολική πολυπλοκότητα O(N^2).

Στο κεφάλαιο αυτό δεχόμαστε την επιβάρυνση log N που προκύπτει από τη έλλειψη πινάκων σε καθαρά συναρτησιακές γλώσσες όπως η Haskell, και προσπαθούμε να μελετήσουμε μια αποδοτική υλοποίηση με χρήση Maps όπου κάθε ενημέρωση θα κοστίζει log N. Όλα αυτά έχοντας ως δεδομένο ότι η σύγκριση αριθμών παίρνει σταθερό χρόνο, ωστόσο αν το N είναι πολύ μεγάλο η σύγκριση αριθμών κοστίζει όσο το πλήθος των ψηφίων τους δηλαδή στη περίπτωσή μας log N.

Θα αναλύσουμε το κεφάλαιο Trie ακολουθώντας την ανάλυση στο κεφάλαιο Trie στο Software Foundations [15]. Έτσι λοιπόν όπως παρουσιάζεται στο [15] θα ορίσουμε ένα τύπο δεδομένων για να αναπαραστήσουμε τους θετικούς διάδικους αριθμούς για να αποδείξουμε χρήσιμα θεωρήματα και ιδιότητες και να μελετήσουμε καλύτερα πως γίνεται η σύγκριση αριθμών. Με τον τύπο που θα ορίσουμε τους φυσικού αριθμούς η σύγκριση αριθμων θα είναι λογαριθμική ως προς τους αριθμούς όπως και στους δυαδικούς αριθμούς. Φυσικά θα μπορούσαμε να χρησιμοποιήσουμε τους built in φυσικούς αριθμούς της Haskell και τις συναρτήσεις σύγκρισης που παρέχονται.

```
{-@ LIQUID "--reflection"                    @-}
```

```
{-@ LIQUID "--ple"                                  @-}
{-@ LIQUID "--exactdc"                              @-}
{-@ LIQUID "--exact-data-con"                       @-}
{-@ LIQUID "--higherorder"                          @-}

{-@ infix : @-}
{-@ infix ++ @-}

module Trie where
import Prelude hiding(abs, (++), succ, lookup, pred, compare)
import Language.Haskell.Liquid.ProofCombinators
import Language.Haskell.Liquid.Bag
import Lists

data Positive = XH | XO Positive | XI Positive deriving (Eq)

{-@ measure plen @-}
{-@ plen:: Positive -> Nat @-}
plen:: Positive -> Int
plen XH = 0
plen (XI q) = 1 + plen q
plen (XO q) = 1 + plen q

--τύπος δεδομένων με τον οποίο αναπαριστούμε θετικούς φυσικούς
--αριθμούς με XH συμβολίζουμε τον αριθμό 1, με (XI n) περιττό αριθμό
--που συμβολίζει τον αριθμό 1+2n και (XO n) τον άρτιο αριθμό 2n
--Για παράδειγμα ο αριθμός 10 συμβολίζεται ως 0+2(1+2(0+2(1)))

{-@ data Positive [plen] = XH | XO Positive | XI Positive @-}

--συνάρτηση μετατροπής Positive αριθμού σε nat
{-@ reflect positive2nat @-}
{-@positive2nat :: Positive -> Nat @-}
positive2nat :: Positive -> Int
positive2nat (XI q) = 1 + 2*positive2nat q
positive2nat (XO q) = 2*positive2nat q
positive2nat  XH    = 1

--συνάρτηση που τυπώνει σε δυαδική μορφή ένα Positive αριθμό
{-@ reflect print_in_binary @-}
{-@ print_in_binary :: Positive -> [Nat] @-}
print_in_binary :: Positive -> [Int]
print_in_binary  XH = [1]
print_in_binary (XI q) = print_in_binary q ++ [1]
print_in_binary (XO q) = print_in_binary q ++ [0]

--H succ δέχεται ένα Positive αριθμό και επιστρέφει τον επόμενο
--Positive αριθμό
{-@reflect succ@-}
succ :: Positive -> Positive
succ  XH    = XO XH
succ (XO p) = (XI p)
succ (XI p) = (XO (succ p))
```

```
--Πρόσθεση με κρατούμενο δύο Poisitive αριθμών
{-@ reflect addc@-}
addc :: Bool -> Positive -> Positive -> Positive
addc False (XI p) (XI q) = XO (addc True p q)
addc False (XI p) (XO q) = XI (addc False p q)
addc False (XI p) (XH)   = XO (succ p)
addc False (XO p) (XI q) = XI (addc False p q)
addc False (XO p) (XO q) = XO (addc False p q)
addc False (XO p) (XH)   = XI p
addc False (XH)   (XI q) = XO (succ q)
addc False (XH)   (XO q) = XI q
addc False (XH)   (XH)   = XO XH
addc True  (XI p) (XI q) = XI (addc True p q)
addc True  (XI p) (XO q) = XO (addc True p q)
addc True  (XI p) (XH)   = XI (succ p)
addc True  (XO p) (XI q) = XO (addc True p q)
addc True  (XO p) (XO q) = XI (addc False p q)
addc True  (XO p) (XH)   = XO (succ p)
addc True  (XH)   (XI q) = XI (succ q)
addc True  (XH)   (XO q) = XO (succ q)
addc True  (XH)   (XH)   = XI XH

--Πρόσθεση δύο Positive αριθμών
{-@ reflect add@-}
add :: Positive -> Positive -> Positive
add p q = addc False p q
```

Τώρα που έχουμε ορίσει των τύπων των θετικών φυσικών αριθμών και διάφορες συναρτήσεις θα προχωρήσουμε στην απόδειξη ορισμένων θεωρημάτων:

```
 --Για κάθε Positive αριθμό ο φυσικός αριθμός που αντιστοιχεί στον
--(succ p) είναι ίσος με το φυσικό αριθμό που αντιστοιχεί στο p
--προσαυξημένος κατά ένα.

{-@lemma_succ_correct :: p:Positive ->
   {positive2nat (succ p) = positive2nat p + 1}
@-}
lemma_succ_correct :: Positive -> Proof
lemma_succ_correct XH = trivial
lemma_succ_correct (XO p) = trivial
lemma_succ_correct (XI p) = positive2nat (succ (XI p))
                          ==. positive2nat (XO (succ p))
                          ==. 2* positive2nat (succ p)
                          ==. 2 * (positive2nat p + 1) ?
lemma_succ_correct p
                          ==. (2* positive2nat p + 1 )+ 1
                          ==. positive2nat (XI p) + 1
                          *** QED


--Ο φυσικός αριθμός που αντιστοιχεί στο άθροισμα δύο Positive p,q με
--κρατούμενο c ισούται με το άθροισμα των φυσικών αριθμών που
--αντιστοιχούν στους p και q προσαυξημένο κατά ένα αν υπήρχε
```

```
--κρατούμενο
{-@lemma_addc_correct :: c: Bool -> p:Positive -> q: Positive
       -> {positive2nat (addc c p q) =  (if c then 1 else 0) +
                positive2nat p + positive2nat q}
@-}
lemma_addc_correct :: Bool -> Positive -> Positive -> Proof
lemma_addc_correct False (XI p) (XI q)
           =   positive2nat (addc False (XI p) (XI q))
           ==. positive2nat (XO (addc True p q))
           ==. 2*positive2nat  (addc True p q)
           ==. 2*(1+ positive2nat p + positive2nat q)
                      ? lemma_addc_correct True p q
           ==. (1+2*positive2nat p) + (1+2*positive2nat q)
           ==. positive2nat (XI p) + positive2nat (XI q)
           *** QED

lemma_addc_correct False (XI p) (XO q)
           =   positive2nat (addc False (XI p) (XO q))
           ==. positive2nat (XI (addc False p q))
           ==. 1+2*positive2nat ((addc False p q))
           ==. 1+2* (positive2nat p + positive2nat q)
                     ? lemma_addc_correct False p q
           ==. (1+2*positive2nat p) + 2*positive2nat q
           ==. positive2nat (XI p)  + positive2nat (XO q)
           *** QED

lemma_addc_correct False (XI p) (XH)
           =   positive2nat (addc False (XI p) (XH))
           ==. positive2nat ( XO (succ p) )
           ==. 2* positive2nat (succ p)
           ==. 2* (positive2nat p + 1)   ? lemma_succ_correct p
           ==. (1+2*positive2nat p) + 1
           ==. positive2nat (XI p) + positive2nat (XH)
           *** QED

lemma_addc_correct False (XO p) (XI q)
           =   positive2nat (addc False (XO p) (XI q))
           ==. positive2nat ( XI (addc False p q) )
           ==. 1+2*positive2nat (addc False p q)
           ==. 1+2* (positive2nat p + positive2nat q)
                     ? lemma_addc_correct False p q
           ==. (2*positive2nat p) + (1+2*positive2nat q)
           ==. positive2nat (XO p) + positive2nat (XI q)
           *** QED

lemma_addc_correct False (XO p) (XO q)
           =   positive2nat (addc False (XO p) (XO q))
           ==. positive2nat (XO (addc False p q))
           ==. 2*positive2nat (addc False p q)
           ==. 2*(positive2nat p + positive2nat q)
                   ? lemma_addc_correct False p q
           ==. 2*positive2nat p + 2*positive2nat q
           ==. positive2nat (XO p) + positive2nat (XO q)
           *** QED
```

```
lemma_addc_correct False (XO p) (XH)
          =   positive2nat (addc False (XO p) XH )
          ==. positive2nat (XI p)
          ==. 1+2*positive2nat p
          ==.positive2nat (XO p) + positive2nat (XH)
          *** QED

lemma_addc_correct False (XH)   (XI q)
          =   positive2nat (addc False XH (XI q))
          ==. positive2nat (XO (succ q))
          ==. 2*positive2nat (succ q)
          ==. 2*(1+positive2nat q)    ?lemma_succ_correct q
          ==. (1+2*positive2nat q) + 1
          ==. positive2nat (XI q) + positive2nat XH
          *** QED

lemma_addc_correct False (XH)   (XO q)
          = positive2nat (addc False XH (XO q))
          ==. positive2nat (XI q)
          ==. 1+2*positive2nat q
          ==.positive2nat (XO q) + positive2nat (XH)
          *** QED

lemma_addc_correct False (XH)   (XH)   = trivial

lemma_addc_correct True  (XI p) (XI q)
           =   positive2nat (addc True (XI p) (XI q))
           ==. positive2nat (XI (addc True p q))
           ==. 1+2*positive2nat (addc True p q)
           ==. 1+2*(1+ positive2nat p + positive2nat q)
                       ? lemma_addc_correct True p q
           ==. 1+(1+2*positive2nat p) + (1+2*positive2nat q)
           ==. 1+positive2nat (XI p) + positive2nat (XI q)
           *** QED

lemma_addc_correct True  (XI p) (XO q)
           =   positive2nat (addc True (XI p) (XO q))
           ==. positive2nat (XO (addc True p q))
           ==. 2*positive2nat ((addc True p q))
           ==. 2* (1+positive2nat p + positive2nat q)
                      ? lemma_addc_correct True p q
           ==. 1+ (1+2*positive2nat p) + 2*positive2nat q
           ==. 1+ positive2nat (XI p)  + positive2nat (XO q)
           *** QED

lemma_addc_correct True  (XI p) (XH)
           =   positive2nat (addc True (XI p) (XH))
           ==. positive2nat ( XI (succ p) )
           ==. 1+2* positive2nat (succ p)
           ==. 1+2* (positive2nat p + 1)
                   ? lemma_succ_correct p
           ==. 1+(1+2*positive2nat p) + 1
           ==. 1+positive2nat (XI p) + positive2nat (XH)
```

```
            *** QED

lemma_addc_correct True  (XO p) (XI q)
            =   positive2nat (addc True (XO p) (XI q))
            ==. positive2nat ( XO (addc True p q) )
            ==. 2*positive2nat (addc True p q)
            ==. 2* (1+ positive2nat p + positive2nat q)
                        ? lemma_addc_correct True p q
            ==. 1+(2*positive2nat p) + (1+2*positive2nat q)
            ==. 1+positive2nat (XO p) + positive2nat (XI q)
            *** QED

lemma_addc_correct True  (XO p) (XO q)
            =   positive2nat (addc True (XO p) (XO q))
            ==. positive2nat (XI (addc False p q))
            ==. 1+2*positive2nat (addc False p q)
            ==. 1+2*(positive2nat p + positive2nat q)
                      ? lemma_addc_correct False p q
            ==. 1+2*positive2nat p + 2*positive2nat q
            ==. 1+positive2nat (XO p) + positive2nat (XO q)
            *** QED

lemma_addc_correct True  (XO p) (XH)
            =   positive2nat (addc True (XO p) XH )
            ==. positive2nat (XO (succ p))
            ==. 2*positive2nat (succ p)
            ==. 2*(1+positive2nat p)   ? lemma_succ_correct p
            ==. 1+2*positive2nat p + 1
            ==. 1+positive2nat (XO p) + positive2nat (XH)
            *** QED

lemma_addc_correct True  (XH)   (XI q)
            =   positive2nat (addc True XH (XI q))
            ==. positive2nat (XI (succ q))
            ==. 1+2*positive2nat (succ q)
            ==. 1+2*(1+positive2nat q)    ?lemma_succ_correct q
            ==. 1+1+(1+2*positive2nat q)
            ==. 1+ positive2nat XH + positive2nat (XI q)
            *** QED

lemma_addc_correct True  (XH)   (XO q)
            =   positive2nat (addc True XH (XO q))
            ==. positive2nat ( XO (succ q) )
            ==. 2*positive2nat (succ q)
            ==. 2* (1 + positive2nat q)    ? lemma_succ_correct q
            ==. 1 + 1 + 2*positive2nat q
            ==. 1 +  positive2nat (XH) + positive2nat (XO q)
            *** QED

lemma_addc_correct True  (XH)   (XH)   = trivial

--Ο φυσικός αριθμός που αντιστοιχεί στο άθροισμα δύο Positive p,q
--ισούται με το άθροισμα των φυσικών αριθμών που αντιστοιχούν στους
--p και q
```

```
{-@ thm_add_correct :: p:Positive -> q:Positive
    -> {positive2nat (add p q) = positive2nat p + positive2nat q }
@-}
thm_add_correct :: Positive -> Positive -> Proof
thm_add_correct p q = [lemma_addc_correct False p q] *** QED
```

Θα συνεχίσουμε την ανάλυση των Positive αριθμών προσθέτοντας συναρτήσεις για σύγκριση Positive αριθμών και αντίστοιχα θεωρήματα με τις αποδείξεις τους.

```
--Ορισμός πιθανών αποτελεσμάτων σύγκρισης, Eq αν δύο αριθμοί είναι
--ίσοι, Gt αν ο πρώτος είναι μεγαλύτερος από το δεύτερο και Lt αν
--είναι αντίστοιχα μικρότερος.
data Comparison = Eq | Gt | Lt deriving (Eq)

--Συνάρτηση που υλοποιεί τη σύγκριση δύο Positive
{-@ reflect compare @-}
compare :: Positive -> Positive -> Comparison
compare (XI p) (XI q) = compare p q
compare (XI p) (XO q) = if (compare p q == Lt) then Lt else Gt
compare (XI p) (XH)   = Gt
compare (XO p) (XI q) = if (compare p q == Gt) then Gt else Lt
compare (XO p) (XO q) = compare p q
compare (XO p) (XH)   = Gt
compare (XH)   (XI q) = Lt
compare (XH)   (XO q) = Lt
compare (XH)   (XH)   = Eq

--Αν p Positive αριθμός τότε ο φυσικός αριθμός στον οποίο
--αντιστοιχεί είναι θετικός
{-@ lemma_positive2nat_pos :: p: Positive ->
    { (positive2nat p) > 0 }
@-}
lemma_positive2nat_pos :: Positive -> Proof
lemma_positive2nat_pos (XH)   =   trivial
lemma_positive2nat_pos (XI p) =   lemma_positive2nat_pos p
lemma_positive2nat_pos (XO p) =   lemma_positive2nat_pos p

--Αντιστοίχιση μεταξύ σύγκρισης δύο Positive αριθμών και των
--αντίστοιχων φυσικών αριθμών.
{-@ reflect comparison_tactic @-}
comparison_tactic :: Positive -> Positive -> Bool
comparison_tactic x y =
   if(compare x y == Lt) then positive2nat x <  positive2nat y
   else if (compare x y == Eq) then positive2nat x == positive2nat y
   else positive2nat x > positive2nat y

--Θεώρημα που αποδεικνύει ότι η παραπάνω τακτική αντιστοίχισης είναι
--ορθή
{-@ compare_correct :: x: Positive -> y: Positive ->
    { comparison_tactic x y }
@-}
```

```
compare_correct :: Positive -> Positive -> Proof
compare_correct (XI p) (XI q) = compare_correct p q
compare_correct (XI p) (XO q) = compare_correct p q
compare_correct (XI p) (XH)   = [lemma_positive2nat_pos p ] *** QED
compare_correct (XO p) (XI q) = compare_correct p q
compare_correct (XO p) (XO q) = compare_correct p q
compare_correct (XO p) (XH)   = [lemma_positive2nat_pos p ] *** QED
compare_correct (XH)   (XI q) = [lemma_positive2nat_pos q ] *** QED
compare_correct (XH)   (XO q) = [lemma_positive2nat_pos q ] *** QED
compare_correct (XH)   (XH)   = trivial
```

Με τα παραπάνω ολοκληρώνουμε τη σύγκριση Positive αριθμών. Στη συνέχεια θα αναφερθούμε σε δομές δεδομένων με τις οποίες μπορούμε να υλοποιήσουμε αποδοτικά Lookup Tables με Positive αριθμούς.

Τα δυαδικά δέντρα αναζήτησης είναι πολύ αποδοτικά, επειδή μπορούν να υλοποιήσουν Lookup Tables από οποιοδήποτε διατεταγμένο τύπο σε οποιοδήποτε άλλο τύπο. Αλλά όταν ο τύπος των κλειδιών είναι γνωστό ότι είναι (μικρού έως μέσου μεγέθους) ακέραιοι, τότε μπορούμε να χρησιμοποιήσουμε μια πιο εξειδικευμένη αναπαράσταση.

Κατ 'αναλογία, στις προστακτικές γλώσσες προγραμματισμού (C, Java, ML), όταν οι δείκτες ενός πίνακα είναι οι ακέραιοι σε ένα ορισμένο εύρος, μπορούμε να χρησιμοποιήσουμε πίνακες. Όταν τα κλειδιά όμως δεν είναι ακέραιοι, πρέπει να χρησιμοποιήσουμε κάτι σαν πίνακες κατακερματισμού ή δυαδικές δέντρα αναζήτησης.

Ένα Trie είναι ένα δέντρο στο οποίο οι ακμές είναι επισημασμένες με γράμματα από ένα αλφάβητο και για να αναζητήσουμε μια λέξη ακολουθούμε τις ακμές που επισημαίνονται με διαδοχικά γράμματα της λέξης. Στην πραγματικότητα, ένα trie είναι μια ειδική περίπτωση ενός Deterministic πεπερασμένου αυτοματισμού (DFA) που συμβαίνει να είναι ένα δέντρο παρά ένα γενικότερο γράφημα.

Ένα δυαδιυκό Trie είναι ένα trie στο οποίο το αλφάβητο είναι μόνο το {0,1}. Η "λέξη" είναι μια ακολουθία δυαδικών ψηφίων, δηλαδή ένας δυαδικός αριθμός. Για την αναζήτηση της "λέξης" 10001, χρησιμοποιούμε το 0 ως σήμα για να "κατευθυνθούμε αριστερά" και 1 ως σήμα για να " κατευθυνθούμε δεξιά". Οι δυαδικοί αριθμοί που θα χρησιμοποιήσουμε θα είναι Positive. Με δεδομένο ένα θετικό αριθμό όπως το δέκα, θα πάμε αριστερά προς τα δεξιά στους constructors xO / xI / (το οποίο είναι από το bit χαμηλής τάξης στο bit υψηλής τάξης), χρησιμοποιώντας το [xO] ως σήμα προς τα αριστερά, [ xI] ως σήμα προς τα δεξιά και [xH] ως σήμα για να σταματήσουμε:

```
--ορισμός δέντρου Trie, αντίστοιχος με τον ορισμό των BST
data Trie a = Leaf | Node (Trie a) a (Trie a)

{-@
data Trie [tlen] a = Leaf
  | Node { tLeft  :: Trie a
         , value  :: a
         , tRight :: Trie a }
@-}


{-@ measure tlen @-}
```

```
tlen :: (Trie a) -> Int
{-@ tlen :: (Trie a) -> Nat @-}
tlen (Leaf)        = 0
tlen (Node l v r) = 1 + (tlen l) + (tlen r)

--Ο τύπος Trie_table είναι μία τούπλα που στο πρώτο στοιχείο δέχεται
--ένα στοιχείο τύπου a, το οποίο χρησιμοποιεί ως default στοιχείο
--και σαν δεύτερο στοιχείο ένα Trie a.
type Trie_table a = (a, Trie a)


{-@ reflect t_empty @-}
t_empty def = (def, Leaf)

--συνάρτηση για αναζήτηση δυαδικής λέξης σε ένα Trie
{-@ reflect look@-}
look:: a -> Positive -> Trie a -> a
look def i (Leaf )      = def
look def (XH)   (Node l v r) =  v
look def (XO i) (Node l v r) =  look def i l
look def (XI i) (Node l v r) =  look def i r

--συνάρτηση αναζήτησης σε Trie_table
{-@ reflect lookup @-}
lookup :: Positive -> Trie_table a -> a
lookup i (def, trie) = look  def i  trie

--Εισαγωγή δυαδικής λέξης σε Trie
{-@ reflect ins @-}
ins :: a -> Positive -> a -> Trie a -> Trie a
ins def (XH)   a (Leaf)       =  Node Leaf a Leaf
ins def (XO i) a (Leaf)       =  Node (ins def i a Leaf) def Leaf
ins def (XI i) a (Leaf)       =  Node Leaf def (ins def i a Leaf)
ins def (XH)   a (Node l v r) =  Node l a r
ins def (XO i) a (Node l v r) =  Node (ins def i a l) v r
ins def (XI i) a (Node l v r) =  Node l v (ins def i a r)

--Εισαγωγή δυαδικής λέξης σε Trie_table
{-@ reflect insert @-}
insert :: Positive -> a -> Trie_table a -> Trie_table a
insert i a (def, trie) = (def, ins def i a trie)
```

Έχοντας ορίσει τα Trie trees, Trie_tables και τις αντίστοιχες συναρτήσεις αναζήτησης και εισαγωγής θα αποδείξουμε διάφορα θεωρήματα εγγυώνται την ορθότητα των Trie tables:

```
--Η αναζήτης σε κενό Trie επιστρέφει το πρώτο όρισμα-default στοιχείο
{-@ lemma_look_leaf :: x:a -> j: Positive -> {look x j Leaf = x} @-}
lemma_look_leaf :: a -> Positive -> Proof
lemma_look_leaf x j = trivial

--Αν εισάγουμε τη τιμή v και το Key k στο Trie t και στη συνέχεια
--αναζητήσουμε τη τιμή που αντιστοιχεί στο k θα πάρουμε το v
{-@ lemma_look_ins_same :: x:a -> k: Positive -> v:a -> t: Trie a ->
```

```
       { look x k (ins x k v t) = v }
@-}
lemma_look_ins_same :: a -> Positive  -> a -> Trie a -> Proof
lemma_look_ins_same x (XH)   v (Leaf)       =   trivial
lemma_look_ins_same x (XO i) v (Leaf)
      =   look x (XO i) (ins x (XO i) v Leaf)
      ==. look x (XO i) (Node (ins x i v Leaf) x Leaf)
      ==. look x i (ins x i v Leaf)
      ==. v ? lemma_look_ins_same x i v Leaf
      *** QED

lemma_look_ins_same x (XI i) v (Leaf)
      =   look x (XO i) (ins x (XI i) v Leaf)
      ==. look x (XO i) (Node Leaf x (ins x i v Leaf))
      ==. look x i (ins x i v Leaf)
      ==. v ? lemma_look_ins_same x i v Leaf
      *** QED

lemma_look_ins_same x (XH)   v (Node l o r) = trivial

lemma_look_ins_same x (XO i) v (Node l o r)
      = look x (XO i) (ins x (XO i) v (Node l o r) )
      ==. look x (XO i) (Node (ins x i v l) o r)
      ==.  look x i (ins x i v l)
      ==. v ? lemma_look_ins_same x i v l
      *** QED

lemma_look_ins_same x (XI i) v (Node l o r)
       = look x (XO i) (ins x (XI i) v (Node l o r) )
       ==. look x (XO i) (Node l o (ins x i v r))
       ==. look x i (ins x i v r)
       ==. v ? lemma_look_ins_same x i v r
       *** QED

--Η εισαγωγή ενός key k και value v σε ένα Trie t δεν επηρεάζει την
--αναζήτησης ενός key x,με x/=k, οπότε το να αναζητήσουμε το x μετά
--την εισαγωγή στο t είναι το ίδιο με το αναζητήσουμε στο t.
{-@ lemma_look_ins_other :: x:a -> k: Positive ->
   j: { Positive | j/= k } -> v:a -> t: Trie a ->
   { look x j (ins x k v t) = look x j t }
@-}
lemma_look_ins_other :: a -> Positive  -> Positive -> a -> Trie a ->
Proof
lemma_look_ins_other x k j v t    | k == j  =   trivial

lemma_look_ins_other x (XH)   (XO j) v (Leaf)
       =   look x (XO j) (ins x XH v Leaf)
       ==. look x (XO j) (Node Leaf v Leaf)
       ==. look x j Leaf
       *** QED

lemma_look_ins_other x (XH)   (XI j) v (Leaf)
       =   look x (XO j) (ins x XH v Leaf)
       ==. look x (XO j) (Node Leaf v Leaf)
```

```
       ==. look x j Leaf
       *** QED

lemma_look_ins_other x (XO i) (XH)   v (Leaf)
       =   look x XH (ins x (XO i) v Leaf)
       ==. look x XH (Node (ins x i v Leaf) x Leaf)
       ==. x
       *** QED

lemma_look_ins_other x (XI i) (XH)   v (Leaf)
       = look x XH (ins x (XI i) v Leaf)
       ==. look x XH (Node Leaf x (ins x i v Leaf))
       ==. x
       *** QED

lemma_look_ins_other x (XO i) (XO j) v (Leaf)
       =   look x (XO j) (ins x (XO i) v Leaf)
       ==. look x (XO j) (Node (ins x i v Leaf) x Leaf)
       ==. look x  j (ins x i v Leaf)
       ==. look x j Leaf   ? lemma_look_ins_other x i j v Leaf
       ==. look x (XO j) Leaf
       *** QED

lemma_look_ins_other x (XI i) (XO j) v (Leaf)
       =   look x (XO j) (ins x (XI i) v Leaf)
       ==. look x (XO j) (Node Leaf x (ins x i v Leaf))
       ==. look x  j Leaf
       ==. look x (XO j) Leaf
       *** QED

lemma_look_ins_other x (XO i) (XI j) v (Leaf)
       =   look x (XI j) (ins x (XO i) v Leaf)
       ==. look x (XI j) (Node (ins x i v Leaf) x Leaf)
       ==. look x  j Leaf
       ==. look x (XO j) Leaf
       *** QED

lemma_look_ins_other x (XI i) (XI j) v (Leaf)
       =   look x (XI j) (ins x (XI i) v Leaf)
       ==. look x (XI j) (Node Leaf x (ins x i v Leaf))
       ==. look x j (ins x i v Leaf)
       ==. look x j Leaf    ? lemma_look_ins_other x i j v Leaf
       ==. look x (XI j) Leaf
      *** QED

lemma_look_ins_other x (XH)   (XO j) v (Node l o r)
       =   look x (XO j) (ins x (XH) v (Node l o r))
       ==. look x (XO j) (Node l v r)
       ==. look x (XO j) (Node l o r)
       *** QED

lemma_look_ins_other x (XH)   (XI j) v (Node l o r)
       =   look x (XI j) (ins x (XH) v (Node l o r))
       ==. look x (XI j) (Node l v r)
```

```
        ==. look x (XI j) (Node l o r)
        *** QED

lemma_look_ins_other x (XO i) (XH)   v (Node l o r)
        =   look x XH (ins x (XO i) v (Node l o r))
        ==. look x XH (Node (ins x i v l) o r)
        ==. o
        *** QED

lemma_look_ins_other x (XI i) (XH)   v (Node l o r)
        =   look x XH (ins x (XI i) v (Node l o r))
        ==. look x XH (Node l o (ins x i v r))
        ==. o
        *** QED

lemma_look_ins_other x (XO i) (XO j) v (Node l o r)
        =   look x (XO j) (ins x (XO i) v (Node l o r))
        ==. look x (XO j) (Node (ins x i v l) o r)
        ==. look x j (ins x i v l)
        ==. look x j l    ? lemma_look_ins_other x i j v l
        ==. look x (XO j) (Node l o r)
        *** QED

lemma_look_ins_other x (XI i) (XO j) v (Node l o r)
        =   look x (XO j) (ins x (XI i) v (Node l o r))
        ==. look x (XO j) (Node l o (ins x i v r))
        ==. look x j l
        ==. look x (XO j) (Node l o r)
        *** QED

lemma_look_ins_other x (XO i) (XI j) v (Node l o r)
        =   look x (XI j) (ins x (XO i) v (Node l o r))
        ==. look x (XI j) (Node (ins x i v l) o r)
        ==. look x j r
        ==. look x (XI j) (Node l o r)
        *** QED

lemma_look_ins_other x (XI i) (XI j) v (Node l o r)
        =   look x (XI j) (ins x (XI i) v (Node l o r))
        ==. look x (XI j) (Node l o (ins x i v r))
        ==. look x j (ins x i v r)
        ==. look x j r    ? lemma_look_ins_other x i j v r
        ==. look x (XI j) (Node l o r)
        *** QED
```

Στη συνέχεια θα αποδείξουμε ότι τα Tries είναι ισοδύναμα με τα Maps και άρα αποτελούν μια σωστή υλοποίηση του γενικού αφηρημένου τύπου Lookup Tables. Για να συνδυάσουμε τα Trie όπου κάνουμε αναζήτηση σε Positives με τα Maps σε φυσικούς αριθμούς θα ήταν χρήσιμο να χρησιμοποιήσουμε μία απεικόνιση από Positives σε φυσικούς και το αντίστροφο. Θα συσχετίσουμε τον αριθμό 1 Positive (δηλ XH) στο 0 τον φυσικών, το 2 Positive στο 1 των φυσικών κλπ…

```
{-@reflect of_succ_nat @-}
```

```
{-@ of_succ_nat :: Nat -> Positive @-}
of_succ_nat :: Int -> Positive
of_succ_nat 0 = XH
of_succ_nat n = succ (of_succ_nat (n-1))

--αντιστοίχιση από φυσικό αριθμό σε Positive
{-@reflect nat2pos@-}
{-@nat2pos :: Nat -> Positive @-}
nat2pos :: Int -> Positive
nat2pos n = of_succ_nat n


{-@ reflect to_nat @-}
{-@ to_nat:: Positive -> Nat -> Nat @-}
to_nat :: Positive -> Int -> Int
to_nat (XH)   a  = a
to_nat (XO p) a  = to_nat p (2*a)
to_nat (XI p) a  = a + to_nat p (2*a)


{-@reflect pred @-}
{-@ pred :: Nat -> Nat @-}
pred :: Int -> Int
pred 0 = 0
pred 1 = 0
pred n = n-1


-- συνάρτηση που δέχεται ένα positive και τον αντιστοιχίζει σε
--φυσικό αριθμό
{-@ reflect pos2nat @-}
{-@ pos2nat :: Positive -> Nat @-}
pos2nat :: Positive -> Int
pos2nat n = pred (positive2nat n)

--Για κάθε φυσικό αριθμό n αν τον αντιστοιχίσουμε σε positive και
--τον positive σε φυσικό, θα πάρουμε πίσω το n
{-@ lemma_nat2pos2nat :: n: Nat -> {pos2nat (nat2pos n) = n} @-}
lemma_nat2pos2nat :: Int -> Proof
lemma_nat2pos2nat 0 = trivial
lemma_nat2pos2nat n = pos2nat (nat2pos n)
                ==. pos2nat (of_succ_nat n)
                ==. pos2nat ( succ (of_succ_nat (n-1)) )
                ==. pred (positive2nat (succ (of_succ_nat (n-1))) )
                ==. pred (positive2nat (of_succ_nat (n-1)) +1)
                        ? lemma_succ_correct (of_succ_nat (n-1))
                ==. positive2nat (of_succ_nat (n-1))
                ==. pred (positive2nat (of_succ_nat (n-1))) +1
                ==. (n-1) +1     ? lemma_nat2pos2nat (n-1)
                ==. n
                *** QED

--ομοίως με το παραπάνω λήμμα αλλά για positive
{-@ lemma_pos2nat2pos:: p: Positive -> {nat2pos (pos2nat p) = p} @-}
```

```
lemma_pos2nat2pos :: Positive -> Proof
lemma_pos2nat2pos p = undefined
lemma_pos2nat2pos (XH)   = trivial
lemma_pos2nat2pos (XO p)  = lemma_pos2nat2pos p
lemma_pos2nat2pos (XI p)  = lemma_pos2nat2pos p

--αν οι positive p,q αντιστοιχίζονται στον ίδιο φυσικό τότε p=q
{-@ lemma_pos2nat_injective :: p: Positive ->
     q: {Positive | pos2nat p = pos2nat q} -> {p=q}
@-}
lemma_pos2nat_injective ::  Positive -> Positive -> Proof
lemma_pos2nat_injective p q = [lemma_pos2nat2pos p,
lemma_pos2nat2pos q] *** QED

--αν οι φυσικοί i,j αντιστοιχίζονται στον ίδιο positive τότε i=j
{-@ lemma_nat2pos_injective :: i: Nat -> j:
     {Nat | nat2pos i = nat2pos j} -> {i=j}
@-}
lemma_nat2pos_injective ::  Int -> Int -> Proof
lemma_nat2pos_injective i j = [lemma_nat2pos2nat i,
lemma_nat2pos2nat j] *** QED

--ιδιότητα για ελέγξουμε αν ένα δέντρο είναι Trie
{-@reflect is_Trie@-}
is_Trie :: Trie_table a -> Bool
is_Trie (def, Leaf)         = True
is_Trie (def, (Node l v r)) = is_Trie (def,l) && is_Trie (def,r)


{-@ theorem_empty_is_trie :: def:a -> {is_Trie (t_empty def) } @-}
theorem_empty_is_trie :: a -> Proof
theorem_empty_is_trie def = trivial
```

--Με το παρακάτω θεώρημα αποδεικνύουμε ότι αν t είναι Trie τότε και η insert
--επιστρέφει Trie

```
{-@ theorem_insert_is_trie :: i:Positive -> x: a ->
    t: {Trie_table a | is_Trie t} ->
    {is_Trie (insert i x t) }
@-}
theorem_insert_is_trie :: Positive -> a -> Trie_table a -> Proof
theorem_insert_is_trie XH x (def, Leaf)          =  trivial
theorem_insert_is_trie XH x (def, Node l v r)    =  trivial

theorem_insert_is_trie (XO p) x (def, Leaf)
          =  is_Trie (insert (XO p) x (def, Leaf))
          ==. is_Trie (def, ins def (XO p) x Leaf)
          ==. is_Trie (def, Node (ins def p x Leaf) def Leaf)
          ==. True ? theorem_insert_is_trie p x (def, Leaf)
          *** QED

theorem_insert_is_trie (XI p) x (def, Leaf)
          =   is_Trie (insert (XI p) x (def, Leaf))
          ==. is_Trie (def, ins def (XI p) x Leaf)
```

```
            ==. is_Trie (def, Node Leaf def (ins def p x Leaf))
            ==. True ? theorem_insert_is_trie p x (def, Leaf)
            *** QED

theorem_insert_is_trie (XO p) x (def, Node l v r)
            =  is_Trie (insert (XO p) x (def, Node l v r))
            ==. is_Trie (def, ins def (XO p) x (Node l v r))
            ==. is_Trie (def, Node (ins def p x l) v r)
            ==. True ? theorem_insert_is_trie p x (def, l)
           *** QED

theorem_insert_is_trie (XI p) x (def, Node l v r)
            =  is_Trie (insert (XI p) x (def, Node l v r))
            ==. is_Trie (def, ins def (XI p) x (Node l v r))
            ==. is_Trie (def, Node l v (ins def p x r))
            ==. True ? theorem_insert_is_trie p x (def, r)
            *** QED
```

Τώρα θα προχωρήσουμε στην ισοδυναμία των Trie_table με τα Maps. Όπως έχουμε αναφέρει ξανά δεν έχουμε μεταφράσει τα Maps σε Liquid Haskell οπότε θα χρησιμοποιήσουμε την υλοποίηση που παρέχεται στο [16] την οποία παρουσιάζουμε στη συνέχεια:

```
 --Αναπαριστούμε τα TM ως τούπλα από ένα default στοιχείο και μία λίστα
--από Keys,Values)
data TotalMap a = TM
  { tmDef  :: a
  , tmVals :: [(Int, a)]
  }

{-@ reflect t_empty_map @-}
tm_empty_map :: a -> TotalMap a
tm_empty_map v = TM v []

{-@ reflect t_update @-}
t_update :: TotalMap a -> Int -> a -> TotalMap a
t_update (TM d kvs) k v = TM d ((k, v) : kvs)


{-@ reflect t_read @-}
t_read :: TotalMap a -> Int -> a
t_read (TM d kvs) key = lookupDefault d key kvs


{-@ reflect lookupDefault @-}
lookupDefault :: (Eq k) => v -> k -> [(k, v)] -> v
lookupDefault d key ((k,v) : kvs)
  | k == key                      = v
  | otherwise                     = lookupDefault d key kvs
lookupDefault d _ [] = d


{-@ t_update_eq :: m:TotalMap a -> v:a -> x:Nat ->
```

```
      { t_read (t_update m x v) x == v }
@-}
t_update_eq :: TotalMap a -> a -> Int -> Proof
t_update_eq m v x = trivial


{-@ t_update_neq :: m:TotalMap a -> v:a -> x1:Nat ->
           x2:{Nat | x1 /= x2} ->
           { t_read (t_update m x1 v) x2 == t_read m x2 }
@-}
t_update_neq :: TotalMap a -> a -> Int -> Int -> Proof
t_update_neq m v x1 x2 = trivial
```

Έχοντας ορίσει τα Total Maps ορίζουμε τη συνάρτηση που εκφράζει ότι ένα Trie table είναι ισοδύναμο με ένα Total Map:

```
{-@ reflect abstract @-}
{-@ abstract ::  Trie_table a -> Nat -> a @-}
abstract ::  Trie_table a -> Int -> a
abstract t n = lookup (nat2pos n) t

{-@ reflect abs @-}
{-@ abs :: (Eq a) => Trie_table a -> TotalMap a -> Nat -> Bool @-}
abs :: (Eq a) => Trie_table a -> TotalMap a -> Int -> Bool
abs t m n = abstract t n == t_read m n

--Αξίωμα για ισότητα συναρτήσεων, αν αποδείξουμε ότι f x == g x για
--κάθε x τότε f == g
{-@ assume ext_axiom :: f:(a -> b) -> g:(a -> b) ->
     pf:(x:a -> {f x == g x}) -> {f == g}
@-}
ext_axiom :: (a -> b) -> (a -> b) -> (a -> pf) -> Proof
ext_axiom f g pf = undefined

{-@ theorem_empty_relate_aux :: (Eq a) => def:a -> n: Nat ->
    {(abstract (t_empty def) n) == ( t_read (t_empty_map def) n)}
@-}
theorem_empty_relate_aux ::  (Eq a) => a -> Int -> Proof
theorem_empty_relate_aux def n = trivial


{-@ theorem_empty_relate :: (Eq a) => def:a ->
    {  (abstract (t_empty def)) == ( t_read (t_empty_map def))}
@-}
theorem_empty_relate ::  (Eq a) => a  -> Proof
theorem_empty_relate def = ext_axiom  (abstract (t_empty def)) (
t_read (TM def [])) (theorem_empty_relate_aux def)

--Το παρακάτω θεώρημα εκφράζει ότι η αναζήτηση ενός positive σε ένα
--Trie_table θα μας δώσει το ίδιο αποτέλεσμα με το να αναζητήσουμε
--τον αντίστοιχο φυσικό αριθμό σε ένα ισοδύναμο Map
{-@ theorem_lookup_relate :: (Eq a) => i:Positive ->
     t: {Trie_table a | is_Trie t} ->
```

```
     m: {TotalMap a | abs t m (pos2nat i)} ->
     { lookup i t == t_read m (pos2nat i) }
@-}
theorem_lookup_relate
  ::  (Eq a) => Positive ->  Trie_table a -> TotalMap a -> Proof
theorem_lookup_relate i t m
   = (lemma_pos2nat2pos i, abs t m (pos2nat i))*** QED


--Αν ένα Trie Table είναι ισοδύναμο με ένα Map τότε εισάγοντας ένα
--(key, value) και στα δύο, παραμένουν ισοδύναμα
{-@ theorem_insert_relate :: (Eq a) =>
       k:Positive -> n:Positive ->  v:a ->
       t: {Trie_table a | is_Trie t} ->
       m: {TotalMap a | abs t m (pos2nat n)} ->
      { abs (insert k v t) (t_update m (pos2nat k) v) (pos2nat n) }
@-}
theorem_insert_relate ::
  (Eq a) => Positive ->  Positive -> a -> Trie_table a -> TotalMap a
-> Proof
theorem_insert_relate k n v (def,t) m

 | (pos2nat k) == (pos2nat n)
  = abs (insert k v (def,t)) (t_update m (pos2nat k) v) (pos2nat n)
  ==. abstract (insert k v (def,t)) (pos2nat n) == t_read (t_update
         m (pos2nat k) v) (pos2nat n)
  ==. abstract (insert k v (def,t)) (pos2nat k) == t_read (t_update
         m (pos2nat k) v) (pos2nat k)
  ==. lookup (nat2pos  (pos2nat k)) (insert k v (def,t)) == t_read
        (t_update m (pos2nat k) v)  (pos2nat k)
  ==. lookup k (insert k v (def,t)) == t_read (t_update m (pos2nat
         k) v)  (pos2nat k)  ? lemma_pos2nat2pos k
  ==. lookup k (def, ins def k v t) == t_read (t_update m (pos2nat
         k) v)  (pos2nat k)
  ==.look def k (ins def k v t) == t_read (t_update m (pos2nat k) v)
         (pos2nat k)
  ==. v  == v
     ? lemma_look_ins_same def k v t &&& t_update_eq m v (pos2nat k)
  ==. True
  *** QED

 | (pos2nat k) /= (pos2nat n)
   = abs (insert k v (def,t)) (t_update m (pos2nat k) v) (pos2nat n)
     ==. abstract (insert k v (def,t)) (pos2nat n) == t_read
            (t_update m (pos2nat k) v) (pos2nat n)
     ==. lookup (nat2pos  (pos2nat n)) (insert k v (def,t)) ==
            t_read (t_update m (pos2nat k) v)  (pos2nat n)
     ==. lookup n (insert k v (def,t)) == t_read (t_update m
            (pos2nat k) v)  (pos2nat n)  ? lemma_pos2nat2pos n
     ==. lookup n (def, ins def k v t) == t_read (t_update m
            (pos2nat k) v)  (pos2nat n)
     ==. look def n (ins def k v t) == t_read (t_update m (pos2nat
            k) v)  (pos2nat n)
     ==. look def n t == t_read m (pos2nat n)
```

```
      ? lemma_look_ins_other def k n v t
      &&& t_update_neq m v (pos2nat k) (pos2nat n)
==. lookup n (def,t) == t_read m (pos2nat n)
==. lookup (nat2pos(pos2nat n)) (def,t) == t_read m (pos2nat n)
      ? lemma_pos2nat2pos n
==. abstract (def,t) (pos2nat n) ==  t_read m (pos2nat n)
==. abs (def,t)  m (pos2nat n)
==. True
*** QED
```

Με τα παραπάνω θεωρήματα αποδείξαμε την ισοδυναμία Trie_tables και Total Maps. Συνεπώς τα Trie Tables αποτελούν μια σωστή υλοποίηση των lookup Tables. Σαν αποτέλεσμα έχοντας αναφέρει παραπάνω ότι τα Trie Tables είναι μια πολύ αποδοτική υλοποίηση, χρησιμοποιώντας τα θα μπορούσαμε να υλοποιήσουμε το αντιστοιχο πρόγραμμα που αναφέραμε στην αρχή του κεφαλαίου για εύρεση φυσικών αριθμών που υπάρχουν τουλάχιστον 2 φορέ στο διάστημα [0, 2N] με Trie Tables και να πετύχουμε πολυπλοκότητα O(NlogN) και για την ακρίβεια αναφέραμε ότι και η σύγκριση αριθμών κοστίζει logN άρα O(N logN  log N). Τέλος τα Trie Tables είναι μια εξαιρετικά αποδοτική λύση για συναρτησιακές γλώσσες για προβλήματα που απαιτούν χρήση lookup tables.

## 4.10 Binomial Queues

Στο παρόν κεφάλαιο θα μελετήσουμε τη δομή δεδομένων Binomial Queue όπως παρουσιάζεται στο Software Foundations [17]. Θα διατυπώσουμε θεωρήματα για τις διάφορες συναρτήσεις χειρισμού των Binomial Queues και στη συνέχεια θα τις επαληθευσουμε με τη Liquid Haskell.

Αρχικά ορίζουμε κάποιους τύπους δεδομένων που έχουμε ξανασυναντήσει και σε προηγούμενα κεφάλαια όπως Pair, Tree κλπ…

```
{-@ LIQUID "--reflection"                          @-}
{-@ LIQUID "--ple"                                 @-}
{-@ LIQUID "--exactdc"                             @-}
{-@ LIQUID "--exact-data-con"                      @-}
{-@ LIQUID "--higherorder"                         @-}
{-@ LIQUID "--noadt"                               @-}

{-@ infix : @-}
{-@ infix ++ @-}

module Binom where
import Prelude hiding(abs, (++), succ, lookup, pred, compare, unzip,
Maybe(..))
import Language.Haskell.Liquid.ProofCombinators
import Lists
import Language.Haskell.Liquid.Bag
import Permutations

data Maybe a = Just a | Nothing  deriving (Eq)
```

```
{-@ measure isJust @-}
isJust :: Maybe a -> Bool
isJust (Just _) = True
isJust Nothing  = False

{-@ measure fromJust @-}
{-@ fromJust :: {v:Maybe a | isJust v } -> a @-}
fromJust :: Maybe a -> a
fromJust (Just x) = x


{-@ type Key = Nat@-}
type Key = Int

data Pair a b = P a b deriving (Eq, Ord)

--συνάρτηση που επιστρέφει το πρώτο στοιχείο ενός Pair
{-@ reflect snd1 @-}
snd1 :: (Pair a b) -> b
snd1 (P a b) = b

--συνάρτηση που επιστρέφει το πρώτο στοιχείο ενός Pair
{-@ reflect fst1 @-}
fst1 :: (Pair a b) -> a
fst1 (P a b) = a

data Tree = Leaf | Node Key Tree Tree deriving (Eq, Ord)

{-@
 data Tree [tlen] = Leaf
   | Node { key    :: Key
          , tLeft  :: Tree
          , tRight :: Tree }
@-}


{-@ measure tlen @-}
{-@ tlen :: Tree -> Nat @-}
tlen :: Tree  -> Int
tlen (Leaf)        = 0
tlen (Node key l r) = 1 + (tlen l) + (tlen r)
```

Τώρα θα ορίσουμε το τύπο δεδομένων queue ως μία λίστα από δυαδικά δέντρα:

```
{-@ type Priqueue = [Tree] @-}
type Priqueue = [Tree]

{-@ reflect p_empty @-}
p_empty :: [Tree]
p_empty = []
```

Θα ορίσουμε επίσης συναρτήσεις για τη παραπάνω δομή δεδομένων:

```
--Η συνάρτηση smash δέχεται δύο δέντρα και τα συμπιέζει σε ένα
{-@ reflect smash @-}
smash :: Tree -> Tree -> Tree
smash (Node x t1 Leaf) (Node y u1 Leaf)
      = if (x>y) then Node x (Node y u1 t1) Leaf
        else Node y (Node x t1 u1) Leaf
smash _ _ = Leaf

--Η συνάρτηση carry δέχεται μία λίστα από δέντρα και ένα δέντρο και
--το εισάγει στη κατάλληλη θέση στη λίστα
{-@ reflect carry @-}
carry :: [Tree] -> Tree -> [Tree]
carry []        Leaf = []
carry []        t    = [t]
carry (Leaf:q') t    = (t:q')
carry (u:q')    Leaf = (u:q')
carry (u:q')    t    = Leaf:carry q' (smash t u)

--η συνάρτηση insert εισάγει ένα φυσικό αριθμό σε μια Priority Queue
--καλώντας την carry.
{-@ reflect insert @-}
{-@ insert :: Nat -> Priqueue -> Priqueue @-}
insert :: Key -> Priqueue -> Priqueue
insert x q = carry q (Node x Leaf Leaf)

--η συνάρτηση join συνενώνει δύο priority queue και ένα δέντρο
{-@ reflect join @-}
join :: Priqueue -> Priqueue -> Tree -> Priqueue
join []       q        c            = carry q c
join p        []       c            = carry p c
join (Leaf:p) (Leaf:q) c            = c    : join p q Leaf
join (Leaf:p) (q1:q)   (Leaf)       = q1   : join p q Leaf
join (Leaf:p) (q1:q)   (Node k l r) = Leaf : join p q (smash (Node k
l r) q1)
join (p1:p)   (Leaf:q) (Leaf)       = p1   : join p q Leaf
join (p1:p)   (Leaf:q) (Node k l r) = Leaf : join p q (smash (Node k
l r) p1)
join (p1:p)   (q1:q)   c            = c    : join p q (smash p1 q1)

--η unzip δέχεται ένα δέντρο και το αποσυμπιέζει σε μικρότερα δέντρα
--κόβοντας πάντα το δεξι υποδέντρο μέχρι αυτό να γίνει Leaf
{-@ reflect unzip@-}
unzip:: Tree -> Priqueue -> Priqueue
unzip Leaf           cont = cont
unzip (Node x t1 t2) cont = unzip t2 ( (Node x t1 Leaf) : cont )


{-@ reflect heap_delete_max @-}
heap_delete_max :: Tree -> Priqueue
heap_delete_max (Node x t1 Leaf) = unzip t1 []
heap_delete_max  _               = []

{-@ reflect find_max'@-}
```

```
{-@find_max' :: Priqueue -> Key -> Nat @-}
find_max' :: Priqueue -> Key -> Key
find_max' []               current  = current
find_max' (Leaf:q)         current  = find_max' q current
find_max' ((Node x _ _):q) current  =
    if(x>current) then find_max' q x else find_max' q current

--Η findmax βρίσκει το μέγιστο στοιχείο-αριθμό σε μία priority
--queue χρησιμοποιώντας τη findmax' ως βοηθητική συνάρτηση
{-@ reflect find_max @-}
{-@ find_max :: p:Priqueue -> Maybe Nat  @-}
find_max :: Priqueue -> Maybe Key
find_max []               = Nothing
find_max (Leaf:q)         = find_max q
find_max ((Node x _ _):q) = Just (find_max' q x)


--Η findmax' δέχεται μια Priority Queue και ένα στοιχείο προς
--αφαίρεση και διατρέχει την Priority Queue μεχρι να το βρει και
--επιστρέφει δύο Queues, η πρώτη έχει τα στοιχεία που προηγούνται
--του στοιχείου που αφαίρεσε και η δευτερη τα υπόλοιπα

{-@ reflect delete_max_aux@-}
delete_max_aux :: Priqueue -> Key -> (Pair Priqueue Priqueue)
delete_max_aux (Leaf: p) m =
  (P (Leaf:(fst1 (delete_max_aux p m))) (snd1 (delete_max_aux p m)))
delete_max_aux ((Node x t1 Leaf):p) m =
   if(m>x) then
    (P ((Node x t1 Leaf):(fst1 (delete_max_aux p m))) (snd1
         (delete_max_aux p m)))
   else (P (Leaf:p) (heap_delete_max (Node x t1 Leaf)))
delete_max_aux  _  m = (P [] [])


--Η delete_max βρίσκει το μέγιστο στοιχείο σε μια Priority Queue και
--επιστρέφει το στοιχείο αυτό και την υπόλοιπη Queue που απομένει

{-@ reflect delete_max @-}
delete_max :: Priqueue -> Maybe (Pair Key Priqueue)
delete_max q =
  if(isJust (find_max q))
  then Just (P (fromJust (find_max q)) (join (fst1 (delete_max_aux q
        (fromJust (find_max q))))  (snd1 (delete_max_aux q (fromJust
        (find_max q))))  Leaf))
  else Nothing


--Η merge δέχεται δύο priority Queues και τις συνενώνει
--χρησιμοποιώντας τη join που έχει οριστεί προηγουμένως
{-@ reflect merge @-}
merge:: Priqueue -> Priqueue -> Priqueue
merge p q = join p q Leaf
```

```
--Η pow2heap ελέγχει αν ένα δέντρο είναι της μορφής (Node m t Leaf)
--και αν έχει την ιδιότητα σορού, δηλαδή το στοιχείο στη ρίζα είναι
--το μέγιστο και το t έχει ύψος n
{-@ reflect pow2heap @-}
{-@ pow2heap :: Nat -> Tree -> Bool @-}
pow2heap :: Int -> Tree -> Bool
pow2heap n (Node m t1 Leaf) = pow2heap' n m t1
pow2heap n _                = False


--Βοηθητική συνάρτηση για την pow2heap
{-@ reflect pow2heap' @-}
{-@ pow2heap':: Nat -> Key -> Tree -> Bool @-}
pow2heap':: Int -> Key -> Tree -> Bool
pow2heap' 0 m Leaf         = True
pow2heap' 0 m (Node _ _ _) = False
pow2heap' n m Leaf         = False
pow2heap' n m (Node k l r) = (m >= k) && pow2heap' (n-1) k l &&
pow2heap' (n-1) m r


--Η συνάρτηση priq’ δέχεται ένα φυσικό αριθμό n και μια priority
--queue εκφράζει την ιδιότητα ότι το πρώτο στοιχείο της queue είναι
--είτε Leaf είτε δέντρο που ικανοποιεί την pow2heap με παράμετρο n
--(δηλαδή έχει ύψος n), και συνεχίζει ανδρομικά στο επόμενο στοιχείο
--της λίστας με παράμετρο n+1

{-@ reflect priq'@-}
{-@ priq' ::  [Tree] -> Nat -> Bool @-}
priq' :: [Tree] -> Int -> Bool
priq' (t:l) i = (t==Leaf || pow2heap i t) && priq' l (i+1)
priq' _     i = True

--Η priq εκφράζει την ιδιότητα ότι μια Priority Queue είναι έγκυρη
--δηλαδή ικανοποιεί τις συνθήκες της priq’ για n = 0
{-@ reflect priq@-}
{-@ priq ::  Priqueue ->  Bool @-}
priq :: Priqueue -> Bool
priq q = priq' q 0
```

Από τα παραπάνω βλέπουμε ότι μια ουρά προτεραιότητας (χρησιμοποιώντας τη δομή δεδομένων binomial queue) είναι μια λίστα δέντρων. Το i-οστό στοιχείο της λίστας είναι είτε Leaf ή είναι ένα power-of-2-σωρός με ακριβώς 2 ^ i κόμβους.

Έχοντας γράψει τις κατάλληλες συναρτήσεις που τροποποιούν μια Queue και συναρτήσεις που ελέγχουν ιδιότητες που πρέπει να πληροί μια Queue για να είναι έγκυρη (πχ priq) θα προχωρήσουμε στην απόδειξη ορισμένων θεωρημάτων:

```
--η empty Queue είναι έγκυρη priority Queue
{-@ thm_empty_priq :: {priq p_empty} @-}
thm_empty_priq :: Proof
```

```
thm_empty_priq = trivial


--Αν για δύο δέντρα t,u ισχύει pow2heap n t και pow2heap n u τότε
--και για το smash t u (δηλαδή η συμπίεση των t,u) ισχύει ότι
--pow2heap (n+1) (smash t u)

{-@ thm_smash_valid :: n: Nat -> t:{Tree | pow2heap n t } -> u:{Tree
| pow2heap n u } -> { pow2heap (n+1) (smash t u) } @-}
thm_smash_valid :: Int -> Tree -> Tree -> Proof
thm_smash_valid n t u = trivial


--η εισαγωγή ενός δέντρου t για το οποίο ισχύει pow2heap n t σε μία
--queue για την οποία ισχύει priq' q n δίνει επίσης queue για την
--οποία ισχύει priq (carry q t) n

{-@ thm_carry_valid :: n: Nat -> q: {Priqueue | priq' q n}->
       t: {Tree | t == Leaf || pow2heap n t} ->
       { priq' (carry q t) n}
@-}
thm_carry_valid :: Int ->  Priqueue -> Tree -> Proof
thm_carry_valid n []     Leaf = trivial
thm_carry_valid n (q:qs) Leaf = trivial
thm_carry_valid n []     t    = trivial
thm_carry_valid n (q:qs) t
          =   priq' (carry (q:qs) t) n
          ==. priq' (Leaf:carry qs (smash t q)) n
          ==. ((Leaf==Leaf || pow2heap n Leaf)
               && priq' (carry qs (smash t q)) (n+1))
          ==. True    ? thm_carry_valid (n+1) qs (smash t q)
          *** QED

--Ομοίως για την insert που εισάγει στοιχείο σε queue καλώντας την
--carry, οπότε χρησιμοποιούμε το προηγούμενο θεώρημα
{-@ thm_insert_priq :: x: Key -> q: {Priqueue | priq q} ->
    { priq (insert x q) }
@-}
thm_insert_priq :: Key -> Priqueue -> Proof
thm_insert_priq x q =
    (pow2heap 0 (Node x Leaf Leaf) ,
     thm_carry_valid 0 q (Node x Leaf Leaf)) *** QED


{-@ thm_join_valid :: n:Nat -> p :{Priqueue | priq' p n} ->
     q:{Priqueue | priq' q n} ->
     c: {Tree | (c==Leaf || pow2heap n c)}
    -> {priq'(join p q c) n}
@-}
thm_join_valid :: Int -> Priqueue -> Priqueue -> Tree -> Proof
thm_join_valid n [] q c
      =   priq' (join [] q c) n
      ==. priq' (carry q c) n
      ==. True     ? thm_carry_valid n q c
```

```
      *** QED

thm_join_valid n  p [] c
      =   priq' (join p [] c) n
      ==. priq' (carry p c) n
      ==. True     ? thm_carry_valid n p c
      *** QED

thm_join_valid n (Leaf:p) (Leaf:q) c
      =   priq' (join (Leaf:p) (Leaf:q) c) n
      ==. priq' (c:join p q Leaf) n
      ==. ((c==Leaf || pow2heap n c) && priq' (join p q Leaf) (n+1))
      ==. (True && True)   ? thm_join_valid (n+1) p q Leaf
      *** QED

thm_join_valid n (Leaf:p) (q1:q) Leaf
      =   priq' (join (Leaf:p) (Leaf:q) Leaf) n
      ==. priq' ( q1   : join p q Leaf ) n
      ==. (  (q1==Leaf || pow2heap n q1)
              && priq' (join p q Leaf) (n+1))
      ==. (True && True)   ? thm_join_valid (n+1) p q Leaf
      *** QED

thm_join_valid n (Leaf:p) (q1:q) (Node k l r)
      =   priq' (join (Leaf:p) (q1:q) (Node k l r) ) n
      ==. priq' ( Leaf : join p q (smash (Node k l r) q1) ) n
      ==. ( (Leaf==Leaf || pow2heap n Leaf)
             && priq' (join p q (smash (Node k l r) q1)) (n+1))
      ==. (True && True)
             ? thm_join_valid (n+1) p q (smash (Node k l r) q1)
      *** QED

thm_join_valid n (p1:p) (Leaf:q) (Leaf)
      =   priq' (join (p1:p)   (Leaf:q) (Leaf)) n
      ==. priq' ( p1   : join p q Leaf ) n
      ==. ( (p1==Leaf || pow2heap n p1)
                &&  priq' (join p q Leaf) (n+1))
      ==. (True && True)   ? thm_join_valid (n+1) p q Leaf
      *** QED

thm_join_valid n (p1:p) (Leaf:q) (Node k l r)
      =   priq' (join (p1:p)   (Leaf:q) (Node k l r)) n
      ==. priq' ( Leaf : join p q (smash (Node k l r) p1) ) n
      ==. ((Leaf==Leaf || pow2heap n Leaf)
             &&  priq' (join p q (smash (Node k l r) p1)) (n+1))
      ==. (True && True)
             ? thm_join_valid (n+1) p q (smash (Node k l r) p1)
      *** QED

thm_join_valid n (p1:p) (q1:q) c
      =   priq' (join (p1:p)   (q1:q)   c  ) n
      ==. priq' ( c : join p q (smash p1 q1) ) n
      ==.  ( (c==Leaf || pow2heap n c)
             &&  priq' ( join p q (smash p1 q1) ) (n+1))
```

```
      ==. (True && True)   ? thm_join_valid (n+1) p q (smash p1 q1)
      *** QED

 --η συνένωση δύο έγκυρων priority queues είναι έγκυρη priority queue
{-@ thm_merge_priq :: p: {Priqueue | priq p} ->
     q: {Priqueue | priq q} -> {priq (merge p q)}
@-}
thm_merge_priq :: Priqueue -> Priqueue -> Proof
thm_merge_priq p q = [thm_join_valid 0 p q Leaf] *** QED


--αν διαγράψουμε ένα στοιχείο από μία queue p για την οποία ισχύει
--priq' p n  τότε για τη πρώτη queue που γυρνάει η delete_max_aux
--ισχύει η ιδιότητα priq' με παράμετρο n
{-@ thm_delete_max_Some_priq_aux :: n:Nat ->
       p:{Priqueue | priq' p n} ->
       m:Key->
       { priq' (fst1( delete_max_aux p m)) n}
@-}
thm_delete_max_Some_priq_aux :: Int -> Priqueue -> Key -> Proof
thm_delete_max_Some_priq_aux n [] m = trivial
thm_delete_max_Some_priq_aux n (Leaf:p) m =
              thm_delete_max_Some_priq_aux (n+1) p m
thm_delete_max_Some_priq_aux n ((Node x t1 Leaf):p) m
            | m>x
              =   thm_delete_max_Some_priq_aux (n+1) p m
            | otherwise
              = thm_delete_max_Some_priq_aux (n+1) p m
thm_delete_max_Some_priq_aux n p m =  trivial


--Για την δεύτερη queue που επιστρέφει η delete_max_aux ισχύει πάντα
--η ιδιότητα piq
{-@ thm_delete_max_Some_priq_aux2 :: p:Priqueue -> m:Key->
    { priq (snd1( delete_max_aux p m))  }
@-}
thm_delete_max_Some_priq_aux2 :: Priqueue -> Key -> Proof
thm_delete_max_Some_priq_aux2  [] m       = trivial
thm_delete_max_Some_priq_aux2  (Leaf:p) m =
             thm_delete_max_Some_priq_aux2 p m
thm_delete_max_Some_priq_aux2  ((Node x t1 Leaf):p) m
            | m>x
              =   thm_delete_max_Some_priq_aux2 p m
            | otherwise = undefined
thm_delete_max_Some_priq_aux2 p m =  trivial


--η priority queue που απομένει μετά από τη διαγραφή του μέγιστου
--στοιχείου είναι μια έγκυρη priority queue
```

Στο παραπάνω θεώρημα (thm_delete_max_Some_priq_aux2) έχουμε χρησιμοποιήσει τη λέξη undefined. Το undefined χρησιμοποιείται όταν έχουμε δυσκολία να αποδείξουμε κάποιο θεώρημα αλλά θέλουμε ο SMT solver να μπορεί να χρησιμοποιήσει ως δεδομένο, σαν να το

είχαμε αποδείξει, σε άλλα θεωρήματα. Συγκεκριμένα η δυσκολία της περίπτωσης που έχουμε αφήσει ως undefined είναι ότι αν η είσοδος είναι μία queue ((Node x t1 Leaf):p) και ένα αριθμός m, για τον οποίο m<=x, θα κληθεί η συνάρτηση heap_delete_max και στη συνέχεια η unzip. Η unzip δέχεται ένα δέντρο (που ικανοποιεί κάποιες ιδιότητες) και το ξετυλίγει σε μία λίστα από μικρότερα δέντρα. Η λίστα από δέντρα που επιστρέφει είναι πάντα μια έγκυρη priority queue πράγμα που αντιμετωπίσαμε όμως δυσκολία στο να το αποδείξουμε.

```
{-@ thm_delete_max_Some_priq::
      p: {Priqueue | priq p &&  delete_max p /= Nothing } ->
      {priq' (snd1 (fromJust ( delete_max p ))) 0}
@-}
thm_delete_max_Some_priq :: Priqueue -> Proof
thm_delete_max_Some_priq []  = trivial
thm_delete_max_Some_priq p
   =   priq' (snd1 (fromJust ( delete_max p ))) 0
   ==. priq' (snd1 ( fromJust (   Just (P (fromJust (find_max p))
      (join  (fst1 (delete_max_aux p (fromJust (find_max p))))
      (snd1 (delete_max_aux p (fromJust (find_max p))))  Leaf))))) 0

   ==. priq' (snd1 ((P (fromJust (find_max p)) (join (fst1
      (delete_max_aux p (fromJust (find_max p))))
      (snd1 (delete_max_aux p (fromJust (find_max p))))  Leaf))))  0

   ==. priq'((join (fst1 (delete_max_aux p (fromJust (find_max p))))
     (snd1 (delete_max_aux p (fromJust (find_max p))))  Leaf)    ) 0

   ==. True
   ?  ( thm_delete_max_Some_priq_aux 0 p (fromJust(find_max p))) &&&
      ( thm_delete_max_Some_priq_aux2  p (fromJust(find_max p))) &&&
(thm_join_valid  0 (fst1 (delete_max_aux p (fromJust (find_max p))))
          (snd1 (delete_max_aux p (fromJust (find_max p))))  Leaf)
   *** QED
```

Παρακάτω ορίζουμε τη συνάρτηση elements που επιστρέφει τα στοιχεία ενός δέντρου:

```
{-@ reflect elements @-}
{-@ elements:: Tree -> [Nat]  @-}
elements:: Tree -> [Key]
elements Leaf = []
elements (Node k l r)  = k: ((elements l) ++ (elements r))
```

Επιπλέον ορίζουμε τη σχέση tree_elems που δέχεται ένα δέντρο και μια λίστα φυσικών αριθμών και αληθευει μόνο αν περιέχουν τα ίδια στοιχεία:

```
{-@ reflect tree_elems @-}
{-@ tree_elems :: Tree -> [Nat] -> Bool @-}
tree_elems :: Tree -> [Key] -> Bool
tree_elems t    l  = permutation (elements t) l


--Βοηθητικό λήμμα σχετικά με permutations
{-@ thmPermProp :: xs:[a] -> ys:{[a] | permutation xs ys} ->
    zs:{[a] | permutation xs zs} -> {permutation ys zs }
```

```
@-}
thmPermProp:: (Ord a) => [a] -> [a] -> [a] -> Proof
thmPermProp xs ys zs = trivial


--Αν μία λίστα e1 περιέχει τα ίδια στοιχεία με ένα δέντρο t και μια
--e2 είναι permutation της e1 τότε θα περιέχει τα ίδια στοιχεία με
--το t
{-@ thm_tree_elems_ext :: t:Tree -> e1: { [Key] | tree_elems t e1}
    -> e2:{ [Key] | permutation e1 e2} ->
    {tree_elems t e2 }
@-}
thm_tree_elems_ext :: Tree -> [Key] -> [Key] -> Proof
thm_tree_elems_ext t e1 e2
      = [thmPermProp (elements t) e1 e2] *** QED

--Αν δύο λίστες e1,e2 περιέχουν τα ίδια στοιχεία με ένα δέντρο t
--τότε θα είναι permutation e1 e2
{-@ thm_tree_perm :: t:Tree -> e1: { [Key] | tree_elems t e1} ->
     e2:{ [Key] | tree_elems t e2} -> {permutation e1 e2 }
@-}
thm_tree_perm :: Tree -> [Key] -> [Key] -> Proof
thm_tree_perm t e1 e2 = [thmPermProp (elements t) e1 e2] *** QED

--Βοηθητικό λήμμα
{-@ thmAppNilR :: xs:[a] -> {( xs ++ []) = xs } @-}
thmAppNilR :: [a] -> Proof
thmAppNilR [] = trivial
thmAppNilR (x:xs) = thmAppNilR xs

--η συνένωση δύο δέντρων σε ένα έχει ως στοιχεία την ένωση των
--στοιχείων των δύο δέντρων
{-@ thm_smash_elems2 ::  t: Tree -> u:{Tree | smash t u /= Leaf}
   -> {fromList (elements (smash t u)) =
       union (fromList (elements u)) (fromList (elements t)) }
@-}
thm_smash_elems2 :: Tree -> Tree  -> Proof
thm_smash_elems2  Leaf Leaf = trivial
thm_smash_elems2  Leaf u    = trivial
thm_smash_elems2  t    Leaf = trivial
thm_smash_elems2  (Node m1 t1 t) (Node m2 u1 u)
        | u/= Leaf || t/= Leaf  = trivial

thm_smash_elems2  (Node m1 t1 Leaf) (Node m2 t2 Leaf)
        | m1 > m2
   = fromList (elements (smash (Node m1 t1 Leaf) (Node m2 t2 Leaf)))
   ==. fromList (elements (Node m1 (Node m2 t2 t1) Leaf))
   ==. fromList (m1: (elements (Node m2 t2 t1) ++ (elements Leaf) ))
   ==. fromList (m1: (elements (Node m2 t2 t1) ++ [] ))
   ==. fromList (m1: (elements (Node m2 t2 t1)) )
            ? thmAppNilR  (elements (Node m2 t2 t1))
   ==. fromList (m1: (m2 :(elements t2 ++ elements t1)) )
   ==. union (fromList (m2:elements t2)) (fromList (m1:elements t1))
            ? appendBag2 m1 m2 (elements t2)  (elements t1)
```

```
   ==. union (fromList (m2: (elements t2 ++ []) )) (fromList (m1:
       (elements t1 ++ [])))
       ?thmAppNilR  (elements t2 ) &&& thmAppNilR  (elements t1 )
   ==. union (fromList (m2: (elements t2 ++ (elements Leaf)) ))
              (fromList (m1: (elements t1 ++ (elements Leaf))))
   ==. union (fromList ( elements (Node m2 t2 Leaf) ))
                (fromList ( elements (Node m1 t1 Leaf) ))
   *** QED

          | otherwise
   = fromList (elements (smash (Node m1 t1 Leaf) (Node m2 t2 Leaf)))
   ==. fromList (elements (Node m2 (Node m1 t1 t2) Leaf))
    ==. fromList (m2:(elements (Node m1 t1 t2) ++ (elements Leaf) ))
    ==. fromList (m2: (elements (Node m1 t1 t2) ++ [] ))
    ==. fromList (m2: (elements (Node m1 t1 t2)) )
                ? thmAppNilR  (elements (Node m1 t1 t2))
    ==. fromList (m2: (m1 :(elements t1 ++ elements t2)) )
    ==. union(fromList (m2:elements t2)) (fromList (m1:elements t1))
                ? appendBag2 m2 m1 (elements t1)  (elements t2)
    ==. union (fromList (m2: (elements t2 ++ []) ))
                (fromList (m1: (elements t1 ++ [])))
          ?thmAppNilR  (elements t2 ) &&& thmAppNilR  (elements t1 )
    ==. union (fromList (m2: (elements t2 ++ (elements Leaf)) ))
                (fromList (m1: (elements t1 ++ (elements Leaf))))
    ==. union (fromList ( elements (Node m2 t2 Leaf) ))
                 (fromList ( elements (Node m1 t1 Leaf) ))
    *** QED

--Βοηθητικό λήμμα για το παραπάνω θεώρημα
{-@ appendBag2 ::(Eq k, Ord k) => x:k -> y:k -> as:[k] -> bs:[k] ->
  {fromList (x:(y:(as ++ bs))) ==
          union (fromList (y:as)) (fromList (x:bs)) }
@-}
appendBag2 :: (Eq k, Ord k) => k-> k-> [k] -> [k] -> Proof
appendBag2 x y [] _      = ()
appendBag2 x y  (_:as) bs = appendBag2 x y  as bs


--η συνένωση δύο δέντρων t,u έχει τα ίδια στοιχεία με τα t,u
{-@ thm_smash_elems :: n:Int -> t: {Tree | pow2heap n t} ->
      u:{Tree | pow2heap n u} ->
      bt:{[Key] | tree_elems t bt} ->
      bu:{[Key]| tree_elems u bu} ->
      {tree_elems (smash t u) (bt ++ bu) }
@-}
thm_smash_elems :: Int -> Tree -> Tree -> [Key] -> [Key] -> Proof
thm_smash_elems n t u bt bu
      = [thm_smash_elems2 t u, appendBag (bt)( bu) ] *** QED
```

Στη συνέχεια κατ αντιστοιχία με τη συνάρτηση elements ορίζουμε τη συνάρτηση priqueue_elements που επιστρέφει τα στοιχεία μιας priority queue.

```
{-@ reflect priqueue_elements @-}
{-@ priqueue_elements :: Priqueue -> [Nat]@-}
priqueue_elements :: Priqueue -> [Key]
priqueue_elements []     = []
priqueue_elements (t:ts) =  (elements t)++ (priqueue_elements ts)
```

Ομοίως κατ αντιστοιχία με τη συνάρτηση tree_elems ορίζουμε τη συνάρτηση priqueue_elems που δέχεται μία priority queue και μία λίστα και αληθευει μόνο αν η λίστα και η queue έχουν τα ίδια στοιχεία:

```
{-@ reflect priqueue_elems @-}
{-@ priqueue_elems :: Priqueue -> [Nat] -> Bool @-}
priqueue_elems :: Priqueue -> [Key] -> Bool
priqueue_elems p    l  = permutation (priqueue_elements p) l
```

Επιπλέον ορίζουμε τη συνάρτηση abs που δέχεται μία queue και μία λίστα και καλεί την priqueue_elems. Η abs εκφράζει την ιδιότητα ισοδυναμίας μεταξύ μιας queue και μιας λίστας καθώς αληθευει μόνο αν έχουν τα ίδια στοιχεία:

```
{-@ reflect abs @-}
{-@abs :: Priqueue -> [Nat] -> Bool@-}
abs :: Priqueue -> [Key] -> Bool
abs p al = priqueue_elems p al
```

Στη συνέχεια θα αποδείξουμε ορισμένες ιδιότητες για τις παραπάνω συναρτήσεις που ορίσαμε:

```
--η empty queue είναι ισοδύναμη με τη κενή λίστα
{-@ thm_empty_relate :: {abs p_empty []}@-}
thm_empty_relate :: Proof
thm_empty_relate = trivial

--Αν μια λίστα e1 έχει τα ίδια στοιχεία με μια queue q, και για μία
--λίστα e2 είναι permutation της e1 τότε έχει τα ίδια στοιχεία με
--την q
{-@ thm_priqueue_elems_ext :: q: Priqueue ->
     e1:{ [Key] | priqueue_elems q e1} ->
     e2:{[Key] | permutation e1 e2 } ->
     { priqueue_elems q e2 }
@-}
thm_priqueue_elems_ext :: Priqueue -> [Key] -> [Key] -> Proof
thm_priqueue_elems_ext q e1 e2
     = [thmPermProp (priqueue_elements q) e1 e2] *** QED


{-@ lemma_tree_can_relate :: t:Tree ->
    {tree_elems t (elements t)}
@-}
lemma_tree_can_relate :: Tree -> Proof
lemma_tree_can_relate t = trivial


{-@ thm_can_relate :: p:{Priqueue | priq p} ->
```

```
   {abs p (priqueue_elements p)}
@-}
thm_can_relate :: Priqueue -> Proof
thm_can_relate t = trivial


--αν οι λίστες e1,e2 είναι ισοδύναμες με μια priority queue q τότε
--ισχύει permutation e1 e2
{-@ thm_abs_perm :: p:{Priqueue | priq p} -> al: { [Key] | abs p al}
     -> bl:{ [Key] | abs p bl}
     -> {permutation al bl }
 @-}
thm_abs_perm :: Priqueue -> [Key] -> [Key] -> Proof
thm_abs_perm p al bl = [thmPermProp (priqueue_elements p) al bl] ***
QED


{-@ thm_abs_perm3 :: p:Priqueue -> al: { [Key] | abs p al} ->
     bl:{ [Key] | abs p bl} -> {abs p al = abs p bl }
@-}
thm_abs_perm3 :: Priqueue -> [Key] -> [Key] -> Proof
thm_abs_perm3 p al bl = trivial


{-@ thm_smash_elems3 ::  t: Tree -> u:{Tree | smash t u /= Leaf}
      -> {tree_elems (smash t u) ((elements t)++(elements u)) }
@-}
thm_smash_elems3 :: Tree -> Tree -> Proof
thm_smash_elems3 t u
      = [thm_smash_elems2 t u ,
          appendBag (elements t) (elements u)] *** QED


{-@ thm_tree :: p:Priqueue -> pl : [Key] ->
     t:{Tree | abs (t:p) pl} ->
     { permutation (pl) (elements t ++ priqueue_elements p) }
@-}
thm_tree :: Priqueue ->[Key] ->  Tree -> Proof
thm_tree p pl t = trivial


{-@ thm_tree2 ::  tl : [Key] -> t:{Tree | tree_elems t tl} ->
      u:{Tree | smash t u /= Leaf} ->
      { tree_elems (smash t u) (tl ++ elements u) }
@-}
thm_tree2 ::  [Key] ->  Tree ->  Tree -> Proof
thm_tree2 tl t u
   = [thm_smash_elems2 t u , appendBag tl (elements u)]*** QED


appendAssoc :: [a] -> [a] -> [a] -> Proof
{-@ appendAssoc :: xs:_ -> ys:_ -> zs:_
               -> { xs ++ (ys ++ zs) = (xs ++ ys) ++ zs }
@-}
```

```
appendAssoc [] _ _          = trivial
appendAssoc (_:xs) ys zs = appendAssoc xs ys zs

--Εισάγοντας ένα δέντρο σε μία queue, η νέα queue που προκύπτει έχει
--ως στοιχεία τα στοιχεία του δέντρου και της αρχικής queue
{-@ thm_carry_elems :: q: Priqueue -> n: {Key |priq' q n} ->
      t:{Tree | (t=Leaf || pow2heap n t)} ->
      eq:{ [Key] | priqueue_elems q eq} ->
      et:{[Key] | tree_elems t et } ->
      { priqueue_elems (carry q t) (et++eq) }
@-}
thm_carry_elems:: Priqueue -> Key -> Tree -> [Key] -> [Key] -> Proof
thm_carry_elems [] n Leaf  eq et
     = [thm1 [] et , thm1 [] eq] *** QED
thm_carry_elems [] n t  eq et
     = [thm1 [] eq , thmAppNilR (elements t), thmAppNilR et] *** QED
thm_carry_elems (Leaf:q') n t  eq et
     = [appendBag (elements t) (priqueue_elements q') ,
        appendBag et eq]
     *** QED
thm_carry_elems (u:q') n Leaf  eq et
     = [thm1 [] et] *** QED
thm_carry_elems (u:q') n t eq et
     =   priqueue_elems (carry (u:q') t) (et++eq)
     ==. priqueue_elems (Leaf:carry q' (smash t u)) (et++eq)
     ==. permutation (priqueue_elements (Leaf:carry q' (smash t u)))
                     (et++eq)

     ==. permutation ( (elements Leaf) ++
            priqueue_elements ( carry q' (smash t u) )  ) (et++eq)

     ==. permutation ([]++priqueue_elements(carry q'(smash t u)))
                      (et++eq)

     ==. permutation (  priqueue_elements ( carry q' (smash t u)))
                      (et++eq)

     ==. permutation (  priqueue_elements ( carry q' (smash t u)))
                    (et++(priqueue_elements (u:q')))
           ? thmPermProp12 et eq (priqueue_elements (u:q'))
                   (priqueue_elements ( carry q' (smash t u)) )

     ==. permutation (  priqueue_elements ( carry q' (smash t u)))
                      (et++(elements u ++ (priqueue_elements q')))

     ==. permutation (  priqueue_elements ( carry q' (smash t u)))
                  ((et++elements u )++ (priqueue_elements q'))
            ? appendAssoc et (elements u) (priqueue_elements q')

     ==. priqueue_elems ( carry q' (smash t u) )
               ((et++elements u )++ (priqueue_elements q'))
     ==. True

      ?(thm_smash_valid n t u)  &&& (thm_tree2 et t u) &&&
```

```
        (thm_can_relate q') &&&
        thm_carry_elems q' (n+1) (smash t u) (priqueue_elements q')
                    ( et ++ elements u)
     *** QED



{-@ thm_insert_relate :: {p: Priqueue | priq p} -> k:Key ->
    al:{[Key] | abs p al} -> {abs (insert k p) (k:al)}
@-}
thm_insert_relate:: Priqueue -> Key -> [Key] -> Proof
thm_insert_relate p k al
     = abs (insert k p) (k:al)
     ==. abs (carry p (Node k Leaf Leaf)) (k:al)
     ==. abs (carry p (Node k Leaf Leaf)) ([k]++al)
     ==. priqueue_elems (carry p (Node k Leaf Leaf))  ([k]++al)
     ==. True ? thm_carry_elems p 0 (Node k Leaf Leaf) al [k]
     *** QED
```

Στη συνέχεια αποδεικνύουμε αρκετά βοηθητικά λήμματα που θα χρειαστούν στην απόδειξη του θεωρήματος thm_join_elems:

```
{-@ thm_perm1 :: xs: {[a] | len xs >0} ->
      { permutation xs [] == False }
@-}
thm_perm1 :: (Eq a, Ord a) => [a] -> Proof
thm_perm1 [] = ()
thm_perm1 (x:xs)
  =   permutation xs []
  ==. fromList xs == fromList []
  ==. put x (fromList xs) == empty
       ? thm_emp x (fromList xs)
  ==. False
  *** QED

{-@ thm1:: l:{[a] | l = []} -> l2:{[a] | permutation l l2}
     -> { l2 = []}
@-}
thm1:: (Eq a, Ord a) => [a] -> [a] -> Proof
thm1 [] [] =trivial
thm1 [] l = thm_perm1 l *** QED


{-@ thm_join_elems_aux :: p:{Priqueue | len p = 1}-> q:Priqueue ->
     pl:{[Int] | priqueue_elems p pl} ->
     ql:{[Int] | priqueue_elems q ql} ->
     {priqueue_elems (p++q) (pl++ql)}
@-}
thm_join_elems_aux:: Priqueue -> Priqueue -> [Int] -> [Int] -> Proof
thm_join_elems_aux [c] q pl ql
   = priqueue_elems ([c]++q) (pl++ql)
```

```
   ==. permutation (priqueue_elements ([c]++q)) (pl++ql)
   ==. permutation (priqueue_elements (c:q)) (pl++ql)
   ==. permutation ( (elements c) ++ priqueue_elements (q)) (pl++ql)
   ==. permutation ( ((elements c) ++ []) ++ priqueue_elements (q))
          (pl++ql) ? thmAppNilR (elements c)
   ==. permutation ((priqueue_elements [c])++ priqueue_elements (q))
          (pl++ql)
   ==. True
       ? thm_join_elems_aux_p (priqueue_elements [c])
              (priqueue_elements q) pl ql
   *** QED


{-@ thm_join_elems_aux_p :: p:[a]-> q:[a]
      -> pl:{[a] | permutation p pl}
      -> ql:{[a] | permutation q ql}
      -> {permutation (p++q) (pl++ql)}
@-}
thm_join_elems_aux_p :: (Ord a) => [a] -> [a] -> [a] -> [a] -> Proof
thm_join_elems_aux_p p q pl ql
     = [appendBag p q, appendBag pl ql] *** QED

{-@ thm_join_elems_aux2 :: c:Tree -> cl:{[Key] | tree_elems c cl}
     -> {priqueue_elems [c] (cl)}
@-}
thm_join_elems_aux2 :: Tree -> [Key] -> Proof
thm_join_elems_aux2 c cl = undefined


{-@ thmPermProp12 :: ws: [a] -> xs:[a] ->
      ys:{[a] | permutation xs ys} -> zs:[a] ->
     {permutation zs (ws++xs) <=> permutation zs (ws++ys)}
@-}
thmPermProp12:: (Ord a) => [a] -> [a] -> [a] -> [a] -> Proof
thmPermProp12 ws xs ys zs = [appendBag ws xs , appendBag ws ys] ***
QED

{-@ thmPermProp13 :: ws: [a] -> xs:[a] ->
     ys:{[a] | permutation xs ys} -> zs:[a] ->
    {permutation zs (ws++xs) <=> permutation zs (ys++ws)}
@-}
thmPermProp13:: (Ord a) => [a] -> [a] -> [a] -> [a] -> Proof
thmPermProp13 ws xs ys zs = [appendBag ws xs , appendBag ys ws] ***
QED

{-@ thmPermProp14 :: xs: [a] -> ys:[a] -> zs:[a] ->
      {permutation xs (ys++zs) <=> permutation xs (zs++ys)}
@-}
thmPermProp14:: (Ord a) => [a] -> [a] -> [a] -> Proof
thmPermProp14 xs ys zs = [appendBag ys zs, appendBag zs ys] *** QED

{-@ thmPermProp15 :: xs: [a] -> ys:[a] ->
     {permutation  (xs++ys) (ys++xs)}
@-}
```

```
thmPermProp15:: (Ord a) => [a] -> [a] -> Proof
thmPermProp15 xs ys = [appendBag xs ys, appendBag ys xs] *** QED

{-@ thmPermProp16 :: ws: [a] -> xs:[a] ->
    ys:{[a] | permutation xs ys} -> zs:[a] ->
    {permutation zs (xs++ws) <=> permutation zs (ys++ws)}
@-}
thmPermProp16:: (Ord a) => [a] -> [a] -> [a] -> [a] -> Proof
thmPermProp16 ws xs ys zs
     = [appendBag xs ws , appendBag ys ws] *** QED

{-@ thmPermProp17 :: xs: [a] -> ys:{[a] | permutation xs ys} ->
    zs:[a] -> {permutation  (xs++zs) (ys++zs)}
@-}
thmPermProp17:: (Ord a) => [a] -> [a] -> [a] -> Proof
thmPermProp17 xs ys zs = [appendBag ys zs, appendBag xs zs] *** QED


{-@ thm_priq_pow2heap:: n:Nat ->
     p:{Priqueue | len p > 0 && priq' p n && lHd p /= Leaf}
    -> { pow2heap n (lHd p)}
@-}
thm_priq_pow2heap :: Int -> Priqueue -> Proof
thm_priq_pow2heap n (p1:p) = trivial
```

Τώρα μπορούμε να προχωρήσουμε στη απόδειξη του θεωρήματος thm_join_elems, κατά το οποίο αν μια priority queue p που έχει την ιδιότητα priq' p n έχει τα ίδια στοιχεία με μία λιστα pe και μια priority queue q που έχει την ιδιότητα priq' q n έχει τα ίδια στοιχεία με μία λιστα qe και ένα δέντρο c για το οποίο ισχύει ότι το c είναι φύλλο ή `pow2heap n c` τότε το `join p q c` έχει τα ίδια στοιχεία με τη λίστα `(ce++pe++qe)`:

```
{-@thm_join_elems :: n:Key -> p:{Priqueue | priq' p n} ->
      q:{Priqueue | priq' q n} ->
      c:{Tree | (c=Leaf || pow2heap n c)} ->
      pe:{[Nat] | priqueue_elems p pe} ->
      qe:{[Nat] | priqueue_elems q qe} ->
      ce:{[Nat] | tree_elems c ce} ->
      { priqueue_elems (join p q c) (ce++pe++qe) } / [len p]
@-}
thm_join_elems :: Key -> Priqueue -> Priqueue -> Tree -> [Key] ->
   [Key] -> [Key] -> Proof

thm_join_elems n [] q c pe qe ce
    =   priqueue_elems (join [] q c) (ce++pe++qe)
    ==. priqueue_elems (carry q c) (ce++(pe++qe))
    ==. priqueue_elems (carry q c) (ce++([]++qe)) ? thm1 [] pe
    ==. priqueue_elems (carry q c) (ce++qe)  ? thmAppNilR ce
    ==. True ? thm_carry_elems q n c qe ce
    *** QED

thm_join_elems n p [] c pe qe ce
    =   priqueue_elems (join p [] c) (ce++(pe++qe))
```

```
    ==. priqueue_elems (carry p c) (ce++(pe++qe))
    ==. priqueue_elems (carry p c) (ce++(pe++[])) ? thm1 [] qe
    ==. priqueue_elems (carry p c) (ce++qe)  ? thmAppNilR (ce++pe)
    ==. True ? thm_carry_elems p n c pe ce
    *** QED

thm_join_elems n (Leaf:p) (Leaf:q) c pe qe ce
     =   priqueue_elems (join  (Leaf:p) (Leaf:q) c) (ce++pe++qe)
     ==. priqueue_elems (join  (Leaf:p) (Leaf:q) c) (ce++(pe++qe))
          ? appendAssoc ce pe qe
     ==. priqueue_elems (c: join p q Leaf) (ce++(pe++qe))
     ==. priqueue_elems ([c] ++ join p q Leaf) (ce++(pe++qe))
     ==. True ? ( thm_join_elems (n+1) p q Leaf pe qe [] ) &&&
                (thmAppNilR (pe++qe)) &&& (thm_join_elems_aux2 c ce)
              &&& thm_join_elems_aux [c] (join p q Leaf) ce (pe++qe)
     *** QED

thm_join_elems n (Leaf:p) (q1:q) Leaf pe qe ce
     =   priqueue_elems (join  (Leaf:p) (q1:q) Leaf) (ce++pe++qe)
     ==. priqueue_elems (join  (Leaf:p) (q1:q) Leaf) (ce++(pe++qe))
            ? appendAssoc ce pe qe
     ==. priqueue_elems (join  (Leaf:p) (q1:q) Leaf) ([]++(pe++qe))
            ? thm1 [] ce
     ==. priqueue_elems ( q1   : join p q Leaf) (pe++qe)
     ==. priqueue_elems ([q1] ++ join p q Leaf) (pe++qe)
     ==. priqueue_elems ([q1] ++ join p q Leaf)
                        (pe++(priqueue_elements (q1:q)))
          ? thmPermProp12 pe qe (priqueue_elements (q1:q))
                        (priqueue_elements ([q1] ++ join p q Leaf))

     ==. priqueue_elems ([q1] ++ join p q Leaf) (pe++ ((elements q1)
                    ++  (priqueue_elements q)))
     ==. priqueue_elems ([q1] ++ join p q Leaf) (( (elements q1) ++
              (priqueue_elements q) ) ++ pe)
          ? thmPermProp13 pe qe (priqueue_elements (q1:q))
              (priqueue_elements ([q1] ++ join p q Leaf))

      ==. priqueue_elems ([q1] ++ join p q Leaf) ((elements q1) ++
              ((priqueue_elements q)  ++ pe))
          ? appendAssoc (elements q1) (priqueue_elements q) pe
      ==. True
      ? (thm_join_elems (n+1) p q Leaf pe (priqueue_elements q) [] )
      &&& (thmPermProp14 (priqueue_elements (join p q Leaf)) (pe)
                   (priqueue_elements q))
      &&& (thm_join_elems_aux2 q1 (elements q1) ) &&&
      thm_join_elems_aux [q1] (join p q Leaf) (elements q1)
             ((priqueue_elements q)++pe)
      *** QED


thm_join_elems n (Leaf:p) (q1:q)  (Node k l r) pe qe ce
       =   priqueue_elems (join (Leaf:p) (q1:q) (Node k l r))
              (ce++pe++qe)
       ==. priqueue_elems (Leaf : join p q (smash (Node k l r) q1))
```

```
       (ce++pe++qe)
==. permutation (priqueue_elements
     (Leaf : join p q (smash (Node k l r) q1))) (ce++pe++qe)
==. permutation ( (elements Leaf) ++ priqueue_elements
        (join p q (smash (Node k l r) q1))) (ce++pe++qe)
==. permutation ( [] ++ priqueue_elements
        (join p q (smash (Node k l r) q1))) (ce++pe++qe)
==. permutation (priqueue_elements
        (join p q (smash (Node k l r) q1))) (ce++pe++qe)
==. priqueue_elems (join p q (smash (Node k l r) q1))
        (ce++pe++qe)
==. priqueue_elems (join p q (smash (Node k l r) q1))
         (ce++(pe++qe)) ?  appendAssoc ce pe qe
==. priqueue_elems (join p q (smash (Node k l r) q1))
         (ce++(qe++pe))
    ?  (thmPermProp15 pe qe) &&&
       thmPermProp12 ce (pe++qe) (qe++pe)
       (priqueue_elements (join p q (smash (Node k l r) q1)))

==. priqueue_elems (join p q (smash (Node k l r) q1))
                    (ce++((priqueue_elements (q1:q))++pe))
     ? (thmPermProp17 qe (priqueue_elements (q1:q)) pe)
   &&& thmPermProp12 ce (qe++pe) ((priqueue_elements
      (q1:q))++pe)
      (priqueue_elements (join p q (smash (Node k l r) q1)))

==. priqueue_elems (join p q (smash (Node k l r) q1))
         ((ce++(priqueue_elements (q1:q)))++pe)
     ? appendAssoc ce (priqueue_elements (q1:q)) pe

==. priqueue_elems (join p q (smash (Node k l r) q1))
     ((ce++((elements q1) ++ (priqueue_elements q)))++pe)

==. priqueue_elems (join p q (smash (Node k l r) q1))
     (((ce++(elements q1)) ++ (priqueue_elements q))++pe)
     ? appendAssoc ce (elements q1) (priqueue_elements q)

==. priqueue_elems (join p q (smash (Node k l r) q1))
     ((ce++(elements q1)) ++ ((priqueue_elements q)++pe))
  ? appendAssoc (ce++(elements q1)) (priqueue_elements q) pe

==. priqueue_elems (join p q (smash (Node k l r) q1))
     ((ce++(elements q1)) ++ (pe++(priqueue_elements q)))
     ? (thmPermProp15 (priqueue_elements q) pe) &&&
     thmPermProp12 (ce++(elements q1))
     ((priqueue_elements q)++pe) (pe++(priqueue_elements q))
     (priqueue_elements (join p q (smash (Node k l r) q1)))

==. priqueue_elems (join p q (smash (Node k l r) q1))
     ((ce++(elements q1)) ++ pe++(priqueue_elements q))
   ? appendAssoc (ce++(elements q1)) pe (priqueue_elements q)

==. True
    ?  (thm_smash_elems n (Node k l r) q1 ce (elements q1))
```

```
          &&& (thm_join_elems (n+1) p q (smash (Node k l r) q1) pe
                 (priqueue_elements q) (ce++(elements q1)) )
        *** QED

thm_join_elems n  (p1:p)   (Leaf:q) (Leaf)  pe qe ce
         =   priqueue_elems (join  (p1:p) (Leaf:q) Leaf)
             (ce++pe++qe)
         ==. priqueue_elems (join  (p1:p) (Leaf:q) Leaf)
             ([]++pe++qe) ? thm1 [] ce
         ==. priqueue_elems (p1   : join p q Leaf) ([]++pe++qe)
         ==. priqueue_elems ([p1]++ join p q Leaf) (pe++qe)
         ==. priqueue_elems ([p1]++ join p q Leaf)
              (priqueue_elements (p1:p) ++ qe)
             ?  thmPermProp16 qe pe (priqueue_elements (p1:p))
                  (priqueue_elements ([p1]++ join p q Leaf))

         ==. priqueue_elems ([p1]++ join p q Leaf)
               (((elements p1) ++ (priqueue_elements p)) ++ qe)

         ==. priqueue_elems ([p1]++ join p q Leaf)
               ((elements p1) ++ ((priqueue_elements p) ++ qe))
              ? appendAssoc (elements p1) (priqueue_elements p) qe

          ==. True
       ? (thm_join_elems (n+1) p q Leaf (priqueue_elements p) qe [])
             &&& (thm_join_elems_aux2 p1 (elements p1))
             &&& thm_join_elems_aux [p1] (join p q Leaf)
                (elements p1) ((priqueue_elements p)++qe)
          *** QED


thm_join_elems n (p1:p) (Leaf:q) (Node k l r) pe qe ce
          =   priqueue_elems (join  (p1:p) (Leaf:q) (Node k l r))
                 (ce++pe++qe)
          ==. priqueue_elems (Leaf:join p q (smash (Node k l r) p1))
                 (ce++pe++qe)
          ==. permutation (priqueue_elements
             (Leaf : join p q (smash (Node k l r) p1))) (ce++pe++qe)
          ==. permutation ( (elements Leaf) ++ priqueue_elements
             (join p q (smash (Node k l r) p1))) (ce++pe++qe)
          ==. permutation ( [] ++ priqueue_elements
             (join p q (smash (Node k l r) p1))) (ce++pe++qe)
          ==. permutation (priqueue_elements
             (join p q (smash (Node k l r) p1))) (ce++pe++qe)
          ==. priqueue_elems (join p q (smash (Node k l r) p1))
             (ce++pe++qe)
          ==. priqueue_elems (join p q
                (smash (Node k l r) p1)) (ce++(pe++qe))
              ? appendAssoc ce pe qe
          ==. priqueue_elems (join p q (smash (Node k l r) p1))
                (ce++((priqueue_elements (p1:p))++qe))
              ? (thmPermProp17 pe (priqueue_elements (p1:p)) qe)
              &&& thmPermProp12 ce (pe++qe)
                ((priqueue_elements (p1:p))++qe)
```

```
            (priqueue_elements (join p q (smash (Node k l r) p1)))

         ==. priqueue_elems (join p q (smash (Node k l r) p1))
               ((ce++(priqueue_elements (p1:p)))++qe)
             ? appendAssoc ce (priqueue_elements (p1:p)) qe

         ==. priqueue_elems (join p q (smash (Node k l r) p1))
             ((ce++((elements p1) ++ (priqueue_elements p)))++qe)

         ==. priqueue_elems (join p q (smash (Node k l r) p1))
            (((ce++(elements p1)) ++ (priqueue_elements p))++qe)
              ? appendAssoc ce (elements p1) (priqueue_elements p)
         ==. priqueue_elems (join p q (smash (Node k l r) p1))
              ((ce++(elements p1)) ++ ((priqueue_elements p)++qe))
             ? appendAssoc (ce++(elements p1))
                        (priqueue_elements p) qe
         ==. True
            ? (thm_smash_elems n (Node k l r) p1 ce (elements p1))
            &&& (thm_join_elems (n+1) p q (smash (Node k l r) p1)
                  (priqueue_elements p) qe (ce++(elements p1)) )
         *** QED


thm_join_elems n (p1:p) (q1:q) c pe qe ce
          =   priqueue_elems (join  (p1:p) (q1:q) c) (ce++pe++qe)
          ==. priqueue_elems (c : join p q (smash p1 q1))
               (ce++pe++qe)
          ==. priqueue_elems (c : join p q (smash p1 q1))
               (ce++(pe++qe))  ? appendAssoc ce pe qe
          ==. priqueue_elems (c : join p q (smash p1 q1))
              (ce++( priqueue_elements (p1:p)++
                  (priqueue_elements (q1:q))) )
              ? thmPermProp18 pe qe (priqueue_elements (p1:p))
                   (priqueue_elements (q1:q)) &&&
                thmPermProp12 ce (pe++qe)
            ( priqueue_elements (p1:p)++(priqueue_elements (q1:q)))
            ( priqueue_elements (c : join p q (smash p1 q1)))

          ==. priqueue_elems (c : join p q (smash p1 q1))
                (ce++( ((elements p1) ++ (priqueue_elements p))
                 ++( ((elements q1) ++ (priqueue_elements q)))) )

          ==. priqueue_elems (c : join p q (smash p1 q1))
                (ce++(  ( (elements p1) ++ (elements q1))
              ++ ( (priqueue_elements p) ++ (priqueue_elements q))))
          ? thmPermProp19 (elements p1) (priqueue_elements p)
               (elements q1) (priqueue_elements q) &&&
          thmPermProp12 ce (((elements p1) ++ (priqueue_elements p))
                   ++( ((elements q1) ++ (priqueue_elements q) )))
                (((elements p1) ++ (elements q1))  ++
                   ((priqueue_elements p) ++ (priqueue_elements q)))
                (priqueue_elements (c:join p q (smash p1 q1)))

         ==. priqueue_elems (c : join p q (smash p1 q1))
```

```
                (ce++(( ((elements p1) ++ (elements q1)) ++
                 (priqueue_elements p)) ++ (priqueue_elements q)))
          ? appendAssoc ((elements p1) ++ (elements q1))
                (priqueue_elements p)  (priqueue_elements q)

         ==. True
          ? thm_smash_elems n p1 q1 (elements p1) (elements q1) &&&
        thm_join_elems (n+1) p q (smash p1 q1) (priqueue_elements p)
           (priqueue_elements q) ((elements p1) ++ (elements q1))
           &&&  (thm_join_elems_aux2 c ce) &&&
          thm_join_elems_aux [c] (join p q (smash p1 q1)) ce
           (((elements p1) ++ (elements q1))++((priqueue_elements p)
                 ++ (priqueue_elements q)) )
          *** QED


{-@ thm_merge_relate :: p:{Priqueue | priq p} ->
       q:{Priqueue | priq q} -> pl:{[Key] | abs p pl} ->
       ql:{[Key] | abs q ql}  ->
       al:{[Key] | abs (merge p q) al} ->
       { permutation al (pl++ql)}
@-}
thm_merge_relate ::
   Priqueue -> Priqueue -> [Key] -> [Key] -> [Key] -> Proof
thm_merge_relate p q pl ql al
          = thm_join_elems 0 p q Leaf pl ql []
```

Στη συνέχεια θα αποδείξουμε ορισμένα βοηθητικά λήμματα με σκοπό να αποδείξουμε το θεώρημα thm_delete_max_None_relate3 κατά το οποίο η διαγραφή του μέγιστου στοιχείου μιας queue επιστρέφει Nothing αν και μόνο αν η queue έχει τα ίδια στοιχεία με τη κενή λίστα. Αυτό δεν σημαίνει ότι αναγκαστικά η priority queue είναι κενή καθώς μπορεί να περιέχει μόνο Leaf ως στοιχεία.

```
--συνάρτηση υπολογισμού μήκου λίστας
{-@ reflect len1@-}
{-@ len1 :: [a] -> Nat@-}
len1 :: [a] -> Int
len1 [] = 0
len1 (x:xs) = 1+ len1 xs

--αν το t δεν είναι φύλλο τότε η λίστα με τα στοιχεία έχει θετικό
--μήκος
{-@ thm_del1 :: t:{Tree | t/=  Leaf} -> {len (elements t)>0} @-}
thm_del1 :: Tree -> Proof
thm_del1 Leaf = trivial
thm_del1 (Node k l r)
     = len1 (elements (Node k l r)) > 0
     ==. (len1 (k: ((elements l) ++ (elements r)))) > 0
     ==. (1+ len1 ((elements l) ++ (elements r))) > 0
     ==. True
     *** QED
```

```
app_length :: [a] -> [a] -> Proof
{-@ app_length ::  xs:[a] -> ys:[a] ->
    { len (xs ++ ys) == len xs + len ys }
@-}

app_length [] ys
    = len1 ([] ++ ys)
    ==. len1 ys
    ==. len1 [] + len1 ys
    *** QED

app_length (x:xs) ys
    = len1 ( (x:xs) ++ ys )
    ==.len1  (x:(xs ++ ys) )
    ==. 1 + len1 (xs ++ ys)
    ==. 1 + len1 xs + len1 ys     ? app_length xs ys
    ==. len1 xs + len1 ys
    *** QED


--Ομοίως αν μια priority queue έχει ως πρώτο στοιχείο ένα δέντρο
--που δεν είναι leaf τότε η λίστα με τα στοιχεία της έχει θετικό
--μήκος
{-@ thm_del2 :: p:{Priqueue | len p>0 && lHd p /=  Leaf} ->
     {len (priqueue_elements p)>0}
@-}
thm_del2 :: Priqueue -> Proof
thm_del2 (p1:p)
     = len1 (priqueue_elements (p1:p))>0
     ==. len1 ( (elements p1) ++ priqueue_elements p) >0
     ==. ((len1 (elements p1)) + (len1 (priqueue_elements p))) >0
               ? app_length (elements p1) (priqueue_elements p)
     ==. True  ? thm_del1 p1
     *** QED

{-@ thm_del3 :: pl:[Key] ->
    p:{Priqueue | len p>0 && lHd p ==  Leaf && abs p pl} ->
    {abs (lTl p) pl}
@-}
thm_del3 :: [Key] -> Priqueue -> Proof
thm_del3 pl (Leaf:p)
     = abs (lTl (Leaf:p)) pl
     ==. abs (lTl p) pl
     ==. permutation (priqueue_elements p) pl
     ==. permutation ([] ++ priqueue_elements p) pl
     ==. permutation (priqueue_elements ((Leaf:p))) pl
     ==. abs (Leaf:p) pl
     ==. True
     *** QED


--Αποδεικνύουμε το ευθύ του θεωρήματος που θέλουμε να αποδείξουμε:
{-@ thm_delete_max_None_relate :: p: { Priqueue | abs p []} ->
       { delete_max p = Nothing}
```

```
@-}
thm_delete_max_None_relate :: Priqueue -> Proof
thm_delete_max_None_relate [] = trivial
thm_delete_max_None_relate (Leaf:p)
      = undefined
thm_delete_max_None_relate (p1:p)
      =  [ thm_del2 (p1:p),
           thm_perm2 (priqueue_elements (p1:p)) [] ]
      *** QED


{-@ thm_perm2 :: (Ord a, Eq a) => xs: [a] ->
        ys: {[a] |len xs /= len ys}  ->
        {permutation xs ys == False}
@-}
thm_perm2 :: (Ord a, Eq a) => [a] -> [a] -> Proof
thm_perm2 xs ys
  =   permutation xs ys
  ==. fromList xs == fromList ys
  ==. bagSize (fromList xs) == bagSize (fromList ys)
         ? thm_size xs &&& thm_size ys
  ==. length xs == length ys
  ==. False
  *** QED


--Αν το πρώτο στοιχείο μιας Priority Queue δεν είναι Leaf τότε η
--findmax επιστρέφει κάποιο στοιχείο
{-@ thm_del5:: p:{Priqueue | len p>0 && lHd p /= Leaf} ->
      {find_max p /= Nothing}
@-}
thm_del5 :: Priqueue -> Proof
thm_del5 (Leaf:p) = trivial
thm_del5 (p1:p)   = trivial

{-@ thm_del4:: p:{Priqueue | len p>0 && lHd p /= Leaf} ->
        {delete_max p /= Nothing}
@-}
thm_del4 :: Priqueue -> Proof
thm_del4 (Leaf:p) = trivial
thm_del4 (p1:p)
  = delete_max (p1:p) /= Nothing
  ==. (if(find_max (p1:p) == Nothing) then Nothing else
  let (P p' q') = delete_max_aux (p1:p) (fromJust (find_max (p1:p)))
    in Just (P (fromJust (find_max (p1:p))) (join p' q' Leaf)) )
    /= Nothing

  ==. (let (P p' q') = delete_max_aux (p1:p) (fromJust (find_max
        (p1:p))) in Just (P (fromJust (find_max (p1:p)))
           (join p' q' Leaf)) ) /= Nothing
     ? thm_del5 (p1:p)
  ==. True
  *** QED
```

```
--αν  βρεθεί κάποιο μέγιστο στοιχείο στην queue (Leaf:p) τότε
--θα βρεθεί και στην p

{-@ thm_delete_max_None_relate_aux :: p: Priqueue ->
   { find_max (Leaf:p) /= Nothing <=> find_max p /= Nothing }
@-}
thm_delete_max_None_relate_aux :: Priqueue -> Proof
thm_delete_max_None_relate_aux p = trivial

--Ορισμένα βοηθητικά λήμματα για το θεώρημα
--thm_delete_max_None_relate2

{-@ thm_delete_max_None_relate4 :: p: Priqueue ->
    { delete_max (Leaf:p) == Nothing ==> delete_max p == Nothing }
@-}
thm_delete_max_None_relate4 :: Priqueue -> Proof
thm_delete_max_None_relate4 [] = trivial
thm_delete_max_None_relate4 p | (find_max p == Nothing ) = trivial
thm_delete_max_None_relate4 p | (find_max p /= Nothing ) =
       [thm_delete_max_None_relate_aux2 (Leaf:p),
        thm_delete_max_None_relate_aux p ] *** QED


{-@ thm_delete_max_None_relate_aux1 :: p: Priqueue ->
    { find_max p == Nothing ==> delete_max p == Nothing }
@-}
thm_delete_max_None_relate_aux1 :: Priqueue -> Proof
thm_delete_max_None_relate_aux1 [] = trivial
thm_delete_max_None_relate_aux1 p
      | (find_max p == Nothing ) = trivial
thm_delete_max_None_relate_aux1 p
      | (find_max p /= Nothing ) = trivial


{-@ thm_delete_max_None_relate1 :: p: Priqueue ->
    { delete_max p == Nothing ==> find_max p == Nothing }
@-}
thm_delete_max_None_relate1 :: Priqueue -> Proof
thm_delete_max_None_relate1 [] = trivial
thm_delete_max_None_relate1 p
       | find_max p == Nothing = trivial
thm_delete_max_None_relate1 q
       | find_max q /= Nothing =
 (thm_delete_max_None_relate_aux3 (find_max q), isJust (find_max q),
   thm_delete_max_None_relate_aux4 (P (fromJust (find_max q)) (join
   (fst1 (delete_max_aux q (fromJust (find_max q))))
   (snd1 (delete_max_aux q (fromJust (find_max q))))  Leaf))  ,
  thm_delete_max_None_relate_aux3 (Just (P (fromJust (find_max q))
   (join  (fst1 (delete_max_aux q (fromJust (find_max q))))
    (snd1 (delete_max_aux q (fromJust (find_max q))))  Leaf))))
 *** QED
```

```
{-@ thm_delete_max_None_relate_aux2 :: p: Priqueue ->
   { find_max p == Nothing <=> delete_max p == Nothing }
@-}
thm_delete_max_None_relate_aux2 :: Priqueue -> Proof
thm_delete_max_None_relate_aux2 p =
  [thm_delete_max_None_relate_aux1 p,
   thm_delete_max_None_relate1 p] *** QED

{-@ thm_delete_max_None_relate_aux3 :: (Eq a) => y: Maybe a ->
   {  y /= Nothing <=>isJust y   }
@-}
thm_delete_max_None_relate_aux3 ::  Maybe a -> Proof
thm_delete_max_None_relate_aux3 (Just y) = trivial
thm_delete_max_None_relate_aux3 Nothing  = trivial

{-@ thm_delete_max_None_relate_aux4 :: (Eq a) =>   x: a ->
     {isJust (Just x) }
@-}
thm_delete_max_None_relate_aux4 ::  a -> Proof
thm_delete_max_None_relate_aux4 x = trivial


--Αποδεικνύουμε τη αντίστροφη κατεύθυνση του θεωρήματος
{-@ thm_delete_max_None_relate2 :: p: Priqueue ->
      { delete_max p = Nothing ==> abs p [] }
@-}
thm_delete_max_None_relate2 :: Priqueue -> Proof
thm_delete_max_None_relate2 [] = trivial
thm_delete_max_None_relate2 (Leaf:p)
      =  abs (Leaf:p) []
      ==. permutation (priqueue_elements (Leaf:p))  []
      ==. permutation (elements Leaf ++ (priqueue_elements p)) []
      ==. permutation ([] ++ (priqueue_elements p)) []
      ==. permutation (priqueue_elements p) []
      ==.  True
            ? thm_delete_max_None_relate4 p  &&&
              thm_delete_max_None_relate2 p
      *** QED
thm_delete_max_None_relate2 (p1:p) = [thm_del4 (p1:p) ] *** QED


--Το τελικό θεώρημα που θέλουμε να αποδείξουμε:
{-@ thm_delete_max_None_relate3 :: p:{Priqueue | priq p} ->
     { abs p [] <=> delete_max p = Nothing }
@-}
thm_delete_max_None_relate3 :: Priqueue -> Proof
thm_delete_max_None_relate3 [] = trivial
thm_delete_max_None_relate3 p =
  [thm_delete_max_None_relate2 p, thm_delete_max_None_relate p]
  *** QED
```

Με τα παραπάνω θεωρήματα κλείνουμε το κεφάλαιο των Binomial Queues που είναι και το τελευταίο κεφάλαιο που εξετάσαμε.

# 5 Συμπεράσματα

## 5.1 Συνεισφορά

Η συνεισφορά της εργασίας και τα συμπεράσματα που προέκυψαν είναι τα παρακάτω:

- Υλοποιήθηκε η επαληθευση ιδιοτήτων αρκετών αλγορίθμων και δομών δεδομένων σε Liquid Haskell. Ως αποτέλεσμα παρουσιάζουμε πως μπορούμε να επαληθεύσουμε την ορθότητα αλγορίθμων, αποδεικνύοντας διάφορες ιδιότητες των αλγορίθμων που εγγυώνται της ορθότητα τους παρουσιάζοντας παράλληλα το τρόπο με τον οποίο αποδεικνείουμε θεωρήματα στη Liquid Haskell.

- Χρησιμοποιώντας τη Liquid Haskell για την απόδειξη των θεωρημάτων καταλήγουμε στο ότι η Liquid Haskell είναι απόλυτα ικανός deductive verifier για την απόδειξη θεωρημάτων σε Haskell και επιπλέον συγκρίνοντας τη με το Coq το οποίο συναντήσαμε στο Software Foundations παρατηρούνται τα εξής πλεονεκτήματα:

    - Η χρήση SMT solver για έλεγχο θεωρημάτων επιτυγχάνει καλύτερη αυτοματοποίηση. Απλά θεωρήματα μπορούν να επαληθευτούν τελείως αυτόματα από τον slover ενώ για τα πιο απαιτητικά θεωρήματα ο χρήστης μπορεί να γράψει κάποιο equational proof ή με χρήση του PLE να παρέχει τα κατάλληλα στοιχεία ώστε ο solver να αποδείξη το ζητούμενο αυτόματα.

    - Επιπλέον η Liquid Haskell παρέχει τη δυνατότητα στον χρήστη να γραφει refinement types για τις συναρτήσεις που ορίζει. Αυτό όπως έχουμε ξαναναφέρει αποτελεί ένα τρόπο επαληθευσης ιδιοτήτων που δεν παρέχεται στο Coq.

## 5.2 Προτάσεις Μελλοντικής Έρευνας

Στο πλαίσιο μια μελλοντικής έρευνας τα εξής οι παρακάτω θεματολογίες θα παρουσίαζαν αρκετό ενδιαφέρον:

- Θα μπορούσαμε να χρησιμοποιήσουμε τη Liquid Haskell προκειμένου να παρέχουμε θεωρήματα που θα μας επιτρέπουν το μετασχηματισμό παραστάσεων να πετύχουμε πιο αποδοτικούς υπολογισμούς. Θα ήταν χρήσιμη η επέκταση του compiler της Haskell προκειμένου να δέχεται τέτοια θεωρήματα και σύμφωνα με αυτά να βελτιστοποιεί περισσότερο τα προγράμματα.

- Επέκταση της Liquid Haskell ώστε να παρέχει πιο ακριβή μηνύματα σε σφάλματα. Μέχρι τώρα αν σε μια απόδειξη ενός θεωρήματος υπάρχει κάποιο λάθος, δεν αναφέρεται σε ποια σημεία υπάρχει το λάθος αλλά απλά ότι η απόδειξη δεν είναι ορθή.

- Τροποποίηση της Liquid Haskell προκειμένου να παρέχεται η δυνατότητα για reflection συναρτήσεων που ανήκουν σε κάποιο typeclass προκειμένου να υπάρχει η δυνατότητα υλοποίησης interface στη Liquid Haskell.

## ΒΙΒΛΙΟΓΡΑΦΙΑ


[1]: https://softwarefoundations.cis.upenn.edu/vfa-current/index.html

[2] : Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: completeverificationwithSMT.PACMPL2,POPL,53:1–53:31. https: //doi.org/10.1145/3158141

[3]: https://ucsd-progsys.github.io/liquidhaskell-tutorial/01-intro.html#/refinement-types

[4]: https://ucsd-progsys.github.io/liquidhaskell-tutorial/02-logic.html#/verification-conditions

[5]: http://goto.ucsd.edu/~nvazou/theorem-proving-for-all/main.pdf

[6]: https://softwarefoundations.cis.upenn.edu/lf-current/Lists.html

[7]: https://softwarefoundations.cis.upenn.edu/vfa-current/Perm.html

[8]: https://softwarefoundations.cis.upenn.edu/vfa-current/Sort.html

[9]: https://softwarefoundations.cis.upenn.edu/vfa-current/Selection.html

[10]: https://softwarefoundations.cis.upenn.edu/vfa-current/SearchTree.html

[11]: https://softwarefoundations.cis.upenn.edu/lf-current/Maps.html

[12]: https://softwarefoundations.cis.upenn.edu/vfa-current/ADT.html

[13]: https://softwarefoundations.cis.upenn.edu/vfa-current/Priqueue.html

[14]: https://softwarefoundations.cis.upenn.edu/vfa-current/Redblack.html

[15]: https://softwarefoundations.cis.upenn.edu/vfa-current/Trie.html

[16]: https://github.com/ucsd-progsys/liquid-sf/blob/master/Maps.hs

[17]: https://softwarefoundations.cis.upenn.edu/vfa-current/Binom.html

[18]: Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Haskell 2017). ACM, New York, NY, USA, 63-74. DOI: https://doi.org/10.1145/3122955.3122963