Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Επαλήθευση πολυπλοκότητας αλγορίθμων σε LiquidHaskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΔΙΑΜΑΝΤΑΚΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2021

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Επαλήθευση πολυπλοκότητας αλγορίθμων σε LiquidHaskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΔΙΑΜΑΝΤΑΚΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Οκτωβρίου 2021.

.....	.....	.....
Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Αριστείδης Παγουρτζής Καθηγητής Ε.Μ.Π.	Δημήτριος Φωτάκης Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2021

Παναγιώτης Διαμαντάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η LiquidHaskell επεκτείνει το σύστημα τύπων της Haskell με λογικά κατηγορήματα που επιτρέπουν την επαλήθευση κρίσιμων ιδιοτήτων κατά την μεταγλώττιση από έναν SMT solver. Επίσης, χάρη στον μηχανισμό του reflection μπορεί να λειτουργήσει και ως σύστημα υποστήριξης αποδείξεων. Στην παρούσα διπλωματική παρουσιάζεται μια βιβλιοθήκη γραμμένη σε LiquidHaskell η οποία αξιοποιεί τις παραπάνω δυνατότητες και παρέχοντας τους κατάλληλους τύπους δεδομένων και συναρτήσεις, επιτρέπει την ανάλυση κατανάλωσης πόρων. Στη συνέχεια παρατίθενται υλοποιήσεις από γνωστούς αλγορίθμους σε Haskell που λειτουργούν ως παραδείγματα, για την επαλήθευση της αναμενόμενης συμπεριφοράς τους σε θέματα χρονικής πολυπλοκότητας.

Λέξεις κλειδιά

LiquidHaskell, refinement types, στατική επαλήθευση, συστήματα υποστήριξης αποδείξεων, equational reasoning, resource analysis

Abstract

LiquidHaskell extends Haskell's type system with logical statements that allow critical properties to be verified by an SMT solver. Also, thanks to the reflection mechanism, it can function as a proof assistant. In this thesis, a library written in LiquidHaskell is introduced which utilizes the above capabilities and by providing the appropriate data types and functions, allows the analysis of resource consumption. Finally, Haskell implementations of well-known algorithms are presented that serve as examples, to verify their expected behavior in terms of time complexity.

Key words

LiquidHaskell, refinement types, static verification, proof assistants, equational reasoning, resource analysis

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή της εργασίας, κ. Νικόλαο Παπασπύρου.

Παναγιώτης Διαμαντάκης,

Αθήνα, 13η Οκτωβρίου 2021

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-5-21, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2021.

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

Περιεχόμενα

Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός της εργασίας

Σκοπός της εργασίας είναι η τυπική επαλήθευση ιδιοτήτων κατανάλωσης πόρων, σε γνωστούς αλγόριθμους [Appe21]. Για την επαλήθευση θα χρησιμοποιηθεί η LiquidHaskell(LH), ένας verifier για προγράμματα γραμμένα σε Haskell, ο οποίος ελέγχει τις αναλλοίωτες που παρέχει ο χρήστης μέσω ενός SMT solver. Ειδικότερα, για την καταγραφή των πόρων θα αξιοποιήσουμε την βιβλιοθήκη RTick [Hand19], η οποία στηρίζεται στην ιδέα των 'ticks'. Ο χρήστης μπορεί να προσθέτει 'ticks' σε ένα πρόγραμμα, όπου θεωρεί ότι καταναλώνονται οι πόροι που θέλει να καταγράψει (απομάκρυνση στοιχείων λίστας, αναδρομικές κλίσεις, αναθέσεις μνήμης, κλπ), και στην συνέχεια να αποδείξει ότι το πλήθος των 'ticks' φράσσεται από κάποια συγκεκριμένη συνάρτηση των παραμέτρων εισόδου (π.χ. το μήκος της λίστα εισόδου).

1.2 Σύνοψη

Στο υπόλοιπο της εργασίας θα αναλυθούν τα εξής:

Κεφάλαιο 2 : θα παρουσιάσουμε την LiquidHaskell, τις αρχές λειτουργίας της, και πως μπορεί να την αξιοποιήσει ο χρήστης.

Κεφάλαιο 3 : θα εισάγουμε την βιβλιοθήκη RTick, η οποία επιτρέπει την ανάλυση κόστους. Αρχικά, παρουσιάζουμε τον τύπο δεδομένων `Tick` και τις συναρτήσεις που επιτρέπουν την ανάθεση κόστους στα προγράμματα προς εξέταση. Επίσης, παρουσιάζουμε χρήσιμους τελεστές για την απόδειξη θεωρημάτων ανάλυσης κόστους.

Κεφάλαιο 4 : θα δούμε πολλά παραδείγματα γνωστών αλγορίθμων, γραμμένα σε Haskell, στα οποία, αξιοποιώντας τις πληροφορίες που αναπτύχθηκαν μέχρι εκείνο το σημείο, έχει γίνει επαλήθευση της πολυπλοκότητά τους, με την βοήθεια της LH.

Κεφάλαιο 5 : έχοντας ολοκληρώσει το κυρίως μέρος της εργασίας, παρουσιάζουμε τα συμπεράσματα που προέκυψαν και προτάσεις για μελλοντική έρευνα

Κεφάλαιο 2

Βασικές Έννοιες στην LiquidHaskell

Η LiquidHaskell (LH) επεκτείνει τους τύπους της Haskell με λογικά κατηγορήματα, επιτρέποντας έτσι την απόδειξη προ- και μετα-συνθηκών, την αναγνώριση αναλλοίωτων και την επαλήθευση ιδιοτήτων συναρτήσεων, κατά την μεταγλώττιση του κάθε προγράμματος [Rond08].

Αντίστοιχες γνωστές γλώσσες - συστήματα υποστήριξης αποδείξεων (interactive theorem provers) όπως η Agda και η Coq, βασίζονται στους *Dependent Types* (εξαρτώμενους τύπους). Τέτοια συστήματα μπορούν να προσφέρουν αυξημένη ασφάλεια και καλύτερη επίδοση (εφόσον παρακάμπτεται ο έλεγχος των refinement κατά τον χρόνο εκτέλεσης). Παρόλα αυτά, ένα σύστημα εξαρτώμενων τύπων που δεν είναι αποκρίσιμο στερεί την δυνατότητα υποστήριξης *αυτόματου Type Inference*. Στην γενική περίπτωση, ο έλεγχος εξαρτώμενων τύπων μπορεί να οδηγεί στον έλεγχο ισότητας εκφράσεων, το οποίο είναι μη-αποκρίσιμο πρόβλημα.

Αντίθετα, στους Liquid Types, *(Logically Qualified Data Types)*, ο συμπερασμός τύπων είναι αποκρίσιμος, καθώς περιορίζεται η εκφραστικότητα των refinements. Τα υποψήφια κατηγορήματα που περιγράφουν τις διάφορες προ- και μετά- συνθήκες των προγραμμάτων πρέπει να είναι πεπερασμένου πλήθους, ώστε μια εξαντλητική αναζήτηση στον χώρο λύσεων να μπορεί φέρει αποτέλεσμα, δοκιμάζοντας όλους τους συνδυασμούς. Αν και στην πράξη η LH επιλέγει πιο αποδοτικές μεθόδους, αφού ακόμα και σε μικρά προγράμματα η brute force προσέγγιση θα οδηγούσε σε χρονοβόρο έλεγχο[Peñ17].

2.1 Γράφοντας Specifications

Τα refinement types, στα οποία η LH στηρίζει την λειτουργία της, μας επιτρέπουν να επεκτείνουμε το σύστημα τύπων της γλώσσας με κατηγορήματα (predicates), περιορίζοντας έτσι το σύνολο των στοιχείων που περιλαμβάνει ο αρχικός τύπος [Jhal20]. Για παράδειγμα, ενώ `Int` μπορεί να είναι κάθε ακέραιος αριθμός, μπορούμε να ορίσουμε το refinement των μη-αρνητικών ακεραίων ως εξής:
`type Nat = { v : Int | v >= 0 }`

Προκειμένου να παραχθούν οι κατάλληλοι refinement types, ο προγραμματιστής θέτει στις συναρτήσεις του τα ζητούμενα specifications (προδιαγραφές) που θα πρέπει ικανοποιούνται. Παρακάτω φαίνονται οι βασικότεροι τρόποι που μπορούμε να κάνουμε annotate σε νέο ή υπάρχοντα κώδικα, για να εκφράσουμε τα specifications που θέλουμε.

- **Inline :** η εντολή `{-@ inline <binding-name> @-}` αντιγράφει τον ορισμό του ονόματος (συνάρτησης) που δεσμεύει στην refinement logic.

  Με αυτόν τον τρόπο το `inline` σου επιτρέπει να χρησιμοποιείς μια Haskell συνάρτηση σε ένα refinement type, επιτρέποντας μεγαλύτερη εκφραστικότητα στα specifications.

```
{-@ inline max @-}
{-@ max :: Int -> Int -> Int @-}
max :: Int -> Int -> Int
max x y = if x > y then x else y
```

Μπορείς για παράδειγμα να χρησιμοποιήσεις την συνάρτηση max στο refinement type , και αυτή αυτόματα θα "αναπτυχθεί" σύμφωνα με τον ορισμό της. Έτσι, γίνεται εύκολη η επαναχρησιμοποίηση απλού κώδικα Haskell για την σύνθεση specifications, και την χρήση κοινού κώδικα τόσο στους ορισμούς όσο και στα refinements.

```
{-@ floor :: x:Int -> {v:Int | max 0 x} @-}
floor :: Int -> Int
floor x
  | x <= 0    = 0
  | otherwise = x
```

– Όλα τα μέρη του ορισμού πρέπει να είναι ήδη ορισμένα στην refinement logic.

– Ο ορισμός δεν μπορεί να είναι αναδρομικός.

Ωστόσο, η εν λόγω 'ανάπτυξη' γίνεται κατά την μεταγλώττιση, και επομένως οι inline συναρτήσεις δεν μπορούν να είναι αναδρομικές. Μπορούν να καλούν άλλες (μη-αναδρομικές) inline συναρτήσεις.

Την ανάγκη για χρήση γενικότερων (αναδρομικών) συναρτήσεων στους τύπους, καλύπτουν οι reflected συναρτήσεις που θα δούμε παρακάτω.

- **Measure :** το `{-@ measure <function-name>[ <refinement-type>] @-}` αντιγράφει την συνάρτηση στην refinement logic, προσθέτει έναν εξαγώμενο refinement τύπο στον constructor του πρώτου ορίσματος της συνάρτησης, και παράγει μία global αναλλοίωτη που ανταποκρίνεται στο refinement.

  - Όλα τα μέρη του ορισμού πρέπει να είναι ήδη ορισμένα στην refinement logic.

  - Η συνάρτηση πρέπει να έχει μόνο ένα όρισμα, και μάλιστα θα πρέπει να πρέπει να υλοποιείται με ταίριασμα προτύπων (pattern matching) στους constructors του τύπου του.

  Σε αντίθεση με το `inline` η συνάρτηση μπορεί να υλοποιείται με αναδρομή στο μοναδικό της όρισμα.

  Με την χρήση των `measures`, μπορούν να συμπεριληφθούν στην refinement logic αναδρομικές συναρτήσεις, όπως η :

```
{-@ measure llen @-}
llen        :: [a] -> Int
llen []     = 0
llen (x:xs) = 1 + llen xs
```

  Ο παραπάνω ορισμός:

  – 'εκλεπτύνει' τους κατασκευαστές του τύπου λίστας με την πληροφορία llen,

  – και επιπλέον 'προσδιορίζει' τον τύπο επιστροφής της llen : `{v:Int | v == llen xs}`.

  Στην συνέχεια ο χρήστης μπορεί να 'προσδιορίσει' περαιτέρω τον τύπο επιστροφής, π.χ. σε `{v:Int | llen xs >= 0}`, και να αντικαταστήσει τον προηγούμενο τύπο που παράχθηκε αυτόματα.

- **Reflect :** η `{-@ reflect <function-name> @-}` δημιουργεί μία συνάρτηση χωρίς ερμηνεία (uninterpreted) με το ίδιο όνομα στην refinement logic, αντιγράφει την υλοποίηση σε ένα συνώνυμο refinement type (alias), και τέλος προσθέτει ένα refinement στον τύπο της ανερμήνευτης συναρτησης ώστε να επιβάλει τον συνώνυμο τύπο (type alias) ως μετα-συνθήκη (post-condition).

  Όλα τα επιμέρους στοιχεία της υλοποίησης πρέπει να είναι ήδη διαθέσιμα στην refinement logic. Η συνάρτηση μπορεί να είναι αναδρομική.

  Περισσότερα για το 'reflection' θα δούμε αναλυτικά σε επόμενη ενότητα.

- **Types :** γράφοντας την έκφραση `{-@ type <type-alias-head> = <refinement-type> @-}` εισάγεται ένας συνώνυμος τύπος (type alias) που συντακτικά μοιάζει με τύπο Haskell αλλά μπορεί επιπλέον να περιλαμβάνει refinements, και μπορεί επίσης να είναι παραμετροποιήσιμος ως προς τους τύπους και τις τιμές.

  Προκειμένου να γράφουμε πιο σύντομα refinements, η LiquidHaskell υποστιρίζει (refinement) type aliases. Έτσι, μπορούμε για παράδειγμα να ορίσουμε το `{-@ type Gt N = {v: Int | N < v} @-}`, ώστε αν εμφανίζεται συχνά στον κώδικα το specification για θετικούς ακεραίους `{v: Int | v > 0}`, μπορούμε να το ορίσουμε συντομότερα ως `{Gt 0}`. Αντίστοιχα, μία συνάρτηση `incr` που επιστρέφει την τιμή εισόδου αυξημένη, μπορεί να έχει signature της μορφής `{-@ incr:: x: Int -> Gt x @-}`.

- **Data :** η έκφραση `{-@ data <data-type-head><termination-measure>[ <data-type-body] @-}` εισάγει ένα refined datatype, και measures για κάθε πεδίο στην εγγραφή του τύπου.

  Για παράδειγμα, για να συμπεριληφθεί στην refinement logic ο τύπος δεδομένων :

```
data LL a = BXYZ { size  :: Int
                 , elems :: [a]
                 }
```

  μπορούμε να χρησιμοποιήσουμε το specification :

```
{-@ data LL a = BXYZ { size  :: {v: Int | v > 0 }
                     , elems :: {v: [a] | (len v) = size }
                     }
@-}
```

  με το οποίο πετυχαίνουμε, να αναθέσουμε στα πεδία τις επιθυμητές προδιαγραφές, ορίζοντας, ο ακέραιος `size` να είναι μη-αρνητικός, και η λίστα `elems` να είναι μεγέθους ίσου με τον ακέραιο.

  - Προαιρετικά, μπορούν να μπουν refinements στα πεδία.

  - Επίσης, προαιρετικά μπορεί να προστεθεί στον τύπο ένα τερματικό (termination) measure.

- **Assume :** το `{-@ assume <binding-signature-with-refinement-type> @-}` εισάγει ένα refinement type για τον αντίστοιχο Haskell ορισμό που δεσμέυει, χωρίς να ελέγχεται η ορθότητά του. - Στην περίπτωση συνάρτησης, τα refinements γίνονται προ- και μετά- συνθήκες, στο περιβάλλον το οποίο γίνεται χρήση της.

- Το `{-@ <binding-signature-with-refinement-type> @-}` εισάγει έναν refinement type για τον αντίστοιχο Haskell ορισμό που δεσμέυει.

  Στην περίπτωση που δεσμεύει συνάρτηση, τα refinements γίνοτναι προ- και μετα- συνθήκες για τις συναρήσεις που την χρησιμοποιούν

  Αυτό είναι ίσως το πιο συχνό Liquid Haskell σχόλιο (annotation). Άλλωστε, τελικό στόχο του χρήστη αποτελεί η επαλήθευση των invariants κάθε συνάρτησής του, τα οποία αποτυπώνονται συνήθως ως μετα-συνθήκες.

2.2 Verification και SMT Solvers

Η LH κάνει την επαλήθευση των ζητούμενων ιδιοτήτων με στατικό έλεγχο, πριν την εκτέλεση του προγράμματος [Vazo14]. Αρχικά, συνδυάζει τους τύπους (με τα refinements) και τον αντίστοιχο κώδικα, για να ορίσει ένα σύνολο από κατηγορήματα που μπορούν να είναι αληθή μόνο αν το πρόγραμμα ικανοποιεί τα ζητούμενα specifications. Οι λογικές προτάσεις που παράγονται από την παραπάνω διαδικασία, ονομάζονται Verification Conditions (VC), και στην συνέχεια ένας SMT solver απαιτείται να ελέγξει την εγκυρότητά τους. Θέλουμε η επαλήθευση των VCs να γίνεται αποδοτικά από τους

solver. Για αυτό, το σύστημα τύπων των Liquid Types, έχει σχεδιαστεί με κατάλληλο τρόπο, ώστε να παράγει μόνο προτάσεις που ανήκουν σε αποκρίσιμες λογικές.

Ένας SMT (Satisfibility Modulo Theories) solver χρησιμοποιείται για τον έλεγχο της ικανοποιησιμότητας μιας λογικής πρότασης, ως προς μία ή περισσότερες θεωρίες. Στην περίπτωση της LH, ελέγχονται προτάσεις από αποκρίσιμες λογικές, οπότε αρκεί να περιορίσουμε τον solver στην QF-EUFLIA λογική (quantifier-free logic of equality, uninterpreted functions and linear integer arithmetic [Vazo16]).

Ουσιαστικά, μια θεωρία είναι ένα σύνολο προτάσεων. Πιο τυπικά, μια $\Sigma-$θεωρία είναι μια συλογή από προτάσεις υπό την υπογραφή Σ. Έτσι δεδομένης μιας θεωρίας T, λέμε ότι το ϕ είναι *ικανοποιήσιμο modulo T (satisfiable modulo T)*, αν είναι ικανοποιήσιμο το σύνολο : $T \cup \{\phi\}$. Για παράδειγμα, έστω Σ η υπογραφή που περιλαμβάνει τα σύμβολά 1, +, - και <, και $\mathbb{Z}$ η δομή που ερμηνεύει αυτά τα σύμβολα με τον συνηθισμένο τρόπο στους ακεραίους, τότε η θεωρία της γραμμικής αριθμητικής (linear arithmetic) είναι το σύνολο προτάσεων πρωτοβάθμιας λογικής που ισχύουν στο $\mathbb{Z}$. Θα λέμε ότι το πρόβλημα της ικανοποιησιμότητας για μια θεωρία είναι αποκρίμιμο (decidable) εάν υπάρχει διαδικασία $\mathcal{G}$ που ελέγχει αν κάθε φόρουλα χωρίς ποσοδείκτες (quantifier-free formula) είναι ικανοποιήσιμη ή όχι. Σε αυτήν την περίπτωση λέμε ότι η $\mathcal{G}$, είναι μια *διαδικασία απόφασης (decision procedure)* για την T.

Όπως είπαμε και παραπάνω, η LH περιορίζεται στην QF-EUFLIA λογική, που περιλαμβάνει τις παρακάτω θεωρίες: Βλ. εδω: [dMou09]

- **Linear Arithmetic**. Η γραμμική αριθμητική(linear arithmetic), γνωστή και ως προσθετική (additive) αριθμητική, είναι η θεωρία όπου οι μόνες αριθμητικές συναρτήσεις είναι η πρόσθεση και η αφαίρεση. Οι συναρτήσεις μπορούν να εφαρμοστούν τόσο σε αριθμητικές σταθερές, όσο και σε μεταβλητές. Επίσης επιτρέπεται ο πολλαπλασιασμός με αριθμητικές σταθερές, τόσο ακέραιες (π.χ. $5 \cdot x$) , όσο και πραγματικές (π.χ. $\frac{2}{3} \cdot x$). Τα κατηγορήματα σχηματίζονται από την σύζευξη ατόμων, που εκφράζουν σχέσεις ισότητας και ανισότητας (=, $\leq$, <).

- **Free functions**. Η 'ελεύθερη' θεωρία σε μία υπογραφή Σ είναι η θεωρία της πρωτοβάθμιας λογικής, με ένα κενό σύνολο προτάσεων. Είναι γνωστή και ως θεωρία των ανερμήνευτων(uninterpreted) συναρτήσεων. Οι διαδικασίες απόκρισης για αυτή τη θεωρία είναι ιδιαίτερα σημαντικές αφού το πρόβλημα απόφασης για πολλές άλλες θεωρίες (e.g., arrays) ανάγεται τελικά σε αυτήν την θεωρία. Δεδομένης μιας σύζευξης από ισότητες μεταξύ όρων που χρησιμοποιούν ελεύθερες συναρτήσεις, μπορεί να χρησιμοποιηθεί μια κλειστότητα σύμπτωσης(congruence closure) για την αναπαράσταση του ελάχιστου συνόλου που υπονοείται από τις ισότητες. Αυτή η αναπαράσταση μπορεί να χρησιμοποιηθεί για να ελεγχθεί η ικανοποιησιμότητα ενός συνδιασμού από ισότητες και ανισότητες. Αρκεί να ελέγξουμε ότι οι όροι στις δύο πλευρές κάθε ανισότητας ανήκουν σε διαφορετικές κλάσεις ισοδυναμίας. Αποδοτικοί αλγόριθμοι για υπολογισμό τέτοιων κλειστοτήτων σύμπτωσης έχουν μελετηθεί αρκετά στο [Down80].

Οι παραπάνω θεωρίες, σε συνδυασμό με τον περιορισμό οι φόρμουλες να μην περιλαμβάνουν υπαρξιακούς και καθολικούς ποσοδείκτες ($\exists$ και $\forall$) συνθέτουν την QF-EUFLIA λογική. Άλλες γνωστές θεωρίες, που όμως δεν σχετίζονται με την επαλήθευση των Verification Conditions της LH, είναι οι **difference arithmetic, non-linear arithmetic, bit vectors, arrays** κ.α.

2.3 Αποδείξεις

Η LH προσφέρει δύο τρόπους για την απόδειξη ιδιοτήτων και θεωρημάτων. Ο πρώτος τρόπος, το **lightweight reasoning**, αφορά τις περιπτώσεις όπου μπορεί να γίνει πλήρως αυτοματοποιημένα η απόδειξη, με την βοήθεια του SMT solver.

Τέτοιες ιδιότητες είναι συνήθως οι απλές αριθμητικές (linear arithmetics), όπως για παράδειγμα, ότι η (αναδρομική) συνάρτηση `length`, που επιστρέφει το μήκος μιας λίστας, δεν μπορεί να επιστρέψει αρνητικό ακέραιο.

Επιπλέον, μπορούμε να ενισχύσουμε τις refinement εκφράσεις, περιλαμβάνοντας, εκτός από αριθμητικές εκφράσεις, και ονόματα συναρτήσεων. Για να μπορέσει μία συνάρτηση να συμπεριληφθεί σε ένα refinement, θα πρέπει να ανήκει στην οικογένεια συναρτήσεων measures, που είδαμε και στο κεφάλαιο 2.1. Θα πρέπει να είναι, λοιπόν, συναρτήσεις με μία μόνο παράμετρο (η οποία να είναι μάλιστα algebraic data type), οι οποίες για κάθε κατασκευαστή του τύπου ορίζονται από μία μόνο 'εξίσωση΄, και οι οποίες κάνουν κλήσεις μόνο σε αριθμητικές συναρτήσεις ή άλλες συναρτήσεις measures.

Χαρακτηριστικό παράδειγμα, είναι η συνάρτηση, `length`, η οποία εφόσον δηλωθεί κατάλληλα, ως measure, επιτρέπει πλέον την αυτόματη απόδειξη, πιο χρήσιμων ιδιοτήτων, όπως για παράδειγμα, ότι η συνάρτηση συνένωσης `(++)` , επιστρέφει μία λίστα μήκους όσο το άθροισμα των δύο αρχικών λιστών :

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

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

Επειδή οι measure συναρτήσεις αποτελούν συνήθως μόνο ένα μικρό μέρος των συναρτήσεων που ορίζει ένας χρήστης, υπάρχουν και άλλοι τρόποι να συμπεριλάβεις συναρτήσεις στα refinments. Για τις υπόλοιπες συναρτήσεις, λοιπόν, η LH μας επιτρέπει να κάνουμε 'reflect' και να πετύχουμε πιο εκφραστικά refinements. Σε αυτήν την περίπτωση, βέβαια, μιλάμε για πιο σύνθετες ιδιότητες, τις οποίες ο SMT solver δεν μπορεί να αποδείξει αυτόματα, οπότε χρειάζεται να γίνει **deep reasoning**.

Έτσι, αν ήθελε ο χρήστης να αποδείξει για την συνάρτηση `reverse` (η οποία δεν είναι measure) ότι reverse [x] == [x], θα έπρεπε πρώτα να δηλώσει τις συναρτήσεις `reverse` και `(++)` ως reflected:

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

Στην συνέχεια θα έπρεπε να εισάγει μία συνάρτηση `singletonP` με refinement τύπο που να εκφράζει την ζητούμενη ιδιότητα:

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

singletonP x
  = reverse [x]
    -- applying reverse on [x]
  ==. reverse [] ++ [x]
    -- applying reverse on []
  ==. [] ++ [x]
    -- applying ++ on [] and [x]
  ==. [x]
  *** QED
```

Παρακάτω, θα εξηγήσουμε τους τελεστές αποδείξεων (proof combinators) όπως αυτούς που εμφανίζονται στο παράδειγμα της `singletonP`, και που είναι ιδιαίτερα χρήσιμοι για τον σχηματισμό αποδείξεων σε μη-τετριμμένες ιδιότητες.

Πρώτα, όμως, να διευκρινίσουμε ότι η παραπάνω απόδειξη, είναι ισοδύναμη με την πολύ πιο συμπυκνωμένη :

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

Επομένως, το module Equational[1], δίνει την δυνατότητα οι 'αποδείξεις' να είναι πιο ευανάγνωστες, αλλά και πιο εύκολες στην σύνταξη, αφού παραπέμπουν σε αποδείξεις που γίνονται με χαρτί και μολύβι.

Το πρώτο πράγμα που ορίζεται στο module, είναι ο τύπος της απόδειξης, που όπως φαίνεται παρακάτω, απότελει απλά ένα συνόνυμο (alias) για τον unit type:

```
type Proof = ()
```

Ο λόγος που επιλέγεται αυτός ο τύπος, είναι ότι δεν ενδιαφερόμαστε για την τιμή επιστροφής που θα είχε μια συνάρτηση-θεώρημα, αλλά μόνο για το αν μπορεί τελικά τα επαληθευτεί η ιδιότητα του refinement.

Ένας από τους τελεστές που χρησιμοποιούνται σε κάθε απόδειξη, είναι ο `(***)`, ο οποίος δέχεται οποιαδήποτε έκφραση, και ανεξάρτητα από την τιμή της, επιστρέφει μία 'απόδειξη' (για την ακρίβεια, επιστρέφει την μοναδική τιμή που μπορεί να πάρει ο τύπος `Proof`, δηλαδή την `()`):

```
data QED = QED

(***) :: a -> QED -> Proof
_ *** QED = ()
infixl 2 ***
```

Σαν δεύτερο όρισμα, όπως είδαμε ο τελεστής παίρνει την τιμή `QED`, και ο λόγος είναι καθαρά αισθητικός - να ολοκληρώνονται δηλαδή οι αποδείξεις με `*** QED` ("quod erat demonstrandum").

Δύο ακόμα χρήσιμοι τελεστές, που θα τους συναντήσουμε και στο κεφάλαιο 4 εκτενώς είναι οι `(==.)` και `(?)`. Ο πρώτος τελεστής δέχεται δύο ορίσματα και από τον refinement τύπο δεσμεύει τα δύο ορίσματα να έχουν ίδια τιμή. Τελικά, η τιμή επιστροφής είναι το δεύτερο όρισμα, έτσι ώστε να μπορεί ο χρήστης να χρησιμοποιήσει, αλυσιδωτά, πολλές διαδοχικές φορές τον τελεστή :

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

(==.) :: a -> a -> a
_ ==. x = x
```

Σχετικά με τον τελεστή 'αιτιολόγησης' (?), το εφαρμόζουμε όταν θέλουμε να αναφέρουμε στην απόδειξή μας ένα άλλο θεώρημα :

```
(?) :: a -> Proof -> a
x ? _ = x
```

Η χρησιμότητά του φαίνεται στο παρακάτω παράδειγμα, όπου για να αποδείξουμε το θεώρημα `singleton1P`, επιλέγουμε να επικαλεστούμε το ήδη γνωστό θεώρημα `singleton1P` ως εξής :

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

Να σημειώσουμε ότι παρότι ο ? τελεστής είθισται να μπαίνει δίπλα στην εξίσωσή που θέλουμε να αιτιολογήσουμε, η τοποθέτησή του είναι πρακτικά αδιάφορη, εφόσον όλο το σώμα μιας απόδειξης ελέγχεται με την μία.

[1] https://github.com/ucsd-progsys/liquidhaskell/blob/develop/include/Language/Haskell/Liquid/Equational.hs

2.3.1 Reflection

Για να ενεργοποιήσουμε την ”απόδειξη θεωρημάτων” στην LH προσθέτουμε σε κάθε αρχείο τα flags : `--reflection` και `--ple`. Το reflection σύμφωνα με το [Vazo17b] στηρίζεται στην ιδέα να αντικατοπτρίζεται η υλοποίηση της ζητούμενη συνάρτησης, στον τύπο εξόδου. Έτσι μπορούμε να ξεφύγουμε από τα απλά predicates τύπου : `type Nat = {v:Int | 0 <= v}` και `Even = {v:Int | v mod 2 == 0}`, και να επαληθεύσουμε πιο ενδιαφέρουσες ιδιότητες.

Η διαδικασία αυτή απαιτεί τρία στάδια. Για παράδειγμα, αν θέλουμε να ασχοληθούμε με την γνωστή συνάρτηση fib, η διαδικασία έχει ώς εξής. Στο πρώτο στάδιο **(definition)** ο SMT solver το μόνο που γνωρίζει για την συνάρτηση είναι ο τύπος της, και έτσι το μόνο που μπορεί να εξάγει είναι το $\forall m, n. m = n \implies \texttt{fib}\ m = \texttt{fib}\ n$. Στο δεύτερο στάδιο **(reflection)** ο τύπος της fib ενισχύεται : `{-@ fib :: n:Nat -> { v:Nat | v = fib n && fibP n } @-}` με την προσθήκη του κατηγορήματος (predicate) `fibP`, το οποίο παραγεται αυτόματα από την LH και αντανακλά την υλοποίηση της αρχικής συνάρτησης όπως φαίνεται και στο παράδειγμα της `fib`:

```
fibP n =
    n == 0 ==> fib n = 0
    && n == 1 ==> fib n = 1
    && n > 1 ==> fib n = fib (n-1) + fib (n-2)
```

Το τρίτο και τελευταίο στάδιο **(application)** έχει να κάνει με την εφαρμογή ορισμάτων στην συνάρτηση. Κάθε εφαρμογή ορίσματος αναπτύσσει τον ορισμό της `fib`. Έτσι η απόδειξη της σχέσης $\texttt{fib}\ 2 = 1$, επαληθεύεται με την απλή συνάρτηση

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

αφού ο SMT solver έχει παράξει τα τρία κατάλληλα predicates, και καλείται να αποδείξει την αντίστοιχη σχέση : $(\texttt{fibP}\ 0 \wedge \texttt{fibP}\ 1 \wedge \texttt{fibP}\ 2) \implies \texttt{fib}\ 2 = 1$.

2.3.2 Proof by Logical Evaluation

Με τον αλγόριθμο PLE(Proof by Logical Evaluation), ο οποίος είναι μη-πλήρης αλλά τερματιζόμενος, μας δίνεται η δυνατότητα να αυτοματοποιήσουμε τετριμμένες αποδείξεις [Vazo17a].

Αρχικά, κάθε reflected συνάρτηση μετασχηματίζεται στο αντίστοιχο guard normal form : $\wedge_i(p_i \implies f(\bar{x}) = b_i)$, όπου p_i το guard, και b_i ο ορισμός του αντίστοιχου κατηγορήματος.

Στην συνέχεια, για την απόδειξη μιας συνθήκης $\Phi \implies p$, αναπτύσσουμε τον ορισμό της συνάρτησης, όπου της εφαρμόζεται όρισμα. Εκεί είναι απαραίτητη η χρήση του SMT solver για να “ενεργοποιηθεί” ο σωστός ορισμός της συνάρτησης, που ανταποκρίνεται στο εφαρμοζόμενο όρισμα. Έτσι, στο παράδειγμα της συνάρτησης `fibonacci`, για την συνθήκη $\texttt{true} \implies \texttt{fib}\ 3 = 2$, ενεργοποιείται ο φρουρός (guard) : $3 > 1$, και επομένως η υπόθεση ενισχύεται με την προσθήκη της σχέσης $\texttt{fib}\ 3 = \texttt{fib}(3-1) + \texttt{fib}(3-2)$.

Τελικά, η διαδικασία ολοκληρώνεται όταν φτάσουμε σε ένα fixpoint, όπου η διαδικασία ανάπτυξης δεν γίνεται να συνεχιστεί. Στο συγκεκριμένο παράδειγμα, ο υπολογισμός του `fib 3`, αναπτύσσει του ορισμούς στα 3, 2, 1, 0 και σταματάει αφού δεν υπάρχει άλλος ”φρουρός” να ενεργοποιηθεί.

2.4 Ελεγχος για Totality και Termination

Η χρήση μιας αναδρομικής συνάρτησης για το σχηματισμό μιας επαγωγής, μπορεί να οδηγήσει σε μη ορθή απόδειξη, αν η συνάρτηση είναι μερική ή δεν τερματίζει η εκτέλεσή της. Για αυτό το λόγο, η LH παρέχει έναν ισχυρό ελεγκτή πληρότητας και τερματισμού, και απορίπτει κάθε ορισμό για τον οποίο δεν μπορεί να επαληθεύσει αυτές τις δύο ιδιότητες. [Vazo18]

2.4.1 Totality

Η Liquid Haskell χρησιμοποιεί τον μηχανισμό pattern completion του μεταγλωττιστή GHC για εξασφαλίσει ότι όλες οι συναρτήσεις είναι πλήρεις.

Για παράδειγμα, αν η `involutionP` ήταν ορισμένη μόνο για την κενή λίστα,

```
involutionP :: [a] -> Proof
involutionP [] = ()
```

τότε θα εμφανιζόταν μήνυμα σφάλματος επειδή η συνάρτηση είναι μερική ('Your function isn't total : some patterns aren't defined').

Ο GHC αυτό το πετυχαίνει συμπληρώνοντας αυτόματα στον ορισμό της involutionP μια κλήση στην συνάρτηση patError :

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

Η LH ενεργοποιεί τον έλεγχο πληρότητας βάζοντας στο refinement της patError την ψευδή προσυνθήκη:

```
{-@ patError :: { i:String | False }->a @-}
```

Επειδή, λοιπόν, δεν υπάρχει όρισμα που να ικανοποιεί την ψευδή συνθήκη, όταν η κλήση στην patError δεν μπορεί να αποδειχθεί νεκρός κώδικας (dead code) η LH εμφανίζει totality error.

2.4.2 Termination

Η LH ελέγχει ότι όλες οι συναρτήσεις τερματίζουν την εκτέλεσή τους, είτε μέσω δομικού (structural) είτε μέσω σημασιολογικού (semantic) ελέγχου.

Structural Termination

Ο δομικός έλεγχος τερματισμού είναι πλήρως αυτοματοποιημένος και εντοπίζει το κοινό μοτίβο αναδρομής όπου το όρισμα της αναδρομικής κλήσης είναι, έμμεσα ή άμεσα, υποόρος του ορίσματος της αρχικής συνάρτησης, όπως συμβάνει χαρακτηριστικά με την συνάρτηση `length`.

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

Στην περίπτωση που η συνάρτηση έχει πολλαπλά ορίσματα, τουλάχιστον ένα όρισμα πρέπει να μειώνεται, ενώ όλα τα υπόλοιπα πριν από αυτό πρέπει να μένουν σταθερά (lexicographic order).

Υπάρχουν πολλές συναρτήσεις που καλύπτει ο δομικός έλεγχος, και δεν χρειάζεται ο χρήστης να ασχοληθεί με αποδείξεις τερματισμού. Ωστόσο, από ένα σημείο και πέρα, ο χρήστεις δεν μπορεί παρά να στραφεί στην τεχνική του σημασιολογικού τερματισμού.

Semantic Termination

Όταν ο δομικός έλεγχος τερματισμού αποτυγχάνει, η LH δοκιμάζει να αποδείξει τον τερματισμό μέσω ενός σημασιολογικού επειχήματος: μία έκφραση δηλαδή, που υπολογίζει έναν φυσικό αριθμό από το όρισμα της συνάρτησης, και ο οποίος να μειώνεται σε κάθε αναδρομική κλήση. Μπορούμε να χρησιμοποιήσουμε αυτόν τον έλεγχο τερματισμού για την παρακάτω απόδειξη που αφορά την `involutionP`, με την σύνταξη `/ [length xs]`:

```
{-@ involutionP :: xs:[a]->{reverse (reverse xs) == xs}/ [length xs] @-}
```

Ένα επιχείρημα τερματισμού είναι της μορφής `/ [e1, ..., en]`, όπου οι εκφράσεις e_i εξαρτόνται από τα ορίσματα της συνάρτησης, και παράγουν φυσικούς αριθμούς. Θα πρέπει να μειώνονται λεξικογραφικά (lexicographically) σε κάθε αναδρομική κλήση. Αυτοί οι περιορισμοί ελέγχονται από

τον SMT solver, μαζί με όλα τα refinement types της συνάρτησης. Όταν ο χρήστης δεν ορίζει συγκεκριμένο termination metric, και ο δομικός έλεγχος τερματισμού αποτυγχάνει, η LH προσπαθεί να μαντέψει ένα termination metric όπου το πρώτο όρισμα (που δεν είναι συνάρτηση) μειώνεται.

Ο σημασιολογικός έλεγχος έχει δύο κύρια οφέλη σε σχέση με τον δομικό. Αφενός, δεν είναι όλες οι συναρτήσεις δομικά αναδρομικές, και για να μετατρέψεις μία - προσθέτοντας επιπλέον παραμέτρους - μπορεί να οδηγήσει μακροσκελή και δυσανάγνωστο κώδικα. Αφετέρου, εφόσον ο έλεγχος τερματισμού γίνεται από τον SMT solver, μπορεί να γίνει αξιοποίηση και των ιδιοτήτων που προέρχονται από τα refinements των ορισμάτων εισόδου(inputs). Όσον αφορά τα μειονεκτήματα του σημασιολογικού ελέγχου, μπορεί να είναι κουραστικό για τον χρήστη να δηλώνει ρητά το termination metric (π.χ. το length για την περίπτωση λίστας), ενώ παράλληλα αχρείαστες κλήσεις στον SMT solver, όταν το επιχείρημα τερματισμού είναι τετριμμένο, στοιχίζουν στον συνολικό χρόνο μεταγλώττισης.

Uncaught termination

Επειδή η Haskell είναι αμιγώς συναρτησιακή, οι μόνες παρενέργειες που επιτρέπει είναι η μησύγκλιση των αναδρομικών κλίσεων και τα ατελή ταιριάσματα τύπων(incomplete patterns). Εάν αποκλείσουμε και τις δύο αυτές παρενέργειες, μέσω ελέγχων τερματισμού και πληρότητας, ο χρήστης θα είναι βέβαιος ότι έχει πλήρεις συναρτήσεις, και άρα σωστά κωδικοποιημένες μαθηματικές αποδείξεις.

Δυστυχώς, η δημιουργική αξιοποίηση χαρακτηριστικών της Haskell, συγκεκριμένα αναδρομικών τύπων που εμφανίζονται σε αρνητικές θέσεις ("type recursion must be covariant" [Turn04]) και ανώτερης τάξης τύπων, όπως στο παρακάτω παράδειγμα[2],

```
module Evil where

data Evil a = Very (Evil a -> a)

{-@
bad :: Evil { _:() | 0 > 1} -> { _:() | 0 > 1} @-}
bad :: Evil () -> ()
bad (Very f) = f (Very f)

{-@
worse :: { _:() | 0 > 1} @-}
worse :: ()
worse = bad (Very bad)
```

επιτρέπουν την συγγραφή μη-τερματιζόμενων συναρτήσεων, που ωστόσο, περνάνε τον έλεγχο της Liquid Haskell. Χρειάζεται, λοιπόν, προσοχή από τον χρήστη, όταν επιλέγει τέτοιους τύπους, μέχρις ότου ο τερματικός έλεγχος να μπορέσει να καλύψει και αυτές τις ιδιαίτερες περιπτώσεις, ανάλογα με τις σχεδιαστικές επιλογές των δημιουργών[3].

[2] https://liquidhaskell.slack.com/archives/C54QAL9RR/p1615244385057800
[3] https://github.com/ucsd-progsys/liquidhaskell/issues/159

Κεφάλαιο 3

Ανάλυση Κόστους

3.1 Υλοποίηση

Ο κύριος τύπος στον οποίο βασίζεται η ανάλυση κόστους της εργασίας είναι ο παραμετρικός τύπος Tick a, που αποτελείται από έναν ακέραιο που καταγράφει την κατανάλωση, και ένα πεδίο τύπου a:

```
data Tick a = Tick { tcost :: Int, tval :: a }
```

Με αυτόν τον τρόπο, στις περιπτώσεις που θέλουμε να επαληθεύσουμε ιδιότητες κόστους, θα τροποποιούμε τους κατάλληλους τύπους δεδομένων από τύπο a σε `Tick a`, και στην συνέχεια στο refinement θα εκφράζουμε τις ζητούμενες ιδιότητες με βάση το `tcost` πεδίο.

Επίσης, χρησιμοποιώντας κατάλληλες συναρτήσεις, ο προγραμματιστής, θα πρέπει να τροποποιήσει τις αρχικές υλοποιήσεις, ώστε σε κάθε στάδιο υπολογισμού να υποδεικνύεται και αντίστοιχη κατανάλωση (ή παραγωγή) πόρων.

Πρώτα θα παρουσιάσουμε τις τέσσερις βασικότερες τέτοιες συναρτήσεις `pure`, `sequential application(<*>)`, `return`, `bind` που πρέπει να διαθέτει κάθε Monad (applicative και monad methods).

```
{-@ pure :: x:a -> { t:Tick a | tval t == x && tcost t == 0 } @-}
pure x = Tick 0 x

{-@ (<*>) :: t1 :Tick (a -> b) -> t2 :Tick a ->
    { t:Tick b | tval t == (tval t1 ) (tval t2 )
        && tcost t == tcost t1 + tcost t2 @-}
Tick m f <*> Tick n x = Tick (m + n) (f x)

{-@ return :: x:a -> { t:Tick a | tval t == x && tcost t == 0 } @-}
return x = Tick 0 x

{-@ (>>=) :: t1 :Tick a -> f:(a -> Tick b) ->
    { t:Tick b | tval t == tval (f (tval t1 ))
        && tcost t == tcost t1 + tcost (f (tval t1 )) } @-}
Tick m x >>= f = let Tick n y = f x in Tick (m + n) y
```

Οι `pure` και `return` επιστρέφουν ένα `Tick` με την τιμή εισόδου στο πεδίο `tval`, και μηδενική τιμή στο πεδίο του κόστους. Οι τελεστές `(<*>)` και `(»=)`, συνδυάζουν τις περιεχόμενες τιμές με τον αναμενόμενο τρόπο για ένα `applied over` και ένα `bind` αντίστοιχα, αθροίζοντας τα κόστη των επιμέρους εκφράσεων.

Η βιβλιοθήκη παρέχει και άλλες βοηθητικές συναρτήσεις, από όπου μπορούμε να επιλέγουμε κάθε φορά την πιο συμβατή με την κατανάλωση που θέλουμε να εκφράσουμε για έναν υπολογισμό. Ξεκινάμε με την `step`, ίσως την πιο απλή συνάρτηση, η οποία αυξάνει την τιμή του κόστους σε ένα `Tick` :

```
{-@ step :: m:Int -> t1 :Tick a ->
    { t:Tick a | tval t == tval t1 && tcost t == m + tcost t1 } @-}
step m (Tick n x) = Tick (m + n) x
```

Όπως φαίνεται από τον ορισμό, στην θέση της παραμέτρου μπορούν να δοθούν και αρνητικοί αριθμοί, εκφράζοντας την παραγωγή πόρων, που μπορούν να 'ακυρώσουν' προηγούμενη ή και μελλοντική κατανάλωση ίσου μεγέθους.

Ο `(</>)` τελεστής συμπεριφέρεται όπως ο `(<*>)` σε επίπεδο τιμών, ενώ για το κόστος δεν αθροίζει απλώς τα επιμέρους αλλά αυξάνει και κατά ένα, χρεώνοντας ουσιαστικά μία μονάδα κόστους στην συγκεκριμένη εφαρμογή.

```
{-@ (</>) :: t1 :Tick (a -> b) -> t2 :Tick a ->
    { t:Tick b | tval t == (tval t1 ) (tval t2 )
&& tcost t == 1 + tcost t1 + tcost t2 } @-}
Tick m f </> Tick n x = Tick (1 + m + n) (f x)
```

Στο ίδιο πνεύμα και η παραλλαγή `(>/=)` του κλασικού `bind`, που και αυτή αθροίζει τα κόστη και προσθέτει και μια παραπάνω μονάδα:

```
{-@ (>/=) :: t1 :Tick a -> f:(a -> Tick b) ->
    { t:Tick b | tval t == tval (f (tval t1 ))
&& tcost t == 1 + tcost t1 + tcost (f (tval t1 )) } @-}
Tick m x >/= f = let Tick n y = f x in Tick (1 + m + n) y
```

Ακόμα υπάρχουν και συναρτήσεις `wait, waitN` που διαφοροποιούνται από τις `pure, return`, καταναλώνοντας 1 ή παραπάνω μονάδες κόστους μετά την ενσωμάτωση του ορίσματος στον τύπο Tick, και οι `go, goN` που κατ' αναλογία παράγουν πόρους.

```
{-@ wait :: x:a -> { t:Tick a | tval t == x && tcost t == 1 } @-}
wait x = Tick 1 x
{-@ waitN :: n:Nat -> x:a -> { t:Tick a | tval t == x && tcost t == n } @-}
waitN n x = Tick n x

{-@ go :: x:a -> { t:Tick a | tval t == x && tcost t == (-1) } @-}
go x = Tick (-1) x
{-@ goN :: n:Nat -> x:a -> { t:Tick a | tval t == x && tcost t == (-n) } @-}
goN n x = Tick (-n) x
```

Να τονίσουμε ότι όλες οι παραπάνω συναρτήσεις μπορούν να εκφραστούν μόνο μέσω των : `return, (»=), step`. Τελικά η ορθότητα της ανάλυσης κόστους που επιτρέπει η βιβλιοθήκη βασίζεται στην ορθότητα αυτών των τριών συναρτήσεων, η οποία αποδεικνύεται στο [Hand19, Ch.5]

3.2 Απόδειξη Ιδιοτήτων και Θεωρήματα

Όπως και στην γενική περίπτωση ιδιοτήτων της LH, υπάρχουν intrinsic και extrinsic αποδείξεις. Για να εστιάσουμε στο κόστος εκτέλεσης, αρκεί να γίνεται αναφορά μέσω της συνάρτησης `tcost` που επιστρέφει το αντίστοιχο πεδίο τύπου `Tick`. Έτσι για τα κάτω και άνω φράγματα του κόστους εκτέλεσης, χρησιμοποιούμε ανισότητες που αφορούν το `tcost` του τύπου επιστροφής της κάθε συνάρτησης. Ταυτόχρονα, στην βιβλιογραφία χρησιοποιούνται οι συναρτήσεις `=!=, <=>, >~>, <~<, >== n ==>, <== n ==<` (που θα μπορούσαν να χρησιμοποιηθούν και στα refinements) για να εκφράσουν σχέσεις : ισοδυναμίας ως προς την τιμή, ισοδυναμίας ως προς το κόστος(μαζί με την ισοδυναμία τιμής), βελτίωσης κόστους, μείωσης κόστους, ποσοτικοποιημένης βελτίωσης κόστους και ποσοτικοποιημένης μείωσης κόστους αντίστοιχα. Σε αντιστοιχία με τις παραπάνω σχέσεις στο module ProofCombinators, ορίζονται οι αντίστοιχοι συνδιαστές, που γενικεύουν τον τελεστή `==.` με

την χρήση των οποίων γίνεται εύκολα η μετάβαση μεταξύ βημάτων μιας απόδειξης με την extrinsic μεθοδο. Η υλοποίηση τους φαίνεται παρακάτω:

```
--
-- Cost equivalence
--
{-@ predicate COSTEQ T1 T2 = tval T1 == tval T2 && tcost T1 == tcost T2 @-}

{-@ (<=>.)
  :: t1:Tick a
  -> { t2:Tick a | COSTEQ t1 t2 }
  -> { t3:Tick a | COSTEQ t1 t2 && COSTEQ t1 t3 && COSTEQ t2 t3 }
@-}
infixl 3 <=>.
(<=>.) :: Tick a -> Tick a -> Tick a
(<=>.) _ t2 = t2
{-# INLINE (<=>.) #-}

--
-- Improvement
--
{-@ predicate IMP T1 T2 = tval T1 == tval T2 && tcost T1 >= tcost T2 @-}

{-@ (>~>.)
  :: t1:Tick a
  -> { t2:Tick a | IMP t1 t2 }
  -> { t3:Tick a | IMP t1 t2 && IMP t1 t3 && t2 == t3 }
@-}
infixl 3 >~>.
(>~>.) :: Tick a -> Tick a -> Tick a
(>~>.) _ t2 = t2
{-# INLINE (>~>.) #-}

--
-- Quantified improvement
--
{-@ predicate QIMP T1 N T2 = tval T1 == tval T2 && tcost T1 == tcost T2 + N @-}

{-@ (.>==)
  :: t1:Tick a
  -> n:Int
  -> { t2:Tick a | QIMP t1 n t2 }
  -> { t3:Tick a | QIMP t1 n t2 && QIMP t1 n t3 && t2 == t3 }
@-}
infixl 3 .>==
(.>==) :: Tick a -> Int -> Tick a -> Tick a
(.>==) _ _ t2 = t2
{-# INLINE (.>==) #-}


--------------------------------------------------------------------------------
-- | Cost separators:
```

```
-------------------------------------------------------------------------------

--
-- Quantified improvement
--
{-@ (==>.) :: (a -> b) -> a -> b @-}
infixl 3 ==>.
(==>.) :: (a -> b) -> a -> b
f ==>. a = f a
{-# INLINE (==>.) #-}
```

Χάριν συντομίας, θα παραλείψω τις υλοποιήσεις που αφορούν την μείωση (ποσοτικοποιημένη ή μη), καθώς η υλοποίηση τους είναι εντελώς ανάλογη με τους συνδιαστές αύξησης κόστους. Μόνη διαφορά είναι η αντικατάσταση του predicate `QIMP` που είδαμε παραπάνω, από το αντίστοιχο μείωσης :

```
{-@ predicate QDIM T1 N T2 = tval T1 == tval T2 && tcost T1 + N == tcost T2 @-}
```

Κεφάλαιο 4

Παραδείγματα από VFA

Σε όλα τα παραδείγματα που θα ακολουθήσουν, ξεκινάμε με την ενεργοποίηση των δύο flags, που επιτρέπουν την απόδειξη θεωρημάτων, όπως είδαμε και στο κεφάλαιο 2 :

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

4.1 Permutations

Ο κώδικας του υποκεφαλαίου είναι ο παρακάτω:

```
module Vfa.Perm where


import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , reverse
  )

--------------------------------------------
-- | Costs on Permutations ----------------
--------------------------------------------

{-@ (+++) :: xs:[a] -> ys:[a]
  -> {t:Tick {zs:[a] | length zs == length xs + length ys} | tcost t == length
  ↪  xs}
@-}
(+++) :: [a] -> [a] -> Tick [a]
[] +++ ys = pure ys
(x : xs) +++ ys = pure (x :) <> (xs +++ ys)


{-@ reflect app @-}
app :: [a] -> [a] -> [a]
```

```
app [] ys     = ys
app (x:xs) ys = cons x (app xs ys)



{-@ lenn :: xs : [a] -> {t : Tick Nat | tcost t == length xs }
@-}
lenn :: [a] -> Tick Int
lenn []     = pure 0
lenn (x:xs) = pure (1 + ) <$> (lenn xs)

{-@ reflect reverse @-}
{-@ reverse :: xs:[a] -> {t:Tick [a] | tcost t == length xs} @-}
reverse :: [a] -> Tick [a]
reverse []      = pure []
reverse (x:xs)  = liftA2 (app) (reverse xs) (Tick 1 [x])

{-@ app_rev :: xs:[a] -> {tcost (reverse xs) == length xs } @-}
app_rev :: [a] -> Proof
app_rev []   = ()
app_rev (x:xs) = app_rev xs
```

Ο τελεστής συνένωσης `(+++)`, παίρνει δύο λίστες ως ορίσματα και επιστρέφει μια τρίτη, η οποία έχει μήκος όσο το άθροισμα των δύο. Βλέπουμε ότι η χρονική πολυπλοκότητα εξαρτάται μόνο από το μήκος της πρώτης λίστας.

Επισης, τόσο η `lenn` οσο και η `reverse`, έχουν $\Theta(n)$ χρονική πολυπλοκότητα.

4.2 Insertion Sort

```
module Vfa.InsertionSort where


import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , (=<<)
  )

--------------------------------------------
-- | Costs on Insertion Sort ---------------
--------------------------------------------
{-@ type OList a = [a]<{ \ x y -> x <= y }> @-}

{-@ reflect insert @-}
```

```
{-@ insert :: Ord a => a -> xs: OList a
  -> {t:Tick {ys:(OList a) | length ys = (length xs)+1}
      | tcost t <= length xs && tcost t >= 0 }
@-}
insert :: Ord a => a -> [a] -> Tick [a]
insert x []      = pure [x]
insert x (y:ys)
    |  x <= y    = wait (x:(y:ys))
    | otherwise = pure (y : ) </> insert x ys

{-@ reflect sort @-}
{-@ sort :: Ord a => xs:[a]
    -> {t:Tick {ys:(OList a)| length ys == length xs}
       | tcost t  <= ((length xs - 1) * length xs) }
@-}
sort :: Ord a => [a] -> Tick [a]
sort [] = pure []
sort (y:ys) = leqBind (length ys) (sort ys) (insert y)

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

Η `insert` βλέπουμε ότι κάνει $O(n)$ συγκρίσεις. Στην `sort`, κάνουμε χρήση του τελεστή `leqBind`, ο οποίος στο paper [Hand19] αναφέρεται ως ”bounded bind operator” και συμβολίζεται με $=<< \{\cdot\}$. Ουσιαστικά είναι μια παραλλαγή του απλού bind : `(>>=) :: m a -> (a -> m b) -> m b`, όπως ορίζεται στο [Huda99].

Ο `leqBind` που ορίζεται στο αρχείο RTick.hs της βιβλιοθήκης:

```
-- Note that leqBind is is written as =<<{n} in the paper and referred to as
-- "bounded bind". It behaves just as =<< but restricts its domain to
-- functions f :: a -> Tick b with resource usage no greater than n.
--
{-@ reflect leqBind @-}
{-@ leqBind :: n:Int -> t1:Tick a
    -> f:(a -> { tf:Tick b | n >= tcost tf })
    -> { t:Tick b | tcost t1 + n >= tcost t }
@-}
leqBind :: Int -> Tick a -> (a -> Tick b) -> Tick b
leqBind _ (Tick m x) f = let Tick n y = f x in Tick (m + n) y
```

με βάση το ‘type signature’ περιορίζει το τελικό κόστος. Διασφαλίζει ότι μετά την εφαρμογή της συνάρτησης στο όρισμα, το νέο κόστος δεν θα υπερβεί το αρχικό, περισσότερο από n μονάδες.

Εδώ, η `insert y` γίνεται “bind” με την λίστα `(sort ys)`, ενώ ταυτόχρονα, ο τελεστής περιορίζει το επιπλέον κόστος να μην ξεπερνά το `(length ys)`.

Στη συνέχεια βλέπουμε κάποια παραδείγματα:

```
---------Paradeigmata-------------
{-@ reflect l_pi @-}
```

```
l_pi= [3,3,4,1,5,9,2,6,5,3,5]
{-@ reflect sl_pi @-}
sl_pi= [1,2,3,3,3,4,5,5,5,6,9]
{-@ reflect des @-}
des = [15,14,13,12,11]
{-@ reflect empty @-}
empty = []

{-@ example_pi :: { (sorted sl_pi) } @-}
example_pi :: Proof
example_pi = ()

{-@ example_sort :: { tcost (sort des) == (1+2+3+4) } @-}
example_sort :: Proof
example_sort = ()
```

όπου φαίνεται ότι η απόδειξη πως μια λίστα είναι ταξινομημένη γίνεται αυτόματα, χωρίς να παρέχουμε κάτι παραπάνω από τον τελεστή `()`. Εξίσου εύκολα επαληθεύουμε ότι η ταξινόμηση της `des=[15,14,13,12,11]`, απαιτεί συνολικά 10 συγκρίσεις , αφού είναι σε φθίνουσα σειρά τα στοιχεία της.

4.3 Selection Sort

Σε αυτό το κεφάλαιο παρουσιάζουμε μια υλοποίηση της ταξινόμησης selection.

```
module Vfa.SelectionSort where

import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , (=<<)
  )
```

Από την διπλωματική [Πέ18] μεταφέρω τα παρακάτω :

```
--Τύπος δεδομένων που συμβολίζει ζεύγος μεταβλητών
data Pair a b = Pair a b
--Συνάρτηση που δέχεται ένα Pair και επιστρέφει το πρώτο στοιχείο
{-@ measure fst1 @-}
fst1 :: (Pair a b) -> a
fst1 (Pair x _) = x
--Συνάρτηση που δέχεται ένα Pair και επιστρέφει το δεύτερο στοιχείο
{-@ measure snd1 @-}
snd1 :: (Pair a b) -> b
snd1 (Pair _ x) = x
```

Στην συνέχεια τροποποιώ τον αλγόριθμο για την selection sort, προσθέτοντας τις κατάλληλες επεκτάσεις (από τύπο α σε $Tick\ \alpha$) ώστε να γίνει εφικτό το resource consumption reasoning.

```
-------------------------------------------
-- | Costs on Seletion Sort ---------------
-------------------------------------------

-- cost based on #(comparisons)
{-@ reflect sorted @-}
{-@ sorted :: Ord a => l:[a] ->{ t:Tick Bool | tcost t <= length l} @-}
sorted :: Ord a => [a] ->  Tick Bool
sorted [] = pure True
sorted [_] = pure True
sorted (x:y:ys) =
    if x<=y then  step 1 (sorted (y:ys))
    else Tick 1 False

{-@ check_sorted :: { tcost (sorted l_pi) == 3 } @-} -- Αρκουν 3 συγκρίσεις
-- {-@ check_sorted :: { tcost (sorted sl_pi) == 10} @-} -- Αρκουν 10 συγκρίσεις
check_sorted :: Proof
check_sorted = ()

{-@ reflect select @-}
{-@ select :: Ord a => a -> xs:[a]
    ->  {t:Tick {p:(Pair a [a])
                | length xs == length (snd1 p) }
        | tcost t <= length xs}
@-}
select :: Ord a => a -> [a] -> Tick (Pair a [a])
select i [] = pure (Pair i [])
select i (x:xs)
    | i <= x    = let Tick c (Pair j l') = select i xs in Tick (c+1) (Pair j (x:l'))
    | otherwise = let Tick c (Pair j l') = select x xs in Tick (c+1) (Pair j (i:l'))

-- {-@ example_select :: { snd1 (select 100 des) == small_des} @-}
-- example_select :: Proof
-- example_select = ()

-- {-@ type OList a = [a]<{ \ x y -> x <= y }> @-}

{-@ reflect selsort @-}
{-@ selsort :: Ord a => l:[a] -> n:Nat
    -> {t:Tick [a] | tcost t <= (length l)*((length l)-1)/2 }
    / [n] @-}                                                -- 1+2+...+(n-1) = n*(n-1)/2 = O(n^2)
selsort :: Ord a => [a] -> Int -> Tick [a]
selsort [] _ = pure []
selsort _  0 = pure [] {- out of fuel -}
selsort (x:r) n = let Tick c1 (Pair y rr) = select x r
                      Tick c2 l' = selsort rr (n-1)
```

```
76                 in Tick (c1 + c2) (cons y l') -- Q: is the 'c1+c2' or 'c1+c2+1'?
77                                                -- A: not sure but I dont see
                                                  ↪  resource cosouming ops
                                                  ↪  outside of lets
78 --  x::r, S n' □ let (y,rr) := select x r
79 --                 in y :: selsort rr n'
80
81
82 {-@ reflect selection_sort @-}
83 {-@ selection_sort :: Ord a => l:[a]
84     -> {t:Tick [a] | tcost t <= (length l)*((length l)-1)/2 }
85     @-}
86 selection_sort :: Ord a => [a] -> Tick [a]
87 selection_sort l = selsort l (length l)
88
89 ---------Paradeigmata-------------
90 {-@ reflect l_pi @-}
91 l_pi= [3,3,4,1,5,9,2,6,5,3,5]
92 {-@ reflect sl_pi @-}
93 sl_pi= [1,2,3,3,3,4,5,5,5,6,9]
94 {-@ reflect small_des @-}
95 small_des = [100,15,14,13,12]
96 {-@ reflect des @-}
97 des = [15,14,13,12,11]
98 {-@ reflect empty @-}
99 empty = []
100
101 -- {-@ example_pi :: { (sorted sl_pi) } @-}
102 -- example_pi :: Proof
103 -- example_pi = ()
104
105 -- {-@ example_sort :: { tcost (sort des) == (1+2+3+4) } @-}
106 -- example_sort :: Proof
107 -- example_sort = ()
```

Ο αλγόριθμος αποτελείται ουσιαστικά από δύο συναρτήσεις, την `select` και την `selsort`. Η `select` διασχίζει όλη την λίστα που δέχεται στο ορίσματα, και επιστρέφει στο τέλος ένα ζεύγος με το ελάχιστο στοιχείο και την υπόλοιπη λίστα., προφανώς σε χρόνο $O(n)$. Η `selsort` στην κλασσική εκδοχή της λειτουργεί ως εξής. Αφαιρεί κάθε φορά το ελάχιστο στοιχείο της λίστας, το προσαρτά στη λίστα αποτελεσμάτων και συνεχίζει επαγωγικά μέχρι να εξαντληθεί η αρχική λίστα. Οπότε τελικά επιστρέφει μια λίστα με τα στοιχεία της αρχικής ταξινομημένα σε αύξουσα σειρά. Προσθετικά σε αυτά, δίνω, ως επιπλέον όρισμα, έναν μη αρνητικό ακέραιο που μειώνεται σε κάθε αναδρομική κλήση, με αποτέλεσμα μόλις φτάσει στο 0 να σταματάει την εκτέλεση. Έτσι, στην υλοποίηση της τελικής συνάρτησης `selection_sortl = selsort l (length l)`, εξάγεται εύκολα ο περιορισμός `tcost t <= (length l)*((length l)-1)/2`, επαληθεύοντας ότι, αν n το μήκος της λίστας, η πολυπλοκότητα του αλγορίθμου είναι $O(n^2)$.

Σε αρκετά refinements, (βλ. `selsort`, γραμμή 72) χρησιμοποιούμε την έκφαση $/[n]$. Η μεταβλητή n είναι ένα explicit termination metric [Vazo14], προκειμένου η απόδειξη για τον τερματισμό να στηριχτεί στην φθίνουσα συμπεριφορά της n.

Στο τέλος του αρχείου παρουσιάζονται και κάποια παραδείγματα, ενδεικτικά, όπου φαίνεται η κατανάλωση για κάποιες πολύ απλές περιπτώσεις.

4.4 Binary Search Trees

Ένα BST είναι ένα δυαδικό δέντρο, που διατηρεί την παρακάτω ιδιότητα. Το κλειδί κάθε κόμβου είναι μεγαλύτερο από όλα τα κλειδιά του αριστερού υποδέντρου, και μεγαλύτερο από όλα τα κλειδιά του δεξιού υποδέντρου [Sedg17, Ch. 3.2].

Η δομή μπορεί να υποστηρίξει διάφορες λειτουργίες όπως: η έλεγχος ύπαρξης στοιχείου στο δέντρο, εύρεση ελάχιστου/μέγιστου, εύρεση επόμενου/προηγούμενου, εσαγωγή, διαγραφή κ.α. Εδώ υλοποιούνται οι : lookup, insert, elements.

Όσον αφορά το υπολογιστικό κόστος, όπως αναφέρεται στο [Corm09], οι βασικές λειτουργίες ενός BST παίρνουν χρόνο ανάλογο του ύψους του δέντρου. Για δέντρα n κόμβων, στην περίπτωση μιας "αλυσίδας" το ύψος είναι $\Theta(n)$, ενώ στα πλήρη διαδικά δέντρα είναι $\Theta(log(n))$. Και παρόλο που το αναμενόμενο ύψος ενός παραγόμενου BST είναι $O(log(n))$, δεν μπορώ να αποδείξω την πολυπλοκότητα της μέσης περίπτωσης, γιατί θέλω η αναλλοίωτη να αφορά οποιοδήποτε instance του τύπου BST.

Έτσι, το επιβεβαιωμένο, από τα refinements, κόστος των παραπάνω συναρτήσεων έχει ως εξής. Οι `lookup` και `insert` απαιτούν $O(height(t))$, ενώ η `elements` $O(blen(t))$.

Με το 'measure' `blen`, εννοούμε το συνολικό αριθμό $n = \#V$ των κόμβων του γράφου $T = G(V, E)$, ενώ με το `bheight`, το ύψος του, το οποίο σαν μέγεθος κυμαίνεται στο εύρος $[log_2(n), n-1]$

```
module Vfa.BST where

import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , (=<<)
  , lookup
  )

data BST k v = Empty
  | Bind { bKey :: k
         , bValue :: v
         , bLeft :: BST k v
         , bRight :: BST k v } deriving (Show)

type Val = Int

data Pair a b = Pair {fst1 :: a, snd1 :: b} deriving (Show)

{-@ 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)

-- hard to proove average complexity.. so i choose height as a parameter
```

```
{-@ measure bheight @-}
bheight :: (BST k v) -> Int
{-@ bheight :: (BST k v) -> Nat @-}
bheight Empty = 0
bheight (Bind _ _ l r) = 1 + ((\x y -> if x < y then y else x) (bheight l) (bheight r)) --could not lift haskell's function max

-------------------------------------------
-- | Costs on Binary Search Trees ----------
-------------------------------------------

-- cost based on #(comparisons)

-- {-@ lookup :: (Ord k) => k -> b:(BST k Val) -> {t:Tick (Maybe Val) | tcost t <= (blen b)} @-}
--prooves UB as tree's size, but tree's heihgt bound is more strict and intuitive
{-@ lookup :: (Ord k) => k -> b:(BST k Val) -> {t:Tick (Maybe Val) | tcost t <= bheight b} @-}
lookup :: (Ord k) => k -> BST k Val -> Tick (Maybe Val)
lookup _ Empty = pure Nothing
lookup k (Bind bk bv bl br)
  | k <  bk = step 1 (lookup k bl)
  | k >  bk = step 1 (lookup k br)
  | k == bk = wait (Just bv) -- same as Tick 1 (just bv)

{-@ insert :: (Ord k) =>  k -> Val -> b:(BST k Val) -> {t:Tick (BST k Val) | tcost t <= bheight b} @-}
insert :: (Ord k) =>  k -> Val -> BST k Val -> Tick (BST k Val)
insert k v Empty = pure (Bind k v Empty Empty)
insert k v (Bind k' v' bl br)
  | k < k'    = let Tick c1 v1 = (insert k v bl) in Tick (c1 + 1) (Bind k' v' v1 br)
  | k > k'    = let Tick c1 v2 = (insert k v br) in Tick (c1 + 1) (Bind k' v' bl v2)
  | otherwise = wait (Bind k v bl br)

{-@ elements :: b:(BST k Val) -> {t:(Tick [(Pair k Val)]) | tcost t <= blen b } @-}
elements :: (BST k  Val) -> Tick [(Pair k Val)]
elements Empty = pure []
elements (Bind k v bl br) =
  let Tick c1 v1 = (elements bl)
      Tick c2 v2 = (elements br)
      in Tick (c1 + c2 + 1) (v1 ++ (cons (Pair k v) v2))
```

4.5 Priority Queues

Μία "ουρά προτεραιότητας" πρέπει να υλοποιεί τις εξής λειτουργίες: δημιουργία άδειας ουράς, προσθήκη στοιχείου και αφαίρεση του μέγιστου στοιχείου[Appe17, Ch. 12]. Σε αυτό το κεφάλαιο, παρουσιάζω μία από τις απλούστερες αλλά και λιγότερο αποδοτικές υλοποιήσεις αυτού του *αφηρημέ-*

νου τύπου δεδομένων(ADT), με λίστες, ενώ θα ακολουθήσουν και αποδοτικότερες υλοποιήσεις στα επόμενα κεφάλαια.

Θα δούμε τις συναρτήσεις `empty` και `insert` να τρέχουν σε σταθερό χρόνο, και την `delete_max` σε χρόνο ανάλογο του μήκους της λίστας. Επίσης, η συνένωση δύο ουρών, γίνεται σε χρόνο $O(n)$ όπου n το μήκος της πρώτης από τις δύο λίστες.

```haskell
module Vfa.Priqueue where

import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , (=<<)
  , lookup
  )

-----------------------------------
-- | Από διπλωματική του Πέτρου ---
-----------------------------------
{-@ type Key = Int @-}
type Key = Int

-- priority queues as unsorted lists
{-@ type  PRIQUEUE = [Key] @-}
type PRIQUEUE = [Key]

data Pair a b = Pair {fst1 :: a, snd1 :: b} deriving (Show)


-------------------------------------------
-- | Costs on Priority Queues --------------
-------------------------------------------

-- why lists is a bad decision : https://arxiv.org/pdf/1808.08329.pdf

-- I use here a the naive impementation of unsorted list
-- select_max : to pull highest element

-- cost based on #(comparisons)

{-@ select_max :: Key -> l:PRIQUEUE -> {t:Tick (Pair Key PRIQUEUE) | tcost t <= length l} @-}
select_max :: Key -> PRIQUEUE -> Tick (Pair Key PRIQUEUE)
select_max i [] = pure (Pair i [])
select_max i (x:xs)
```

```
  | i >= x    = let Tick c1 (Pair j l') = select_max i xs in Tick (c1+1) (Pair j (x:l'))
  | otherwise = let Tick c1 (Pair j l') = select_max x xs in Tick (c1+1) (Pair j (i:l'))

{-@ empty :: {t:Tick PRIQUEUE | tcost t == 0} @-}
empty :: Tick PRIQUEUE
empty = pure []

{-@ insert :: Key -> PRIQUEUE -> {t:Tick PRIQUEUE | tcost t == 1} @-}
insert :: Key -> PRIQUEUE -> Tick PRIQUEUE
insert k p = wait (k:p)


{-@ delete_max :: l:PRIQUEUE -> {t : Tick (Maybe PRIQUEUE) | (tcost t == 0) ||(tcost t < length l)} @-}
delete_max :: PRIQUEUE -> Tick (Maybe PRIQUEUE)
delete_max [] = pure Nothing
delete_max (x:xs) = let Tick c1 (Pair _ l') = select_max x xs in Tick c1 (Just l')

{-@ merge :: x:PRIQUEUE -> PRIQUEUE -> { t:Tick PRIQUEUE | tcost t <= length x } @-}
merge :: PRIQUEUE -> PRIQUEUE -> Tick PRIQUEUE
merge p q = append p q

-- For (append) I use as cost, the # of recursive calls (or # of concatination)
-- instead of the usual suspect : # of comparisons
{-@ append :: x:[a] -> [a] -> {t:Tick [a] | tcost t <= length x } @-}
append :: [a] -> [a] -> Tick [a]
append [] ys = pure ys
append (x:xs) ys = wait (x :) <*> append xs ys
```

4.6 RedBlack Trees

Σύμφωνα με το [Corm09, Ch. 13] ένα Red-Black Tree είναι ένα δυαδικό δέντρο με επιπλέον τις ακόλουθες ιδιότητες

1. Κάθε κόμβος είναι είτε μαύρος είτε κόκκινος
2. Η ρίζα έχει μαύρο χρώμα
3. Κάθε φύλλο έχει επίσης μαύρο χρώμα
4. Οι κόκκινοι κόμβοι έχουν μόνο παιδιά μαύρου χρώματος
5. Για κάθε κόμβο, όλα τα απλά μονοπάτια που οδηγούν από εκείνον σε κάποιον απόγονο-φύλλο θα πρέπει να περιλαμβάνουν τον ίδιο αριθμό μαύρων κόμβων.

Θα περιγράψουμε ένα Red-Black Tree t με την $bh(t)$, που επιστρέφει το πλήθος των μαύρων κόμβων ενός μονοπατιού από τη ρίζα του t μέχρι κάποιο φύλλο. Με βάση την ιδιότητα 5, η bh είναι καλά ορισμένη, και στην συνέχεια θα δείξουμε ότι αποτελεί καλή προσέγγιση του πραγματικού ύψους του δέντρου.

```haskell
module Vfa.Redblack where

import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , (=<<)
  , lookup
  )


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

{-@ data Color = Red | Black @-}
data Color = Red | Black deriving (Eq)

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

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

-- "Measuring the sizeof a Red-Black tree would take linear time,
-- so I measure the approximate log of the size (the "black-node height") in logNtime."
-- https://www.cs.princeton.edu/~appel/papers/redblack.pdf
{-@ measure bh @-}
```

```
{-@ bh :: Tree v -> Nat @-}
bh :: Tree v -> Int
bh E = 1
↪  ----https://www.cs.tufts.edu/~nr/cs257/archive/chris-okasaki/redblack99.pdf
bh (T c tl _ _ tr) = bh tl + if c==Red then 0 else 1

{-@ measure height @-}
{-@ height :: Tree v -> Nat @-}
height :: Tree v -> Int
height E = 1
height (T _ t1 _ _ t2) = (\x y -> if x < y then y else x) (height t1) (height
↪  t2) + 1

{-@ measure true_rh @-}
{-@ true_rh :: Tree v -> Nat @-}
true_rh :: Tree v -> Int
true_rh E = 0
true_rh (T Red tl _ _ tr) = (\x y -> if (true_rh x) < (true_rh y) then true_rh y
↪  else true_rh x) tl tr  + 1
true_rh (T Black tl _ _ tr) = (\x y -> if (true_rh x) < (true_rh y) then true_rh
↪  y else true_rh x) tl tr  + 0

{-@ measure rh @-}
{-@ rh :: rb : RBTree v -> {n:Nat | n <= true_rh rb} @-}
rh :: Tree v -> Int
rh E = 0
rh (T c tl _ _ tr) = (rh tl) + if c==Red then 1 else 0

{-@ measure bh_star @-}
{-@ bh_star :: rb:RBTree v -> Nat @-}
bh_star :: Tree v -> Int
bh_star E = 1
bh_star (T Red tl _ _ tr) = (\x y -> if (true_rh x) < (true_rh y) then bh_star y
↪  else bh_star x) tl tr + 0
bh_star (T Black tl _ _ tr) = (\x y -> if (true_rh x) < (true_rh y) then bh_star
↪  y else bh_star x) tl tr + 1

{-@ measure no_redchain @-}
{-@ no_redchain ::  Tree v -> Bool @-}
no_redchain ::  Tree v -> Bool
no_redchain E = True
no_redchain (T c t1 _ _ t2) =
    if c == Red && (col t1 == Red || col t2 == Red) then  False
    else no_redchain t1 && no_redchain t2

{-@ reflect col @-}
{-@ col :: Tree v -> Color @-}
col :: Tree v -> Color
col E = Black
col (T c _ _ _ _) = c

```

```
{-@ measure bl_balance @-}
bl_balance :: Tree v -> Bool
bl_balance E = True
bl_balance (T _ t1 _ _ t2) = bl_balance t1  && bl_balance t2 && bh t1 == bh t2


{-@ type RBTree v = { rb : Tree v | no_redchain rb && bl_balance rb} @-}

{-@ thm_no_red_seq :: rb:RBTree v -> {col rb == Red => (col(bLeft(rb)) == Black && col(bRight(rb)) == Black )} @-}
thm_no_red_seq :: Tree v -> Proof
thm_no_red_seq E = ()
thm_no_red_seq (T Black t1 k v t2) = ()
thm_no_red_seq (T Red t1 k v t2) = ()

{-@ thm_balanced :: rb : RBTree v -> {rb == E || bh (bLeft rb) == bh (bRight rb) } @-}
thm_balanced :: Tree v -> Proof
thm_balanced E = ()
thm_balanced _ = ()

{-@ thm_true_rh_Black :: {rb : RBTree v| (col rb == Black)}
    -> { ((rb == E )
      || ( true_rh (bRight rb) <  true_rh (bLeft rb) && (true_rh rb == true_rh (bLeft rb) ))
      || ( true_rh (bRight rb) >= true_rh (bLeft rb) && (true_rh rb == true_rh (bRight rb) )) )} @-}
thm_true_rh_Black :: Tree v -> Proof
thm_true_rh_Black E = ()
thm_true_rh_Black (T Red _ _ _ _) = ()
thm_true_rh_Black (T Black _ _ _ _) = ()

{-@ thm_true_rh_Red :: {rb : RBTree v | (col rb == Red)}
    -> {((true_rh (bRight rb) <  true_rh (bLeft rb) && (true_rh rb == true_rh (bLeft rb) + 1 ))
      || (true_rh (bRight rb) >= true_rh (bLeft rb) && (true_rh rb == true_rh (bRight rb) + 1)) ) } @-}
thm_true_rh_Red :: Tree v -> Proof
thm_true_rh_Red E = ()
thm_true_rh_Red (T Red _ _ _ _) = ()
thm_true_rh_Red (T Black _ _ _ _)= ()

{-@ thm_bh_EQ_bh_star :: rb:RBTree v -> {bh rb == bh_star rb} @-}
thm_bh_EQ_bh_star :: Tree v -> Proof
thm_bh_EQ_bh_star E = ()
thm_bh_EQ_bh_star (T Red t1 k v t2)
  | true_rh t1 >= true_rh t2
  = bh (T Red t1 k v t2)
  ==. bh t1 ? thm_bh_EQ_bh_star t1
  ==. bh_star t1
  ==. bh_star (T Red t1 k v t2)
```

```
  *** QED
  | otherwise
  = bh (T Red t1 k v t2)
  ==. bh t1 ? thm_balanced (T Red t1 k v t2)
  ==. bh t2 ? thm_bh_EQ_bh_star t2
  ==. bh_star t2
  ==. bh_star (T Red t1 k v t2)
  *** QED
thm_bh_EQ_bh_star (T Black t1 k v t2)
  | true_rh t1 >= true_rh t2
  = bh (T Black t1 k v t2)
  ==. bh t1 + 1 ? thm_bh_EQ_bh_star t1
  ==. bh_star t1 + 1
  ==. bh_star (T Black t1 k v t2)
  *** QED
  | otherwise
  = bh (T Black t1 k v t2)
  ==. bh t1 + 1? thm_balanced (T Black t1 k v t2)
  ==. bh t2 + 1 ? thm_bh_EQ_bh_star t2
  ==. bh_star t2 + 1
  ==. bh_star (T Black t1 k v t2)
  *** QED

{-@ thm_true_rh_UB :: rb : RBTree v -> {true_rh rb <= bh_star rb} / [true_rh rb]
↪  @-}
thm_true_rh_UB :: Tree v -> Proof
thm_true_rh_UB E = ()
thm_true_rh_UB (T Black t1 k v t2)
  | true_rh t1 < true_rh t2
  = true_rh (T Black t1 k v t2)
  ==. true_rh t2 ? thm_true_rh_UB t2
  <=. bh_star t2
  <=. bh_star t2 + 1
  ==. bh_star (T Black t1 k v t2)
  *** QED
  | otherwise
  = true_rh (T Black t1 k v t2)
  ==. true_rh t1 ? thm_true_rh_UB t1
  <=. bh_star t1
  <=. bh_star t1 + 1
  ==. bh_star (T Black t1 k v t2)
  *** QED
thm_true_rh_UB (T Red E k v E) = ()
thm_true_rh_UB (T Red t k v  E)
  | true_rh (bLeft t) >= true_rh (bRight t)
  = true_rh (T Red t _ _ E) ? thm_no_red_seq (T Red t _ _ E)
  ==. true_rh t + 1 ? toProof (col t == Black)
  ==. true_rh (bLeft t) + 1 ? thm_true_rh_UB (bLeft t)
  <=. bh_star (bLeft t) + 1 ? toProof (col t == Black)
  <=. bh_star t
  <=. bh_star (T Red t k v E)
```

```
  *** QED
  | otherwise
  = true_rh (T Red t _ _ E) ? thm_no_red_seq (T Red t _ _ E)
  ==. true_rh t + 1 ? toProof (col t == Black)
  ==. true_rh (bRight t) + 1 ? thm_true_rh_UB (bRight t)
  <=. bh_star (bRight t) + 1 ? toProof (col t == Black)
  <=. bh_star t
  <=. bh_star (T Red t k v E)
  *** QED
thm_true_rh_UB _ = ()

{-@ corollary1 :: rb : RBTree v -> {true_rh rb <= bh rb} / [true_rh rb] @-}
corollary1 :: Tree v -> Proof
corollary1 t = thm_true_rh_UB t ? thm_bh_EQ_bh_star t *** QED

{-@ thm_true_height_UB :: rb:RBTree v -> {true_rh rb + bh_star rb <= 2*bh rb} @-}
thm_true_height_UB :: Tree v -> Proof
thm_true_height_UB t = corollary1 t ? thm_bh_EQ_bh_star t *** QED

{-@ thm_height_EQ :: rb : RBTree v -> {height rb == bh_star rb + true_rh rb} @-}
thm_height_EQ :: Tree v -> Proof
thm_height_EQ E = ()
thm_height_EQ (T Red t1 k v t2)
  = height (T Red t1 k v t2)
  ==. 1 + ((\x y -> if x < y then y else x) (height t1) (height t2))
  ==. 1 + ((\x y -> if (height x) < (height y) then (height y) else (height x)) t1 t2) ? thm_height_EQ t1 ? thm_height_EQ t2
  ==. 1 + ((\x y -> if (bh_star x + true_rh x) < (bh_star y + true_rh y) then (bh_star y + true_rh y) else (bh_star x + true_rh x)) t1 t2)
  ? thm_bh_EQ_bh_star t1 ? thm_bh_EQ_bh_star t2
  ==. 1 + ((\x y -> if (bh x + true_rh x) < (bh y + true_rh y) then (bh y + true_rh y) else (bh x + true_rh x)) t1 t2) ? thm_balanced (T Red t1 k v t2)
  ==. 1 + ((\x y -> if (bh x + true_rh x) < (bh x + true_rh y) then (bh x + true_rh y) else (bh x + true_rh x)) t1 t2)
  ==. 1 + ((\x y -> if (true_rh x) < (true_rh y) then (true_rh y) else (true_rh x)) t1 t2) + bh t1
  ==. true_rh (T Red t1 k v t2) + bh t1 ? thm_bh_EQ_bh_star t1
  ==. true_rh (T Red t1 k v t2) + bh_star t1
  ==. true_rh (T Red t1 k v t2) + bh_star (T Red t1 k v t2)
  ***QED
thm_height_EQ (T Black t1 k v t2)
  = height (T Black t1 k v t2)
  ==. 1 + ((\x y -> if x < y then y else x) (height t1) (height t2))
  ==. 1 + ((\x y -> if (height x) < (height y) then (height y) else (height x)) t1 t2) ? thm_height_EQ t1 ? thm_height_EQ t2
  ==. 1 + ((\x y -> if (bh_star x + true_rh x) < (bh_star y + true_rh y) then (bh_star y + true_rh y) else (bh_star x + true_rh x)) t1 t2)
  ? thm_bh_EQ_bh_star t1 ? thm_bh_EQ_bh_star t2
```

```
  ==. 1 + ((\x y -> if (bh x + true_rh x) < (bh y + true_rh y) then (bh y + true_rh y) else (bh x + true_rh x)) t1 t2) ? thm_balanced (T Black t1 k v t2)
  ==. 1 + ((\x y -> if (bh x + true_rh x) < (bh x + true_rh y) then (bh x + true_rh y) else (bh x + true_rh x)) t1 t2)
  ==. 1 + ((\x y -> if (true_rh x) < (true_rh y) then (true_rh y) else (true_rh x)) t1 t2) + bh t1
  ==. 1 + true_rh (T Black t1 k v t2) + bh t1 ? thm_bh_EQ_bh_star t1
  ==. 1 + true_rh (T Black t1 k v t2) + bh_star t1
  ==. true_rh (T Black t1 k v t2) + bh_star (T Black t1 k v t2)
  ***QED

{-@ thm_height_UB :: rb:RBTree v -> {height rb <= 2*bh rb} @-}
thm_height_UB :: Tree v -> Proof
thm_height_UB x = thm_true_height_UB x ? thm_height_EQ x *** QED

{-@ lemma_height_LB :: rb:RBTree v -> {bh rb <= height rb } @-}
lemma_height_LB :: Tree v -> Proof
lemma_height_LB E = ()
lemma_height_LB (T Red t1 k v t2)
    = bh (T Red t1 k v t2)
  ==. bh t1 + 0
  ==. bh t1 ? lemma_height_LB t1
  <=. height t1
  <=. height (T Red t1 k v t2)
  *** QED
lemma_height_LB (T Black t1 k v t2)
  = bh (T Black t1 k v t2)
  ==. bh t1 + 1 ? lemma_height_LB t1
  <=. height  (T Black t1 k v t2)
  *** QED
```

Στο πρώτο τμήμα του κώδικα ορίζω μαζί με τους τύπους δεδομένων και κάποιες βοηθητικές συναρτήσεις, για την καλύτερη περιγραφή των πόρων που καταναλώνονται. Επίσης δείχνω με extrinsic proofs τις ανισότητες : $bh(rb) \leq height(rb) \leq 2 \times bh(rb)$.

Από την διπλή ανισότητα, το αριστερό τμήμα αποδεικνύεται με μία απλή extrisic proof σε LH (`lemma_height_LB`), που μοιάζει με μια κλασική επαγωγή που θα κάναμε με χαρτί και μολύβι. Αντίθετα, το δεύτερο τμήμα, εκφράζει μια λιγότερο προφανή ιδιότητα, και επομένως χρειαστήκαν αρκετά βοηθητικά θεωρήματα μέχρι να καταλήψουμε στο `thm_height_UB`.

Κατ αρχάς, για την απόδειξη, χρειάστηκε πρώτα να ορίσω κάποιες συναρτήσεις για τον προσδιορισμό του ύψους ενός δέντρου. Πέρα από την κλασσική `height` που υπολογίζει το βαθύτερο μονοπάτι, ορίζω και τις `bh`, `rh` που επιστρέφουν το πλήθος των μαύρων και κόκκινων κόμβων, αντίστοιχα, που υπάρχουν στο μονοπάτι από τη ρίζα στο αριστερό φύλλο. Η αναδρομή στο αριστερό υποδέντρο που κάνουν κάθε φορά οι `bh` και `rh` γίνεται αυθαίρετα, για λόγους απλότητας, και έτσι για την απόδειξη κάποιων θεωρημάτων χρειάστηκαν επιπλέον οι `true_rh` και `bh_star` οποίες επιστρέφουν το πλήθος των κόκκινων και μαύρων κόμβων αντίστοιχα, στο βαθύτερο μονοπάτι ως προς τους κόκκινους κόμβους.

Τα πρώτα (βοηθητικά) θεωρήματα από τα οποία ξεκίνησα, εκφράζουν τις ιδιότητες 4 και 5. Ειδικότερα, με το `thm_no_red_seq` επιβεβαιώνεται ότι για κάθε κόκκινο κόμβο και τα δύο παιδιά του έχουν μαύρο χρώμα. Με το `thm_balanced` επιβεβαιώνεται ότι σε κάθε μη κενό δέντρο το "μαύρο ύψος" `bh` του αριστερού υποδέντρου ισούται με το `bh` του δεξιού.

Παρακάτω συνεχίζω με το θεώρημα το thm_bh_EQ_bh_star όπου δείχνω ότι σε ένα **RBTree** η bh ισούται με την bh_star, για την απόδειξη του οποίου χρειάστηκε να αναφέρω και το προηγούμενο θεώρημα thm_balanced. Αξιοποιώντας αυτά τα δύο θεωρήματα, μπορώ να δείξω πλέον το thm_height_EQ, σύμφωνα με το οποίο το ύψος του δέντρου ισούται με το μήκος του μονοπατιού εκείνου που (από την ρίζα μέχρι κάποιο φύλλο) μεγιστοποιεί το αριθμό των κόκκινων κόμβων ($height(rb) = bh\_star(rb) + true\_rh(rb)$).

Με την βοήθεια του thm_no_red_seq δείχνω το θεώρημα thm_true_rh_UB, σύμφωνα με το οποίο το "πραγματικό κόκκινο ύψος" true_rh δεν μπορεί να υπερβεί το bh_star. Στην συνέχεια, προκύπτει σχεδόν άμεσα το πόρισμα corollary1, δηλαδή ότι : $true\_rh(rb) \leq bh(rb)$, και σε συνδυασμό με το προαναφερθέν thm_bh_EQ_bh_star, συνεπάγεται το thm_true_height_UB : $true\_height(rb) + bh\_star(rb) \leq 2 \times bh(rb)$. Τελικά, από το παραπάνω, και αξιοποιώντας ξανά το thm_height_EQ, αποδεικνύω το thm_height_UB, που είναι και το ζητούμενο : $height(rb) \leq 2 \times bh(rb)$.

```
---------------------------------------------
-- | Costs on Red Black Trees --------------
---------------------------------------------

-- cost based on #(comparisons)

{-@ empty_tree :: Tick (Tree v) @-}
empty_tree :: Tick (Tree v)
empty_tree = pure E

-- cost constrain based on height.. bh bounds as lemma's, done above...
{-@ lookup :: Key -> v -> rb: RBTree v ->{t:Tick v | tcost t <= height rb } @-}
lookup :: Key -> v -> Tree v -> Tick v
lookup x def E = pure def
lookup x def (T _ tl tk tv tr)
--    | x < tk = step 1 $ lookup x def tl
  | x < tk = let Tick c v = lookup x def tl in Tick (c+1) v
  | x > tk = let Tick c v = lookup x def tr in Tick (c+1) v
  | otherwise    = wait (tv)


{-@ balance :: Tree v -> Tree v @-}
balance :: Tree v -> Tree v
balance E = E
balance (T Red t1 k vk t2)
    = T Red t1 k vk t2
balance (T _ (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) -- right-rotation ()
balance (T _ (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) -- right-rotation
    ↪ (left-rotation ())
balance (T _ 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 (T _ 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 (T _ t1 k vk t2) = T Black t1 k vk t2

{-@ makeBlack :: Tree v -> Tree v @-}
makeBlack :: Tree v -> Tree v
```

```
302 makeBlack E = E
303 makeBlack (T _ t1 x vx t2) = T Black t1 x vx t2
304
305 {-@ ins :: Key -> v -> rb : Tree v -> {t:Tick (Tree v) | tcost t <= height rb}
    ↪  @-}
306 ins :: Key -> v -> Tree v -> Tick (Tree v)
307 ins x vx E = pure $ T Red E x vx E
308 ins x vx (T c a y vy b) =
309     if x < y      then let Tick c1 t1 = (ins x vx) a
310                        in Tick (c1+1) $ balance (T c t1 y vy b )
311     else if x > y then let Tick c1 t1 = (ins x vx) b
312                        in Tick (c1+1) $ balance (T c a  y vy t1)
313     else wait $ T c a x vx b
314
315 {-@ insert :: Key -> v -> rb : RBTree v -> {t:Tick (Tree v) | tcost t <= height
    ↪  rb } @-}
316 insert :: Key -> v -> Tree v -> Tick (Tree v)
317 insert x vx s = let Tick c1 t1 = ins x vx s in Tick c1 $ makeBlack t1
```

Η `insert` προσθέτει στο υπάρχον Red-Black δεντρο μια νέα εγγραφή με την βοήθεια της `ins`, και χρωματίζει με μαύρο την ρίζα του τελικά διαμορφωμένου δέντρου (Ιδιοτητα 2).

Η `ins` με την σειρά της, εντοπίζει την θέση-υποδέντρο όπου πρέπει να τοποθετηθεί η νέα εγγραφή, και θα επιστρέψει το συνολικό δέντρο, αφού πρώτα εφαρμόσει κάποιον μετασχηματισμό-περιστροφή. Τον ρόλο αυτό αναλαμβάνει η `balance`. Αρχικά, όταν εισάγεται ένα στοιχείο στο δέντρο, του αποδίδεται κόκκινο χρώμα. Έτσι οι μόνες ιδιότητες που μπορεί να παραβιαστούν με την εισαγωγή, είναι οι 2 (ρίζα μαύρου χρώματος) και 4 (όχι κόκκινος κόμβος με παιδί κόκκινου χρώματος). Με την `balance`, θεωρούμε ευθύνη του κόμβου-"παππού" να διορθώσει αυτές τις παραβιάσεις. Όπως φαίνεται από την υλοποίησή της, αλλά και πιο ξεκάθαρα από το σχήμα 4.1, σε όλες τις περιπτώσεις εισαγωγής όπου παραβιάζεται η ιδιότητα 4 εφαρμόζουμε την ίδια λύση: ανακατατάσσουμε τον μαύρο κόμβο με τους δύο κόκκινους, δημιουργώντας ένα υποδέντρο με κόκκινη ρίζα, που κατά τα άλλα έχει όλες τις επιθυμητές ιδιότητες ενός Red Black Tree. Έχοντας ισορροπήσει το υποδέντρο, ο κόκκινος κόμβος-ρίζα, υπάρχει τώρα περίπτωση (στο συνολικό δέντρο) να αποτελεί παιδί ενός άλλου κόκκινου κόμβου. Αυτός είναι και ο λόγος που η `ins` γράφτηκε με τέτοιο τρόπο ώστε να συνεχίζει η εξισορρόπηση από τον κόμβο εισαγωγής προς τα πάνω, μέχρι την πραγματική ρίζα. [Okas99]

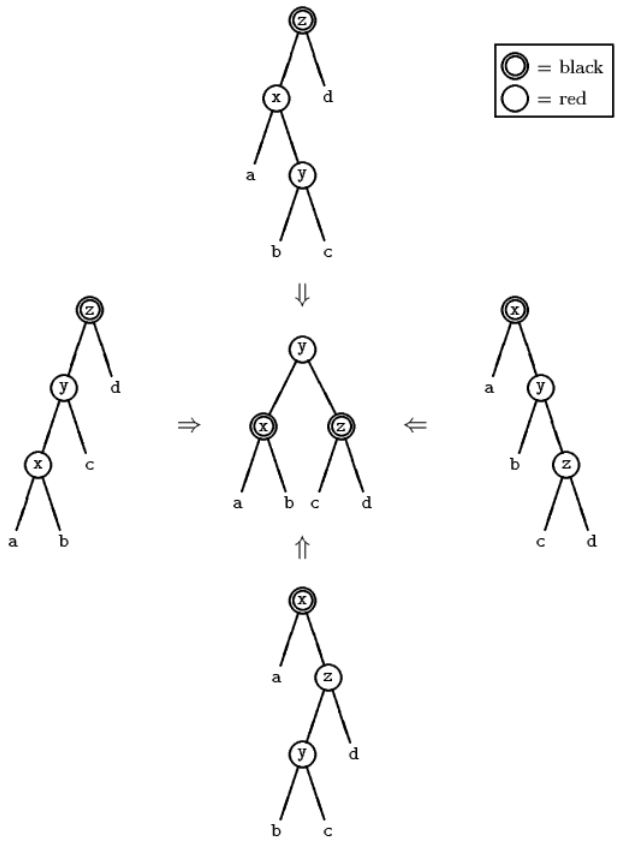
Να σημειώσουμε ότι στην `insert` ο τύπος αρχικού δέντρου, που δίνεται ως όρισμα, δεν είναι απλώς ο **`Tree`** αλλά ένα refinement αυτού που βρίσκεται στην γραμμή 106:

```
106 {-@ type RBTree v = { rb : Tree v | no_redchain rb && bl_balance rb} @-}
```

Με δεδομένο λοιπόν ότι το αρχικό δέντρο έχει την **`RBTree`** ιδιότητα, μπορεί να επιβεβαιωθεί από τον verifier ότι η εισαγωγή στοιχείου δεν χρειάζεται παραπάνω χρόνο από όσο είναι το συνολικό ύψος του δέντρου.

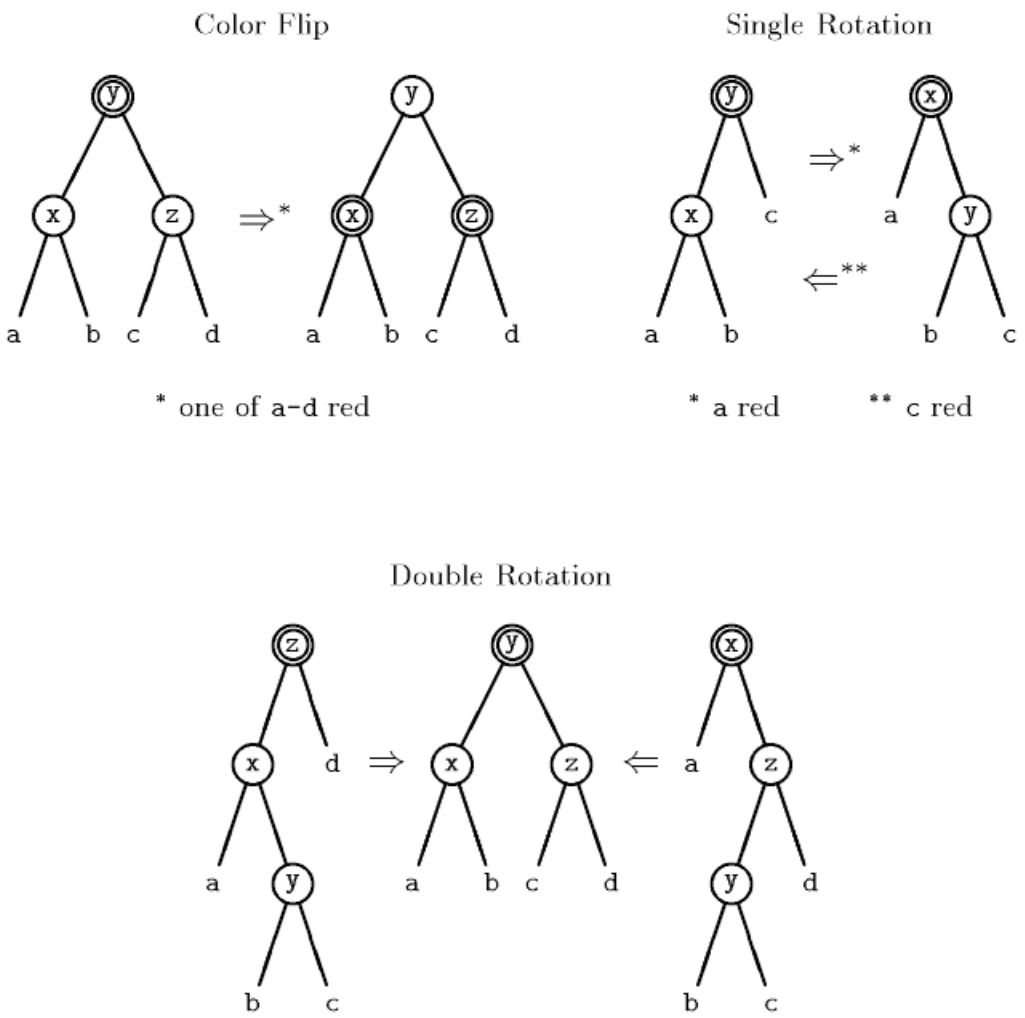
Σύγκριση συναρτησιακής υλοποίησης με προστακτική

Η έννοια της "περιστροφής", που αναφέρθηκε παραπάνω, προέρχεται κυρίως από τις προστακτικές υλοποιήσεις όπου η εξισορρόπηση του δέντρου γίνεται μέσα από συνδυασμούς αριστερών/δεξιών στροφών ανάλογα με την περίπτωση (Σχήμα 4.2). Τέτοιες υλοποιήσεις, συνήθως είναι πιο περίπλοκες, αφού αναγνωρίζουν διπλάσιες "ενδιαφέρουσες" κατηγορίες υποδέντρων που χρειάζονται εξισορρόπηση (οκτώ αντί για τέσσερις που έχουμε στην συναρτησιακή υλοποίηση), αλλά και γιατί δεν αντιμετωπίζονται όλες οι περιπτώσεις με τον ίδιο μετασχηματισμό. Προτιμούνται, ωστόσο, γιατί με αυτές η εξισορρόπηση απαιτεί αισθητά λιγότερες ενέργειες όπως ανάθεση διαφορετικού χρώματος, ή εναλλαγή δεικτών. Επίσης, αν φανταστούμε την διαδικασία εισαγωγής να αποτελείται από την top-down



Σχήμα 4.1: Σχηματική απεικόνιση του μετασχηματισμού που εφαρμόζει η `balance`. Από [Okas99]

φάση αναζήτησης και την bottom-up φάση εξισορρόπησης, η δεύτερη μπορεί να επιταχυνθεί στις προστακτικές υλοποιήσεις. Κάθε μετασχηματισμός-στροφή που οδηγεί σε υποδέντρο μαύρης ρίζας, μπορεί να τερματίσει την διάσχιση νωρίτερα, χωρίς να χρειάζεται να φτάσουμε μέχρι την κορυφή του δέντρου. Ωστόσο, για γλώσσες όπως η Haskell, οι παραπάνω λειτουργίες δεν θα είχαν ιδιαίτερο όφελος, καθώς μία αμιγώς συναρτησιακή γλώσσα δεν υποστηρίζει destructive updates. Επομένως, η δημιουργία νέων κόμβων αντί για την τροποποίηση των προηγούμενων δεν αποτελεί επιπλέον κόστος. Το ίδιο και για την διάσχιση εξισορρόπησης, αφού δεν μπορούμε να παρακάμψουμε την bottom-up κατασκευή της δομής, δεν έχουμε λόγο να επιδιώξουμε μετασχηματισμούς που παράγουν υποδέντρα με μαύρες ρίζες. [Okas99]



Σχήμα 4.2: Εναλλακτικοί μετασχηματισμοί εξισορρόπησης, που συναντάμε σε προστακτικές γλώσσες. Τα υποδέντρα a-d έχουν μαύρες ρίζες, εκτός αν αναφέρεται κάτι άλλο. [Okas99]

4.7 Trie

Σε αυτό το κεφάλαιο θα υλοποιήσουμε (συναρτησιακά) ένα πρόβλημα που, σε μια προστακτική γλώσσα, θα είχε πολύ εύκολη υλοποίηση γραμμικού χρόνου. Πιο συγκεκριμένα θέλουμε να υπολογίσουμε σε δοσμένη ακολουθία αριθμών, πόσες φορές επαναλαμβάνεται κάποιος όρος. Αν γνωρίζουμε ότι το μήκος της ακολουθίας είναι N και το εύρος των όρων το $[0, 2N]$, αρκεί ένας πίνακας $visited$ μεγέθους $2N$ όπου θα φαίνονται οι τιμές που έχουν προσπελαστεί μέχρι την i-οστή επανάληψη, και ένας μετρητής που θα αυξάνεται κάθε φορά που το $visited[input[i]] > 0$.

Στην υλοποίηση που ακολουθεί, χρησιμοποιώ τον τύπο `data Positive = XI Positive | X0 Positive | XH` για την αναπαράσταση των φυσικών αριθμών, αντί του `Int`. Έτσι, κάθε αναζήτηση (lookup) ενός αριθμού γίνεται στο `Trie_table` - διασχίζοντας ουσιαστικά ένα prefix tree - και απαιτεί χρόνο το πολύ όσο του βάθος του δέντρου. Με την συνάρτηση `t_insert`, θέτουμε στο σημείο του `Trie_table` a που μας υποδεικνύει το πρώτο όρισμα τύπου `Positive`, την τιμή τύπου `a`. Στην περίπτωση που το δέντρο δεν ήταν αρκετά μεγάλο, κάνουμε την κατάλληλη επέκτασή του πρώτα, καλύπτοντας έτσι όχι μόνο την `update` λειτουργία ενός `lookup table`, αλλά και την `insert`.

Για λόγους σύγκρισης προηγείται μια απλοϊκή υλοποίηση του lookup table, με λίστες. Ελέγχουμε κάθε νέο αριθμό, αν υπάρχει στην λίστα με τα "$visited$ κλειδιά", και αναλόγως, είτε αυξάνουμε τον

μετρητή ”συγκρούσεων”, είτε κάνουμε `update` το `lookup table`, προσαρτώντας ουσιαστικά το νέο στοιχείο στο τέλος της λίστας. Αυτή η διαδικασία λύνει το αρχικό πρόβλημα σε $O(n^2)$, αν n το μήκος της λίστας εισόδου. (βλ. κόστος στην `t_loop`)

Η πολυπλοκότητα αυτή προκύπτει από τα $O(n)$ διαδοχικά `lookup` σε ένα `looup_table` μεγέθους $O(n)$. Αν, λοιπόν, αντικαταστούσαμε την λίστα με κάποιο ισορροπημένο δέντρο, για παράδειγμα με την χρήση των RedBlack Trees που είδαμε παραπάνω, το παραπάνω κόστος θα μπορούσε να πέσει σε $O(n \cdot log(n))$.

Ωστόσο, σε αυτό το υποκεφάλαιο, θεωρούμε ότι οι πράξεις μεταξύ ακεραίων, δεν γίνονται σε σταθερό χρόνο, αλλά ανάλογο με το μήκος την αναπαράστασής τους. Με αυτό το δεδομένο, η επιλογή της δομής (Binary) Trie πετυχαίνει χρόνο $O(n \cdot log(n))$, χωρίς να επιβαρύνεται με το overhead της σύγκρισης ακεραίων (βλ. κόστος στην `fast_collisions`) .

```
module Vfa.Trie_costs where

import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Examples.Arithmetic
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , (=<<)
  , lookup
  )



import Vfa.Trie


--------------------------------------------
-- | 0. Helping functions for refinements --
--------------------------------------------


{-@ reflect my_max @-}
{-@ my_max :: Ord a => a -> a -> a  @-}
my_max :: Ord a => a -> a -> a
my_max a b = if a >= b then a else b

{-@ measure trie_size @-}
{-@ trie_size :: Trie a -> Nat @-}
trie_size :: Trie a -> Int
trie_size Leaf = 0
trie_size (Node l _ r) = 1 + trie_size l + trie_size r

{-@ measure trie_depth @-}
{-@ trie_depth :: Trie a -> Nat @-}
```

```
trie_depth :: Trie a -> Int
trie_depth Leaf = 0
trie_depth (Node l _ r) = 1 + max
  where dl = trie_depth l
        dr = trie_depth r
        max = if dl >= dr then dl else dr

{-@ measure pos_len @-}
{-@ pos_len :: Positive -> Nat @-}
pos_len :: Positive -> Int
pos_len XH = 1
pos_len (XI q) = 1 + pos_len q
pos_len (X0 q) = 1 + pos_len q

{-@ measure max_pos_len @-}
{-@ max_pos_len :: [Positive] -> Nat @-}
max_pos_len :: [Positive] -> Int
max_pos_len [] = 0
max_pos_len (l:ls) = max
  where l1  = pos_len l
        l2  = max_pos_len ls
        max =  if l1 >= l2 then l1 else l2




-- ---------------------------------------------
-- -- | 1. Costs on Maps (before Trie) --------
-- ---------------------------------------------

{-@ t_update :: t1:Tick (TotalMap a) -> Int -> a
      -> {t2 : Tick (TotalMap a) | tcost t2 = tcost t1 + 1
             && (tm_length(tval t2) = 1+ tm_length(tval t1) )} @-}
t_update :: Tick (TotalMap a) -> Int -> a -> Tick (TotalMap a)
t_update (Tick n x) k v = Tick (n+1) (update x k v)

{-@ t_lookup :: a -> Int -> l:[(Int, a)]
      -> {t : Tick a | tcost t <= length l}  @-}
t_lookup :: a -> Int -> [(Int, a)] -> Tick a
t_lookup def _ []  = pure def
t_lookup def x ((k,v):kvs)
  | x == k    = wait v
  | otherwise = step 1 (t_lookup def x kvs)


-- tcost $ t_lookup False 1 [(2, True), (1, True),(2, True), (2, True), (2, True), (2, True)] --> 2
-- tcost $ t_lookup False 1 [(2, True), (2, True),(2, True), (2, True), (2, True), (2, True)] --> 6
-- tcost $ t_lookup False 1 [(2, True), (2, True),(2, True), (2, True), (2, True), (1, True)] --> 6
```

```haskell


{-@
t_lookup_total :: a -> Int -> tm : (TotalMap a)
  -> {tick : Tick a | tcost tick <= tm_length tm } @-}
t_lookup_total :: a -> Int -> TotalMap a -> Tick a
t_lookup_total def x (TM d kvs) = t_lookup def x kvs

{-@ t_loop :: l1 :[Int] -> Int -> m1: TotalMap Bool
      -> {t:Tick Int | tcost t <= (length l1)*(length l1 + tm_length m1) + 1  }
↪  @-}
t_loop :: [Int] -> Int -> TotalMap Bool -> Tick Int
t_loop []   c _ = pure c
t_loop (a:al) c table =
  step (cst + 1)
    (if val
        then t_loop al (c+1) table
        else t_loop al c     (update table a True))
  where Tick cst val = t_lookup_total False a table

-- t_collisions input = t_loop input 0 (TM False [])

---------------------------------------------
-- | 2. Cost on Trie  ----------------------
---------------------------------------------


{-@ t_look :: a -> Positive -> tr:Trie a
      -> {t:Tick a | tcost t <= trie_depth tr } @-}
t_look :: a -> Positive -> Trie a -> Tick a
t_look def _ Leaf = pure def
t_look def (XH)   (Node l x r) = wait x
t_look def (X0 j) (Node l x r) = step 1 $ t_look def j l
t_look def (XI j) (Node l x r) = step 1 $ t_look def j r

-- sta trie_table to 'trie' einai to 2o stoixeio ths tuple
{-@ t_lookup_pos :: Positive -> tt : Trie_table a
      -> {t : Tick a | tcost t <= trie_depth (snd tt) } @-}
t_lookup_pos :: Positive -> Trie_table a -> Tick a
t_lookup_pos i table = t_look (fst table) i (snd table)

{-@ reflect t_ins @-}
{-@ t_ins :: a -> p:Positive -> a -> tr:Trie a
    -> {t:Tick (Trie a) | (tcost t <= pos_len p )
    && (trie_depth (tval t) == my_max (trie_depth tr) (pos_len p) ) } @-}
t_ins :: a -> Positive -> a -> Trie a ->  Tick (Trie a)
t_ins def (XH)   a Leaf         = wait (Node Leaf a Leaf)
t_ins def (X0 q) a Leaf         = let Tick c n = t_ins def q a Leaf
                                  in Tick (c+1) (Node n def Leaf)
t_ins def (XI q) a Leaf         = let Tick c n = t_ins def q a Leaf
                                  in Tick (c+1) (Node Leaf def n)
```

```
t_ins def (XH)   a (Node l x r) = wait (Node l a r)
t_ins def (X0 q) a (Node l x r) = let Tick c n = t_ins def q a l
                                  in Tick (c+1) (Node n x r)
t_ins def (XI q) a (Node l x r) = let Tick c n = t_ins def q a r
                                  in Tick (c+1) (Node l x n)

{-@ t_insert :: p : Positive -> a -> tt:Trie_table a
    -> {t: Tick (Trie_table a) | (tcost t <= pos_len p)
    && ((trie_depth (snd (tval t))) == my_max (trie_depth (snd tt)) (pos_len p)
↪  )  } @-}
t_insert :: Positive -> a -> Trie_table a -> Tick (Trie_table a)
-- t_insert p a table = ((fst table), t_ins (fst table) p a (snd table))
t_insert p a (def, trie) = let Tick c n = t_ins def p a trie in Tick c (def, n)

-- -- t_naive_loop :: [Positive] -> Int -> Trie_table Bool -> Tick Int
-- -- t_naive_loop []     c table = pure c
-- -- t_naive_loop (a:al) c table =
-- --   step (cst + 1)
-- --     (if val
-- --         then t_naive_loop al (c+1) table
-- --         else t_naive_loop al c     (t_insert a True table))
-- --   where Tick cst val = t_lookup_pos a table

--------------------------------------------
-- | 3. Reduce Trie Cost by logN  ----------
--------------------------------------------


{-@ t_fast_loop :: plist : [Positive] -> Int -> ttable : Trie_table Bool
      -> {t:Tick Int | tcost t <= (len plist) *
                         ( (my_max  (trie_depth (snd ttable)) (max_pos_len
↪  plist)) +
                            (max_pos_len plist)
                         )} @-}
t_fast_loop :: [Positive] -> Int -> Trie_table Bool -> Tick Int
t_fast_loop []     c table = pure c
t_fast_loop (a:al) c table =
  let Tick cost1 b = t_lookup_pos a table
      Tick cost2 table2 = (t_insert a True table)
  in
    if b
    then
      step cost1 (t_fast_loop al (c+1) table)
    else
      step (cost1+cost2) (t_fast_loop al c table2)

{-@ fast_collisions :: p:[Positive]
      -> {t:Tick Int | tcost t <= (len p) *
                         (2*max_pos_len p + 1) } @-}
fast_collisions :: [Positive] -> Tick Int
fast_collisions input = t_fast_loop input 0 (False, Leaf)
```

```
---------------------------------------------
-- | Theorems- Bounds of tree depth  ----
---------------------------------------------

-- Assume ' s :
-- https://liquid.kosmikus.org/03-sets.html
--
↪  http://goto.ucsd.edu/~gridaphobe/liquid/haskell/blog/blog/2013/03/04/bounding-
↪  vectors.lhs/




{-@ assume log2Mul :: m:Int -> n:Int -> { log2 (m * n) == log2 m + log2 n } @-}
log2Mul :: Int -> Int -> Proof
log2Mul _ _ = ()

-- log2 2 = 1 assumed in Arithmetic
{-@ log2_times2 :: m:Int -> { log2(m*2) == log2 m + 1} @-}
log2_times2 :: Int -> Proof
log2_times2 x = (log2Mul x 2)


{-@ assume thm1 :: {n:Nat | n > 0} -> { log2 n + 1 == pos_len (nat2pos n)} @-}
thm1 :: Int -> Proof
thm1 _ = ()

-- -- το τρέχει αλλά του παίρνει ώρα...
-- {-@ assume thm2 :: p:Positive -> { log2 (pos2nat p) + 1 == pos_len p } @-}
-- thm2 :: Positive -> Proof
-- thm2 _ = ()


-- {-@
-- thm_bin_tree :: tr : Trie Bool -> {trie_depth tr <= 1 + (trie_depth tr)  }
-- @-}
-- thm_bin_tree :: Trie Bool -> Proof
-- thm_bin_tree _ = ()



-- {-@
-- thm_bin_tree' :: tr : Trie Bool -> {twoToPower (trie_depth tr) <= (trie_size
↪  tr)  }
-- @-}
-- thm_bin_tree' :: Trie Bool -> Proof
-- thm_bin_tree' _ = ()
```

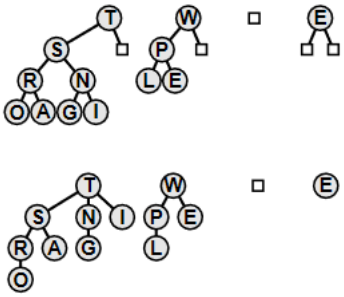
4.8 Binomial Queues

Όπως περιγράφεται στο [Sedg02, Ch. 9.7] αυτή η δομή χρησιμοποιείται όταν απαιτούνται συχνά και μεγάλα `join`, οπότε δεν θέλουμε να κοστίσουν πάνω από $O(log(n))$. Αυτός είναι και ο λόγος που ψάχνουμε κάτι αποδοτικότερο από την δομή `RedblackTree`, η οποία δεν μπορεί να πετύχει λιγότερες από $\Theta(mlog(\frac{n}{m}+1))$ συγκρίσεις [Brow79], με m, n τα μεγέθη των δύο δέντρων και $m \leq n$.

Ένα Binomial Queue είναι μια λίστα από `left-heap-ordered`, `power-of-2` σωρούς. Ωστόσο, μπορούμε να πούμε ότι η ουρά αφορά όλους τους κλασσικούς σωρούς, αφού κάθε N-ary heap μπορεί εύκολα να αναπαρασταθεί ως δυαδικό δέντρο (από το σχήμα $\swarrow$: "left child", $\rightarrow$: "right sibling"), μετατρέποντας την κλασική αναλλοίωτη του σωρού στην `left-heap-ordered` invariart, όπου κάθε κόμβος έχει μεγαλύτερη τιμή από το κάθε κόμβο του *αριστερού* υποδέντρου του. Προκειμένου κάθε σωρός να έχει τις ζητούμενες ιδιότητες, έχει στην ρίζα του δέντρο το στοιχείο με την μεγαλύτερη τιμή, ως αριστερό υποδέντρο ένα `left-heap-ordered` πλήρες δυαδικό δέντρο, ενώ το δεξί υποδέντρο είναι κενό. Παρατηρώντας την υλοποίηση της `smash` από το **`module Binom`**, γίνεται ξεκάθαρο πως οι ιδιότητες διατηρούνται κατά την συνένωση τέτοιων σωρών ίσου μεγέθους.

```
118 smash :: Tree -> Tree -> Tree
119 smash (Node x t1 Leaf) (Node y u1 Leaf)
120   = if x <= y
121     then (Node y (Node x t1 u1) Leaf)
122     else (Node x (Node y u1 t1) Leaf)
123 smash _ _ = (Node 0 Leaf Leaf)  -- arbitrary bogus tree
```

Σε ότι αφορά την δομή της λίστας, λοιπόν, στο i-οστό στοιχείο της είτε θα υπάρχει κενή εγγραφή, είτε θα υπάρχει σωρός μεγέθους 2^i της παραπάνω μορφής (`left-heap-ordered`, `power-of-2`). Έτσι ένα Binomial Queue μεγέθους n θα έχει μη κενή εγγραφή στις θέσεις της λίστα που η αντιστοιχούν στα μη μηδενικά bit της δυαδικής αναπαράστασης του n.

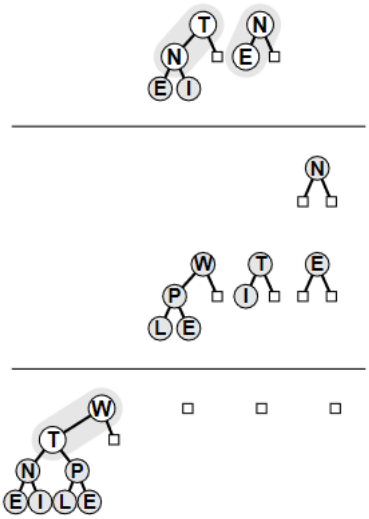


Σχήμα 4.3: Η Binomial Queue μεγέθους 13 έχει μη κενά δέντρα στις κατάλληλες θέσεις κατά αντιστοιχία με την δυαδική αναπαράσταση του αριθμού $13 = 1101_2$.Εικόνα 9.15 από [Sedg02].

Η υλοποίηση αφορά τις τρεις στοιχειώδεις πράξεις : εισαγωγή, διαγραφή μέγιστου στοιχείου, και ένωση ουρών.

Για την εισαγωγή νέου στοιχείου x στην ουρά q, χρησιμοποιούμε την `insert x q = carry q (Node x Leaf Leaf)`. Με την `carry` αναδεικνύεται η σχέση του αλγορίθμου με την δυαδική πρόσθεση. Αν το πρώτο στοιχείο της λίστας είναι κενό `Leaf`, το νέο δέντρο (μεγέθους $2^0 = 1$) παίρνει την θέση του, και η εκτέλεση ολοκληρώνεται. Στην ενδιαφέρουσα περίπτωση, όπου το πρώτο στοιχείο της λίστας είναι το μη κενό δέντρο u, τότε το αντικαθιστούμε με κενό, και συνεχίζουμε αναδρομικά με `carry (u:q') t = (Leaf : carry q' (smash t u))`. Στην i-στη αναδρομική κλίση θα έχουν ήδη τοποθετηθεί κενά δέντρα στις $i-1$ πρώτες (λιγότερο σημαντικές) θέσεις της ενώ παράλληλα θα έχουμε να προσθέσουμε δέντρο μεγέθους 2^i, στην ουρά. Εάν, λοιπόν, υπάρχει κενό δέντρο

στην τρέχουσα θέση της ουράς, η εκτέλεση τελικά θα τερματιστεί αφήνοντας στην κατάλληλη θέση το δέντρο (με τελικό μέγεθος 2^i), ενώ το συνολικό μήκος της ουράς θα παραμείνει n. Αντίθετα, αν είχαμε μία πλήρη ουρά, σε κάθε επανάληψη, η (λιγότερο σημαντική) άκρη της υπόλοιπης ουράς θα ήταν κατειλημμένη, θα ενώναμε τα δύο δέντρα για να πάρουμε δέντρο διπλάσιου μεγέθους 2^i, και θα συνεχίζαμε μέχρι τελικά να πάρουμε την νέα ουρά, μεγέθους $n+1$ όπου σε όλες της θέσεις έχει κενά δέντρα, εκτός από την "πιο σημαντική" όπου βρίσκεται το δέντρο μεγέθους 2^n.

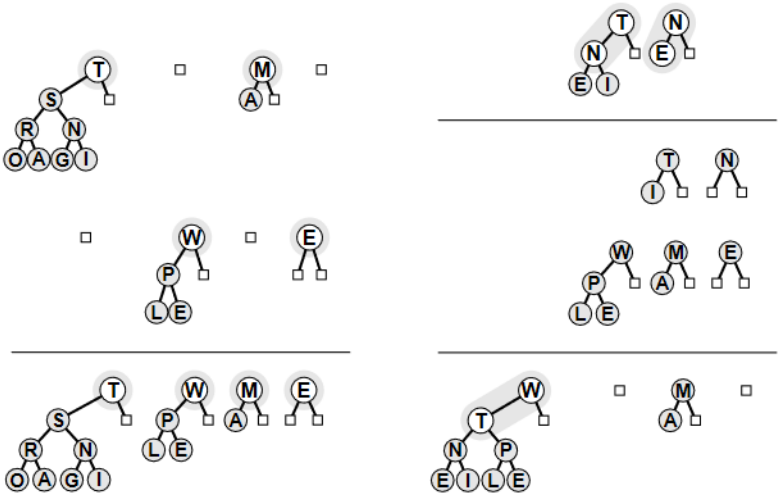


Σχήμα 4.4: Στο παράδειγμα φαίνεται η διαδικασία εισαγωγής σε υπάρχουσα Binomial Queue του στοιχείου N, που μοιάζει με την δυαδική πρόσθεση $111_2 + 1 = 1000_2$. Στην πρώτη γραμμή φαίνονται τα "κρατούμενα" που σχηματίζονται κατά την διαδικασία. Εικόνα 9.17 από [Sedg02].

Για την συνένωση δύο `prique` καλώ την `merge p q = join p q Leaf`. Η `join` σε κάθε κλίση εξετάζει την τρέχουσα "κορυφή" από δύο ουρές, και το δέντρο του τρίτου ορίσματος (κρατούμενο). Στην περίπτωση όπου 3 ή 1 από αυτά είναι μη κενά, επιλέγουμε κάποιο να μπει μπροστά, και συνεχίζουμε αναδρομικά, με ή χωρίς κρατούμενο, για το υπόλοιπο των ουρών. Αντίστοιχα και στην περίπτωση 2 ή 0 μη κενών δέντρων, μόνο που τότε θα μπει κενό δέντρο στη θέση. (Βλ. σχήμα 4.5)

Τέλος, για την διαγραφή μέγιστου στοιχείου η `delete_max` καλεί τις βοηθητικές `delete_max_aux`, `find_max` και την `join`. Με την `find_max` διασχίζουμε την λίστα, μέχρι να βρεθεί η μέγιστη τιμή από όλα τα δέντρα της δομής. Η `delete_max_aux`, στην συνέχεια εντοπίζει το δέντρο με ρίζα αυτήν την τιμή, και επιστρέφει την λίστα απαλλαγμένη από το δέντρο αυτό. Στην συνέχεια, μεσώ της `heap_delete_max` και της `unzip`, αφού αφαιρεθεί η ρίζα (μέγιστο entry), εξάγουμε από το αριστερό υποδέντρο(πλήρες δυαδικό) ένα binomial queue (παίρνοντας κάθε φορά ρίζα και αριστερό υποδέντρο, και συνεχίζοντας αναδρομικά με το δεξί υποδέντρο). Τέλος, μέσω της `merge` "προσθέτω" τις δύο binomial queues για να πάρω την τελική απάντηση.

Για το συγκεκριμένο καφάλαιο επέλεξα να φτιάξω δύο modules, τα "Binom" και "Binom_costs". Μέχρι τώρα έχω περιγράψει το πρώτο. Στο "Binom_costs", τροποποιώ τις παραπάνω υλοποιήσεις, στο ελάχιστο δυνατό, μετατρέποντας κάθε τελικό τύπο α σε Tick α, προκειμένου να μπορέσει - στα refinements - να γίνει reasoning που να αφορά την ανάλυση κόστους. Δεδομένου ότι οι Binomial queues αποτελούν λίστες, όπου στην θέση i είτε έχουμε ένα κενό δέντρο, είτε ένα left-heap-ordered μεγέθους 2^i, αν $n = 2^k$ είναι το μέγεθος του μέγιστου δέντρου, συνολικά το μέγεθος των κόμβων του binomial queue θα είναι $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 = O(n)$. Επίσης, τόσο το μήκος της λίστας, όσο και το ύψος των δέντρων είναι $O(\log n)$ Για την εισαγωγή στοιχείων, βλέπουμε ότι η `insert` έχει refinement `tcost tout <= tcost tin + 2*(length (tval tin)) + 1`. Με δεδομένο ότι το μήκος μια έγκυρης λίστα είναι $O(\log n)$, κάθε insert επιβαρύνει το κόστος μιας λί-



Σχήμα 4.5: Εδώ βλέπουμε δύο περιπτώσεις συνένωσης ουρών, η πρώτη χωρίς και η δεύτερη με κρατούμενο. Αντιστοιχία με τις προσθέσεις $1010_2 + 0101_2 = 1111_2$ και $0011_2 + 0111_2 = 1010_2$. Εικόνες 9.19 και 9.20 από [Sedg02].

στας κατά $2 * \log n + 1 = O(\log n)$. Η ένωση δύο λιστών που υλοποιείται ουσιαστικά με την `join :: p1-> p2 -> {t:Tick Prique | tcost t <= 2*(length p1 + length p2) + 1 }`, δηλαδή $O(max\{\log n_1, \log n_2\})$. Για την διαγραφή του μέγιστου στοιχείου ο χρόνος εκτέλεσης εξαρτάται από τις `join` και `find_max` που φράσσονται από το μήκος της ουράς, και από την `delete_max_aux` που φράσσεται από το μήκος της ουράς και από το μέγιστο ύψος δέντρου. Σε κάθε περίπτωση, ο συνολικός χρόνος εκτέλεσης επιβεβαιώνεται ότι είναι $O(\log n)$.

```
module Vfa.Binom_costs where

import Examples.RTick
import Examples.ProofCombinators
import Examples.Lists
import Examples.Arithmetic
import Prelude hiding
  ( Functor(..)
  , Applicative(..)
  , Monad(..)
  , drop
  , length
  , take
  , (=<<)
  , lookup
  , unzip
  )

import qualified Data.Maybe as M

import qualified Vfa.Binom as B
import Vfa.Binom (Tree(..), Prique, Key, tree_depth)

```

```haskell
{-@ reflect isBounded @-}
{-@ isBounded :: Prique -> Int-> Bool @-}
isBounded :: Prique -> Int-> Bool
isBounded []      _ = True
isBounded (x:xs) n = tree_depth x <= n && isBounded xs n


{-@ reflect smash @-}
{-@ smash ::  Tree -> Tree -> {t:Tick (Tree) | tcost t == 1 || tcost t == 0 }
@-}
smash :: Tree -> Tree -> Tick Tree
smash (Node x t1 Leaf) (Node y u1 Leaf)
  = wait $ B.smash (Node x t1 Leaf) (Node y u1 Leaf)
smash _ _ = pure Leaf  -- arbitrary bogus tree

{-@ reflect carry @-}
{-@ carry :: l:[Tree] -> Tree -> {t:Tick [Tree] | tcost t <= 2*(length l) + 1 &&
↪  tcost t >= 0  } @-}
carry :: [Tree] -> Tree -> Tick [Tree]
carry []        Leaf = pure []
carry []        t    = wait [t]
carry (Leaf:q') t    = wait (t : q')
carry (u:q')    Leaf = wait (u : q')  -- 1st case can be unified with this
-- carry (u:q')    t    = let Tick m x = (smash t u) in (waitN m (Leaf :)) <*>
↪  (carry q' x)
carry (u:q')    t    = let Tick m x = (smash t u)
                           Tick n y = carry q' x
                       in waitN (m+n) (Leaf : y)
-- carry (u:q')    t    = ((smash t u) >>= (carry q')) >>= ( \x -> pure ( Leaf :
↪  x)) -- does not accept the refinement....

{-@ reflect insert @-}
-- {-@ insert :: Key -> tin:Tick Prique -> {tout:Tick Prique | tcost tout >=
↪  tcost tin } @-}
{-@ insert :: Key -> tin:Tick Prique
  -> {tout:Tick Prique | tcost tout <= tcost tin + 2*(length (tval tin)) + 1 }
↪  @-}
insert :: Key -> Tick Prique -> Tick Prique
insert x q = let Tick m q'   = q
                 priq = carry q' (Node x Leaf Leaf)
             in step m priq
-- insert x q = q >>= ins                              -- ATTENTION : this
↪  Monad's >>= function essentially
--   where ins = (\p -> carry p (Node x Leaf Leaf))    --             combines
↪  the resource usage of its two subexprs

--  Eval compute in fold_left (fun x q  insert q x) [3;1;4;1;5;9;2;3;5] empty.
{-@ compute ::  Tick Prique @-}
compute ::  Tick Prique
compute = foldl (\x q -> insert q x) (pure []) [3, 1, 4, 1, 5, 9, 2, 3, 5]
```

```
{-@ join :: p1:Prique -> p2:Prique -> tree:Tree
      -> {t:Tick Prique | tcost t <= 2*(length p1 + length p2) + 1 } @-}
join :: Prique -> Prique -> Tree -> Tick Prique
join [] q c                 = carry q c
join p [] c                 = carry p c
join (Leaf:p') (Leaf:q') c    = let Tick n x = join p' q' Leaf
                                in  pure (c:) </> (Tick n x)
join (Leaf:p') (q1:q')   Leaf = let Tick n x = join p' q' Leaf --q1   : (join p' q' Leaf)
                                in  pure (q1 :) </> (Tick n x)
join (Leaf:p') (q1:q')   c    = let Tick n x = smash c q1  -- Leaf : (join p' q' (smash c q1))
                                    Tick m y = join p' q' x
                                in  Tick (n+m+1) (Leaf : y)
join (p1:p')   (Leaf:q') Leaf = let Tick n x = join p' q' Leaf  --  p1   : (join p' q' Leaf)
                                in  Tick (n+1) (p1:x)
join (p1:p')   (Leaf:q') c    = let Tick n x = smash c p1 -- Leaf : (join p' q' (smash c p1))
                                    Tick m y = join p' q' x
                                in  Tick (n+m+1) (Leaf : y)
join (p1:p')   (q1:q')   c    = let Tick n x = smash p1 q1  --  c    : (join p' q' (smash p1 q1))
                                    Tick m y = join p' q' x
                                in  Tick (n+m+1) (c:y)


-- Θα πάρω την εκδοχή του Πέτρου για unzip (αντι του αρχικού από SoftwareFoundations)

{-@ unzip :: tree:Tree -> q:Prique
      -> {t:Tick Prique | tcost t <= tree_depth tree && tcost t >= 0
               && length (tval t) <= tree_depth tree + (length q) } @-}
unzip :: Tree -> Prique  -> Tick Prique
unzip (Node x t1 t2) cont = step 1 $ unzip t2 ((Node x t1 Leaf): cont )
unzip Leaf           cont = pure cont

{-@ heap_delete_max :: tree:Tree
      -> {t:Tick Prique | tcost t <= tree_depth tree && tcost t >= 0
               && length (tval t) <= tree_depth tree} @-}
heap_delete_max :: Tree -> Tick Prique
heap_delete_max (Node x t1 Leaf) = let Tick n x = unzip t1 [] in Tick n x
heap_delete_max _ = pure [] -- bogus value for ill-formed or empty trees


{-@ find_max' :: Key -> l:Prique
      -> {t:Tick Key | tcost t <= length l } @-}
find_max' :: Key -> Prique -> Tick Key
find_max' current []                = pure current
find_max' current (Leaf:q')         = step 1 $ find_max' current q'
```

```
find_max' current ((Node x _ _):q') = step 1 $ find_max' (if x>current then x else current) q'


{-@ find_max :: l:Prique -> {t:Tick (Maybe Key)
                                | tcost t <= length l } @-}
find_max :: Prique -> Tick (Maybe Key)
find_max [] = pure Nothing
find_max (Leaf:q')          = step 1 $ find_max q'
find_max ((Node x _ _):q') = let Tick n y = (find_max' x q') in Tick n (Just y)

{-@ thm_help :: q:{Prique | (isBounded q (length q)) && q /= [] } -> {isBounded (tail q) (length q)} @-}
thm_help :: Prique -> Proof
thm_help []     = ()
thm_help [_]    = ()
thm_help (x:xs) = ()

{-@ delete_max_aux :: l: Nat -> Key -> p:{Prique |  isBounded p l}
            -> t:{Tick (Prique,Prique) | (tcost t <= length p + l + 1)
                                  && length (fst (tval t)) + length (snd (tval t)) <= length p + l}
                @-}
delete_max_aux :: Int -> Key -> Prique -> Tick (Prique,Prique)
delete_max_aux l m [Leaf]             = pure ([],[])  -- bogus value --
delete_max_aux l m (Leaf:p')            = let Tick n (j,k) = delete_max_aux l m p' in Tick (n+1) (Leaf:j,k)
delete_max_aux l m ((Node x t1 Leaf):p') = if m > x
                                        then let Tick n (j,k) = delete_max_aux l m p'
                                          in Tick (n+1) ((Node x t1 Leaf):j, k)
                                       else let Tick n y = heap_delete_max (Node x t1 Leaf)
                                              in Tick (n+1) (Leaf:p', y)
delete_max_aux l m _                      = pure ([],[])  -- bogus value

{-@ delete_max :: {p:Prique | isBounded p (length p)} -> t:{Tick (Maybe (Key, Prique))
           |  tcost t <= (length p) + (2*length p +1) + (2*(length p + length p) + 1) + 1 } @-}
delete_max :: Prique -> Tick (Maybe (Key, Prique))
delete_max q
  | M.isNothing f = pure Nothing
  | otherwise     = let m                = M.fromJust f
                        Tick n2 (p', q') = delete_max_aux (length q) m q
                        Tick n3 j        = join p' q' Leaf
                  --   in  Tick (n1+n2+n3+1) (Just (m, j))
                    in  Tick (n1+n2+n3+1) (Just (m, j))
    where Tick n1 f = find_max q


```

```
{-@ merge :: p1:Prique -> p2:Prique
      -> {t:Tick Prique | tcost t <= 2*(length p1) + 2*(length p2) + 1 } @-}
merge :: Prique -> Prique -> Tick Prique
merge p q = join p q Leaf
```

Κεφάλαιο 5

Συμπεράσματα

5.1 Συνεισφορά

Στα πλαίσια της παρούσας εργασίας είχαμε την δυνατότητα να κάνουμε μια εισαγωγή στην Liquid-Haskell, και δούμε πως μπορεί να αξιοποιηθεί ως proof assistant, τόσο με intrinsic όσο και με extrinsic proofs. Επίσης είδαμε πως μπορεί κανείς να σχεδιάσει μια βιβλιοθήκη, ώστε να μπορεί να συμπεριληφθεί στην υλοποίηση του προγράμματος η κατανάλωση υπολογιστικών πόρων, και χρήσιμους τελεστές για την απόδειξη θεωρημάτων αναφορικά με το κόστος κάθε συνάρτησης. Τέλος, θέσαμε σε εφαρμογή τις παραπάνω προτάσεις, και υλοποιώντας γνωστούς αλγορίθμους σε LH δοκιμάσαμε να επαληθεύσουμε τις αναμενόμενες ιδιότητες κόστους που είναι γνωστές από την θεωρία αλγορίθμων.

5.2 Προτάσεις μελλοντικής Έρευνας

Στην παρούσα εργασία είδαμε εφαρμογές σχετικά με την χρονική πολυπλοκότητα των αλγορίθμων. Ακολουθώντας την ίδια μεθοδολογία, θα μπορούσαμε να επαληθεύσουμε ιδιότητες σχετικά με την μνήμη που δεσμεύει κάθε αλγόριθμος. Επίσης, δεδομένου ότι τα παραδείγματα είναι υλοποιημένα σε Haskell, θα είχε ενδιαφέρον να μελετήσουμε την πολυπλοκότητα προγραμμάτων, υπό την υπόθεση οκνηρής αποτίμησης.

Ακόμα, ως μελλοντική έρευνα θα μπορούσαμε να υλοποιήσουμε αντίστοιχες βιβλιοθήκες που καταγράφουν την κατανάλωση πόρων, για άλλες γλώσσες - proof assistants, όπως οι Adga, Coq, Idris. Στην συνέχεια, θα ήταν χρήσιμο να συγκρίνουμε τα σύστημα υποστήριξης αποδείξεων μεταξύ τους κοιτώντας χαρακτηριστικά όπως ο χρόνος επαλήθευσης ή το μήκος των αποδείξεων και specification για τις ιδιότητες κατανάλωσης που θα επιχειρούσαμε να αποδείξουμε.

Βιβλιογραφία

[Appe11] Andrew W. Appel, "Efficient Verified Red-Black Trees", 2011.

[Appe17] Andrew W. Appel, "Software Foundations, Volume 3: Verified Functional Algorithms", `https://softwarefoundations.cis.upenn.edu/vfa-current/index.html`, 2017.

[Appe21] Andrew W. Appel, *Verified Functional Algorithms*, vol. 3 of *Software Foundations*, Electronic textbook, 2021. Version 1.5, `http://softwarefoundations.cis.upenn.edu`.

[Brow79] Mark R. Brown and Robert E. Tarjan, "A Fast Merging Algorithm", *J. ACM*, vol. 26, no. 2, p. 211–226, April 1979.

[Corm09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd edition, 2009.

[dMou09] Leonardo de Moura and Nikolaj Bjørner, "Satisfiability Modulo Theories: An Appetizer", in Marcel Vinícius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, pp. 23–36, Berlin, Heidelberg, 2009, Springer Berlin Heidelberg.

[Down80] Peter J. Downey, Ravi Sethi and Robert Endre Tarjan, "Variations on the Common Subexpression Problem", *J. ACM*, vol. 27, no. 4, p. 758–771, October 1980.

[Hand] Martin A. T. Handley, Niki Vazou and Graham Hutton, "RTick Library and Proofs Described in the Paper".

[Hand19] Martin A. T. Handley, Niki Vazou and Graham Hutton, "Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell", *Proc. ACM Program. Lang.*, vol. 4, no. POPL, December 2019.

[Huda99] Paul Hudak, John Peterson and Joseph Fasel, "A Gentle Introduction to Haskell 98", 1999.

[Jhal20] Ranjit Jhala and Niki Vazou, "Refinement Types: A Tutorial", 2020.

[Nels80] Charles Gregory Nelson, *Techniques for Program Verification*, Ph.D. thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

[Okas99] Chris Okasaki, "Red-Black Trees in a Functional Setting", *J. Funct. Program.*, vol. 9, no. 4, p. 471–477, July 1999.

[Peñ17] Ricardo Peña, "An Introduction to Liquid Haskell", *Electronic Proceedings in Theoretical Computer Science*, vol. 237, p. 68–80, Jan 2017.

[Rond08] Patrick M. Rondon, Ming Kawaguci and Ranjit Jhala, "Liquid Types", *SIGPLAN Not.*, vol. 43, no. 6, p. 159–169, June 2008.

[Sedg02] Robert Sedgewick and Michael Schidlowsky, *Algorithms in Java, Parts 1–4 (Fundamental Algorithms, Data Structures, Sorting, Searching)*, Addison-Wesley, 3rd edition, 2002.

[Sedg17] Robert Sedgewick and Kevin Wayne, "Algorithms, 4th Edition", `https://algs4.cs.princeton.edu/home/`, 2017.

[Turn04] D. A. Turner, "Total Functional Programming", *Journal of Universal Computer Science*, vol. 10, no. 7, pp. 751–768, jul 2004. `http://www.jucs.org/jucs_10_7/total_functional_programming`.

[Vazo13] Niki Vazou, Patrick M. Rondon and Ranjit Jhala, "Abstract Refinement Types", in *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, p. 209–228, Berlin, Heidelberg, 2013, Springer-Verlag.

[Vazo14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis and Simon Peyton-Jones, "Refinement Types for Haskell", *SIGPLAN Not.*, vol. 49, no. 9, p. 269–282, August 2014.

[Vazo16] Niki Vazou, *Liquid Haskell: Haskell as a Theorem Prover*, UC San Diego, 2016. Permalink: `https://escholarship.org/uc/item/8dm057ws`.

[Vazo17a] Niki Vazou, Leonidas Lampropoulos and Jeff Polakow, "A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq", *SIGPLAN Not.*, vol. 52, no. 10, p. 63–74, September 2017.

[Vazo17b] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler and Ranjit Jhala, "Refinement Reflection: Complete Verification with SMT", *Proc. ACM Program. Lang.*, vol. 2, no. POPL, December 2017.

[Vazo18] Niki Vazou, Joachim Breitner, Will Kunkel, David Van Horn and Graham Hutton, "Functional Pearl: Theorem Proving for All (Equational Reasoning in Liquid Haskell)", 2018.

[Wald18] Johannes Waldmann, "When You Should Use Lists in Haskell (Mostly, You Should Not)", 2018.

[Πέ18] Γεώργιος Πέτρου, *ΕΠΑΛΗΘΕΥΣΗ ΙΔΙΟΤΗΤΩΝ ΑΛΓΟΡΙΘΜΩΝ ΣΕ LIQUID HASKELL*, Diploma thesis, Εθνικό Μετσόβιο Πολυτεχνίο, 2018.