Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Επαλήθευση Ιδιοτήτων Πολυπλοκότητας Δυαδικών Δέντρων Αναζήτησης σε Liquid Haskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρύσα Οικονόμου

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

Αθήνα, Μάρτιος 2021

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Επαλήθευση Ιδιοτήτων Πολυπλοκότητας Δυαδικών Δέντρων Αναζήτησης σε Liquid Haskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρύσα Οικονόμου

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26η Μαρτίου 2021.

Νικόλαος Σ. Παπασπύρου	Πέτρος Παπαδόπουλος	Γεώργιος Νικολάου
Καθηγητής Ε.Μ.Π.	Επίκ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Κ.Π.Α.

Αθήνα, Μάρτιος 2021

(Υπογραφή)

.....

Χρύσα Οικονόμου

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

Περίληψη

Τα τελευταία χρόνια υπάρχει μεγάλη έρευνα γύρω από την ανάπτυξη τεχνικών για την τυπική επαλήθευση των προγραμμάτων. Μας ενδιαφέρει η στατική ανάλυση των ενδεχόμενων σφαλμάτων του κώδικα αποδεικνύοντας ή απορρίπτοντας ιδιότητες ορθότητας, τερματισμού, ασφάλειας μνήμης αλλά και όχι μόνο. Είναι σημαντικό να γνωρίζουμε επίσης, το κόστος εκτέλεσης ή/και την χρήση υπολογιστικών πόρων των προγραμμάτων. Τα οφέλη είναι προφανή, από την μείωση των διαφόρων σφαλμάτων του κώδικα μέχρι την αξιοπιστία του αποδεδειγμένου συστήματος. Ωστόσο, η τυπική επαλήθευση των προγραμμάτων δεν είναι συνηθισμένη και ως επί το πλείστον προορίζεται για ορισμένες ειδικές περιπτώσεις χρήσης. Οι πιο γνωστές γλώσσες προγραμματισμού με δυνατότητες τυπικής επαλήθευσης, όπως η Coq, η Agda και η Idris, χρησιμοποιούνται κυρίως από την ακαδημαϊκή κοινότητα. Ο κύριος λόγος κρύβεται πίσω από την δυσκολία στη χρήση τους.

Στην παρούσα διπλωματική εργασία, εξερευνούμε την χρηστικότητα και τις δυνατότητες της Liquid Haskell (LH) και επιχειρούμε να συνεισφέρουμε στην βιβλιοθήκη της. Η LH είναι μία επέκταση στο σύστημα τύπων της γλώσσας προγραμματισμού Haskell. Η επέκταση αυτή αφορά την εισαγωγή των εκλεπτυσμένων τύπων (refinement types) στο υπάρχον σύστημα τύπων, με τους οποίους εκφράζουμε ιδιότητες ορθότητας και πολυπλοκότητας. Ένα υποσύνολο των εκλεπτυσμένων τύπων είναι οι Liquid Τύποι (Logical Qualified Data Types) οι οποίοι περιορίζουν ακόμη περισσότερο τη γλώσσα των κατηγορημάτων που μπορούν να υπάρχουν στους εκλεπτυσμένους τύπους, ώστε να επιτυγχάνεται αυτόματα ο συμπερασμός τύπων στα ενδιάμεσα στάδια ενός προγράμματος, χαρακτηριστικό πολύ χρήσιμο για ένα σύστημα τύπων. Συστήματα τύπων με αυτόματο συμπερασμό τύπων καθίστανται χρήσιμα πέρα από θεωρητικά και πρακτικά για την προγραμματίστρια.

Η δική μας συνεισφορά επικεντρώνεται στην ανάλυση της πολυπλοκότητας και της ορθότητας κάποιων δημοφιλών δομών δεδομένων (Binary Search Trees, Red-Black Trees) με χρήση της Liquid Haskell. Για τις περισσότερες αποδείξεις αρκεί η intrinsic μέθοδος, δηλαδή τόσο το κόστος της συνάρτησης όσο και οι ιδιότητες ορθότητας που θέλουμε να αποδείξουμε δηλώνονται στον τύπο της μέσω εκλεπτυσμένων κατηγορημάτων και αποδεικνύονται αυτόματα από τον SMT Solver. Ωστόσο, όταν τα κατηγορήματα που

συνθέτουν τους εκλεπτυσμένους τύπους δεν ανήκουν στην SMT-αποφασίσιμη λογική, χρησιμοποιούμε την extrinsic μέθοδο. Δηλαδή εκφράζουμε θεωρήματα και λήμματα για τα κόστη και τις ιδιότητες ορθότητας των συναρτήσεων χρησιμοποιώντας εκλεπτυσμένους τύπους και έπειτα γράφουμε αποδείξεις για αυτά σε Haskell, χρησιμοποιώντας συνδυαστές απόδειξης (proof combinators).

Ο κώδικάς μας μπορεί να βρεθεί στη διεύθυνση: `https://github.com/linen101/ticked-binary-search-trees`

Λέξεις κλειδιά

Liquid Haskell, Εκλεπτυσμένοι Τύποι, Τυπική Επαλήθευση, Ανάλυση Κόστους, Δυαδικά Δέντρα Αναζήτησης, Red-Black Δέντρα.

Abstract

In recent years there has been significant research for the development of techniques of formally verifying computer programs. We are interested in the static analysis of potential bugs in the code by means of proving or rejecting properties of correctness, termination, memory safety and more. It is also important to know the execution cost and/or the computational resource usage of various programs. The benefits are obvious, from significantly reducing the occurrence of bugs to the reliability of the formally proven system. Nevertheless, formal verification remains unusual and is mostly intended for special use cases. The most common programming languages that can be used for formal verification, like Coq, Agda and Idris, are mostly used by the academic community, the main reason being the difficulty in their usage.

In this dissertation we explore the usability and usefulness of Liquid Haskell (LH), while attempting to contribute to its growing library. Liquid Haskell is an extension to the type system of the Haskell programming language, pertaining to the introduction of refinement types to the existing type system, which are used to express correctness, complexity, termination and completeness properties. A subset of refinement types are Liquid Types (Logical Qualified Data Types), that limit the language of expressible predicates for refinement types, in order to enable automatic type inference in the intermediate stages of a program, a very useful characteristic of type systems, not only for its theoretical properties but also for the practicality it offers the programmer.

Our contribution is focused on the complexity and correctness analysis of specific popular data structures (Binary Search Trees, Red-Black Trees) by use of Liquid Haskell. For most proofs, the intrinsic method is enough: in this method, the function cost as well as the correctness invariants that we want to prove are declared in the function's type through refinement predicates and automatically proven by the SMT Solver. Additionally, when the predicates that compose refinement types are not part of the SMT-decidable logic, we use the extrinsic method: we express theorems and lemmas for the costs and correctness properties of the functions using refinement types and then write their proofs in Haskell, by utilizing proof combinators.

Our code can be found at:
`https://github.com/linen101/ticked-binary-search-trees`

Key words

Liquid Haskell, Refinement Types, Formal Verification, Cost Analysis, Binary Search Trees, Red-Black Trees.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Νίκο Παπασπύρου για την υποστήριξη και την καθοδήγηση που μου προσέφερε και την εμπιστοσύνη που μου έδειξε. Θέλω να ευχαριστήσω επίσης, τους συμφοιτητές και φίλους μου, Δημήτρη και Χρήστο, που με στήριξαν ανιδιοτελώς από την αρχή μέχρι το τέλος των σπουδών μου. Ακόμη, ευχαριστώ τον φίλο μου Μάνο που ήταν δίπλα μου - είτε φυσικά, είτε απομακρυσμένα - συνεχώς τα τελευταία χρόνια και τον γάτο μου Μόρρισον που ήταν μαζί μου από την αρχή σχεδόν της ακαδημαϊκής μου πορείας. Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου, που παρά τις όποιες δυσκολίες, με βοήθησε και όλες τις φίλες και τους φίλους μου που είτε πέρασαν από τη ζωή μου, είτε βρίσκονται ακόμη στο πλάι μου, δίχως την ύπαρξη των οποίων δεν θα υπήρχα.

Χρύσα Οικονόμου,

Αθήνα, 26η Μαρτίου 2021

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-21, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Μάρτιος 2021.

URL: `http://www.softlab.ntua.gr/techrep/`

FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

Το πρώτο σπουδαίο μαθηματικό θεώρημα που αποδείχθηκε μέσω υπολογιστή ήταν το θεώρημα τεσσάρων χρωμάτων, το 1976 [Appe77]. Τότε, η απόδειξη των Appel και Haken γέννησε όχι μόνο φιλοσοφικούς προβληματισμούς αλλά και προβληματισμούς που αφορούσαν την ορθότητά της. Από τους πρώτους που τοποθετήθηκαν σε σχέση με τους κινδύνους της απόδειξης μαθηματικών θεωρημάτων με τη βοήθεια μηχανών ήταν ο Thomas Tymoczko[Tymo79]. Υποστήριξε πως, από τη μία οι εκτενείς μαθηματικές αποδείξεις εξαντλητικής απαρίθμησης από περίπλοκα προγράμματα δεν είναι στην πραγματικότητα μαθηματικές αποδείξεις επειδή περιλαμβάνουν πολλές περιπτώσεις και κρύβουν μέρος της συλλογιστικής σε επιχειρήματα που δημιουργούνται από υπολογιστή και δεν μπορούν να επαληθευτούν με το χέρι. Συγκεκριμένα, η τότε εξαντλητική απόδειξη του θεωρήματος 4 χρωμάτων περιλάμβανε 1834 περιπτώσεις ενώ σήμερα η μικρότερη γνωστή παρόμοια απόδειξη λίγες περισσότερες από 600[Robe96]. Από την άλλη παρατήρησε πως η εμπιστοσύνη στον πειραματισμό και τις εμπειρικές υπολογιστικές διαδικασίες αντικαθιστά τους παραγωγικούς συλλογισμούς από αξιώματα. Άλλοι μαθηματικοί θεώρησαν πως αποτελούν υπολογισμούς και όχι αποδείξεις. Για να αποτελέσουν όντως αποδείξεις ο ίδιος ο αλγόριθμος απόδειξης πρέπει να αποδειχθεί έγκυρος, ώστε η χρήση του να μπορεί να θεωρηθεί έπειτα ως απλή επαλήθευση. Κάτι ακόμη είναι πως αυτές οι αποδείξεις υστερούν σε "ομορφιά" σε σχέση με τις παραδοσιακές μαθηματικές αποδείξεις των προηγούμενων αιώνων, αλλά αυτό είναι κάτι εντελώς υποκειμενικό..

Σήμερα, πολλά από αυτά τα ερωτήματα έχουν απαντηθεί. Πώς μπορεί κάποια να είναι απόλυτα σίγουρη πως η υλοποίηση μιας απόδειξης ή ενός περίπλοκου αλγόριθμου είναι ορθή; Εκτός από τις συμβατικές δοκιμές λογισμικού, μία απάντηση σε αυτήν την ερώτηση θα μπορούσε να είναι η επιλογή της κατάλληλης μεθόδου τυπικής επαλήθευσης. Ήδη το 2005 οι Werner και Gonthier [Gont07] είχαν δημοσιεύσει μία τυπική απόδειξη του θεωρήματος 4 χρωμάτων στην γλώσσα προγραμματισμού Coq (interactive proof assistant). Αυτό εξάλειψε την ανάγκη εμπιστοσύνης στα διάφορα προγράμματα υπολογιστών που χρησιμοποιούνται για την επαλήθευση όλων των περιπτώσεων. Αρκεί να εμπιστευόμαστε τον πυρήνα της Coq. Έκτοτε, έχουν γίνει σπουδαία βήματα στον το-

μέα της τυπικής επαλήθευσης και του ευρύτερου πεδίου των τυπικών μεθόδων (formal methods) που αφορούν ποικίλα επιστημονικά πεδία. Έχει επαληθευτεί η ορθότητα των αποδείξεων πολλών μαθηματικών θεωρημάτων αλλά και όχι μόνο. Αρκετά λειτουργικά συστήματα έχουν επαληθευτεί, ο σχεδιασμός μεγάλων δικτύων υπολογιστών έχει επαληθευτεί μέσω ενός μαθηματικού μοντέλου του δικτύου και ακόμη έχει αναπτυχθεί ένα πρωτόκολλο τυπικής επαλήθευσης για το blockchain. Εκτός από αυτά, η διαδικασία της τυπικής επαλήθευσης είναι κομμάτι πλέον της κατασκευής ιατρικού εξοπλισμού αλλά και των αεροσκαφών.

Έπειτα, όσον αφορά το ερώτημα που σχετίζεται με το διαχρονικό δίπολο της μαθηματικής κοινότητας, δηλαδή από τη μία η ύπαρξη μιας απόλυτης, αμετάβλητης έννοιας των μαθηματικών αντικειμένων που διαχωρίζεται από την ανθρώπινη εμπειρία και από την άλλη οι μαθηματικές ιδέες που προέρχονται από την ανθρώπινη εμπειρία και παρατήρηση, ο Paul Halmos [Halm85] το θέτει αρκετά απλά, κατανοητά και επεξηγηματικά.

> Mathematics is not a deductive science—that's a cliché. When you try to prove a theorem, you don't just list the hypotheses, and then start to reason. What you do is trial and error, experimentation, guesswork. You want to find out what the facts are, and what you do is in that respect similar to what a laboratory technician does.

Φυσικά, γνωρίζουμε πως στη σύγχρονη επιστήμη, υπάρχουν πολλά πεδία των μαθηματικών που έχουν αποκοπεί από την ανθρώπινη εμπειρία. Ωστόσο, ίσως ένας υπολογιστής που χρησιμοποιείται ως εργαλείο για μαθηματικό πειραματισμό, αλλά και για την παραγωγή μερικών αποτελεσμάτων και αποδείξεων, θα μπορούσε να προσφέρει ακόμη και στο πιο αφηρημένο μαθηματικό πεδίο ένα μέσο σύνδεσης με τον κόσμο της εμπειρίας.

1.1 Αντικείμενο της εργασίας

Ένα κοινό πρόβλημα που συναντάμε στην διαδικασία ανάπτυξης ενός λογισμικού ή δυστυχώς και στα λογισμικά που κυκλοφορούν ήδη, πέραν των σφαλμάτων ορθότητας, είναι η ύπαρξη σφαλμάτων απόδοσης. Δηλαδή, η ύπαρξη αναποτελεσματικών ακολουθιών κώδικα που μπορούν να προκαλέσουν σημαντική υποβάθμιση των επιδόσεων και σπατάλη πόρων. Τα σφάλματα απόδοσης υπάρχουν ευρέως στα συστήματα λογισμικού. Για παράδειγμα, οι προγραμματιστές του Mozilla έχουν διορθώσει 5-60 σφάλματα απόδοσης που αναφέρουν οι χρήστες κάθε μήνα, τα τελευταία 10 χρόνια. Ακόμη, οι ερευνητές Jin, Song, Shi, Scherpelz και Lu του Πανεπιστημίου του Ουισκόνσιν εντόπισαν 332 προηγούμενα άγνωστα προβλήματα απόδοσης στις τελευταίες εκδόσεις των εφαρμογών MySQL, Apache και Mozilla. Παρόλο που, τα τελευταία χρόνια έχει μελετηθεί ένα ευρύ φάσμα τεχνικών στατικής ανάλυσης του κόστους εκτέλεσης ή της χρήσης υπολογιστικών

πόρων των προγραμμάτων και ένα μεγάλο πλήθος αυτών των τεχνικών θεωρούνται σημαίνουσες, δεν επιχειρήθηκε με συστηματικό τρόπο η εφαρμογή μίας ή και περισσότερων από αυτών από την κοινότητα των προγραμματιστών. Αυτό γιατί, η τυπική επαλήθευση είτε της ορθότητας είτε της απόδοσης είτε και των δύο, στην τωρινή της μορφή, είναι μια δύσκολή συνήθως διαδικασία κυρίως διότι στοχεύει γλώσσες ειδικού σκοπού όπως η Coq και η Agda.

Η επιθυμία για πειραματισμό με νέα εργαλεία τυπικής επαλήθευσης, ελεύθερου λογισμικού, που δημιουργήθηκαν για να είναι πιο φιλικά προς την προγραμματίστρια, ώστε να διευκολύνουν αντί να εμποδίζουν την ανάπτυξη λογισμικού και γράφονται για αυτό το λόγο σε γλώσσες γενικού σκοπού (Haskell,C) αλλά και η αναγκαιότητα συνεισφοράς σε τέτοιες βιβλιοθήκες, αποτέλεσαν το βασικότερο κίνητρο για την εκπόνηση της παρούσας διπλωματικής. Είτε για να καλύψουμε μία ανάγκη, είτε για να δημιουργήσουμε μία επιθυμία, υλοποιήσαμε τα απλά δέντρα δυαδικής αναζήτησης, τα δέντρα red-black και μία ειδική μορφή των τελευταίων (left leaning red-black) και κατασκευάσαμε αποδείξεις ορθότητας και αποδείξεις κόστους για αυτά σε Liquid Haskell. Χρησιμοποιήσαμε τις εκφραστικές δυνατότητες που προσφέρει η Liquid Haskell για να ορίσουμε νέους, πιο ισχυρούς τύπους (refinement types) που κωδικοποιούν τις ιδιότητες ορθότητας και πολυπλοκότητας που θέλουμε να αποδείξουμε. Όπου δεν ήταν δυνατό να ελεγχθεί αυτόματα η ορθότητα αυτών των προδιαγραφών μέσω του Satisfiability Modulo Theories (SMT) solver - σε λίγες περιπτώσεις σε σχέση με το συνολικό αποτέλεσμα - εκφράσαμε θεωρήματα και λήμματα μέσω refinement types και έπειτα γράψαμε αποδείξεις για αυτά σε Haskell, χρησιμοποιώντας συνδυαστές απόδειξης (proof combinators). Πιο συγκεκριμένα, η ανάλυση κόστους έγινε μετρώντας αφηρημένα υπολογιστικά «βήματα» μέσα σε αμιγείς Haskell συναρτήσεις, μέσω της Μονάδας Tick (Tick Monad), εκφράζοντας το ανώτατο ή/και κατώτατο όριο αυτής της μέτρησης κάθε φορά μέσω εκλεπτυσμένων τύπων και αποδεικνύοντας αυτά τα όρια - όταν δεν είναι λογαριθμικά - μέσω του SMT solver. Το σημαντικό είναι πως ο τύπος δεδομένων Tick στη Liquid Haskell μας επέτρεπε να ενσωματώνουμε τις ιδιότητες ορθότητας στην ανάλυση κόστους, πράγμα που καθιστά αμέσως την ίδια την ανάλυση κόστους έγκυρη.

1.2 Διάρθρωση της εργασίας

Το υπόλοιπο της εργασίας διαμορφώνεται ως εξής:

Κεφάλαιο 2.

Παρουσιάζουμε μία θεωρητική εισαγωγή στις μεθόδους τυπικής επαλήθευσης.

Κεφάλαιο 3.

Παρουσιάζουμε διάφορες τεχνικές ανάλυσης κόστους. Δίνουμε έμφαση σε αυτές που σχετίζονται με κάποιο τρόπο με το μοντέλο ανάλυσης κόστους της Liquid Haskell.

Κεφάλαιο 4.

Παρουσιάζουμε εκτενώς την υποκείμενη θεωρία και την πρακτική χρήση της Liquid Haskell.

Κεφάλαιο 5.

Παρουσιάζουμε μία απλή μελέτη περίπτωσης. Συγκεκριμένα υλοποιούμε τη δομή των δυαδικών δέντρων αναζήτησης σε Liquid Haskell και αποδεικνύουμε το κόστος των συναρτήσεων εισαγωγής, αναζήτησης και διαγραφής.

Κεφάλαιο 6.

Παρουσιάζουμε μία λιγάκι πιο σύνθετη μελέτη περίπτωσης. Εδώ, υλοποιούμε τη δομή των red-black δέντρων και μιας ειδικής μορφής τους, των left-leaning red-black δέντρων σε Liquid Haskell. Αποδεικνύουμε το κόστος των συναρτήσεων εισαγωγής, αναζήτησης και διαγραφής για την πρώτη δομή και το κόστος των συναρτήσεων εισαγωγής και αναζήτησης για την τελευταία.

Κεφάλαιο 7. Εισάγουμε κάποιες ιδέες για μελλοντική έρευνα και επιπλέον συνοψίζουμε και ολοκληρώνουμε αυτήν την διπλωματική εργασία.

Κεφάλαιο 2

Θεωρητική Εισαγωγή στην Τυπική Επαλήθευση

Σε αυτήν την ενότητα παρουσιάζουμε μία επισκόπηση των βασικότερων μοντέλων επαλήθευσης και εργαλείων. Η ενότητα αυτή δεν προσδοκά να παρουσιάσει τα εξεταζόμενα θέματα στην πληρότητά τους, παρά μόνο να επεξηγήσει βασικές έννοιες με όσο γίνεται πιο εύκολο τρόπο. Για πιο βαθιά μελέτη ο ενδιαφερόμενος αναγνώστης μπορεί να κοιτάξει ένα από τα σχετικά βιβλία, όπως το [Alme11], στο οποίο βασίζεται σε μεγάλο κομμάτι του η παρούσα ενότητα.

2.1 Ένας Ορισμός της Τυπικής Επαλήθευσης Προγράμματος

Η τυπική επαλήθευση προγράμματος είναι ο τομέας της επιστήμης των υπολογιστών που μελετά μαθηματικές μεθόδους για τον έλεγχο της συμμόρφωσης ενός προγράμματος με τις προδιαγραφές του. Είναι μέρος του ευρύτερου πεδίου των τυπικών μεθόδων, οι οποίες ομαδοποιούν αυστηρές, ετερογενείς μεταξύ τους προσεγγίσεις, για την ανάπτυξη συστημάτων και λογισμικού. Η επαλήθευση προγράμματος αφορά τις ιδιότητες του κώδικα, οι οποίες μπορούν να μελετηθούν με περισσότερους από έναν τρόπους. Από μεθόδους με βάση τη λογική, όπως ο συνδυασμός λογικής προγράμματος και θεωριών πρώτης τάξης, σε άλλες προσεγγίσεις όπως ο έλεγχος μοντέλου λογισμικού, η αφηρημένη ερμηνεία και η συμβολική εκτέλεση.

Ο απλός ορισμός που δίνει η Catherine Meadows [Mead11] είναι ο εξής:

> Η επαλήθευση προγράμματος είναι η διαδικασία της εφαρμογής τυπικών μεθόδων απευθείας στην επαλήθευση του κώδικα, και όχι σε προδιαγραφές υψηλού επιπέδου.

2.2 Τυπικές Μέθοδοι

Οι τυπικές μέθοδοι αποτελούν μια εναλλακτική, μαθηματική προσέγγιση για την εξασφάλιση αξιοπιστίας, στην οποία η επιστημονική κοινότητα αποδίδει σταδιακά περισσότερη προσοχή. Ένα σύνολο φορμαλισμών και μαθηματικών εργαλείων για μοντελοποίηση και συλλογιστική συστημάτων (συγκεκριμένα, συστήματα λογισμικού και συστήματα υλικού) συγκροτεί τις τυπικές μεθόδους. Οι τυπικές μέθοδοι προσφέρουν τη δυνατότητα παραγωγής λογικών συλλογισμών για τον πλήρη χώρο δεδομένων και το σύνολο των διαμορφώσεων ενός συστήματος, αντί για σύνολα δειγματοληψίας.

Ο John Rushby [Rush97] ορίζει τις τυπικές μεθόδους ως εξής:

> Ο όρος τυπικές μέθοδοι αναφέρεται στη χρήση μαθηματικών μοντέλων, υπολογισμού και πρόβλεψης για τις προδιαγραφές, το σχεδιασμό, την ανάλυση και τη διασφάλιση συστημάτων υπολογιστών και λογισμικού. Ο λόγος που χρησιμοποιείται ο όρος τυπικές μέθοδοι και όχι ο όρος μαθηματική μοντελοποίηση λογισμικού είναι ακριβώς για να τονίσει τον χαρακτήρα των μαθηματικών θεωριών που εμπλέκονται.

Η κύρια απαίτηση για τις τυπικές μεθόδους είναι, να μπορούν να εγγυηθούν τη συμπεριφορά ενός δεδομένου υπολογιστικού συστήματος ακολουθώντας κάποια αυστηρή προσέγγιση. Στην καρδιά των τυπικών μεθόδων βρίσκει κανείς την έννοια της προδιαγραφής. Μια προδιαγραφή είναι ένα μοντέλο ενός συστήματος που περιέχει μια περιγραφή της επιθυμητής συμπεριφοράς του - τι πρέπει να υλοποιηθεί, σε αντίθεση με το πώς. Αυτή η προδιαγραφή μπορεί να είναι εντελώς αφηρημένη (στην περίπτωση αυτή το μοντέλο είναι η περιγραφή της συμπεριφοράς) ή μπορεί να είναι πιο λειτουργική, οπότε η περιγραφή περιέχεται κάπως ή υπονοείται από το μοντέλο.

Ίσως θα ήταν πιο εύκολο ή/και σύντομο να παρουσιάσουμε μόνον τις τυπικές μεθόδους που εμπλέκονται ρητά με το σύστημα της Liquid Haskell το οποίο χρησιμοποιούμε σε αυτήν την εργασία. Μελετώντας την υπάρχουσα βιβλιογραφία για τα εργαλεία τυπικών μεθόδων, συνειδητοποιήσαμε πως υπάρχει μία ιστορική συνέχεια σε σχέση με την ορολογία, τις τεχνικές και τις έννοιες. Αυτήν ακριβώς την ιστορική συνέχεια θέλουμε να αναδείξουμε σε ένα βαθμό, ώστε να κατανοήσουμε με περισσότερη σιγουριά τα εργαλεία που θα χρησιμοποιήσουμε και τις έννοιες που θα συναντήσουμε στην πορεία αυτής της διπλωματικής.

2.3 Γλώσσες Προδιαγραφών

Όπως προαναφέραμε, στην καρδιά των τυπικών μεθόδων βρίσκει κανείς την έννοια της προδιαγραφής. Μια προδιαγραφή περιγράφει στην ουσία τα δεδομένα που χρησιμοποιούμε

και πώς αυτά αλλάζουν, δηλαδή τις λειτουργίες που τα μετασχηματίζουν.

2.3.1 Προδιαγραφή Βάσει Μοντέλου

Οι γλώσσες που χρησιμοποιούνται για αυτήν την κατηγορία προδιαγραφών χαρακτηρίζονται από την ικανότητα περιγραφής της έννοιας της εσωτερικής *κατάστασης* του συστήματος και από την εστίασή τους στην περιγραφή του τρόπου με τον οποίο οι λειτουργίες του συστήματος τροποποιούν αυτήν την κατάσταση, πώς δηλαδή το σύστημα *μεταβαίνει* από μία κατάσταση σε μία άλλη. Τα διακριτά μαθηματικά, η θεωρία συνόλων, η θεωρία κατηγοριών και η λογική αποτελούν τα βασικά θεμέλια αυτών των γλωσσών.

Θα επιμείνουμε λίγο στην Θεωρία Κατηγοριών. Εδώ οι καταστάσεις περιγράφονται με όρους μαθηματικών δομών όπως σχέσεις ή συναρτήσεις. Οι μεταβάσεις εκφράζονται ως *αναλλοίωτες*, καθώς και *προϋποθέσεις* και *μετα-προϋποθέσεις*. Η θεωρία κατηγοριών έχει πρακτικές εφαρμογές στη θεωρία γλωσσών προγραμματισμού, για παράδειγμα με τη χρήση *μονοειδών*, *μονάδων*, *συναρτητών*, *φυσικών μετασχηματιστών* στον συναρτησιακό προγραμματισμό (Haskell, Charity). Ο Wadler [Wadl92] παρατήρησε πως οι μονάδες μπορούν να χρησιμοποιηθούν για τη δημιουργία συναρτησιακών προγραμμάτων, με τον ίδιο ακριβώς τρόπο που χρησιμοποιήθηκαν από τον Moggi [Mogg91] για την κατασκευή της δηλωτικής σημασιολογίας (denotational semantics) των γλωσσών προγραμματισμού.

2.3.2 Αλγεβρικές Προδιαγραφές

Μια δεύτερη κλασική προσέγγιση στις προδιαγραφές βασίζεται στη χρήση αλγεβρών πολλαπλών ταξινομήσεων. Μια *άλγεβρα πολλαπλών ταξινομήσεων* ή αλλιώς ετερογενής άλγεβρα, αποτελείται από μια συλλογή δεδομένων ομαδοποιημένων σε σύνολα (ένα σύνολο για κάθε τύπο δεδομένων), μια συλλογή συναρτήσεων πάνω σε αυτά τα σύνολα, που αντιστοιχούν στις λειτουργίες του προγράμματος που μοντελοποιείται και ένα σύνολο αξιωμάτων που καθορίζουν τις βασικές ιδιότητες των συναρτήσεων στην άλγεβρα. Ένας τέτοιος φορμαλισμός επιτρέπει σε κάποια να επικεντρωθεί στην αναπαράσταση των δεδομένων και στη συμπεριφορά εισόδου-εξόδου των συναρτήσεων, αντί για τον εκάστοτε αλγόριθμο που κωδικοποιεί τις ιδιότητες. Ένα παράδειγμα χρήσης αυτής της δομής στην επιστήμη των υπολογιστών είναι η μοντελοποίηση των αφηρημένων τύπων δεδομένων ως αλγεβρικές δομές πολλαπλών ταξινομήσεων. Εκτός από την άλγεβρα πολλαπλών ταξινομήσεων, η μαθηματική επαγωγή και η εξισωτική λογική βρίσκονται στα θεμέλια των αλγεβρικών προδιαγραφών.

2.3.3 Δηλωτική μοντελοποίηση

Από την μία η προσέγγιση των λογικών γλωσσών προγραμματισμού βασίζεται στην έννοια του *κατηγορήματος (predicate)*. Εδώ, τα δεδομένα αναπαρίστανται μέσω εκφραστικών τύπων δεδομένων και οι λειτουργίες εκφράζονται μέσω των συμπεριφορικών ιδιοτήτων τους, όπως τα αξιώματα στις αλγεβρικές προδιαγραφές.

Από την άλλη, η προσέγγιση των συναρτησιακών γλωσσών προγραμματισμού βασίζεται στην έννοια της συνάρτησης. Εδώ, ο πυρήνας τους βασίζεται στον *λ-λογισμό* (lambda calculi) και συγκεκριμένα στον λ-λογισμό με τύπους (typed lambda calculi). Ο λ-λογισμός, όντας ένα σύστημα με πολύ απλή σύνταξη και σημασιολογία, αλλά ταυτόχρονα με μεγάλη εκφραστικότητα, μπορεί να λειτουργήσει ως ενδιάμεση μορφή, στην οποία να μετασχηματίζεται ένα οποιοδήποτε συναρτησιακό πρόγραμμα. Για την υλοποίηση των συναρτησιακών γλωσσών αρκεί λοιπόν η υλοποίηση ενός πολύ περιορισμένου συνόλου από κανόνες μετασχηματισμού, και ειδικότερα από κανόνες αναγωγής, για τον υπολογισμό εκφράσεων.

Οι εκφράσεις στον λ-λογισμό ονομάζονται *λ-εκφράσεις*, μία ειδική κατηγορία των οποίων είναι οι *λ-αφαιρέσεις*. Μια λ-αφαίρεση είναι ο ορισμός μιας ανώνυμης συνάρτησης. Οι ανώνυμες συναρτήσεις - τις οποίες θα χρησιμοποιήσουμε κι εμείς στη συνέχεια - αντιμετωπίζονται ως τιμές πρώτης κατηγορίας (*functions as first-class values*), δηλαδή με τον ίδιο ακριβώς τρόπο που αντιμετωπίζεται οποιοδήποτε άλλη βαθμωτή τιμή.

Γλώσσες όπως η Haskell και proof assistants όπως η Agda, PVS, Coq που θα μας απασχολήσουν στη συνέχεια βασίζονται σε παραλλαγές του λ-λογισμού με τύπους ή επεκτάσεις του.

2.4 Τεχνικές Απόδειξης

Παραπάνω, παρουσιάσαμε μία συνοπτική ανάλυση των διαφορετικών προσεγγίσεων συγγραφής προδιαγραφών. Εδώ, θα κάνουμε μία σύνοψη των τρόπων με τους οποίους μπορούμε να αποδείξουμε ιδιότητες για αυτές τις προδιαγραφές. Θα συνεχίσουμε με τους τρόπους με τους οποίους εξασφαλίζουμε ότι η υλοποίηση του κώδικά μας είναι ορθή σε σχέση με αυτές τις προδιαγραφές. Έτσι θα έχουμε φτάσει να έχουμε μία συνολική εικόνα για το πεδίο της τυπικής επαλήθευσης.

2.4.1 Μηχανές Απόδειξης

Από τη μία πλευρά μηχανές απόδειξης με ισχυρή εκφραστικότητα επιτρέπουν σε ένα βαθμό την έκφραση σημασιολογικών προδιαγραφών των προγραμμάτων. Η ορθότητα

αυτών των προδιαγραφών όμως συνήθως δεν αποδεικνύεται αυτόματα, αντίθετα η απόδειξη τους επαφίεται στον χρήστη. Από την άλλη πλευρά μηχανές απόδειξης με λογικούς φορμαλισμούς και λιγότερο ισχυρή εκφραστικότητα επιτρέπουν την αυτόματη απόδειξη προδιαγραφών, κατά το χρόνο μεταγλώττισης. Στη σύγχρονη βιβλιογραφία υπάρχουν πολλές επιτυχημένες απόπειρες ενοποίησης αυτών των δύο σε ένα σύστημα, ώστε να μην θυσιάζεται η εκφραστικότητα για την αυτοματοποίηση ή το αντίστροφο. Εδώ θα κάνουμε αρχικά μία παρουσίαση αυτών των διαφορετικών προσεγγίσεων.

2.4.1.1 Αυτοματοποιημένη Απόδειξη Θεωρημάτων

Εδώ η κατασκευή της απόδειξης είναι αυτόματη, εφόσον η μηχανή απόδειξης έχει παραμετροποιηθεί επαρκώς. Ένα τέτοιο σύστημα πρέπει να βασίζεται στην αποφασιστότητα τουλάχιστον ενός κομματιού της υποκείμενης θεωρίας (λογική πρώτης τάξης, horn cluases κλπ).

Οι Satisfiability Modulo Theory (SMT) solvers ανήκουν σε αυτήν την κατηγορία. Ο SMT Solver αποφασίζει εάν ένα κατηγόρημα είναι έγκυρο χωρίς να απαριθμεί και να αξιολογεί όλες τις πιθανές αναθέσεις. Πράγματι, αυτό θα ήταν αδύνατο υπολογιστικά, καθώς υπάρχουν συνήθως απείρως πολλές αναθέσεις όταν τα κατηγορήματα αναφέρονται σε ακέραιους αριθμούς ή λίστες και ούτω καθεξής . Αντ' αυτού, ο SMT solver χρησιμοποιεί ένα σύνολο συμβολικών αλγορίθμων για να συμπεράνει αν ένα κατηγόρημα είναι έγκυρο ή όχι. Αυτή η διαδικασία είναι το αποτέλεσμα δεκαετιών εργασίας στη μαθηματική λογική και στις διαδικασίες λήψης αποφάσεων. Κάποια εκφραστικά συστήματα έχουν ακόμη, χρησιμοποιήσει SMT solvers για την επαλήθευση προγραμμάτων χωρίς τον περιορισμό για την αποφασιστότητα. Σε αυτά τα συστήματα οι SMT solvers χρησιμοποιούνται για να αποδείξουν την εγκυρότητα ορισμένων αφηρημένων - όχι απαραίτητα αποφασιστών - λογικών, δημιουργώντας έτσι εκφραστικές προδιαγραφές αλλά και την πιθανότητα για μη αποφασιστή και απρόβλεπτη επαλήθευση. Η γλώσσες F*, Danfy και Sage αποτελούν τέτοια παραδείγματα. Οι τρόποι τους οποίους η καθεμία χρησιμοποιεί για τον έλεγχο της απρόβλεπτης επαλήθευσης έχουν αποτελέσει πηγή έμπνευσης για την Liquid Haskell.

2.4.1.2 Βοηθοί Απόδειξης

Τα εργαλεία απόδειξης που ανήκουν σε αυτήν την κατηγορία βασίζονται σε ισχυρά εκφραστικές λογικές, όπως οι λογική ανώτερης τάξης. Συνήθως συνδυάζουν 2 μέρη. Έναν *ελεγκτή απόδειξης*, ο οποίος αποδεικνύει αν οι θεωρίες, που έχουν οριστεί στην διαδικασία μοντελοποίησης, είναι καλώς ορισμένες ή όχι. Επίσης, επειδή πρόκειται για συστήματα των οποίων οι λειτουργία εξαρτάται από τη συνεργασία ανθρώπου - μηχανής, το 2ο μέρος αποτελείται από ένα *διαδραστικό σύστημα ανάπτυξης αποδείξεων* το οποίο

βοηθά τις προγραμματίστριες κατά τη διαδικασία κατασκευής αποδείξεων. Όταν ολοκληρωθεί η κατασκευή μιας απόδειξης, μπορεί να αποθηκευτεί ένα σενάριο απόδειξης, το οποίο περιγράφει αυτήν την κατασκευή.

Οι εξαρτώμενοι τύποι αποτελούν τη βάση πολλών βοηθών απόδειξης. Η επαλήθευση κώδικα γραμμένου σε Haskell είναι δυνατή με πλήρως εξαρτημένα συστήματα τύπων όπως αυτό της Coq, της Agda, της Idris, της Omega κλπ. Ενώ αυτά τα συστήματα είναι εξαιρετικά εκφραστικά, αυτή η εκφραστικότητά τους, τους κοστίζει σε αυτοματοποίηση, καθιστώντας τον έλεγχο λογικής εγκυρότητας μη αποφασίσιμο, κι έτσι η επαλήθευση γίνεται αρκετά πολύπλοκή για την προγρμματίστρια. Η ίδια το Haskell μπορεί να θεωρηθεί γλώσσα εξαρτημένων τύπων, καθώς οι υπολογισμοί στο επίπεδο τύπων επιτρέπεται μέσω των Type Families , των Singleton Types, των Generalized Algebraic Datatypes (GADTs) και μέσω type-level συναρτήσεων. Ωστόσο, η επαλήθευση στην ίδια τη Haskell μπορεί να αποδειχθεί πολύ επώδυνη.

2.4.2 Έλεγχος Μοντέλων

Ο έλεγχος μοντέλου ή αλλιώς έλεγχος ιδιοτήτων είναι μια μέθοδος για την επαλήθευση συστημάτων πεπερασμένων καταστάσεων (finite-state systems), δηλαδή αν ένα μοντέλο συστήματος πληρεί μια δεδομένη προδιαγραφή. Η ιδέα είναι η εξής: Οι αναμενόμενες ιδιότητες του μοντέλου εκφράζονται με τύπους *χρονικής λογικής* (temporal logic) και χρησιμοποιούνται αποτελεσματικοί *συμβολικοί αλγόριθμοι* για να διασχίσουν το μοντέλο στο σύνολό του, έτσι ώστε να επαληθευτεί εάν όλοι οι πιθανοί σχηματισμοί επικυρώνουν αυτές τις ιδιότητες. Το σύνολο όλων των καταστάσεων ονομάζεται *χώρος καταστάσεων* του μοντέλου. Όταν ένα σύστημα διαθέτει χώρο πεπερασμένων καταστάσεων, οι αλγόριθμοι ελέγχου μοντέλων μπορούν θεωρητικά να χρησιμοποιηθούν για την πραγματοποίηση της αυτόματης απόδειξης ιδιοτήτων. Εάν μια ιδιότητα δεν είναι έγκυρη, παρουσιάζεται ένα αντιπαράδειγμα για αυτήν. Το βασικό πρόβλημα αυτής της προσέγγισης είναι το μέγεθος του χώρου καταστάσεων, το οποίο μεγαλώνει εκθετικά σε σχέση με το μέγεθος του συστήματος. Οπότε και η εξερεύνηση αυτού είναι κάθε άλλο παρά πραγματοποιήσιμη. Για να την αντιμετώπιση αυτού του προβλήματος (state explosion problem) έχουν προταθεί διάφορες τεχνικές. Μία από αυτές είναι η πρόταση για μία απλοποιημένη μορφή του μοντέλου, ένα *αφηρημένο μοντέλο* στο οποίο η αναζήτηση στο χώρο καταστάσεων μπορεί να πραγματοποιηθεί. Η *αφαίρεση*, στα πλαίσια του ελέγχου μοντέλου, στοχεύει στη μείωση του χώρου καταστάσεων του συστήματος παραλείποντας λεπτομέρειες που δεν σχετίζονται με την ιδιότητα που επαληθεύεται. Ταυτόχρονα, το αφηρημένο μοντέλο ικανοποιεί κάποιες συγκεκριμένες ιδιότητες, έτσι ώστε μία έγκυρη ιδιότητα στο αφηρημένο μοντέλο να είναι επίσης έγκυρη στο αρχικό μοντέλο.

Αυτή η προσέγγιση, της αφαίρεσης, είναι επίσης χρήσιμη και εκτός του ελέγχου μο-

ντέλου. Συχνά υπάρχει η ανάγκη να μετασχηματίσουμε κάποιες προδιαγραφές ώστε να κρύψουμε κάποιες λεπτομέρειες ή ακόμη και να εμπλουτίσουμε κάποιες από αυτές. Η θεωρία πίσω από αυτούς τους μετασχηματισμούς είναι η θεωρία της *αφηρημένης ερμηνείας* (abstract interpretation). Η αφηρημένη ερμηνεία μπορεί να θεωρηθεί ως μερική εκτέλεση ενός προγράμματος μέσω της οποίας αποκτούμε πληροφορίες σχετικά με τη σημασιολογία του χωρίς να εκτελούμε όλους τους υπολογισμούς. Άλλωστε, η μελέτη μιας μεμονωμένης πτυχής του μοντέλου είναι ευκολότερη από τη μελέτη της γενικής συμπεριφοράς, και υπό ορισμένες προϋποθέσεις μπορεί να είναι ισοδύναμη από την άποψη των αποτελεσμάτων που καταγράφουμε.

2.4.3 Σχολιασμός Προγράμματος

Κάποια εργαλεία απόδειξης μας δίνουν την δυνατότητα να προσθέσουμε σε ένα κομμάτι κώδικα του προγράμματός μας σχολιασμούς, οι οποίοι εκφράζει τις συνθήκες που πρέπει να ικανοποιούνται πριν την εκτέλεσή του, αλλά και ταυτόχρονα να περιγράφει τη λογική κατάσταση του προγράμματος αφού αυτό εκτελεστεί. Αυτό μας δίνει το εξής πλεονέκτημα: Ο ίδιος ο πηγαίος κώδικας αποτελεί την βάση της επαλήθευσης και όχι κάποιο ξεχωριστό πρόγραμμα το οποίο περιλαμβάνει όλες τις απαιτούμενες προδιαγραφές. Στην πραγματικότητα, εδώ το μοντέλο κατασκευάζεται με εισόδους ένα πρόγραμμα και τους σχολιασμούς του, μαζί με το υποκείμενο μοντέλο της γλώσσας προγραμματισμού. Αυτή η προσέγγιση θεωρείται όλο και περισσότερο ότι παρέχει μια ικανοποιητική εναλλακτική λύση στο κεντρικό πρόβλημα των τυπικών μεθόδων, δηλαδή να μπορούν αυτές να εγγυηθούν τη συμπεριφορά ενός δεδομένου υπολογιστικού συστήματος ακολουθώντας κάποια αυστηρή προσέγγιση. *Σχολιασμό προγράμματος* ονομάζουμε την τεχνική απόδειξης που βασίζεται σε αυτούς τους σχολιασμούς, που υπάρχουν εντός του κώδικα προς επαλήθευση. Οι υποκείμενη θεωρία αυτής της προσέγγισης είναι κάποια λογική προγράμματος όπως η λογική Hoare (ή αλλιώς λογική Floyd-Hoare). Αυτή η προσέγγιση χρησιμοποιείται για την επαλήθευση χρονικής πολυπλοκότητας σε αμιγείς συναρτησιακές δομές δεδομένων στην γλώσσα Agda. Συγκεκριμένα, ο N. A. Danielsson [Dani08] χρησιμοποιεί το σύστημα τύπων της Agda (εξαρτημένοι τύποι) για να προσθέσει σε κάθε συνάρτηση σχολιασμούς - συγκεκριμένα προσθέτει "νι" τα οποία αναπαριστούν αφηρημένα υπολογιστικά βήματα - με τον χρόνο που απαιτείται για να υπολογίσει την τιμή της. Ορίζει το Thunk Monad, το οποίο περιέχει αυτούς τους σχολιασμούς και οι γνωστοί συνδυαστές μονάδας (bind operator κλπ) χρησιμοποιούνται για να συνδυάσουν τις χρονικές πολυπλοκότητες των υποεκφράσεων. Αντίστοιχα οι [Hand19] εμπνεόμενες από το παραπάνω, χρησιμοποιούν σχολιασμούς για την επαλήθευση πολυπλοκότητας σε LiquidHaskell. Ορίζουν το Tick Monad, το οποίο περιέχει αυτούς τους σχολιασμούς και οι γνωστοί συνδυαστές μονάδας μαζί με καινούργιους χρησιμοποιούνται για να συν-

δυάσουν τις πολυπλοκότητες των υποεκφράσεων.

Η εκλογίκευση που προσφέρει η μέθοδος του σχολιασμού προγράμματος από τη μία και η συμπόρευσή της με τις αρχές της μηχανικής λογισμικού από την άλλη, οδήγησε στη συστημική χρήση ενός προτύπου, που βασίζεται σε έναν ειδικής μορφής σχολιασμό, το *συμβόλαιο*. Τα συμβόλαια είναι προυποθέσεις και μέτα-συνθήκες οι οποίες ορίζουν τυπικές, αυστηρές και επαληθεύσιμες προδιαγραφές για κομμάτια κώδικα οι οποίες ελέγχονται δυναμικά. Τα συμβόλαια είναι της μορφής $\langle\{v : \tau \mid p\}\rangle^l$. Η μορφή εσωτερικά των αγκυλών, $\{v : \tau \mid p\}$, περιγράφει τις τιμές v με τύπο τ που ικανοποιούν το λογικό κατηγόρημα p. Ο εκθέτης l είναι μία *επισημείωση υπαιτιότητας* που χρησιμοποιείται για να εντοπίζει την πηγή των σφαλμάτων. Για παράδειγμα ένα συμβόλαιο για τους θετικούς ακέραιους είναι το εξής $\langle\{v : Int \mid v > 0\}\rangle^l$. Έστω πως εφαρμόζουμε το συμβόλαιο σε 2 τιμές 2 και 0.

$$\langle\{v : Int \mid v > 0\}\rangle^{l'} 2 \to 2$$
$$\langle\{v : Int \mid v > 0\}\rangle^{l} 0 \to \Uparrow l$$

Στην περίπτωση της εφαρμογής της τιμής 2 στο συμβόλαιο, ο έλεγχος είναι επιτυχής και επιστρέφεται αυτή η τιμή. Στην περίπτωση της εφαρμογής της τιμής 0 στο συμβόλαιο, ο έλεγχος δεν είναι επιτυχής και το πρόγραμμα δημιουργεί μία εξαίρεση που ονομάζεται "υπαιτιότητα l" .

Κεφάλαιο 3

Στατική Ανάλυση Κόστους

Η ενότητα αυτή σκοπό έχει να κάνει μία σύντομη επισκόπηση στις πιο σημαντικές για εμάς προσεγγίσεις για την επαλήθευση ιδιοτήτων πολυπλοκότητας και την ανάλυση κόστους των προγραμμάτων που είναι γραμμένα κυρίως σε συναρτησιακές γλώσσες. Συγκεκριμένα μελετάμε τις μεθόδους που υπολογίζουν (αυτόματα) το κόστος εκτέλεσης χρησιμοποιώντας εκφραστικούς τύπους. Έμφαση δίνεται σε αυτές που θα χρειαστούμε στη συνέχεια για να μελετήσουμε το μοντέλο ανάλυσης κόστους της Liquid Haskell. Ειδάλλως, θα έπρεπε να αφιερώσουμε το μεγαλύτερο μέρος αυτής της διπλωματικής για την συγγραφή αυτής της ενότητας. Υπάρχουν πραγματικά πολλές τεχνικές στατικής ανάλυσης κόστους ή/και πολυπλοκότητας των προγραμμάτων. Σημασιολογική ερμηνεία [Bonf11, Bonf11], συστήματα τύπων και συνεπειών [Hold10, Reis94], λογικές προγράμματος όπως η λογική διαχωρισμού [Atke11] και η λογική τύπου VDM [Aspi07], τεχνικές βασισμένες σε τύπους, όπως οι (γραμμικά) εξαρτημένοι τύποι [Lago12, Dall13], οι τύποι με μέγεθος [Hugh96, Avan15, Avan17], οι τύποι (μονάδας) με δείκτες [Wang17, McCa17, Dani08], τεχνικές που χρησιμοποιούν μονάδα κόστους με εκλεπτυσμένους τύπους [Radi17, Hand19, Grob01], τεχνικές βασισμένες στην αυτόματη ανάλυση πόρων απόσβεσης [Hoff15, Hoff12a, Hoff12b, Hofm03a, Hofm03b]. Για πιο βαθιά μελέτη ο ενδιαφερόμενος αναγνώστης μπορεί να ανατρέξει στις σχετικές δημοσιεύσεις.

3.1 Εισαγωγή

Η στατική ανάλυση πολυπλοκότητας προγραμμάτων αποτελεί από τον προηγούμενο αιώνα αντικείμενο επιστημονικής έρευνας και από τις αρχές του αιώνα έχει κερδίσει ακόμη περισσότερη προσοχή λόγω σημαντικών ανακαλύψεων στον τομέα αλλά και λόγω της αυξημένης αναγκαιότητας για ασφάλεια και διασφάλιση ποιότητας των συστημάτων. Ακόμη, αυτό το πεδίο έρευνας, έχει τη δυνατότητα να γίνει μια γέφυρα μεταξύ των ερευνητικών κοινοτήτων των γλωσσών προγραμματισμού και των αλγορίθμων, μεταφέροντας πληροφορίες και εργαλεία μεταξύ τους. Η ιδιαιτερότητα της στατικής ανάλυσης του κόστους των προγραμμάτων σε σχέση με την επαλήθευση της λειτουργικής ορθότητας αυτών

εντοπίζεται στο εξής σημείο. Η πρώτη δεν μπορεί να αντικατασταθεί με ευκολία από την μέθοδο δοκιμών, διότι τα σφάλματα κόστους εμφανίζονται συνήθως σε αρκούντως μεγάλες εισόδους γεγονός το οποίο καθιστά την ίδιες τις δοκιμές κόστους πολύ κοστοβόρες. Από την άλλη όμως η στατική ανάλυση κόστους μοιράζεται συχνά τα ίδια προβλήματα με την τυπική επαλήθευση ιδιοτήτων ορθότητας. Δηλαδή, όπως περιγράψαμε παραπάνω στην ανάλυση των τεχνικών τυπικών μεθόδων, είτε θα στοχεύει στην ισχυρή εκφραστικότητα του συστήματος είτε στην πλήρη αυτοματοποίηση της απόδειξης. Η πρώτη προσέγγιση φιλοδοξεί να επαληθεύσει όρια πέρα από τα πολυωνυμικά, χρησιμοποιώντας τεχνικές όπως λογικές προγραμμάτων και εργαλεία όπως βοηθοί απόδειξης, με κόστος όμως την αναγκαιότητα συγγραφής αποδείξεων από την προγραμματίστρια. Η δεύτερη προσέγγιση στοχεύει σε περιορισμένα πεδία αλγορίθμων (π.χ πολυωνιμικοί αλγόριθμοι) αλλά είναι πιο εύκολη στη χρήση, χωρίς όμως να δίνει τη δυνατότητα στο χρήστη για υποδείξεις όταν η αυτοματοποιημένη διαδικασία αποτυγχάνει. Μια κλάση προσεγγίσεων που βασίζεται στην μέση των δύο παραπάνω κερδίζει δημοτικότητα στην επαλήθευση λογισμικού. Πρωτοεμφανίστηκε από τους Xi και Pfenning το 1999 με την δημοσίευση της γλώσσας Dependent ML (DML) και διαδόθηκε το 2008 κι έπειτα με την δημοσίευση των Liquid Types[Rond08] από τους Rondon, Kawaguci και Jhala και την δημοσίευση της γλώσσας Dafny[Lein10] από τους Rustan και Leino, οι οποίοι με την προσέγγιση τους περιορίζουν τη δύναμη των εξαρτημένων τύπων ή/και των λογικών προγραμμάτων και σε αντάλλαγμα κερδίζουν κάποιο βαθμό αυτοματοποίησης. Πιο πρόσφατα παραδείγματα είναι η TiML (Timed ML) [Wang17] και η Liquid Haskell [Vazo14a].

3.2 Εκλεπτυσμένοι Τύποι

Το σύστημα των *εκλεπτυσμένων τύπων* [Free91] αναπτύχθηκε ως ένα σύστημα υποτύπων για την γλώσσα ML. Αργότερα οι εκλεπτυσμένοι τύποι χρησιμοποιήθηκαν στην ανάπτυξη της DML [Xi99]. Το σύστημα διατηρεί τις υπάρχουσες ιδιότητες της ML, όπως η αποφασισιμότητα του συμπερασμού τύπων, ενώ ταυτόχρονα επιτρέπει τον εντοπισμό περισσότερων σφαλμάτων κατά τη μεταγλώττιση. Οι εκλεπτυσμένοι τύποι μπορούν να κατασκευαστούν από τους τύπους που ανήκουν στο σύστημα τύπων της ML, από αναδρομικούς τύπους με τον κατασκευαστή τύπων " $\rightarrow$ " της ML και από τις λειτουργίες ένωση " $\wedge$ " και τομή " $\vee$ ". Διαισθητικά μία έκφραση έχει τύπο $\sigma \vee \tau$ αν ικανοποιεί και τους δύο τύπους, σ και τ. Όμοια, έχει τύπο $\sigma \wedge \tau$ αν ικανοποιεί τουλάχιστον έναν από τους τύπους σ και τ. Ωστόσο εδώ, δεν είναι δυνατόν κάποιες φορές να αποφανθούμε κατά τη μεταγλώττιση ποιον ακριβώς από τους 2 τύπους ικανοποιεί. Η γραμματική των

εκλεπτυσμένων τύπων είναι η εξής:

$$
\begin{aligned}
refty ::= {} & refty \vee refty \mid refty \wedge refty \mid \\
& refty \rightarrow refty \mid \ \bot \ \mid \\
& < \ refty \ > mltyname \mid \\
& < \ refty \ > reftyname \mid \\
& reftyvar :: mltyvar
\end{aligned}
$$

όπου,

$reftyvar$	Μεταβλητές εκλεπτυσμένων τύπων, όπως ra, rβ, κ.ο.κ
$reftyname$	Εκλεπτυσμένοι τύποι δεδομένων, όπως singleton ή pair στην ML

Κάθε μεταβλητή εκλεπτυσμένων τύπων είναι δεσμευμένη από μια μεταβλητή τύπου της ML και έτσι η εμβέλεια της είναι εντός των εκλεπτύνσεων ενός τύπου στην ML.

Πιο πρόσφατα έχουμε την ανάπτυξη του συστήματος των Liquid Types (Logically Qualified Data Types)[Rond08] και των αφηρημένων εκλεπτυσμένων τύπων, τα οποία συστήματα αποτελούν υποσύνολα των εκλεπτυσμένων τύπων και θα τα μελετήσουμε στην επόμενη ενότητα.

3.2.1 Μονάδα Κόστους με Εκλεπτυσμένους Τύπους

Η τεχνική της ανάλυσης κόστους μέσω ενός τύπου δεδομένων εντός του προγράμματος έχει δύο κατευθύνσεις. Στη μία - την οποία θα αναλύσουμε παρακάτω - τα αφηρημένα υπολογιστικά βήματα μετρώνται μέσω ενός δείκτη στο επίπεδο τύπων. Στην άλλη, το μέτρημα των βημάτων γίνεται στο επίπεδο τιμών με έναν μετρητή ακεραίων. Οι τιμές συχνά μπορούν να επηρεάσουν την ακριβή ανάλυση κόστους. Επίσης, η λειτουργική επαλήθευση του προγράμματος είναι συχνά προαπαιτούμενη για την ανάλυση κόστους. Για παράδειγμα, το κόστος εισαγωγής ενός στοιχείου σε ένα red-black δέντρο εξαρτάται τόσο από την τιμή αυτού του στοιχείου, αλλά και από τις τιμές των στοιχείων του δέντρου. Ένα άλλο παράδειγμα είναι η ανάλυση του σχετικού κόστους δύο εκτελέσεων μιας υλοποίησης της insertion sort, η οποία ανάλυση απαιτεί να αποδειχθεί η λειτουργική ορθότητα του αλγορίθμου.

Τα θεωρητικά θεμέλια της μέτρησης του κόστους στο επίπεδο των τιμών μέσω ενός τύπου δεδομένων βρίσκονται στην ερευνητική δουλειά των Radicek, Barthie, Gaboardi, Garg και Zuleger [Radi17].

Για τη συσχετιστική ανάλυση κόστους εκκινούν από μία ισχυρά εκφραστική συσχετιστική λογική ανώτερης τάξης, ικανή για να εκφράζει συλλογισμούς για αμιγή προγράμματα, την RHOL (Relational Higher Order Logic) [Agui19]. Η RHOL μπορεί να εκφράζει συλλογισμούς συσχετιστικών ιδιοτήτων για ανώτερης τάξης προγράμματα, γραμμένα σε μία παραλλαγή της PCF (Programming Computable Functions') με απλούς τύπους. Η λογική αυτή χειρίζεται ισχυρισμούς της μορφής:

$$\Gamma \mid \Psi \ \vdash \ t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi$$

όπου Γ ένα περιβάλλον με απλούς τύπους, σ_1 και σ_2 απλοί τύποι - πιθανώς διαφορετικοί μεταξύ τους - των t_1 και t_2 αντίστοιχα, t_1 και t_2 όροι, Ψ ένα σύνολο υποθέσεων και ϕ μια υπόθεση για τα t_1 και t_2. Η ϕ μπορεί να αποτελεί είτε μέτα-συνθήκη των όρων t_1 και t_2 είτε μία συσχετιστική εκλέπτυνση των τύπων σ_1 και $sigma_2$.

Έπειτα η λογική RHOL επεκτείνεται με τον ορισμό μίας μονάδας. Η μονάδα εμπεριέχει τους σχετικούς με το κόστος υπολογισμούς. Το ακριβές κόστος (συσχετιστικό) υπολογίζεται με τη βοήθεια εκλεπτυσμένων τύπων που προστίθενται στη μονάδα. Οι εκφράσεις μονάδας διαχωρίζονται από τις απλές εκφράσεις και οι ισχυρισμοί μονάδας έχουν την εξής μορφή:

$$\Gamma \mid \Psi \ \vdash \ m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi$$

Η σημασία αυτού του ισχυρισμού είναι η εξής. Αν αποτιμηθούν οι όροι m_1 και m_2 με κόστη n_1 και n_2 αντίστοιχα, τότε ισχύει ότι $n_1 - n_2 \leq n$. Δηλαδή το n είναι ένα άνω όριο για την διαφορά κόστους του m_1 από το m_2.

Για τη μοναδιαία ανάλυση κόστους εκκινούν από την UHOL (Unary Higher Order Logic) [Agui19].Η UHOL διατηρεί την έννοια των εκλεπτυσμένων τύπων, αλλά διαχωρίζει την ανάθεση τύπων από τις υποθέσεις. Η μορφή των ισχυρισμών στην UHOL είναι η εξής:

$$\Gamma \mid \Psi \ \vdash \ t : \tau \mid \phi$$

όπου, μία ξεχωριστή μεταβλητη $\mathbf{r}$ που δεν ανήκει στο περιβάλλον Γ, μπορεί να είναι ελεύθερη μεταβλητή στην υπόθεση ϕ ως ένα συνώνυμο του όρου t με τύπο τ.Ένας ισχυρισμός είναι καλώς ορισμένος εάν, ο όρος t έχει τύπο τ, το Ψ είναι ένα έγκυρο σύνολο υποθέσεων στο περιβάλλον Γ και η ϕ είναι μία έγκυρη υπόθεση στο περιβάλλον Γ, $\mathbf{r} : \tau$

Αντίστοιχα, η λογική UHOL επεκτείνεται με τον ορισμό μίας μονάδας. Η μονάδα εμπεριέχει τους σχετικούς με το κόστος υπολογισμούς. Το ακριβές κόστος (μοναδιαίο) υπολογίζεται με τη βοήθεια εκλεπτυσμένων τύπων που προστίθενται στη μονάδα. Οι ισχυρισμοί μονάδας εδώ έχουν την εξής μορφή:

$$\Gamma \mid \Psi \ \vdash \ m \div \tau \mid k \mid l \mid \phi$$

Ο παραπάνω ισχυρισμός σημαίνει ότι αν αποτιμηθεί ο όρος m τότε το άνω και κάτω όριο του κόστους της αποτίμησης είναι k και l αντίστοιχα. Ο ισχυρισμός ϕ αναπαριστά τις λειτουργικές ιδιότητες της εξόδου, δηλαδή τις μέτα-συνθήκες που πρέπει να ικανοποιούνται.

3.3 Τύποι με Δείκτες

Οι *τύποι με δείκτες* (indexed types) έχουν χρησιμοποιηθεί ευρέως για την ανάλυση κόστους. Ο πρώτος που εισήγαγε τον όρο είναι ο Zenger [Zeng97]. Ο ίδιος πρότεινε μία επέκταση του συστήματος τύπων Hindley/ Milner, ώστε το νέο σύστημα τύπων με δείκτες, να περιέχει αλγεβρικούς τύπους με παραμέτρους τύπων αλλά και παραμέτρους τιμών (δείκτες). Η γλώσσα των τύπων είναι κατασκευασμένη σύμφωνα με τις προδιαγραφές των τύπων με περιορισμούς (qualified types), όπου τα κατηγορήματα είναι πολυωνυμικές εξισώσεις. Όμως, εδώ, οι τύποι μπορεί επίσης να έχουν πολυώνυμα ως παραμέτρους. Αυτό το σύστημα επιτρέπει να ελεγχθούν πριν την εκτέλεση κάποιες ιδιότητες. Για παράδειγμα μπορούν να παραμετροποιηθούν οι πίνακες και τα διανύσματα με το μέγεθός τους ώστε να ελεγχθεί η συμβατότητα του μεγέθους τους, στατικά. Παρόλο που αυτός ο ορισμός έχει το πλεονέκτημα πως είναι πιο γενικός - λόγω της ελαστικότητας στην επιλογή της σημασιολογίας επιτρέπει την ύπαρξη αυθαίρετων δεικτών - δεν προσφέρεται για την ανάλυση του χρόνου εκτέλεσης όπως τα συστήματα που θα μελετήσουμε στο εξής.

3.3.1 Τύποι με Μέγεθος

Οι *τύποι με μέγεθος* (sized types) αποτελούν μια εξειδίκευση των τύπων με δείκτες. Η εξειδίκευση αυτή αφορά την επιλογή των δεικτών. Αυτοί δεν μπορεί να είναι αυθαίρετοι αλλά πρέπει πάντα να εκφράζουν το άνω όριο στο εκάστοτε μέγεθος των αλγεβρικών τύπων. Οι Hughes, Pareto και Sabry [Hugh96] σχεδίασαν και υλοποίησαν πρώτοι ένα σύστημα τύπων με μέγεθος για να αποδείξει κάποιες βασικές ιδιότητες των συστημάτων αντίδρασης (ασφάλεια μνήμης, deadlocks, τερματισμό αναδρομής, αποδοτικότητα λίστας). Απέδειξε την ορθότητα του παραπάνω συστήματος αναπτύσσοντας ένα κατάλληλο σημασιολογικό μοντέλο για τους τύπους με μέγεθος. Οι Vaskonelos και Hammond [Vasc08] επηρεάστηκαν άμεσα από την ερευνητική δουλειά των τριών επιστημόνων παραπάνω και χρησιμοποίησαν τους τύπους με μέγεθος για να αναλύσουν αυτόματα άνω όρια για τα κόστη εκτέλεσης (χώρου - χρόνου) αναδρομικών συναρτήσεων, συναρτήσεων ανώτερης τάξης και πολυμορφικών συναρτήσεων. Το κλασικό σύστημα τύπων Hindley - Milner και ο αντίστοιχος αλγόριθμος Damas - Milner επεκτείνονται με "συνέπειες" (effects) που περιγράφουν τα κόστη αποτίμησης. Παρακάτω παρουσιάζουμε την σημειογραφία της

γλώσσας που χρησιμοποιείται και τη σημασιολογία του μοντέλου κόστους.

- **Γραμματική της Γλώσσας**

$$\begin{aligned}\mathcal{L} ::= \; & x \mid n \mid b \mid [] \mid e_1 :: e_2 \mid p(e) \mid \lambda x.e \mid fix\ x.e \mid e_1 e_2 \qquad \text{όροι} \\ & \mid if\ e_1\ then\ e_2\ else\ e_3 \mid let\ x\ =\ e_1\ in\ e_2\end{aligned}$$

όπου, x, n, b, p είναι οι συνταχτικές κατηγορίες για μεταβλητές, φυσικούς αριθμούς, λογικές τιμές και primitive λειτουργίες, αντίστοιχα. Ο όρος $x.e$ είναι αφαίρεση συνάρτησης, το $fix\ x.e$ είναι μια αναδρομική συνάρτηση που ικανοποιεί την εξίσωση $x = e$. Το $let..in$ επιτρέπει τον ορισμό πολυμορφικών τοπικών μεταβλητών όπως στο σύστημα Hindley-Milner.

- **Μοντέλο Κόστους της $\mathcal{L}$** Η σημασιολογία της $\mathcal{L}$ είναι κλήση κατά τιμή αναγωγική σημασιολογία. Το κόστος μιας $\mathcal{L}$ - έκφρασης ορίζεται μέσω του αριθμού των βημάτων β-αναγωγής $(\lambda x.e)e' \rightarrow_\beta e[e'/x]$ και ορίζουν μηδενικό κόστος σε άλλους κανόνες αναγωγής. Αυτή η μετρική κόστους έχει το πλεονέκτημα να είναι εύκολα κατανοητή και να καταγράφει το ασυμπτωτικό κόστος για αναδρομικούς ορισμούς.

- **Εκφράσεις Κόστους**

$$\begin{aligned}z ::= \; & l \mid n \mid e \mid \omega \mid z_1 + z_2 \mid z - n \mid z_1 \times z_2 \qquad \text{σύνολο } \mathbf{ZExp} \\ & \mid max\ (z_1, z_2) \mid f_i(z)\end{aligned}$$

όπου, οι τιμές του συνόλου ανήκουν στο $\overline{\mathcal{N}} = \mathcal{N} \cup \{e, \omega\}$ οι φυσικοί αριθμοί αντιπροσωπεύουν πεπερασμένα μεγέθη και κόστη, το e αντιπροσωπεύει την απροσδιόριστη τιμή και το ω αντιπροσωπεύει την τιμή χωρίς όριο. Η συνήθης διάταξη leq στους φυσικούς εκτείνεται στο $\overline{\mathcal{N}}$ ορίζοντας $x \leq$ και $e \leq x$. Οι όροι $l \in \mathbf{ZVar}$ είναι η συντακτική κατηγορία των μεταβλητών με συνέπειες και $\{f_1,\ f_2, ...\ f_n\}$ ένα μετρήσιμο σύνολο από ονόματα συναρτήσεων.

- **Σημασιολογία για τις Εκφράσεις Κόστους**
Μία *εκτίμηση* ρ είναι μία ολική απεικόνιση από μεταβλητές με συνέπειες σε τιμές κόστους $\rho : \mathbf{ZVar} \rightarrow \overline{\mathcal{N}}$. Δεδομένης εκτίμησης ρ, η σημασιολογία μιας έκφρασης κόστους ορίζεται από την συνάρτηση αποτίμησης $[\![\,\cdot\,]\!]\rho : \mathbf{ZExp} \rightarrow \overline{\mathcal{N}}$

- **Τύποι με Μέγεθος**
Για τύπους συναρτήσεων, ένα *κρυμμένο κόστος* είναι συνημμένο στο βέλος της

συνάρτησης. Αυτό το κρυμμένο κόστος είναι ένα άνω όριο του κόστος αποτίμησης του σώματος της συνάρτησης.

$$\tau ::= \ \alpha \mid Bool \mid Nat^z \mid List^z \tau \mid \tau_1 \xrightarrow{z} \tau_2 \qquad \text{τύποι με μέγεθος}$$

όπου, α η συντακτική κατηγορία για τις μεταβλητές τύπων.

Οι τύποι με μέγεθος μας επιτρέπουν να περιγράψουμε τα μεγέθη των στοιχείων μιας δομής, καθώς και της ίδιας της δομής. Για παράδειγμα, η έκφραση $List^5(Nat^{10})$ ορίζει μια λίστα από φυσικούς αριθμούς μικρότερους από 10, της οποίας το μήκος είναι το πολύ 5.

Τυπική σημασιολογία για τους τύπους με μέγεθος δεν έχει κατασκευαστεί για αυτό το σύστημα, κυρίως διότι η επιδιωκόμενη σημασιολογία για αυτό περιλαμβάνει διαφορετικές μεταξύ τους τιμές.

- **Αποτελέσματα**
  Με αυτό το σύστημα μελετήθηκε το κόστος συναρτήσεων λιστών του Prelude της Haskell και απλών αναδρομικών συναρτήσεων, ωστόσο δεν καθίστατο δυνατόν να μελετηθεί το κόστος ορισμένων από το χρήστη τύπων δεδομένων. Κάποιες φορές επίσης, το κόστος των συναρτήσεων μπορεί και να υπερεκτιμηθεί. Αυτό συμβαίνει γιατί, η ορθότητα των αναδρομικών τύπων αποδεικνύεται αυτόματα με επαγωγή στο δείκτη μεγέθους και όχι με σχολιασμούς στον κώδικα. Τέλος, αυτό το σύστημα δεν μπορεί να συνδυάσει την θεωρία των τύπων με δείκτες με την λειτουργική ορθότητα, οπότε δεν καθίσταται δυνατή μια πιο ακριβής ανάλυση των προγραμμάτων μέσω αυτού.

3.3.2 Εκλεπτυσμένοι Δείκτες

Ένα σύστημα που συνδυάζει την ανάλυση πολυπλοκότητας με την ανάλυση λειτουργικής ορθότητας είναι η γλώσσα προγραμματισμού TiML (Timed ML). Η TiML χρησιμοποιεί τύπους με δείκτες για να εκφράσει τα μεγέθη των δομών δεδομένων - αφού ο χρόνος εκτέλεσης μιας συνάρτησης μπορεί να εξαρτάται από το μέγεθος της εισόδου της - αλλά και τα άνω όρια του χρόνου εκτέλεσης των συναρτήσεων.

Οι συναρτήσεις μπορεί να είναι παραμετρικές ως προς το μέγεθος της εισόδου τους. Δηλαδή η προγραμματίστρια μπορεί να επιλέξει διάφορες έννοιες μεγέθους όπως το ύψος ενός δέντρου, το μαύρο ύψος ενός red - black δέντρου, το μεγαλύτερο στοιχείο μιας λίστας και ούτω καθεξής. Παράλληλα χρησιμοποιεί *εκλεπτυσμένα sorts*, που εκφράζουν αναλλοίωτες των δομών δεδομένων (αυτές οι αναλλοίωτες αξιοποιούνται μονάχα για την ανάλυση κόστους), προϋποθέσεις και μετα-συνθήκες, για να περιορίσει αυτούς τους δείκτες. Τα sorts που εφαρμόζονται στους δείκτες σαν έννοια και πρακτική είναι όμοια με

την ανάθεση τύπων στους όρους. Προσφέρουν κάποιου είδους κατηγοριοποίηση με βάσει την μορφή που έχουν οι δείκτες. Τα εκλεπτυσμένα sorts είναι σαν τους εκλεπτυσμένους τύπους. Περιορίζουν ακόμη περισσότερο τους αποδεκτούς δείκτες όπως οι εκλεπτύνσεις στους τύπους περιορίζουν τους τύπους που θεωρούνται αποδεκτοί ως είσοδος ή ως έξοδος [1]. Εδώ, μία εκλέπτυνση δηλώνει ένα υποσύνολο δεικτών που ικανοποιούν ένα κατηγόρημα. Εφόσον το εκάστοτε κατηγόρημα μπορεί να αναφέρεται σε άλλους δείκτες στο πλαίσιο έκφρασης σχεσιακών περιορισμών, το υποσύνολο δεικτών είναι ένα εξαρτώμενο σύστημα τύπων. Σημαντικό είναι επίσης το γεγονός ότι προσφέρει τη δυνατότητα στην προγραμματίστρια να ορίζει νέους τύπους δεδομένων με δείκτες.

Ένα παράδειγμα είναι ο ορισμός των red-black δέντρων (Σχήμα 3.1). Ένα red-black δέντρο `rbt` στην TiML έχει 3 δείκτες, το μέγεθός του, το χρώμα της ρίζας και το μαύρο ύψος του. Κατά τα γνωστά, τα φύλλα του δέντρου, `Leaf`, είναι μαύρα με μηδενικό μαύρο ύψος. Τα παιδιά ενός κόμβου, `Node`, πρέπει να έχουν το ίδιο μαύρο ύψος, και αν το χρώμα του κόμβου είναι κόκκινο τότε τα παιδιά του πρέπει να είναι χρώματος μαύρου.

```
1  datatype color : {Bool} =
2           Black of color {true}
3           | Red of color {false}
4
5  datatype rbt 'a : {Nat(*size*)} {Bool} {Nat(*black-height*)} =
6           Leaf of rbt 'a {0} {true} {0}
7         | Node {lcolor color rcolor : Bool}
8                {lsize rsize bh (*black-height*) : Nat}
9                {color = false -> lcolor = true /\ rcolor = true }
10               { ... (*other invariants*)}
11        of color {color} * rbt 'a {lsize} {lcolor} {bh} * (key * 'a)
12           * rbt 'a {rsize} {rcolor} {bh}
13           --> rbt 'a {lsize + 1 + rsize} {color} {bh + b2n color}
```

Σχήμα 3.1: Υλοποίηση Red-Black δέντρων σε TiML.

Όσον αφορά την επαλήθευση των ορίων πολυπλοκότητας, αυτή γίνεται αυτόματα. Η προγραμματίστρια προσθέτει σχολιασμούς τύπων στον κώδικα και ο ελεγκτής τύπων δημιουργεί τις συνθήκες επαλήθευσης ώστε αυτές με τη σειρά τους να αποδειχθούν από τον SMT solver. Το πιο ιδιαίτερο χαρακτηριστικό της TiML στο κομμάτι της επαλήθευσης είναι ότι παρέχει αυτοματοποιημένη υποστήριξη για την επίλυση αναδρομικών σχέσεων,

[1] Εδώ ο όρος εκλεπτυσμένοι τύποι χρησιμοποιείται με την στενή έννοια δηλαδή, τύποι της μορφής $\{x : t \mid P(x)\}$. Ο όρος αυτός μπορεί να χρησιμοποιηθεί από άλλες πηγές με την πιο χαλαρή έννοια, δηλαδή τύποι εμπλουτισμένοι με επιπλέον πληροφορίες. Εμείς γενικά υιοθετούμε την πιο στενή έννοια του όρου.

μέσω ευρετικού ταιριάσματος μοτίβων με περιπτώσεις του Master Theorem . Έτσι, δεν χρειάζεται να παρέχει ο χρήστης καμία απόδειξη για τα θεωρήματα πολυπλοκότητας. Για παράδειγμα, εάν ο solver συναντήσει ένα αναδρομικό πρότυπο $T(n-1)+3 \leq T(n)$, όπου το $T(\cdot)$ είναι άγνωστο, συμπεραίνει ότι η πολυπλοκότητα της συνάρτησης είναι $O(n)$. Οι συμβολισμοί μεγάλου-Ο εκφράζονται στην TiML ως τύποι που έχουν εκλεπτυνθεί από το κατηγόρημα μεγάλου-Ο, το οποίο είναι μια δυαδική σχέση μεταξύ δύο δεικτών ενός τύπου συνάρτησης.

Όσον αφορά την ορθότητα του συστήματος, έχει αποδειχθεί στην γλώσσα Coq.

Ο ελεγκτής τύπων της TiML έχει δοκιμαστεί σε 17 δοκιμές απόδοσης, συμπεριλαμβανομένου κλασσικών δομών δεδομένων και αλγορίθμων (δέντρα, διπλά συνδεδεμένες λίστες, αλγόριθμοι ταξινόμησης και αναζήτησης), αλγορίθμων για την εύρεση συντομότερων μονοπατιών και συναρτησιακές ουρές.

3.3.3 Μονάδα Κόστους με Δείκτες

Μία άλλη προσέγγιση για την ανάλυση του κόστους εκτέλεσης αλλά και της λειτουργικής ορθότητας των προγραμμάτων, που βασίζεται στην θεωρία των τύπων με δείκτες, είναι αυτή που εισάγει το κόστος μέσα στον ορισμό του προγράμματος μέσω ενός τύπου δεδομένων μονάδας ο οποίος μετράει τα αφηρημένα υπολογιστικά βήματα, στο επίπεδο των τύπων. Αυτή η τεχνική έχει εφαρμοστεί στην Agda [Dani08] και στην Coq [McCa17].

3.3.3.1 Βιβλιοθήκη THUNK

Στην Agda, υλοποιήθηκε μία απλή βιβλιοθήκη, η `THUNK`, για την επαλήθευση της πολυπλοκότητας αμιγών συναρτησιακών δομών δεδομένων. Η βασική ιδέα κρύβεται πίσω από τον σχολιασμό του προς εκτέλεση κώδικα με "νι" ή αλλιώς "τικ", τα οποία αναπαριστούν τα αφηρημένα υπολογιστικά βήματα. Σημειώνουμε πως η προγραμματίστρια πρέπει να εισάγει τα "νι" στον κώδικα καθώς δεν υποστηρίζεται αυτόματος συμπερασμός.

$$\checkmark : Thunk\ n\ \alpha \rightarrow Thunk\ (1+n)\ a$$

Ο τύπος δεδομένων $Thunk$ είναι μία μονάδα με σχολιασμούς. Οι συνδυαστές μονάδας $(bind, return)$, χρησιμοποιούνται για να συνδυάσουν τις χρονικές πολυπλοκότητες των υποεκφράσεων. Οι τύποι για τις συναρτήσεις $return$ και $bind$ είναι οι εξής.

$$return :\ \alpha\ \rightarrow\ Thunk\ 0\ a$$
$$(\gg=) :\ Thunk\ m\ \alpha\ \rightarrow\ (\alpha\ \rightarrow\ Thunk\ n\ b) \rightarrow\ Thunk\ (m+n)\ b$$

Επίσης είναι απαραίτητες δύο ακόμη συναρτήσεις. Η συνάρτηση $force$ η οποία λειτουργεί ως διεπιφάνεια μεταξύ του κώδικα με σχολιασμούς και του απλού κώδικα, που δεν τρέχει μέσα στο $Thunk$ monad. Και η συνάρτηση pay η οποία επιτρέπει την αξιοποίηση της οκνηρής αποτίμησης της Agda ώστε να μην γίνεται στις περιπτώσεις οκνηρής αποτίμησης μιας συνάρτησης υπερεκτίμηση του κόστους. Οι τύποι αυτών των δύο συναρτήσεων είναι οι εξής.

$$force : Thunk\ n\ \alpha\ \rightarrow\ \alpha$$
$$pay : (m : \mathcal{N})\ \rightarrow\ Thunk\ n\ \alpha\ \rightarrow\ Thunk\ m\ (Thunk\ (n - m)\ \alpha)$$

Θα δώσουμε ένα παράδειγμα της συνάρτησης *append* για ακολουθίες ή αλλιώς διανύσματα, για να καταλάβουμε καλύτερα πως εισέρχονται οι σχολιασμοί κόστους στο επίπεδο των τύπων μέσω του $Thunk$ monad. Ο απλός ορισμός της συνάρτησης append σε Agda, η οποία δέχεται ως εισόδους δύο πολυμορφικές ακολουθίες μήκους n_1 και n_2 αντίστοιχα και επιστρέφει μία λίστα μήκους $n_1 + n_2$ είναι ο εξής.

$$(+\!\!+\ ) : Seq\ \alpha\ n_1 \rightarrow Seq\ \alpha\ n_2 \rightarrow Seq\ \alpha\ (n_1 + n_2)$$
$$nil +\!\!+\ ys \qquad = ys$$
$$(x :: xs) +\!\!+\ ys = x :: (xs +\!\!+\ ys)$$

Ενώ ο ορισμός της *append* στο $Thunk$ monad είναι ο εξής.

$$(+\!\!+\ ) : Seq\ \alpha\ m \rightarrow Seq\ \alpha\ n$$
$$\quad \rightarrow Thunk\ (1 + 2 * m)\ (Seq\ a\ (m + n))$$
$$nil +\!\!+\ ys \qquad = \checkmark return\ ys$$
$$(x :: xs) +\!\!+\ ys = \checkmark xs +\!\!+\ ys \gg\!= (\lambda xsys \rightarrow \checkmark return(x :: xsys))$$

Χρησιμοποιώντας τους συνδυαστές της βιβλιοθήκης, αποδεικνύεται ότι η συνάρτηση *append* είναι γραμμική ως προς μήκος της πρώτης ακολουθίας.

3.3.3.2 Βιβλιοθήκη C

Στην Coq δημιουργήθηκε επίσης μια βιβλιοθήκη που εισάγει μια ακριβή έννοια του χρόνου εκτέλεσης μιας συνάρτησης στον τύπο της, μέσω ενός τύπου δεδομένων μονάδας. Ο δείκτης της μονάδας είναι ένα λογικό κατηγόρημα (και όχι ένας εξαρτημένος τύπος) το οποίο μετρά το χρόνο εκτέλεσης. Η διαφοροποιήσεις που εισήγαγαν οι McCarthy, Fetscher, New, Feltey και Findler [McCa17] σε σχέση με την υλοποίηση που περιγράψαμε παραπάνω έγκεινται στα εξής σημεία. Αφενός, το λογικό κατηγόρημα που προστίθεται

στη μονάδα μπορεί να χρησιμοποιηθεί για να εκφράσει αναλλοίωτες των δομών δεδομένων, επιτρέποντας έτσι την εξέταση πιο σύνθετων περιπτώσεων (όπως τα δέντρα Braun του Okasaki [Okas97]). Αφετέρου, οι σχολιασμοί κόστους μπορούν να προστεθούν αυτόματα στον κώδικα και έπειτα, όταν εξαχθεί ο Ocaml κώδικας, να διαγραφούν.

Πιο συγκεκριμένα, λόγω της απαίτησης να μην εξάγονται τα κόστη, δεν είναι δυνατό να υπάρχει στη μονάδα ως δείκτης ένας φυσικός αριθμός που αναπαριστά το κόστος, δηλαδή μία τιμή Set [2]. Αντί αυτού, το κόστος πρέπει να αναπαρίσταται με κάποιο τρόπο από μία τιμή $Prop$ η οποία εξ ορισμού δεν είναι εξαγόμενη. Έτσι, η νέα μονάδα είναι ουσιαστικά μία συνάρτηση, C, η οποία δέχεται έναν τύπο και μία πρόταση. Η πρόταση αυτή παραμετροποιείται ως προς τις τιμές του τύπου και ως προς φυσικούς αριθμούς.

$$Definition\ C\ (A : Set)\ (P : A \to nat \to Prop)\ :\ Set\ :=$$
$$a : A \mid exists(an : nat),\ (P\ a\ an).$$

Για ένα δεδομένο A και P, η έκφραση $C\ A\ P$ είναι ένα εξαρτημένο ζεύγος του a, αποτελείται από μια τιμή τύπου A και μια απόδειξη ότι υπάρχει κάποιος φυσικός αριθμός an που σχετίζεται με το a μέσω της P. Ο φυσικός αριθμός an είναι ο χρόνος εκτέλεσης του και η τιμή P είναι μία μέτα-συνθήκη που περιλαμβάνει κάποια προδιαγραφή χρόνου (και ορθότητας) για τη συγκεκριμένη λειτουργία.

3.4 Αυτόματη Ανάλυση Πόρων Απόσβεσης

Γενικά, η Ανάλυση Πόρων Απόσβεσης είναι η ανάλυση χειρότερης περίπτωσης μιας *ακολουθίας λειτουργιών*. Το κίνητρο για την Ανάλυση Πόρων Απόσβεσης είναι ότι η εξέταση του άνω ορίου χρόνου εκτέλεσης ανά λειτουργία - ανεξάρτητα από τις υπόλοιπες - μπορεί να είναι πολύ απαισιόδοξη, εάν ο μόνος τρόπος για να φτάσουμε στην εκτέλεση μίας πολύ κοστοβόρας λειτουργίας είναι να έχουμε εκτελέσει εκ των προτέρων έναν μεγάλο αριθμό φτηνών λειτουργιών. Το κόστος απόσβεσης n λειτουργιών είναι το συνολικό κόστος όλων των λειτουργιών δια n.

[2] Το σύστημα τύπων της Coq (the Calculus of Inductive Constructions) έχει τις ακόλουθες ιδιότητες.

- Κάθε καλώς ορισμένος όρος έχει τουλάχιστον έναν τύπο.
- Κάθε καλώς ορισμένος τύπος είναι όρος.

Αυτό σημαίνει πως οι καλώς ορισμένοι τύποι έχουν επίσης τύπους. Ο τύπος ενός τύπου είναι πάντα μία σταθερά, η οποία ονομάζεται sort. Το sort Set είναι ένα από τα προκαθορισμένα sorts της Coq. Χρησιμοποιείται κυρίως για την περιγραφή τύπων δεδομένων και προδιαγραφών προγράμματος. Εξ ορισμού, όλες οι τιμές ενός τύπου που έχει τύπο Set μπορούν να εξαχθούν. Το sort $Prop$ μπορεί να χρησιμοποιηθεί για τον ορισμό προτάσεων και αποδείξεων με τον ίδιο τρόπο που χρησιμοποιείται το sort Set για προδιαγραφές και προγράμματα. Εξ ορισμού, όλες οι τιμές ενός τύπου που έχει τύπο $Prop$ δεν μπορούν να εξαχθούν. Για περισσότερες πληροφορίες ο αναγνώστης προτρέπεται να ανατρέξει στο βιβλίο [Bert04]

Η τεχνική αυτή παρουσιάστηκε για πρώτη φορά από τον Robert Tarjan [Tarj85] ο οποίος έθεσε την ανάγκη για μια πιο χρήσιμη μορφή ανάλυσης από τις κοινές πιθανοτικές μεθόδους που χρησιμοποιούνται. Ο Tarjan σημείωσε ότι η δήλωση και η απόδειξη των ορίων πολυπλοκότητας λειτουργιών σε ορισμένες δομές δεδομένων μπορούν να απλοποιηθούν εάν μπορούμε εννοιολογικά να θεωρήσουμε ότι η δομή δεδομένων είναι σε θέση να αποθηκεύει "πιστώσεις" που χρησιμοποιούνται από μεταγενέστερες λειτουργίες. Εισάγοντας σε μία δομή δεδομένων μία πίστωση για να χρησιμοποιηθεί σε μεταγενέστερο χρόνο, αποσβένεται το κόστος της λειτουργίας με την πάροδο του χρόνου. Η *απόσβεση* ή αλλιώς *κατά μέσο όρο με την πάροδο του χρόνου* χρησιμοποιήθηκε αρχικά για πολύ συγκεκριμένους τύπους αλγορίθμων, ιδίως αλγορίθμων για λειτουργίες δυαδικών δέντρων και λειτουργίες ένωσης. Ωστόσο, πλέον προσφέρεται για την ανάλυση πολλών άλλων αλγορίθμων.

Η Αυτόματη Ανάλυση Πόρων Απόσβεσης (ΑΑΠΑ) στοχεύει στην αυτόματη εξαγωγή ορίων απόσβεσης του κόστους εκτέλεσης. Αυτό επιτυγχάνεται χρησιμοποιώντας ένα σύστημα τύπων που δημιουργεί ανισότητες για πόρους οι οποίες επιλύονται από έναν solver γραμμικού προγραμματισμού. Το αρχικό σύστημα [Hofm03b] δημιουργήθηκε για την στατική πρόβλεψη της χρήσης χώρου από τον σωρό σε ένα πρόγραμμα γραμμένο σε συναρτησιακή γλώσσα πρώτης τάξης. Η ανάλυσή των Hofmann και Jost συσχετίζει κάθε στοιχείο μιας δομής δεδομένων με μια άδεια χρήσης ενός πόρου (στην συγκεκριμένη περίπτωσή το σύνολο των πόρων που μελετάται είναι ο σωρός). Αυτός ο πόρος διατίθεται στο πρόγραμμα όταν η δομή δεδομένων αποσυντίθεται χρησιμοποιώντας ταίριασμα μοτίβου. Κατά την κατασκευή ενός μέρους μιας δομής δεδομένων, οι απαιτούμενοι πόροι πρέπει να είναι *πραγματικά* διαθέσιμοι. Γι' αυτό, χρησιμοποιείται ένα γραμμικό σύστημα τύπων που διασφαλίζει ότι οι δομές δεδομένων που κρατάνε πόρους δεν αντιγράφονται, διότι αυτό θα σήμαινε ότι οι πόροι που καταναλώνονται μπορούν να αντιγραφούν και να ξαναχρησιμοποιηθούν.

Έκτοτε, έχουν αναπτυχθεί συστήματα βασισμένα στην παραπάνω προσέγγιση, που επιτρέπουν την αυτόματη εξαγωγή πολυωνυμικών ορίων απόσβεσης [Hoff12b, Hoff12a], την αυτόματη ανάλυση κόστους σε συναρτήσεις ανώτερης τάξης [Jost10], σε παράλληλα προγράμματα [Hoff15] και σε γλώσσες με οκνηρή σημασιολογία τύπου Haskell [Jost17]

Κεφάλαιο 4

Liquid Haskell

Αυτή η ενότητα εισάγει τον αναγνώστη στην υποκείμενη θεωρία και την πρακτική χρήση της Liquid Haskell. Υποθέτουμε πως η αναγνώστρια είναι εξοικειωμένη με κάποια συναρτησιακή γλώσσα προγραμματισμού (Haskell, Idris, ML, OCaml).

4.1 Liquid Haskell

Η Liquid Haskell (LH) παρουσιάστηκε για πρώτη φορά στα [Vazo14a, Vazo14b]. Η LH είναι μία επέκταση στο σύστημα τύπων της Haskell. Η επέκταση αυτή αφορά την εισαγωγή των εκλεπτυσμένων τύπων στο υπάρχον σύστημα τύπων - με τους οποίους εκφράζουμε ιδιότητες ορθότητας και πολυπλοκότητας - και πιο συγκεκριμένα την εισαγωγή των liquid τύπων και των αφηρημένων εκλεπτυσμένων τύπων. Οι Liquid τύποι εκφράζουν ένα υποσύνολο των εκλεπτυσμένων τύπων, οι οποίοι περιορίζουν ακόμη περισσότερο τη γλώσσα των κατηγορημάτων που μπορούν να υπάρχουν στους εκλεπτυσμένους τύπους ώστε να επιτυγχάνεται αυτόματα ο συμπερασμός τύπων στα ενδιάμεσα στάδια του προγράμματος, χαρακτηριστικό πολύ κρίσιμο ενός συστήματος τύπων χρήσιμου όχι μόνο θεωρητικά αλλά και πρακτικά για την προγραμματίστρια. Έπειτα, η εισαγωγή *αφηρημένων εκλεπτυσμένων τύπων* είναι ένας τρόπος για να μην υστερεί το εκλεπτυσμένο με liquid τύπους σύστημα σε εκφραστικότητα ενώ παραμένει αποφασίσιμο για έναν SMT solver. Η βασική ιδέα εδώ είναι η ποσοτικοποίηση των τύπων δεδομένων και συναρτήσεων μέσω της κωδικοποίησης εκλεπτυσμένων παραμέτρων ως μη ερμηνεύσιμες προτάσεις εντός της εκλεπτυσμένης λογικής.

4.2 Μία Σύνοψη της Εκτέλεσης της LH

Η Liquid Haskell από τον Αύγουστο του 2020 αποτελεί plugin του μεταγλωττιστή GHC [1]. Αυτό είναι σημαντικό γιατί μπορούμε να:

[1] `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/extending_ghc.html/#compiler-plugins`

1. Αναλύουμε ολόκληρα πακέτα με την ελάχιστη επανα-μεταγλώττιση, όπου είναι απαραίτητη, χρησιμοποιώντας την ήδη υπάρχουσα επίλυση εξαρτήσεων του GHC.

2. Εισάγουμε προδιαγραφές μεταξύ των πακέτων.

3. Χρησιμοποιούμε εργαλεία βασισμένα στο ghci (π.χ gchid ή το ghcide που είναι και πιο διαδραστικό).

Μπορούμε να ξεχωρίσουμε την εκτέλεση της LH πάνω σε ένα πρόγραμμα σε Haskell σε 3 φάσεις.

1. Αρχικά, η LH χρησιμοποιεί τον μεταγλωττιστή GHC της Haskell για να επιλύσει τις εξωτερικές αναφορές του προγράμματος (π.χ αναφορές σε άλλα modules όπως το Prelude ή modules που έχει ορίσει η προγραμματίστρια), να κάνει έλεγχο τύπων του προγράμματος της μορφής Hindley-Milner και να το μετατρέψει στην εσωτερική αναπαράσταση του πυρήνα του, ώστε έπειτα η δουλειά της LH να απλοποιηθεί.

2. Έπειτα, η Liquid Haskell συνδυάζει τον κώδικα και τους εκλεπτυσμένους τύπους σε ένα σύνολο από verification conditions (VC), οι οποίες είναι έγκυρες μόνο αν το πρόγραμμά ικανοποιεί μια δεδομένη ιδιότητα.

3. Τέλος, οι VC επιλύονται από έναν SMT solver (Z3, CVC4, MathSat). Εάν είναι έγκυρες όλες,το πρόγραμμά μας είναι ασφαλές, τυπώνει "SAFE" και δηλώνει πόσους περιορισμούς έχει ελέγξει, τυπώνει "N constraints checked", διαφορετικά απορρίπτει το πρόγραμμα, τυπώνει "UNSAFE" και το σημείο όπου βρέθηκε το λάθος.

4.3 Σύνταξη

4.3.1 Εκλεπτυσμένοι Τύποι

Η τυπική γραμματική των κατηγορημάτων που συνθέτουν τους εκλεπτυσμένους τύπους είναι η εξής:

$$c := 0, 1, 2, ... \qquad \text{σταθερές}$$
$$v := x, y, z, ... \qquad \text{μεταβλητές}$$

$e :=$	εκφράσεις
v	σταθερές
$\mid c$	μεταβλητές
$\mid (e + e)$	πρόσθεση
$\mid (e - e)$	αφαίρεση
$\mid (c * e)$	πολ/μος με σταθερά
$\mid (v\ e_1\ e_2\ ...\ e_n)$	εφαρμογή ανερμήνευτης συνάρτησης
$\mid (if\ p\ then\ e\ else\ e)$	if-then-else

$r :=$	σχέσεις
$==$	ισότητα
$\mid /=$	ανισότητα
$\mid (>=)$	μεγαλύτερο ή ίσο
$\mid (<=)$	μικρότερο ή ίσο
$\mid (>)$	αυστηρά μεγαλύτερο
$\mid (<)$	αυστηρά μικρότερο

$p :=$	κατηγορήματα
$(e\ r\ e)$	δυαδική σχέση
$\mid (v\ e_1\ e_2\ ...\ e_n)$	εφαρμογή κατηγορήματος ή ψευδωνύμου
$\mid (p\ \&\&\ p)$	λογικό και
$\mid (p\ \&\&\ p)$	λογικό ή
$\mid (p \Rightarrow p)$	λογική συνεπαγωγή
$\mid (not\ p)$	λογική άρνηση
$\mid true$	αληθές
$\mid false$	ψευδές

4.3.2 Προδιαγραφές

Οι σχολιασμοί που περιέχουν τους εκλεπτυσμένους τύπους γράφονται από την προγραμματίστρια στο αρχείο εισόδου ως σχόλια Haskell της μορφής:

```
{-@  annotation @-}
```

Προφανώς, εφόσον είναι εντός των άγκιστρων ο GHC τα αγνοεί, όπως κάνει με όλα τα σχόλια αυτής της μορφής. Όμως η LH τα επεξεργάζεται στο 2ο στάδιο της εκτέλεσης όπως περιγράψαμε στην προηγούμενη ενότητα.

Κάποιοι από τους σχολιασμούς που μπορεί να προσθέσει η προγραμματίστρια σε ένα αρχείο εισόδου είναι οι εξής: [2]

- `data` Είναι όμοιο με την σύνταξη σε Haskell όπου δηλώνονται αλγεβρικοί τύποι, μόνο που εδώ η προγραμματίστρια μπορεί επίσης να δηλώσει ότι ο τύπος μιας παραμέτρου εξαρτάται από την τιμή μιας προηγούμενης παραμέτρου. Δηλαδή, μπορεί να προσθέσει εκλεπτύνσεις στα πεδία ενός κατασκευαστή τύπου δεδομένων. Μπορεί επίσης, προσθέσει μία μέτρηση (`measure`) για τον τερματισμό του.

  Για παράδειγμα, ο τύπος $BSTree\ k\ v$ περιγράφει τα ordered trees. Δηλαδή, αυτά των οποίων οι τιμές σε ένα αριστερό υπόδεντρο είναι μικρότερες της ρίζας και ταυτόχρονα οι τιμές σε ένα δεξιό υπόδεντρο είναι μεγαλύτερες της ρίζας. To height είναι measure τερματισμού.

```
{-@ data BSTree k v =
          Nil
        | Node { tkey :: k
               , tval :: v
               , tleft :: BSTree {key:k | key < tkey } v
               , tright :: BSTree {key:k | key > tkey } v
               }
@-}
```

- `type` Επιτρέπει τον ορισμό ενός ψευδόνυμου (alias) για έναν εκλεπτυσμένο τύπο. Το χρησιμοποιούμε όταν στο πρόγραμμα μας θέλουμε να χρησιμοποιήσουμε πολλές φορές τον ίδιο τύπο. Μοιάζει με την σύνταξη σε Haskell μόνο που εδώ ο ορισμός μπορεί να περιέχει και τιμές (συμβολίζονται με κεφαλαία γράμματα) και τύπους (συμβολίζονται με πεζά γράμματα). Για παράδειγμα,

  ```
  {-@ type Nat = {v:Int | v >= 0 } @-}
  ```

  ο τύπος Nat περιγράφει όλους τους φυσικούς αριθμούς.

  ```
  {-@ type GeNum a  N = {v:a | \ N <= v} @-}
  ```

[2] Για μία αναλυτικότερη παρουσίαση, εδώ: `http://ucsd-progsys.github.io/liquidhaskell/specifications/`

ο τύπος GeNum περιγράφει όλες τις τιμές τύπου a που είναι μεγαλύτερες από την τιμή N.

Αντίστοιχα,

```
{-@ type RBTN k v N = {v: RBT k v  | bh v = N } @-}
```

ο τύπος $RBTN\ k\ v$ περιγράφει τα Red-Black Trees των οποίων το black height είναι ίσο με την τιμή N.

- `measure` Ορίζει το όνομα μιας Haskell συνάρτησης ως measure. Ένα measure είναι πιθανώς μια αναδρομική συνάρτηση που μπορεί να χρησιμοποιηθεί στους ορισμούς εκλεπτυσμένων τύπων. Ωστόσο, για να οριστεί μία συνάρτηση ως measure πρέπει να ικανοποιεί κάποιους περιορισμούς[Vazo14b]. Η συνάρτηση πρέπει να περιγράφει ιδιότητες αλγεβρικών δομών δεδομένων, πρέπει να έχει ως είσοδο μόνο μία παράμετρο, για κάθε κατασκευαστή πρέπει να έχει μόνον μία εξίσωση και στο εσωτερικό της πρέπει να καλεί μόνο primitive συναρτήσεις. Παραδείγματα measures είναι το μήκος μιας λίστας, και το ύψος ενός δέντρου. Δηλαδή,

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

  οι κατασκευαστές της λίστας και του δέντρου εκλεπτύνονται με την πληροφορία length και height αντίστοιχα

- `inline` Αντιγράφει έναν ορισμό συνάρτησης σε Haskell στην εκλεπτυσμένη λογική - ώστε να μπορεί να χρησιμοποιηθεί στο κατηγόρημα ενός τύπου. Ο ορισμός πρέπει να είναι στο σύνολό του διαθέσιμος στην εκλεπτυσμένη λογική κι επίσης δεν μπορεί να είναι αναδρομικός. Ένα παράδειγμα είναι το εξής:

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

  Γράφοντας το παραπάνω μπορούμε να χρησιμοποιήσουμε την max σε ένα measure.

  ```
  {-@ measure height @-}
  {-@ height :: t: Tree k v -> {n:Nat | n <= size t} @-}
  height :: Tree k v -> Int
  height Nil            = 0
  height (Node k v l r) = 1 + max (height l) (height r)
  ```

- `predicate` Επιτρέπει τον ορισμό ενός ψευδόνυμου (alias) για ένα κατηγόρημα. Χρησιμοποιείται συνήθως για τον ορισμό μακροσκελών κατηγορημάτων ή/και κατηγορημάτων που πρόκειται να χρησιμοποιηθούν πάνω από μία φορά. Οι παράμετροι τιμών γράφονται με κεφαλαία. Για παράδειγμα,

  ```
  {-@ predicate IsB T = not (col T == R) @-}
  ```

  το κατηγόρημα IsB ελέγχει αν το χρώμα ενός κόμβου ενός δέντρου είναι μαύρο.

- `invariant` Εισάγει μια καθολική αναλλοίωτη που σχετίζεται με έναν τύπο δεδομένων στο πρόγραμμα, που μπορεί να χρησιμοποιηθεί από την Liquid Haskell. Ωστόσο η εγκυρότητα αυτού του τύπου δεν ελέγχεται, πράγμα το οποίο μπορεί να οδηγήσει στη μη ασφάλεια του προγράμματος. Για παράδειγμα,

  ```
  {-@ invariant {v:[a] | (len v >= 0)} @-}
  ```

  το μήκος μίας λίστας δεν μπορεί να είναι αρνητικό. Η Liquid Haskell έπειτα υποθέτει πως κάθε λίστα που κατασκευάζεται στο προγραμμα ικανοποιεί αυτήν την αναλλοίωτη.

- `using` Εισάγει μια τοπική αναλλοίωτη που σχετίζεται με έναν συγκεκριμένο τύπο δεδομένων στο πρόγραμμα, η οποία μπορεί να χρησιμοποιηθεί από την Liquid Haskell. Η Liquid Haskell ελέγχει ότι αυτή η αναλλοίωτη είναι έγκυρη για τον συγκεκριμένο τύπο. Συγκεκριμένα, ελέγχει ότι *κάθε* εμφάνιση αυτού του τύπου στο πρόγραμμα ικανοποιεί αυτήν την αναλλοίωτη. Για παράδειγμα,

  ```
  {-@ using (Color) as {v: Color | v = R || v = B } @-}
  ```

  το χρώμα ενός δέντρου είναι είτε κόκκινο είτε μαύρο.

  ```
  {-@ using (BlackRBT k v) as {t:BlackRBT k v | rh t <= bh t}
  ```

  Το κοκκινο ύψος ενός διατεταγμένου, ισορροπημένου, με μαύρη ρίζα red-black tree είναι πάντοτε μικρότερο από το μαύρο ύψος του.

  Η Liquid Haskell αποδεικνύει πως η παραπάνω αναλλοίωτη είναι έγκυρη, αποδεικνύοντας πως όλες οι κλήσεις των κατασκευαστών του δέντρου (Nil, Node ..) την ικανοποιούν και έπειτα υποθέτει πως κάθε δέντρο στο πρόγραμμα την ικανοποιεί.

- `assume` Εισάγει μια υπόθεση - δεν ελέγχεται - που μπορεί να χρησιμοποιηθεί από την Liquid Haskell. Η λέξη-κλειδί "assume" υποδεικνύει ότι δεν έχουμε αποδείξει την ορθότητα αυτού του εκλεπτυσμένου τύπου. Αντ 'αυτού *ισχυριζόμαστε* ότι ο τύπος είναι αληθινός. Για αυτό πρέπει να είμαστε ιδιαίτερα προσεκτικές όπου τη χρησιμοποιούμε. Για παράδειγμα,

```
{-@ assume logComp ::   x:Int -> y: Int
                    -> { x <= y => log x <= log y }
@-}
logComp :: Int -> Int -> Proof
logComp _ _ = assumption
```

- `reflect` Χρησιμοποιείται για τις συναρτήσεις που δεν ικανοποιούν τις απαιτήσεις των measures, αλλά θέλουμε να τις εισάγουμε στην refinement λογική[Vazo16]. Έστω f μία τέτοια συνάρτηση. Το reflection δημιουργεί μία ανερμήνευτη συνάρτηση με το ίδιο όνομα στην εκλεπτυσμένη λογική, έστω $\mathbf{f}$ αυτή, αντιγράφει ολόκληρη την υλοποίηση της συνάρτησης f σε έναν εκλεπτυσμένο τύπο, έστω τ, ο οποίος εκλεπτυσμένος τύπος τ είναι ο τύπος που επιστρέφει η ανερμήνευτη συνάρτηση. Με τον όρο ανερμήνευτη (uninterpreted) εννοούμε ότι η $\mathbf{f}$ που ορίστηκε στην εκλεπτυσμένη λογική δεν συνδέεται με τη συνάρτηση f που έχουμε ορίσει στο πρόγραμμά μας. Στην εκλεπτυσμένη λογική, για τη $\mathbf{f}$ το μόνο που γνωρίζουμε είναι ότι ικανοποιεί το αξίωμα της ισότητας (congruence axiom). $\forall x, y.\ n == m \Rightarrow \mathbf{f}\ n == \mathbf{f}\ m$. Το reflection παρέχει στον SMT solver μόνο την τιμή της συνάρτησης για τα ορίσματα με τα οποία καλείται. Περιορίζοντας τις πληροφορίες που παρέχουμε στον solver κατά αυτόν το τρόπο ο έλεγχος των refinement types παραμένει αποκρίσιμος. Οι reflected συναρτήσεις εξυπηρετούν κυρίως την ανάγκη να μπορούν να χρησιμοποιηθούν οι συναρτήσεις από την Liquid Haskell σε εξισωτικές, ανισωτικές και αλγεβρικές αποδείξεις. Για παράδειγμα,

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

Η συνάρτηση fib μπορεί να χρησιμοποιηθεί πλέον σε οποιαδήποτε απόδειξη. Μία τετριμμένη (trivial) απόδειξη είναι η εξής:

```
{-@ fibCongr :: i:Nat -> j:Nat -> \{i == j => fib i == fib j} @-}
fibCongr _ _
    = trivial
    *** QED
```

και μία απλή απόδειξη είναι η εξής:

```
{-@ fibTwo :: _ -> {fib 2 == 1} @-}
fibTwo _
    = fib 2
    ==. fib 1 + fib 0
    *** QED
```

- `ple` Η Liquid Haskell έχει μία ακόμη λειτουργία που ονομάζεται *Proof by Logical Evaluation (PLE)* που επιτρέπει την παράλειψη κάποιων βημάτων στις εξισωτικές/ανισοτικές αποδείξεις ούτως ώστε αυτές παρόλο που γράφονται από την προγραμματίστρια να είναι πιο αυτοματοποιημένες. Η σύνταξη είναι απλώς η εξής.

```
{-@ ple lemma1 @-}
```

4.3.3 Κατασκευή Αποδείξεων

Η Liquid Haskell διαθέτει ένα πολύ εκφραστικό σύστημα τύπων με το οποίο μπορούμε να εκφράζουμε ιδιότητες πολυπλοκότητας και ορθότητας. Τα κατηγορήματα που εκφράζουν αυτές τις ιδιότητες, όταν ανήκουν στην SMT-αποφασίσιμη λογική, αποδεικνύονται αυτόματα από τον SMT solver. Κάποιες φορές θέλουμε να αποδείξουμε ιδιότητες συναρτήσεων, τις οποίες δεν μπορούμε να εκφράσουμε στην SMT-αποφασίσιμη λογική ή ακόμη και να συγκρίνουμε τις υλοποιήσεις δύο αλγορίθμων. Για να κατασκευάσουμε τέτοιες αποδείξεις χρησιμοποιούμε το refinement reflection και συναρτήσεις από την βιβλιοθήκη των *συνδυαστών απόδειξης* (proof combinators).

Συγκεκριμένα, τα θεωρήματα για reflected συναρτήσεις εκφράζονται ως νέες Haskell συναρτήσεις, χρησιμοποιώντας την reflected συνάρτηση στον εκλεπτυσμένο τύπο της συνάρτησης-θεωρήματος. Οι αντίστοιχες εξισωτικές/ανισοτικές αποδείξεις κωδικοποιούνται ως ορισμοί των νέων συναρτήσεων χρησιμοποιώντας συνδυαστές απόδειξης. Οι συνδυαστές αυτοί μας επιτρέπουν να κατασκευάζουμε αποδείξεις με εξισωτική ή αλγεβρική συλλογιστική. Μερικοί από αυτούς είναι:

- `Proof` Είναι ένας τύπος, $type\ Proof = ()$, ψευδώνυμο του τύπου unit, (), το χρησιμοποιούμε για να δηλώσουμε μία απόδειξη. Ο τύπος unit είναι αρκετός γιατί ένα θεώρημα, εκφράζεται ως μία εκλέπτυνση στις παραμέτρους εισόδου και εξόδου μίας συνάρτησης. Με άλλα λόγια η "τιμή" ενός θεωρήματος δεν έχει κάποια σημασία. Ο τύπος unit στον ίδιο τον ορισμό μίας περίπτωσης μιας απόδειξης, δηλώνει ότι η απόδειξη για αυτήν την περίπτωση είναι τετριμμένη (trivial) και δεν χρειάζεται κάποια επιπλέον πληροφορία.

- `trivial` Η συνάρτηση $trivial = ()$ χρησιμοποιείται στον ορισμό των τετριμμένων αποδείξεων.

- `QED` Εξυπηρετεί αισθητικό σκοπό, ούτως ώστε οι αποδείξεις σε Haskell να μοιάζουν με τις κλασσικές μαθηματικές αποδείξεις που γίνονται με το χέρι. Οι (ολοκληρωμένες) αποδείξεις τελειώνουν πάντοτε με την έκφραση $***QED$

- `ASS` Εξυπηρετεί ακριβώς τον ίδιο σκοπό με τον τύπο QED αλλά για μη ολοκληρωμένες αποδείξεις. Όλες οι μη ολοκληρωμένες αποδείξεις τελειώνουν πάντοτε με την έκφραση $***ASS$

- `***` Παίρνει οποιαδήποτε ως είσοδο έκφραση, αγνοεί την τιμή της και επιστρέφει στην έξοδο μία απόδειξη.

- `?` Συνδυάζει λήμματα μέσα σε μεγαλύτερα θεωρήματα. Αυτό είναι απαραίτητο, διότι μερικές φορές πρέπει να αναφερθούμε σε άλλα θεωρήματα στις αποδείξεις μας. Επειδή τα θεωρήματα είναι απλώς συναρτήσεις Haskell, το μόνο που χρειαζόμαστε είναι ένας τελεστής που δέχεται ένα όρισμα τύπου Proof, δηλαδή,

$$(?) :: a \to Proof \to a$$
$$x\ ?\ \_ = x$$

- `==.` Ο εκλεπτυσμένος τύπος της συνάρτησης $==$. εξασφαλίζει ότι οι παράμετροί της εισόδου της είναι ίσες. Στην έξοδο δίνει την δεύτερη παράμετρο της εισόδου ούτως ώστε να είναι δυνατή η αλυσιδωτή χρήση της συνάρτησης.

- `<=.` , `>=.` , `<.` , `>.` Οι εκλεπτυσμένοι τύποι αυτών των συναρτήσεων εξασφαλίζουν ότι οι παράμετροι της εισόδου τους ικανοποιούν την εκάστοτε σχέση διάταξης. Όπως και στην περίπτωση της $==$. έτσι κι εδώ, οι ανισοτικές αυτές συναρτήσεις δίνουν στην έξοδο την δεύτερη παράμετρο της εισόδου ούτως ώστε να είναι δυνατή η αλυσιδωτή χρήση τους.

4.3.3.1 Όλότητα και Τερματισμός

Όσον αφορά την απόδειξη για την ολότητα των συναρτήσεων, αυτή γίνεται με το κλασικό ταίριασμα μοτίβων του GHC και η προγραμματίστρια δεν χρειάζεται να δώσει κάποια επιπλέον πληροφορία πέραν των εκλεπτυσμένων τύπων στη Liquid Haskell.

Τα πράγματα δεν είναι ίδια για την απόδειξη τερματισμού. Όταν η παράμετρος μιας αναδρομικής κλήσης είναι άμεσα ή έμμεσα υποέκφραση της αρχικής παραμέτρου με τις οποίες κλήθηκε η συνάρτηση τότε η Liquid Haskell μπορεί να αποδείξει αυτόματα πως η συνάρτηση τερματίζει και αυτήν την μέθοδο την ονομάζουμε *κατασκευαστικός τερματισμός*. Στην περίπτωση που η συνάρτηση καλείται με περισσότερες από μία παραμέτρους

τότε πρέπει τουλάχιστον μία από αυτές να "μικραίνει" και όλες οι παράμετροι πριν από αυτή πρέπει να παραμένουν ίδιες. Ωστόσο ο κατασκευαστικός τερματισμός δεν είναι πάντα εφικτός. Τότε, πρέπει να παρέχει η προγραμματίστρια στο πρόγραμμα μία παράμετρο τερματισμού. Μια έκφραση, δηλαδή, που υπολογίζει έναν φυσικό αριθμό από την παράμετρο της συνάρτησης και ο οποίος μειώνεται σε καθένα αναδρομική κλήση. Αυτή η μέθοδος ονομάζεται *σημασιολογικός τερματισμός*

4.3.3.2 Μελέτη Περίπτωσης: Απόδειξη Μονοτονίας της Συνάρτησης fib

Στο Σχήμα 4.1, βλέπουμε πως συνδυάζονται όσα έχουμε αναφέρει σε αυτήν την υποενότητα, για την απόδειξη της μονοτονίας της συνάρτησης fib. Αρχικά, αποδεικνύουμε πως η fib είναι τοπικά αύξουσα κι έπειτα χρησιμοποιώντας αυτό το λήμμα αποδεικνύουμε πως η fib είναι μονότονη το οποίο μπορεί να γενικευτεί για οποιασδήποτε συνάρτηση.

Συγκεκριμένα, η απόδειξη fibUp γίνεται με επαγωγή στο n, πράγμα το οποίο δηλώνουμε με τον σχολιασμό $/[n]$. Η έκφραση $/[n]$ δηλώνει ότι το n είναι μία καλώς ορισμένη μετρική τερματισμού, δηλαδή το n μειώνεται σε κάθε αναδρομική κλήση. Η βάσεις της επαγωγής είναι η περίπτωση που το n είναι 0 και η περίπτωση που το n είναι 1, όπου απλώς εισάγουμε τις σχετικές ανισότητες με βάση τον ορισμό της συνάρτησης fib. Σε αυτές τις δύο περιπτώσεις το reflected refinement "ξεδιπλώνει" τον ορισμό της fib σε 0 και 1 αντίστοιχα, κι έτσι ο SMT επαληθεύει αυτόματα ότι $0 < 1$ και αντίστοιχα $1 + 0 \leq 1$. Όμοια, στην επαγωγική υπόθεση το $fib\ n$ μέσω του reflection ξεδιπλώνεται σε $fib(n-1) + fib(n-2)$, το οποίο με την εφαρμογή της επαγωγικής υπόθεσης στις δύο υποεκφράσεις, ολοκληρώνει την απόδειξη.

Η απόδειξη του $fibMonotonic$ γίνεται με επαγωγή στο y που είναι μία καλώς ορισμένη μετρική τερματισμού γιατί μειώνεται σε κάθε αναδρομική κλήση. Αν $x + 1 == y$ τότε χρησιμοποιούμε την απόδειξη $fibUp\ x$. Ειδάλλως, $x + 1 < y$ και χρησιμοποιούμε την επαγωγική υπόθεση $fibMonotonic\ (y-1)$ κι έπειτα την απόδειξη $fibUp\ (y-1)$.

```
{-@ fibUp :: n:Nat -> {fib n <= fib (n+1)} / [n] @-}
fibUp :: Int -> Proof
fibUp n | n == 0
    =.  fib 0
    <=. fib 1
    *** QED
    fibUp n | n == 1
    =.  fib 1
    <=. fib 1 + fib 0
    <=. fib 2
    *** QED
    fibUp n | n >= 1
    =.  fib n
    ==. fib (n-1) + fib (n-2)
    ?   fibUp (n-1)
    ?   fibUp (n-2)
    <=. fib n + fib (n-1)
    <=. fib (n+1)
    *** QED

{-@ fibMonotonic :: x:Nat -> y:{Nat | x < y } -> {fib x <= fib y}  / [y] @-}
fibMonotonic :: Int -> Int -> Proof
fibMonotonic x y | x + 1 == y
    =.  fib x
    ?   fibUp x
    <=. fib (x+1)
    <=. fib y
    *** QED
fibMonotonic x y | x + 1 < y
    =.  fib x
    ?   fibMonotonic x (y-1)
    <=. fib (y-1)
    ?   fibUp (y-1)
    <=. fib y
    *** QED
```

Σχήμα 4.1: Απόδειξη της αύξουσας μονοτονίας της fib σε Liquid Haskell.

4.4 Liquid Types

Οι Liquid Types (Logically Qualified Data Types)[Rond08] είναι ένα σύστημα που συνδυάζει τον συμπερασμό τύπων *Hindley-Milner* με την *Αφαίρεση Κατηγορήματος*, για να εξάγει αυτόματα εκλεπτυσμένους τύπους, με τέτοια ακρίβεια ώστε να μπορεί να αποδείξει ιδιότητες ορθότητας. Πίσω από αυτόν τον ορισμό υπάρχουν διάφορες τεχνικές.

- Ο αλγόριθμος συμπερασμού τύπων Hindley-Milner συχνά είναι συνδεδεμένος με σύγχρονες συναρτησιακές γλώσσες προγραμματισμού. Ωστόσο αυτό δεν είναι περιοριστικό απαραίτητα για το σύστημα. Οι Liquid τύποι θα μπορούσαν να εφαρμοστούν και σε γλώσσες με διαφορετικά συστήματα τύπων.

- Η τεχνική της Αφαίρεσης Κατηγορήματος βασίζεται στην αφηρημένη ερμηνεία. Εδώ γίνεται αναζήτηση του πιο ισχυρού κατηγορήματος, δηλαδή αυτού που ικανοποιεί ένα σύνολο περιορισμών σε ένα πεπερασμένο πλήρες σύνολο κατηγορημάτων που συσχετίζονται με μία σχέση συνεπαγωγής. Αυτό είναι πολύ ουσιώδες κομμάτι της προσέγγισης των Liquid τύπων.

- Οι Liquid τύποι είναι ένα υποσύνολο των εκλεπτυσμένων τύπων στο οποίο ο συμπερασμός τύπων είναι αποφασίσιμος. Η γλώσσα των λογικών κατηγορημάτων των liquid τύπων περιορίζεται σε αποφασίσιμες λογικές χωρίς ποσοδείκτες συμπεριλαμβανομένου λογικές φόρμουλες, γραμμική αριθμητική και ανερμήνευτες συναρτήσεις. Οι εκλεπτυσμένοι τύποι είναι τύποι της μορφής $\{x \ : \ t \mid P(x)\}$, δηλαδή τύποι που ικανοποιούν λογικά κατηγορήματα. Για παράδειγμα, αν έχουμε τον ορισμό της συνάρτησης pred

  $$pred\ n = n - 1$$

  η LH μπορεί να συμπεράνει ότι ο τύπος της pred είναι

  $$pred :: n : Int \rightarrow \{v : Int \mid v = n - 1\}$$

- Το σύστημα μπορεί να αποδείξει ιδιότητες ορθότητες, δηλαδή να κάνει έλεγχο τύπων για τύπους συναρτήσεων που έχει προσθέσει η προγραμματίστρια με τρόπο όμοιο με τον συμπερασμό τύπων. Για παράδειγμα, αν παρατηρήσουμε τον τύπο της pred θα δούμε ότι δεν ορίζει κάποιον περιορισμό για την παράμετρο της. Αυτό είναι σωστό προφανώς εφόσον ο ορισμός της pred δεν περιέχει κάποιον τέτοιο περιορισμό. Αυτό είναι κάτι που μπορεί να κάνει ο χρήστης όμως. Δηλαδή, μπορεί κάποια να περιορίσει τις αποδεκτές παραμέτρους στο σύνολο των θετικών ακεραίων.

  $$pred :: \{n : Int \mid n > 0\} \rightarrow \{v : Int \mid v = n - 1\}$$

- Ένας λογικός προσδιορισμός είναι μία έκφραση που έχει δυαδική τιμή, δηλαδή ένα κατηγόρημα, που αποτελείται από μεταβλητές προγράμματος, τη μεταβλητή ειδικής τιμής $\mathbf{v}$ που διαφέρει από τις μεταβλητές προγράμματος και την ειδική μεταβλητή $\star$ που μπορεί να αρχικοποιηθεί με μεταβλητές προγράμματος . Λέμε ότι ένας προσδιορισμός q ταιριάζει με τον q', αν αντικαθιστώντας κάποια υποσύνολα ελεύθερων μεταβλητών στον q με το $\star$ παράγεται ο προσδιορισμός q'. Για παράδειγμα ο προσδιορισμός $x \leq \mathbf{v}$ ταιριάζει με τον $\star \leq \mathbf{v}$. $\mathcal{Q}^*$ ονομάζεται το σύνολο των προσδιορισμών που δεν περιέχουν το $\star$ και ταιριάζουν με κάποιον προσδιορισμό στο $\mathcal{Q}$. Έστω τώρα ότι η μεταβλητή $\mathbf{v}$ είναι τύπου int, το $\mathcal{Q}$ ορίζεται ως $\{0 \leq \mathbf{v}, \star \leq \mathbf{v}, \mathbf{v} < \star, \mathbf{v} < len\ \star\}$ και οι μεταβλητές προγράμματος είναι οι x, y, k τύπου int και η α τύπου array. Τότε το $\mathcal{Q}^*$ είναι το $\{0 \leq \mathbf{v}, x \leq \mathbf{v}, y \leq \mathbf{v}, k \leq \mathbf{v}, \mathbf{v} < n, \mathbf{v} < len\ \alpha\}$. Ένας liquid τύπος στο Q είναι ένας εκλεπτυσμένος τύπος όπου τα κατηγορήματα είναι αυστηρά συζεύξεις προσδιορισμών από το $\mathcal{Q}^*$ Το σύστημα χρησιμοποιεί μία παραλλαγή του λ-λογισμού με πολυμορφισμό τύπου ML η οποία επεκτείνεται με Liquid Types. Συγκεκριμένα, η σύνταξη των κατηγορημάτων είναι η εξής.

$$\mathcal{Q} ::= true \mid q \mid \mathcal{Q} \wedge \mathcal{Q}$$

όπου, q προσδιορισμός στο $\mathcal{Q}^*$

Ένας liquid τύπος έχεις την μορφή $\{\mathbf{v} : \tau \mid e\}$ όπου τ είναι τύπος Hindley-Milner και e μία δυαδική έκφραση η οποία μπορεί να περιέχει την μεταβλητή $\mathbf{v}$ και ελεύθερες μεταβλητές του προγράμματος. Αυτός ο τύπος αναπαριστά όλες τις τιμές u με τύπο τ τέτοιες ώστε η έκφραση $e[u/\mathbf{v}]$ να είναι αληθής. Για παράδειγμα ο τύπος $\{\mathbf{v} : int \mid \mathbf{v} > x\}$ αναπαριστά τον τύπο των ακεραίων που είναι μεγαλύτεροι από την ελεύθερη μεταβλητή x. Αυτό ονομάζεται ένας εκλεπτυσμένος τύπος του τύπου int. Η $\mathbf{v}$ είναι η μεταβλητή τιμής και θεωρούμε ότι μπορεί παίρνει τις τιμές που ορίζει ο εκλεπτυσμένος τύπος και μόνον αυτές.

Σε έναν ορισμό τύπου συνάρτησης, ο τύπος του αποτελέσματος της συνάρτησης μπορεί να εξαρτάται από τις τιμές της εισόδου της. Έπειτα, ο τύπος μιας παραμέτρου της εισόδου μπορεί να εξαρτάται από τον τύπο μιας παραμέτρου που προηγείται. Για παράδειγμα η συνάρτηση range δέχεται μία παράμετρο τύπου Int, μια δεύτερη παράμετρο τύπου Int η οποία είναι επίσης μικρότερη της πρώτης και επιστρέφει μία λίστα με τιμές που κυμαίνονται μεταξύ των 2 παραμέτρων.

$$range :: lo : Int \rightarrow hi : \{Int \mid lo \leq hi\} \rightarrow [v : \{Int \mid lo \leq v \ \&\& \ v < hi\}]$$

Το σημαντικό είναι ότι η προγραμματίστρια δεν χρειάζεται να παρέχει στο πρόγραμμα τους liquid τύπους για όλες τις μεταβλητές προγράμματος, ώστε το Liquid Type System

(LTS) να αποδείξει αυτόματα όλες τις φόρμουλες, εφαρμόζοντας τους κανόνες τύπων της γλώσσας, εφόσον αυτές ανήκουν σε μια αποφασίσιμη λογική. Ειδάλλως θα χρειαζόταν τρομερή δουλειά από την μεριά της. Το LTS απαιτεί κάποιους ελάχιστους σχολιασμούς όμως. Στις περισσότερες περιπτώσεις μόνον οι τύποι των συναρτήσεων πρέπει να γραφτούν από την προγραμματίστρια, δηλαδή οι τύποι των παραμέτρων και ο τύπος την εξόδου. Σε αυτές τις περιπτώσεις πρέπει να περιγράψει τις σχέσεις μεταξύ των παραμέτρων και το πως εξαρτάται η έξοδος από αυτές, ούτως ώστε να έχουμε όλες τις απαραίτητες προυποθέσεις και μέτα-συνθήκες, δηλαδή τις προδιαγραφές του προγράμματος. Με αυτές τις πληροφορίες το LTS συμπεραίνει τους liquid τύπους όλων των ενδιάμεσων μεταβλητών και των υποεκφράσεων του προγράμματος. Για αυτό το σκοπό χρησιμοποιούνται αυστηροί κανόνες για τους λογικούς προσδιορισμούς, δηλαδή τα κατηγορήματα που συναντούμε στους liquid τύπους.

4.5 Αφηρημένοι Εκλεπτυσμένοι Τύποι

Οι εκλεπτυσμένοι τύποι, όταν ανήκουν στην SMT-αποφασίσιμη λογική μπορούν να επαληθευτούν αυτόματα. Αυτή η δυνατότητα για αυτόματη επαλήθευση στερούσε βέβαια έως τώρα τη δυνατότητα για ποσοτικοποίηση των εκλεπτύνσεων των τύπων δεδομένων και συναρτήσεων και κατά συνέπεια τη δυνατότητα επέκτασης των εκλεπτυσμένων τύπων στη σφαίρα των συναρτήσεων ανώτερης τάξης. Το κενό αυτό έρχονται να καλύψουν οι αφηρημένοι εκλεπτυσμένοι τύποι. Η βασική ιδέα είναι ότι οι αφηρημένες εκλεπτυσμένες σχέσεις (παράμετροι) κωδικοποιούνται στον ορισμό των τύπων δεδομένων και των τύπων συναρτήσεων ως *ανερμήνευτες* προτάσεις εντός της εκλεπτυσμένης λογικής. Αυτές οι προτάσεις παραμένουν SMT-αποφασίσιμες.

Για παράδειγμα, έστω ότι θέλουμε να κατασκευάσουμε ένα *εξαρτημένο ζεύγος* , του οποίου τα στοιχεία θέλουμε να έχουν σχέση διάταξης. Αυτό μπορούμε να το κάνουμε με δύο τρόπους. Ο ένας είναι ο εξής:

```
{-@ data OPair a
    = P { pairX :: a
        , pairY :: {a:a | pairX <= a}
        }
 @-}
data OPair a = P a a
```

Όπου, προσθέτουμε μία εκλέπτυνση στον τύπο a του δεύτερου στοιχείου του ζεύγους, δηλαδή ένα κατηγόρημα το οποίο είναι αληθές αν το πρώτο στοιχείο είναι μικρότερο ή ίσο από το δεύτερο στοιχείο του ζεύγους.

Αυτό όμως είναι κάπως περιοριστικό. Διότι, αν θέλουμε στη συνέχεια να χρησιμοποιήσουμε ένα ζεύγος του οποίου τα στοιχεία συνδέονται με μία διαφορετική σχέση διάταξης ή όποια άλλη σχέση τότε πρέπει να ορίσουμε νέο *data* που να ικανοποιεί αυτήν την απαίτηση.

Ο δεύτερος τρόπος καλύπτει αυτό το κενό, χρησιμοποιώντας abstract refinements.

```
{-@ data Pair a b <p :: a -> b -> Bool>
    = P { pairX :: a
        , pairY :: b <p pairX>
        }
@-}
data Pair a b = P a b
{-@ type OPair a = P <{\px py -> px <= py}> a a  @-}
```

Όπου, ο εκλεπτυσμένος ορισμός του *data Pair* εδώ λέει ότι, για κάθε κατηγόρημα p πάνω σε δύο τιμές τύπου a και τύπου b αντίστοιχα, ο κατασκευαστής P του τύπου Pair παίρνει μία τιμή τύπου a, έστω pairX και μία τιμή τύπου b, έστω pairY. Η τιμή $pairY$ ικανοποιεί το κατηγόρημα p, όταν το p είναι ένα κατηγόρημα πάνω στην $pairX$. Η γενική ιδέα είναι ότι μπορούμε να ορίζουμε τύπους δεδομένων, όπου μπορεί να υπάρχει μία αφηρημένη σχέση μεταξύ των συνθετικών στοιχείων του εκάστοτε τύπου, μέσω της παραμετροποίησης του ορισμού του τύπου δεδομένων με μία εκλεπτυσμένη παράμετρο p.

Ο τύπος *Opair* εκφράζει το σύνολο των ζευγών τύπου a όπου το δεύτερο συνθετικό στοιχείο είναι μεγαλύτερο ή ίσο από το πρώτο.

Αντίστοιχα, μπορούμε να εκφράσουμε διάφορα άλλα σύνολα ζευγών, όπως το σύνολο των σχετικά πρώτων ακεραίων [3]:

```
{-@ type CPPair a = Pair <{\px py -> (gcd px py) == 1 }> a a  @-}
```

Σε κάθε περίπτωση, η παράμετρος που εκφράζει αυτήν την σχέση επιβεβαιώνεται όταν κατασκευάζεται το ζεύγος. Δηλαδή, η συνάρτηση *okpair* επαληθεύεται ως ορθή

```
{-@ okpair :: x:Nat -> CPPair Nat @-}
okpair :: Int -> Pair Int Int
okpair x = P x 1
```

ενώ η *notokpair* απορρίπτεται.

Αντίστοιχα, όταν αποσυντίθεται το ζεύγος στον κατασκευαστή του, η LH υποθέτει πως ισχύει η σχέση μεταξύ των συνθετικών του στοιχείων. Δηλαδή, το παρακάτω επαληθεύεται.

[3] Εδώ, αν θέλουμε να χρησιμοποιήσουμε τον τύπο CPPair a πρέπει να γίνει reflected στην refinement λογική η συνάρτηση gcd.

```
{-@ notokpair :: x:Nat -> CPPair Nat @-}
notokpair :: Int -> Pair Int Int
notokpair x = P x 2

{-@ checkPair :: CPPair Nat -> {v:Nat | v == 1} @-}
checkPair :: Pair Int Int -> Int
checkPair (P' x y) = gcd x y
```

Έχει επίσης αξία να δούμε, πως μπορούμε να συνδυάσουμε τους αναδρομικούς τύπους δεδομένων με τις αφηρημένες εκλεπτύνσεις για να εκφράσουμε διάφορες ενδιαφέρουσες ιδιότητες χωρίς όμως να περιορίζουμε εξ'αρχής των ορισμό των τύπων με αυτές, προτού δηλαδή τις χρειαστούμε. Για παράδειγμα, ας ανακαλέσουμε τον ορισμό των δυαδικών δέντρων αναζήτησης από την υποενότητα 4.3.2. Αυτός ο ορισμός εξυπηρετεί τον σκοπό της επαλήθευσης της συγκεκριμένης ιδιότητας που αφορά την διάταξη των στοιχείων του δέντρου. Τί γίνεται όμως αν θέλουμε εκτός από τον ορισμό για τα δέντρα δυαδικής αναζήτησης, έναν ορισμό για τα δέντρα που έχουν τη μορφή δυαδικού σωρού; Πρέπει να παρέχουμε έναν εντελώς καινούριο ορισμό ακόμη κι αν οι διαφορές στη σχέση διάταξης είναι ελάχιστες;

Οι αφηρημένες παράμετροι στους τύπους δεδομένων μας επιτρέπουν να ορίσουμε μέσω μιας αφηρημένης σχέσης την εκάστοτε σχέση διάταξης των στοιχείων του δέντρου. Δηλαδή, μπορούμε να ορίσουμε γενικά τα δέντρα ως εξής:

```
{-@ data Tree k v < l :: root:k -> x:k -> Bool
                  , r :: root:k -> x:k -> Bool >
    = Nil
    | Node { key :: k
           , val :: v
           , tl  :: Tree <l, r> (k <l key>) v
           , tr  :: Tree <l, r> (k <r key>) v }
@-}
```

Έστω ότι έχουμε ένα δέντρο με ρίζα key. Οι αφηρημένες σχέσεις l και r συσχετίζουν το κλειδί key της ρίζας με *όλα* τα κλειδιά στο αριστερό και δεξιό υπόδεντρο του δέντρου. Τα κλειδιά στο αριστερό και στο δεξί υπόδεντρο πρέπει να είναι τύπου $k\ <l\ key>$ και $k\ <r\ key>$ αντίστοιχα.

Σημειώνουμε ότι η σύνταξη:

```
k <l key>
k <r key>
```

είναι απλώς μία απλοποίηση για τους αντίστοιχους εκλεπτυσμένους τύπους:

```
{lkey : k | l (key, lkey)}
{rkey : k | r (key, rkey)}
```

Δηλαδή, το πρώτο περιγράφει το σύνολο των τιμών τύπου k που ικανοποιούν το κατηγόρημα l με παραμέτρους το κλειδί key και μία τιμή, έστω lkey, από αυτό το σύνολο. Αντίστοιχα το δεύτερο περιγράφει το σύνολο των τιμών τύπου k που ικανοποιούν το κατηγόρημα r με παραμέτρους το κλειδί key και μία τιμή, έστω rkey, από αυτό το σύνολο τιμών.

Έτσι μπορούμε να ορίσουμε τα δέντρα δυαδικής αναζήτησης και τον δυαδικό σωρό (ελάχιστο-μέγιστο), δίνοντας συγκεκριμένη μορφή στις σχέσεις l και r.

```
{-@ type BST k v      = Tree < {\root k -> k < root}
                             , {\root k -> k > root} > k v
@-}
{-@ type MinHeap k v = Tree < {\root k -> root <= k}
                             , {\root k -> root <= k} > k v
@-}
{-@ type MaxHeap k v = Tree < {\root k -> root >= k}
                             , {\root k -> root >= k} > k v
@-}
```

4.6 Μοντέλο Ανάλυσης Κόστους

Η ιδέα για την μονάδα κόστους από το [Dani08] που υλοποιήθηκε στην Agda στο επίπεδο τύπων εφαρμόζεται στην LiquidHaskell στο επίπεδο των τιμών όμως, μέσω του *TickMonad*, με τις δημιουργούς να εμπνέονται παράλληλα από τα θεωρητικά θεμέλια του [Radi17].

Όπως αναλύσαμε στην υποενότητα 3.3.3, η βασική ιδέα κρύβεται στον σχολιασμό του προς εκτέλεση κώδικα με "νι" ή αλλιώς "τικ", τα οποία αναπαριστούν τα αφηρημένα υπολογιστικά βήματα. Κι εδώ, η προγραμματίστρια πρέπει να εισάγει τα "νι" στον κώδικα καθώς δεν υποστηρίζεται αυτόματος συμπερασμός. Γίνεται αντιληπτό, λοιπόν, πως η ορθότητα μιας ανάλυσης κόστους εξαρτάται άμεσα από την προσθήκη κατάλληλων σχολιασμών κόστους σε ένα πρόγραμμα από την προγραμματίστρια. Ως εκ τούτου, είναι ευθύνη του χρήστη να διασφαλίσει ότι αυτοί οι σχολιασμοί μοντελοποιούν σωστά την πρόβλεψη της χρήση ενός πόρου.

Γενικά, υποθέτουμε ότι το κόστος των συγκρίσεων, των αριθμητικών και λογικών υπολογισμών και το κόστος των αναδρομικών κλήσεων συνάρτησης υπερβαίνουν το κόστος όλων των άλλων λειτουργιών που πραγματοποιούνται κατά την εκτέλεση μιας

συνάρτησης. Έτσι, μπορούμε να συμπεραίνουμε ασυμπτωτικά άνω και κάτω όρια στον χρόνο εκτέλεσης μιας υλοποίησης ενός αλγορίθμου.

4.6.1 Η βιβλιοθήκη RTick

Η βιβλιοθήκη RTick αποτελείται από δύο modules. Το πρώτο είναι το *RTick* στο οποίο ορίζεται το Tick Monad και οι συναρτήσεις πάνω σε αυτό που μετρούν και τροποποιούν το κόστος. Το δεύτερο είναι το *ProofCombinators* στο οποίο ορίζονται επιπλέον συνδυαστές απόδειξης από αυτούς που ορίσαμε στην υποενότητα 4.3.3,που κωδικοποιούν τα βήματα της εξισωτικής/ ανισωτικής συλλογιστικής για τις τιμές και για το κόστος των εκφράσεων.

4.6.1.1 Module RTick

Το **Tick Monad**, που είναι και ο βασικός τύπος σε αυτήν την ανάλυση, αποτελείται από μία τιμή τύπου Int για το κόστος και μία πολυμορφική τιμή τύπου a.

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

Οι μέθοδοι εφαρμοστή είναι οι εξής:

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

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

Και οι μέθοδοι μονάδας είναι οι εξής:

Αυτές οι συναρτήσεις δεν μεταβάλλουν το κόστος των εκφράσεων παρά μόνο το καταγράφουν. Οι συναρτήσεις **pure** και **return** εισάγουν τις απλές τιμές τύπου a στο $Tick/Monad$ με μηδενικό κόστος. Οι συναρτήσεις $(<*>)$ και $(\gg=)$ αθροίζουν τα κόστη των υποεκφράσεων. Κατά τα άλλα η λειτουργία τους, αν "ξεχάσουμε" το κόστος είναι ίδια με τις Haskell συναρτήσεις, $pure, return, (<*>), (\gg=)$

Για να μεταβάλλουμε το κόστος χρησιμοποιούμε την συνάρτηση **step** για να δηλώσουμε ότι το κόστος αυξάνεται ή μειώνεται. Το ότι μπορεί να μειώνεται το κόστος, με μια πρώτη σκέψη φαίνεται παράδοξο, αν το σκεφτούμε λίγο καλύτερα όμως, θα δούμε

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

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

πως είναι όντως δυνατό να γίνεται αυτό σε μία γλώσσα οκνηρής αποτίμησης με ισχυρές δυνατότητες memoization όπως η Haskell.

Η συνάρτηση **step** δέχεται 2 ορίσματα. Το ένα είναι μία τιμή τύπου Int, έστω n, δηλαδή το κατά πόσο θέλουμε να αυξήσουμε (ή να μειώσουμε) το κόστος και το δεύτερο είναι μία τιμή τύπου $Tick\ x$. Αν το n είναι ένας θετικός ακέραιος τότε η εφαρμογή της συνάρτησης $step$ δηλώνει την κατανάλωση ενός πόρου από το x, ειδάλλως δηλώνει την παραγωγή ενός πόρου από το x. Επιστρέφει μία τιμή τύπου $Tick\ x$ της οποίας το κόστος έχει τροποποιηθεί κατά n.

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

Συχνά, εκτός από την τροποποίηση του κόστους μίας έκφρασης, θέλουμε να αθροίσουμε τα κόστη κάποιων υποεκφράσεων (όπως περιγράψαμε παραπάνω με την χρήση των τελεστών $(<*>)$ και $(\gg=)$) και ταυτόχρονα να τροποποιήσουμε το τελικό αποτέλεσμα είτε αυξάνοντάς το είτε μειώνοντας το κατά μία τιμή. Αυτό είναι εφικτό με το συνδυασμό των συναρτήσεων $(<*>)$ και $(\gg=)$ από τη μία και της συνάρτησης step από την άλλη.

Ας πάρουμε για παράδειγμα, την συνάρτηση append από την υποενότητα 3.3.3. Με χρήση των μεθόδων Εφαρμοστή και της συνάρτησης $step$ η append ορίζεται ως εξής:

Αντίστοιχα, με χρήση των συναρτήσεων Μονάδας $return, \gg=$ και της συνάρτησης $step$ το σώμα της append μπορεί να οριστεί επίσης ως εξής:

Παρατηρούμε ότι αν χρησιμοποιήσουμε τον τελεστή $(\gg=)$ η Liquid Haskell δεν μπορεί να αποδείξει αυτόματα το άνω όριο του κόστους. Αυτό γιατί, η LH δεν μπορεί να γνωρίζει ποιο είναι το κόστος της συνάρτησης $(\lambda xy \rightarrow return(x : xy))$ προτού

```
1   {-@ append :: x:[a] -> y:[a]
2         -> {t : Tick {xy:[a] | length xy == length x + length y}
3               | tcost t <= length x }
4   @-}
5   append :: [a] -> [a] -> Tick [a]
6   append [] ys     = pure ys
7   append (x:xs) ys = step 1 $ pure ( x :) <*> append xs ys
```

```
1   {-@ append' :: x:[a] -> y:[a]
2         -> {t : Tick {xy:[a] | length xy == length x + length y}
3   @-}
4   append' :: [a] -> [a] -> Tick [a]
5   append' [] ys     = return ys
6   append' (x:xs) ys =
7       step 1 $ (append' xs ys) >>= (\xy -> return (x : xy))
```

την εφαρμόσει στην τιμή xy. Ακόμη κι αν εμείς διαισθητικά γνωρίζουμε ότι αυτό το κόστος είναι 0. Το κόστος αυτό μπορεί να υπολογιστεί μονάχα εκτελώντας υπολογισμούς επιπέδου τύπων, πράγμα το οποίο δεν υποστηρίζεται, γιατί αυτοί οι υπολογισμοί δεν είναι δυνατόν να γίνουν αυτόματα από το σύστημα. Για τις περιπτώσεις όπου δεν είναι δυνατό να χρησιμοποιοήσουμε τις μεθόδους Εφαρμοστή, έχουμε δύο επιλογές. Είτε θα γράψουμε την απόδειξη για το κόστος σε Haskell χρησιμοποιώντας συνδυαστές απόδειξης, είτε θα χρησιμοποιήσουμε έναν εναλλακτικό τελεστή, μία παραλλαγή του τελεστή $(\gg\!=)$, στον οποίο ορίζουμε ένα άνω όριο κόστους στη συνάρτηση f, η οποία στην προκειμένη είναι η συνάρτηση $(\lambda xy \rightarrow return(x : xy))$. Ο τελεστής αυτός ονομάζεται leqBind και ορίζεται ως εξής:

```
{-@ 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
```

Από λειτουργικής άποψης, η έκφραση $leqBind\ n\ t\ f$ είναι ίδια με την έκφραση $t \gg\!= f$. Όμως, ο εκλεπτυσμένος τύπος του leqBind περιορίζει το σύνολο συναρτήσεων f σε αυτές των οποίων το κόστος δεν ξεπερνά τον αριθμό n. Έτσι, το συνολικό κόστος εκτέλεσης της έκφρασης $leqBind\ n\ t\ f$ δεν μπορεί να ξεπερνά το κόστος του t αυξημένο κατά n.

Έτσι οι γραμμές 6-7 της $append'$ μπορούν να γραφτούν ως:

```
append' (x:xs) ys =
    step 1 $ leqBind 0 (append' xs ys) (\xy -> return (x : xy))
```

Σημειώνουμε ότι, αν αλλάξουμε την τάξη μεγέθους στο n, π.χ αν το 0 γίνει 1 ή (length xs), τότε η LiquidHaskell δεν μπορεί να αποδείξει το κόστος και στην έξοδο δίνει "UNSAFE"

Επειδή, είναι πολύ συχνή η απαίτηση για αύξηση (ή και μείωση) του αθροίσματος του κόστους δύο υποεκφράσεων κατά 1 και αρκετές φορές κατά δύο - πράγμα το οποίο παρατηρήσαμε και εμείς τόσο κατά την πειραματική διαδικασία όσο και κατά την μελέτη της βιβλιογραφίας - στην RTick υπάρχουν κάποιοι τελεστές που ενσωματώνουν αυτήν την απαίτηση. Έτσι, στην *append* η γραμμή 7 μπορεί να γραφτεί πιο απλά ως εξής με την μέθοδο Εφαρμοστή ($< / >$):

```
append (x:xs) ys = pure ( x :) </> append xs ys
```

Και με τη μέθοδο Μονάδας ($> / =$) ως εξής:

```
append (x:xs) ys =  (append xs ys) >/= (\xy -> pure (x : xy))
```

Κάποιες από τις συναρτήσεις που μεταβάλλουν το κόστος και χρησιμοποιούμε στην συνέχεια είναι οι εξής:

- **wait** Εισάγει μία τιμή στο $Tick\ Monad$, όπως οι $pure, return$ αλλά ταυτόχρονα αυξάνει το κόστος της κατά 1, όπως οι $step\ 1\ \$\ return, step\ 1\ \$\ pure$.
- **waitN** Εισάγει μία τιμή στο $Tick\ Monad$, όπως οι $pure, return$ αλλά ταυτόχρονα αυξάνει το κόστος της κατά n, όπως οι $step\ n\ \$\ return, step\ n\ \$\ pure$. Το n πρέπει να είναι αυστηρά θετικό.
- **(</>)** Στο επίπεδο τιμών συμπεριφέρεται ακριβώς όπως η $(< * >)$, ωστόσο αυξάνει το αθροιστικό κόστος των υποεκφράσεων κατά ένα. Θα μπορούσαμε να γράφουμε στη θέση της, $step\ 1\ \$\ (f < * >)$
- **(>/=)** Όμοια με την πάνω περίπτωση, συμπεριφέρεται όπως η $(\gg=)$ όσον αφορά τις τιμές, αλλά αυξάνει το κόστος των υποεκφράσεων κατά ένα. Από άποψη λειτουργικότητας γενικά, είναι ίδιο με την έκφραση $step\ 1\ \$\ (\gg=)f$

4.6.1.2 Module ProofCombinators

Πέρα από τους συνδυαστές απόδειξης που περιγράψαμε στην υποενότητα 4.3.3, ορίζονται και κάποιοι νέοι, μέσω των οποίων επιτρέπεται η συσχετιστική ανάλυση κόστους. Δηλαδή, μπορεί να αποδεικνύεται η αποδοτικότητα μιας υλοποίησης ενός αλγορίθμου

έναντι μιας άλλης αλλά και να καταγράφεται όταν είναι δυνατό η ακριβής διαφορά κόστους ανάμεσα σε αυτές τις υλοποιήσεις. Όταν μία υλοποίηση είναι πιο αποδοτική από μία άλλη λέμε ότι έχουμε *βελτίωση κόστους*. Η ιδέα για την απόδειξη και την καταγραφή της βελτίωσης κόστους στη Liquid Haskell είναι από την έρευνα των [Mora99]

Ας πάρουμε για παράδειγμα το εξής. Για κάθε τύπο a, ο τύπος $[a]$ είναι ένα Μονοειδές. Ο τελεστής $(++)$, είναι ο τελεστής που ικανοποιεί τους μονοειδείς κανόνες για το σύνολο L που περιέχει όλα τα στοιχεία τύπου $[a]$ και το ουδέτερο στοιχείο για την πράξη $(++)$ επί του L είναι η κενή λίστα []. Οι εξής μονοειδείς κανόνες για τον τελεστή $(++)$

$$
\begin{aligned}
[] \;++\; ys &== ys && \text{αριστερή ταυτότητα} \\
xs \;++\; [] &== xs && \text{δεξιά ταυτότητα} \\
(xs \;++\; ys) \;++\; zs &== xs \;++\; (ys \;++\; zs) && \text{προσεταιριστικότητα}
\end{aligned}
$$

μπορούν να αποδειχθούν στην Liquid Haskell μέσω αποδείξεων εξισωτικής συλλογιστικής. Όμως, παρόλο που οι δύο πλευρές σε κάθε κανόνα δίνουν το ίδιο αποτέλεσμα, δεν μπορούμε να ισχυριστούμε ότι ισχύει το ίδιο και για το κόστος. Αυτή η τελευταία παρατήρηση, μπορεί να επαληθευτεί τυπικά από τους παρακάτω ισχυρισμούς χρησιμοποιώντας τον ορισμό του τελεστή $(++)$ στο Tick Monad από την συνάρτηση *append* που ορίσαμε παραπάνω. Για συντομία συμβολίζουμε την συνάρτηση *append* ως την infix συνάρτηση $\underline{++}$.

$$
\begin{aligned}
[] \;\underline{++}\; ys &\quad <=> \quad pure\ ys \\
xs \;\underline{++}\; [] &\quad >==\ length\ xs\ ==>\quad pure\ xs \\
(xs \;\underline{++}\; ys) \;\gg=\; (\underline{++}\; zs) &\quad >==\ length\ xs\ ==>\quad (xs \;\underline{++}) \;=\ll\; (ys \;\underline{++}\; zs)
\end{aligned}
$$

Η πρώτη ιδιότητα δηλώνει ότι ο κανόνας για την αριστερή ταυτότητα αποτελεί επιπλέον και μία *ισοδυναμία κόστους*. Αυτό σημαίνει ότι, οι εκφράσεις $[] ++ ys$ και ys δίνουν το ίδιο αποτέλεσμα και επιπλέον απαιτούν τον ίδιο αριθμό αναδρομικών κλήσεων της συνάρτησης *append*, δηλαδή 0. Προφανώς, εδώ η λίστα ys πρέπει να ενσωματωθεί στο Tick Monad για να κάνει type-check η ιδιότητα.

Η δεύτερη ιδιότητα δηλώνει ότι, στον κανόνα για την δεξιά ταυτότητα, η αριστερή πλευρά της εξίσωσης απαιτεί $length\ xs$ περισσότερους πόρους, δηλαδή $length\ xs$ περισσότερες αναδρομικές κλήσεις στην συνάρτηση *append*, από την δεξιά πλευρά. Εδώ, είναι προφανές, από τον ορισμό της *append*, εφόσον η μεταβλητή που "μειώνεται" σε κάθε κλήση είναι η λίστα xs, ότι και το κόστος της συνάρτησης θα εξαρτάται από το μήκος της λίστας xs. Δηλαδή, ο κανόνας αυτός αποτελεί *βελτίωση κόστους* από τα αριστερά

προς τα δεξιά. Και εδώ χρησιμοποιούμε την μέθοδο pure για να εισάγουμε την λίστα ys στο Tick Monad ώστε να κάνει type-check και να μπορούμε να μετρήσουμε το κόστος.

Η τρίτη ιδιότητα αποτελεί και αυτή *βελτίωση κόστους* από τα αριστερά προς τα δεξιά. Στην αριστερή πλευρά η λίστα xs διατρέχεται δύο φορές, μία φορά για να προσαρτηθεί στην λίστα ys και άλλη μία φορά για να προσαρτηθεί στη λίστα zs μαζί με την λίστα ys. Στην αριστερή πλευρά διατρέχουμε αντιθέτως μόνο μία φορά την λίστα xs για να προσαρτηθεί στην λίστα $ys \mathbin{+\!\!+} zs$.

Αυτές οι ιδιότητες αποδεικνύονται με την κατασκευή επαγωγικών αποδείξεων από την προγραμματίστρια. Έστω ότι θέλουμε να αποδείξουμε την δεύτερη ιδιότητα. Τότε κατασκευάζουμε μία συνάρτηση-θεώρημα, έστω rightIdCostImp αυτή και στον εκλεπτυσμένο τύπο της δηλώνουμε ότι δέχεται στην είσοδο μία πολυμορφική λίστα xs και ως έξοδο δίνει ένα κατηγόρημα ανάμεσα σε λίστες που εκφράζει αυτήν την ιδιότητα. Στο σώμα της συνάρτησης χρησιμοποιούμε τους συνδυαστές απόδειξης και με την επαγωγική μέθοδο καταλήγουμε στο ζητούμενο.

```
{-@ rightIdCostImp :: xs:[a]
                   ->  {QIMP (append xs []) (length xs) (pure xs)} @-}
rightIdCostImp :: [a] -> Proof
rightIdCostImp []
   =.   append [] []
   <=>. pure []
   ***  QED
rightIdCostImp (x:xs)
   =.   append (x:xs) []
   <=>. pure ( cons x  ) </> (append xs [])
   ? rightIdCostImp xs
   .>== length xs ==>. pure (cons x ) </> pure xs
   <=>. Tick 0 (cons x ) </> Tick 0 xs
   <=>. Tick 1 (cons x xs)
   <=>. Tick 1 (x:xs)
   .>== 1 ==>. Tick 0 (x:xs)
   <=>. pure (x:xs)
   ***  QED
```

Σχήμα 4.2: Απόδειξη βελτίωσης κόστους στον κανόνα δεξιάς ταυτότητας για το Μονοειδές List.

όπου *cons* είναι μια reflected συνάρτηση που εισάγει τον τελεστή (:) στην εκλεπτυσμένη λογική. Η Liquid Haskell πολλές φορές δεν μπορεί να αποδείξει την ισοδυναμία σε

αποδείξεις πάνω σε ήδη ορισμένους τύπους της Haskell (π.χ. List, Vector). Το QIMP (Quantified Improvement) είναι ένα κατηγόρημα που εκφράζει την ποσοτικοποιημένη βελτίωση κόστους, από τα αριστερά προς τα δεξιά, ανάμεσα σε δύο εκφράσεις που στο επίπεδο τιμών δίνουν το ίδιο αποτέλεσμα. Ορίζεται ως εξής.

$$\{-@\ predicate\ QIMP\ T1\ N\ T2 = \\ tval\ T1 == tval\ T2\ \&\&\ tcost\ T1 == tcost\ T2 + N\ @-\}$$

Συγκεντρωτικά, οι συνδυαστές για την συσχετιστική ανάλυση κόστους των συναρτήσεων είναι οι εξής:

- <=> . Ο εκλεπτυσμένος τύπος της συνάρτησης <=> . εξασφαλίζει ότι οι παράμετροί της εισόδου της, τύπου Tick a, είναι ίσες και επιπλέον ότι έχουν το ίδιο κόστος. Στην έξοδο δίνει την δεύτερη παράμετρο της εισόδου ούτως ώστε να είναι δυνατή η αλυσιδωτή χρήση της συνάρτησης.

- >∼> . Ο εκλεπτυσμένος τύπος της συνάρτησης >∼> . εξασφαλίζει ότι οι παράμετροί της εισόδου της, τύπου Tick a, είναι ίσες αλλά η πρώτη παράμετρος απαιτεί περισσότερους πόρους από την δεύτερη. Δηλαδή, έχουμε βελτίωση κόστους από τα αριστερά προς τα δεξιά, ή απλώς βελτίωση κόστους. Στην έξοδο δίνει την δεύτερη παράμετρο της εισόδου ούτως ώστε να είναι δυνατή η αλυσιδωτή χρήση της συνάρτησης.

- <∼< . Η συνάρτηση <∼< . είναι αντίστοιχη της συνάρτησης >∼> ., μόνο που εδώ έχουμε βελτίωση κόστους από τα δεξιά προς τα αριστερά.

- .>== Ο εκλεπτυσμένος τύπος της συνάρτησης .>== εξασφαλίζει ότι η πρώτη και η τρίτη παράμετρος της εισόδου της, τύπου Tick a, είναι ίσες αλλά η πρώτη παράμετρος απαιτεί συγκεκριμένα n περισσότερους πόρους από την τρίτη, όπου n είναι η δεύτερη παράμετρος εισόδου. Στην έξοδο δίνει την τρίτη παράμετρο της εισόδου ούτως ώστε να είναι δυνατή η αλυσιδωτή χρήση της συνάρτησης.

- .<== Η συνάρτηση .<== είναι αντίστοιχη της συνάρτησης .>==, μόνο που εδώ έχουμε βελτίωση κόστους κατά n από τα δεξιά προς τα αριστερά.

Κεφάλαιο 5

Μελέτη Περίπτωσης: Επαλήθευση Κόστους Απλών Δυαδικών Δέντρων Αναζήτησης

Σε αυτήν την ενότητα, ορίζουμε την δομή των απλών δυαδικών δέντρων αναζήτησης και συναρτήσεις επάνω σε αυτά, σε Liquid Haskell. Αυτές τις συναρτήσεις, τις ενσωματώνουμε στο *Tick Monad* για να αποδείξουμε σε κάθε περίπτωση το άνω όριο κόστους. Ξεκινάμε, δίνοντας τους βασικούς ορισμούς για τα δυαδικά δέντρα αναζήτησης και συνεχίζουμε με την υλοποίηση και την επαλήθευση των αντίστοιχων ιδιοτήτων σε Liquid Haskell. Για μία πιο ολοκληρωμένη μελέτη της θεωρίας των δυαδικών δέντρων αναζήτησης η αναγνώστρια μπορεί να ανατρέξει στα [Corm09, Sedg15].

5.1 Δυαδικά Δέντρα Αναζήτησης

Ένα δυαδικό δέντρο αναζήτησης (BST) είναι ένα δυαδικό δέντρο, το οποίο σε κάθε κόμβο έχει ένα συγκρίσιμο *κλειδί (key)* και μία *τιμή (value)* που σχετίζεται με αυτό το κλειδί. Εκτός από το κλειδί και την τιμή, κάθε κόμβος περιέχει και τα πεδία *left (αριστερό παιδί)* και *right (δεξί παιδί)* που αντιστοιχούν στο αριστερό και δεξιό υπόδεντρο αντίστοιχα. Επιπλέον, ικανοποιεί τον περιορισμό ότι το κλειδί σε κάθε κόμβο είναι μεγαλύτερο από όλα τα κλειδιά των κόμβων που ανήκουν στο αριστερό υπόδεντρο του κόμβου και μικρότερο από όλα τα κλειδιά των κόμβων που ανήκουν στο δεξί υπόδεντρο του κόμβου. Ο τελευταίος περιορισμός είναι γνωστός και ως *ιδιότητα δυαδικού δέντρου αναζήτησης (search tree property)*:

Ορισμός 5.1.1: *Έστω x ένας κόμβος σε ένα δυαδικό δέντρο αναζήτησης και kx το κλειδί σε αυτόν τον κόμβο. Αν y είναι οποιοσδήποτε κόμβος στο αριστερό υπόδεντρο του x και ky το κλειδί σε αυτόν τον κόμβο, τότε $ky \leq kx$. Αν y είναι οποιοσδήποτε κόμβος στο δεξί υπόδεντρο του x και ky το κλειδί σε αυτόν τον κόμβο, τότε $ky \geq kx$.*

Θεώρημα 5.1.1: *Μπορούμε να υλοποιήσουμε τις λειτουργίες αναζήτηση, εισαγωγή, διαγραφή, έτσι ώστε η καθεμία να εκτελείται σε $\mathcal{O}(h)$ χρόνο, σε οποιοδήποτε δυαδικό δέντρο αναζήτησης ύψους h*

5.2 Υλοποίηση της δομής δυαδικού δέντρου αναζήτησης σε LH

Ένα απλό δυαδικό δέντρο ορίζεται στην Haskell ως εξής:

```
data Tree k v = Nil
              | Node k v (Tree k v) (Tree k v) deriving Show
```

Στον τύπο $Tree$ προσθέτουμε τον αφηρημένο εκλεπτυσμένο τύπο:

```
{-@ data Tree k v < l :: root:k -> x:k -> Bool
                  , r :: root:k -> x:k -> Bool >
    = Nil
    | Node { key :: k
           , val :: v
           , tl  :: Tree <l, r> (k <l key>) v
           , tr  :: Tree <l, r> (k <r key>) v }
@-}
```

Για να εξασφαλίσουμε ότι ισχύει η ιδιότητα δυαδικού δέντρου αναζήτησης δίνουμε συγκεκριμένη μορφή στις σχέσεις l, r και ορίζουμε τον τύπο BST. Διαισθητικά, τα l και

```
{-@ type BST k v = Tree < {\root k -> k < root}
                        , {\root k -> k > root}> k v
@-}
```

r είναι σχέσεις μεταξύ του $root\ key$ και του καθενός κλειδιού στο αριστερό και το δεξιό του υπόδεντρο αντίστοιχα.

Έπειτα, ορίζουμε measures πάνω στον τύπο Tree - όντας ο γενικός τύπος, αν τα measures οριστούν για τον τύπο $Tree$ τότε θα ισχύουν και για τον τύπο BST - τα οποία θα χρειαστούμε για να επαληθεύσουμε τις ιδιότητες στη συνέχεια. Εφόσον έχουμε εισάγει την ιδιότητα search tree στον ίδιο τον τύπο δεν χρειαζόμαστε κάποιο measure για αυτήν την ιδιότητα, παρά μόνο για το ύψος και για το μέγεθος του δέντρου. Αυτά ορίζονται με τις γνωστές συναρτήσεις size και height και εμείς προσθέτουμε τους εκλεπτυσμένους τύπους ώστε να εξασφαλίσουμε ότι το μέγεθος και το ύψος οποιουδήποτε

```
{-@ size :: Tree k v -> Nat @-}
{-@ height :: t: Tree k v -> {h:Nat | h <= size t} @-}
```

δέντρου δεν μπορούν να είναι αρνητικά και επιπλέον ότι για οποιοδήποτε δέντρο T με n εσωτερικούς κόμβους ισχύει ότι $height\ T <= n$

Στη συνέχεια ορίζουμε τις βασικές συναρτήσεις get, set και $delete$ στο Tick Monad, για την αναζήτηση, την εισαγωγή και την διαγραφή μιας τιμής σε δυαδικά δέντρα αναζήτησης αντίστοιχα, για τις οποίες επαληθεύουμε μέσω των εκλεπτυσμένων τύπων τους διάφορες χρήσιμες ιδιότητες.

Αναζήτηση Η συνάρτηση get δέχεται ως ορίσματα ένα κλειδί και ένα δυαδικό δέντρο αναζήτησης, έστω key και t αντιστοιχα, εκτελεί αναζήτηση στο δέντρο για το key και επιστρέφει μία Maybe τιμή εντός του Tick Monad. Ο εκλεπτυσμένος τύπος της get εξασφαλίζει ότι, το κόστος αναζήτησης είναι μικρότερο ή ίσο του ύψους του δέντρου εισόδου και δεν χρειάζεται να παρέχουμε κάποια άλλη απόδειξη. Η διαδικασία αναζήτησης, η οποία ξεκινά από την ρίζα του δέντρου ακολουθώντας μία διαδρομή προς τα κάτω, είναι η εξής:

- Αν το t είναι κενό (Nil) τότε επιστρέφει την τιμή που επιστρέφει η κλήση της συνάρτησης $pure\ Nothing$, δηλαδή $Tick\ 0\ Nothing$ και η αναζήτηση τελειώνει. Αυτή η περίπτωση είναι η μία βάση της αναδρομής.

- Αν το key ισούται με το κλειδί του κόμβου που εξετάζεται, έστω k αυτό, τότε επιστρέφει την τιμή $wait\ Just\ v$, δηλαδή $Tick\ 1\ (Just\ v)$ όπου v η τιμή που αντιστοιχεί στο k. Εδώ εισάγουμε την τιμή $Just\ v$ στο Tick Monad, προσθέτωντας επίσης 1 στο κόστος γιατί έχει εκτελεσεί μία σύγκριση. Αυτή η περίπτωση είναι η δεύτερη βάση της αναδρομής.

- Αν το key είναι μεγαλύτερο από το κλειδί k του κόμβου που εξετάζουμε, τότε εκτελούμε αναδρομικά αναζήτηση στο δεξί υπόδεντρο του κόμβου, προσθέτοντας 1 στο κόστος γιατί έχουμε εκτελέσει μία σύγκριση.

- Αν το key είναι μικρότερο από το κλειδί k του κόμβου που εξετάζουμε, τότε εκτελούμε αναδρομικά αναζήτηση στο αριστερό υπόδεντρο του κόμβου, προσθέτοντας παράλληλα 1 στο κόστος γιατί έχουμε εκτελέσει μία σύγκριση.

Εισαγωγή Η συνάρτηση set δέχεται σαν ορίσματα ένα κλειδί, μία τιμή και ένα δυαδικό δέντρο αναζήτησης, έστω key, val και t αντίστοιχα και επιστρέφει ένα δυαδικό δέντρο αναζήτησης εντός του Tick Monad. Στην περίπτωση που το key ανήκει στο δέντρο εισόδου τότε η set επιστρέφει ένα δέντρο ίδιου μεγέθους με το δέντρο εισόδου, ειδάλλως επιστρέφει ένα δέντρο με ένα επιπλέον στοιχείο το οποίο περιέχει τα key, val, ένα κενό δεξί υπόδεντρο και ένα κενό αριστερό υπόδεντρο. Ο εκλεπτυσμένος τύπος της set εξασφαλίζει ότι, το κόστος εισαγωγής μίας τιμής σε ένα δυαδικό δέντρο αναζήτησης είναι μικρότερο ή ίσο του ύψους του δέντρου και δεν χρειάζεται να παρέχουμε κάποια άλλη απόδειξη.

```
{-@ reflect get @-}
{-@ get :: Ord k => k:k -> ts: BST k v
        -> { t:Tick (Maybe v) | tcost t <= height ts } @-}
get :: Ord k => k -> Tree k v -> Tick (Maybe v)
get _ Nil          = pure Nothing
get k' (Node k v l r)
    | k' < k       = step 1 (get k' l)
    | k' > k       = step 1 (get k' r)
    | otherwise    = wait (Just v)
```

Σχήμα 5.1: Η συνάρτηση get εντός του Tick Monad για ένα δυαδικό δέντρο αναζήτησης.

Η διαδικασία εισαγωγής είναι όμοια με την διαδικασία αναζήτησης. Οι βάσεις της αναδρομής είναι οι περιπτώσεις που είτε φτάνουμε στο κενό δέντρο, όπου δημιουργείται ένας κόμβος με τα key, val με κόστος 0, είτε φτάνουμε σε έναν κόμβο με κλειδί, έστω k, ίδιο με το key όπου απλώς ενημερώνουμε την τιμή που αντιστοιχεί στο k με την τιμή val. Εδώ προσθέτουμε 1 στο κόστος γιατί έχει εκτελεστεί σύγκριση μεταξύ των δύο κλειδιών. Η αναδρομή γίνεται και εδώ στο ύψος του δέντρου. Σε κάθε κλήση της set με παραμέτρους που δεν αντιστοιχούν στις 2 παραπάνω περιπτώσεις, το ύψος του δέντρου μειώνεται κατά 1, είτε πηγαίνοντας στο αριστερό υπόδεντρο, είτε πηγαίνοντας στο δεξί υπόδεντρο ανάλογα με το αποτέλεσμα της σύγκρισης μεταξύ των κλειδιών και προσθέτουμε 1 στο συνολικό κόστος.

Σημειώνουμε ότι, η συνάρτηση singleton δημιουργεί ένα δυαδικό δέντρο αναζήτησης με ένα στοιχείο, με κόστος 0.

Διαγραφή Η συνάρτηση $delete$ δέχεται σαν ορίσματα ένα κλειδί και ένα δυαδικό δέντρο αναζήτησης, έστω key, t αντίστοιχα και επιστρέφει ένα δυαδικό δέντρο αναζήτησης εντός του Tick Monad. Γενικά, στην περίπτωση που το key ανήκει στο t τότε η $delete$ επιστρέφει ένα δέντρο με ένα λιγότερο στοιχείο από το δέντρο εισόδου, ειδάλλως επιστρέφει ένα δέντρο ίδιου μεγέθους με αυτό της εισόδου. Ο εκλεπτυσμένος τύπος της $delete$ εξασφαλίζει ότι, το κόστος διαγραφής ενός κόμβου από ένα δυαδικό δέντρο αναζήτησης είναι μικρότερο ή ίσο του ύψους του δέντρου και δεν χρειάζεται να παρέχουμε κάποια άλλη απόδειξη.

Οι περιπτώσεις όπου το t είναι Nil ή $(Node\ k\ v\ l\ r)$ με k αυστηρά μικρότερο ή αυστηρά μεγαλύτερο από το key, είναι όμοιες με τις αντίστοιχες περιπτώσεις της συνάρτησης set. Στην περίπτωση που το key ταιριάξει με το k, καλούμε την $deleteI$ και κάνουμε pattern matching στο t με την εξής διαδικασία για να διατηρήσουμε την

```
{-@ reflect set @-}
{-@ set :: Ord k => ts:BST k v -> k -> v
        -> { t:Tick { ts':(BST k v)
                    | size ts + 1 == size ts'
                      || size ts  == size ts' }
           | tcost t <= height ts }
@-}
set :: Ord k => Tree k v -> k -> v -> Tick (Tree k v)
set Nil k v     = singleton k v
set (Node k v l r) k' v'
    | k' < k    = pure (\l' -> Node k v l' r) </> set l k' v'
    | k' > k    = pure (\r' -> Node k v l r') </> set r k' v'
    | otherwise = wait (Node k v' l r)
```

Σχήμα 5.2: Η συνάρτηση set εντός του Tick Monad για ένα δυαδικό δέντρο αναζήτησης.

```
{-@ reflect singleton @-}
{-@ singleton :: k -> v
              -> { t:Tick { ts:(BST k v) | size ts == 1 }
                 | tcost t == 0 }
@-}
singleton :: Ord k => k -> v -> Tick (Tree k v)
singleton k v = pure (Node k v Nil Nil)
```

Σχήμα 5.3: Η συνάρτηση singleton.

ιδιότητα δυαδικού δέντρου αναζήτησης.

- Ο κόμβος προς διαγραφή δεν έχει κανένα παιδί. Τότε τον διαγράφουμε, επιστρέφοντας στη θέση του απλώς το κενό δέντρο.
- Ο κόμβος έχει μόνον ένα παιδί, είτε αριστερό είτε δεξί. Τότε τον διαγράφουμε, επιστρέφοντας απλώς το δέντρο-παιδί.
- Ο κόμβος έχει 2 παιδιά, ένα δεξί και ένα αριστερό. Τότε, πρέπει να αναζητήσουμε τον "διάδοχο" του, έστω y, να αντιγράψουμε τα πεδία κλειδιού και τιμής του y, έστω k, v αυτά, στα αντίστοιχα πεδία στο t και να διαγράψουμε τον y, ο οποίος δεν έχει κανένα παιδί. Ο y είναι είτε ο δεξιότερος κόμβος στο αριστερό υπόδεντρο (δηλαδή ο αμέσως μικρότερος του t) είτε ο αριστερότερος κόμβος στο δεξι υπόδεντρο του t (δηλαδή ο αμέσως μεγαλύτερος του t). Εμείς υλοποιούμε την 2η περίπτωση

για τον y με την συνάρτηση $delMinKey$.

```
{-@ reflect delete @-}
{-@ delete :: Ord k => k:k -> ts:BST k v
           -> { t:Tick { ts':(BST k v)
                       | size ts'   == size ts - 1
                        || size ts' == size ts }
              | tcost t <= height ts }
@-}
delete :: Ord k => k -> Tree k v -> Tick (Tree k v)
delete _ Nil = pure Nil
delete k' t@(Node k v l r)
    | k' < k    = pure (\l' -> Node k v l' r) </> (delete k' l)
    | k' > k    = pure (\r' -> Node k v l r') </> (delete k' r)
    | otherwise = (deleteI t)
```

Σχήμα 5.4: Η συνάρτηση delete στο Tick Monad για τα δυαδικά δέντρα αναζήτησης.

```
{-@ reflect deleteI @-}
{-@ deleteI :: Ord k => ts: NEBST k v
           -> { t:Tick { ts':(BST k v)
                       | size ts' == size ts - 1  }
              | tcost t <= height ts }
@-}
deleteI :: Ord k => Tree k v -> Tick (Tree k v)
deleteI (Node _ _ Nil Nil) = pure Nil
deleteI (Node _ _ l Nil)   = pure l
deleteI (Node _ _ Nil r)   = pure r
deleteI (Node _ _ l r)     = pure f </> delMinKey r
    where f (MinKey k v r')  = Node k v l r'
```

Σχήμα 5.5: Η συνάρτηση deleteI.

Για να επαληθεύσει η Liquid Haskell ότι όντως η delMinKey και κατά συνέπεια η deleteI επιστρέφει ένα δυαδικό δέντρο που ικανοποιεί την ιδιότητα *search tree* πρέπει να επαληθεύσει ότι ο κόμβος που έχουμε διαγράψει έχει αντικατασταθεί όντως από τον διάδοχο του, στην προκειμένη από τον αμέσως "μεγαλύτερο" του στη διάταξη. Δηλαδή, πρέπει να επαληθευτεί ότι το κλειδί k που επιστρέφει η delMinKey, η οποία καλείται την πρώτη φορά από την deleteI με είσοδο το δεξί υπόδεντρο r, είναι το *minimum* κλειδί του r

Γι αυτό ορίζουμε έναν νέο τύπο, τον $MinKey\ k\ v$ ο οποίος ενσαρκώνει αυτήν την ιδιότητα.

```
{-@ data MinKey k v = MinKey { mkv_key  :: k
                             , mkv_val  :: v
                             , mkv_tree :: BST {x:k | mkv_key < x} v }
@-}
data MinKey k v = MinKey { mkv_key :: k
                         , mkv_val :: v
                         , mkv_tree :: Tree k v }
```

Έτσι, μέσω του εκλεπτυσμένου τύπου της συνάρτησης $delMinKey$ εξασφαλίζουμε ότι έχουμε υλοποιήσει σωστά την διαγραφή σε ένα BST, ότι διατηρούμε δηλαδή την ιδιότητα *search tree*.

```
{-@ reflect delMinKey @-}
{-@ delMinKey :: Ord k => ts:NEBST k v
              -> { t:Tick { m:(MinKey k v)
                          | size (mkv_tree m) == size ts - 1 }
                 | tcost t <= height ts }
@-}
delMinKey :: Ord k => Tree k v -> Tick (MinKey k v)
delMinKey (Node k v Nil r) = pure (MinKey k v r)
delMinKey (Node k v l r)   = pure f </> delMinKey l
    where f (MinKey k' v' l') = MinKey k' v' (Node k v l' r)
```

Σχήμα 5.6: Η συνάρτηση delMinKey.

5.3 Άνω και Μέσο Όριο Ύψους ενός Δυαδικού Δέντρου Αναζήτησης

Το άνω όριο του ύψους ενός δέντρου T με n εσωτερικούς κόμβους είναι n - κατ' επέκταση το άνω όριο κόστους εισαγωγής, αναζήτησης και διαγραφής στο δέντρο είναι n.

Αυτό το άνω όριο είναι, όμως, πολύ συντηρητική επιλογή, αν υποθέσουμε ότι το δέντρο T έχει κατασκευαστεί από (ομοιόμορφα) τυχαία κλειδιά ή, ισοδύναμα, ότι τα κλειδιά εισάγονται στο δέντρο T με τυχαία σειρά.

Ορισμός 5.3.1: *Ορίζουμε ως τυχαία κατασκευασμένο δυαδικό δέντρο αναζήτησης n κλειδιών ένα δέντρο που προκύπτει μέσω εισαγωγής των κλειδιών με τυχαία σειρά σε*

ένα αρχικά κενό δέντρο. Στην περίπτωση αυτή οι $n!$ μεταθέσεις των κλειδιών εισόδου είναι όλες ισοπίθανες.

Θεώρημα 5.3.1: *Το αναμενόμενο ύψος ενός τυχαία κατασκευασμένου δυαδικού δέντρου αναζήτησης με n διαφορετικά κλειδιά είναι $\mathcal{O}(logn)$.*

Σημειώνουμε ότι η ακριβής σταθερά είναι 1.39, δηλαδή η αναζήτηση σε ένα τυχαία κατασκευασμένο δυαδικό δέντρο αναζήτησης, απαιτεί $1.39 \cdot logn$ συγκρίσεις.

Η απόδειξη για αυτό το θεώρημα παρουσιάζεται αναλυτικά στο [Corm09]. Λόγω της έκτασης της και της πολυπλοκότητας της, δεν θα επιχειρήσουμε εδώ να την κατασκευάσουμε σε Liquid Haskell.

Κεφάλαιο 6

Μελέτη Περίπτωσης: Επαλήθευση Κόστους Red-Black Δέντρων

Σε αυτήν την ενότητα, ορίζουμε την δομή των red-black δέντρων και συναρτήσεις επάνω σε αυτά, σε Liquid Haskell. Αυτές τις συναρτήσεις, τις ενσωματώνουμε στο *Tick Monad* για να επαληθεύσουμε ιδιότητες ορθότητας και χρόνου εκτέλεσης. Ξεκινάμε, δίνοντας τους βασικούς ορισμούς για τα red-black δέντρα και συνεχίζουμε με την υλοποίηση και την ενσωμάτωση των ιδιοτήτων ορθότητας και πολυπλοκότητας εντός της υλοποίησης, σε Liquid Haskell. Για μία πιο ολοκληρωμένη μελέτη της θεωρίας των red-black δέντρων η αναγνώστρια μπορεί να ανατρέξει στο [Corm09]. Η ιδέα για την υλοποίηση της μεθόδου *insert* με τη βοήθεια των συναρτήσεων εξισορρόπησης είναι από το [Okas99] ενώ η ιδέα για την υλοποίηση της μεθόδου *delete* με τη βοήθεια των συναρτήσεων εξισορρόπησης και συγχώνευσης είναι από το [Kahr01].

6.1 Red-Black Trees

Ένα red-black δέντρο είναι ένα δυαδικό δέντρο αναζήτησης, αλλά με εξισορρόπηση. Κάθε κόμβος του δέντρου έχει ένα επιπλέον πεδίο πληροφορίας, το χρώμα του. Το χρώμα οποιουδήποτε κόμβου μπορεί να είναι είτε μαύρο είτε κόκκινο. Μέσω των περιορισμών που επιβάλλονται στην ανάθεση χρωμάτων στους κόμβους επιτυγχάνεται η εξισορρόπηση. Ένα red-black δέντρο εκτός από την ιδιότητα *search tree* πρέπει να ικανοποιεί 2 επιπλέον ιδιότητες.

- **Χρώμα:** Οποιοσδήποτε κόκκινος κόμβος έχει δύο μαύρα παιδιά, όπου οι κόμβοι Nil θεωρούμε ότι είναι μαύροι εξ ορισμού.
- **Ύψος:** Όλα τα μονοπάτια από την ρίζα του δέντρου - η οποία θεωρούμε ότι είναι μαύρη - σε οποιοδήποτε κόμβο Nil, περιέχουν τον ίδιο αριθμό μαύρων κόμβων.

Ορισμός 6.1.1: *Έστω τυχαίο red-black δέντρο t. Σε οποιαδήποτε απλή διαδρομή από την ρίζα του t μέχρι ένα φύλλο, το πλήθος των μαύρων κόμβων - εξαιρουμένης της*

ίδιας της ρίζας - ονομάζεται μαύρο ύψος του t. Συμβολίζουμε με $bh\ t$.

Θεώρημα 6.1.1: *Ένα red-black δέντρο με n εσωτερικούς κόμβους έχει ύψος μικρότερο ή ίσο του $2\log(n+1)$.*

Λήμμα 6.1.2: *Έστω τυχαίο red-black δέντρο t. Το t περιέχει τουλάχιστον $2^{bh\ t} - 1$ εσωτερικούς κόμβους.*

Θεώρημα 6.1.3: *Σε ένα red-black δέντρο, οι λειτουργίες αναζήτηση, εισαγωγή, διαγραφή εκτελούνται σε λογαριθμικό χρόνο, σε σχέση με το μέγεθος του δέντρου, στη χειρότερη περίπτωση.*

6.2 Υλοποίηση της δομής red-black δέντρου σε LH

Στην Haskell ένα red-black δέντρο μπορεί να αναπαρισταθεί ως εξής:

```
data Color      = R | B
data RBTree k v = Nil
                | Node Color k v (RBTree k v) (RBTree k v)
```

6.2.1 Τύποι και Measures

Αναλλοίωτη Διάταξης Εμείς, όμοια με την περίπτωση των απλών δυαδικών δέντρων αναζήτησης, προσθέτουμε τον εκλεπτυσμένο τύπο, στον τύπο RBTree, ο οποίος παραμετροποιείται με abstract refinements. και εξασφαλίζουμε ότι δεν παραβιάζεται η

```
{-@ data RBTree k v < l :: root:k -> x:k -> Bool
                    , r :: root:k -> x:k -> Bool>
    = Nil
    | Node { tCol :: Color
           , key  :: k
           , val  :: v
           , tl   :: RBTree <l, r> (k <l key>) v
           , tr   :: RBTree <l, r> (k <r key>) v
          }
@-}
```

ιδιότητα δυαδικών δέντρων αναζήτησης για τον τύπο $RBTree$ χρησιμοποιώντας τον εξής τύπο.

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

Πρέπει όμως να εξασφαλίσουμε ότι δεν παραβιάζονται οι 2 επιπλέον ιδιότητες που ορίσαμε παραπάνω για τα red-black δέντρα.

Αναλλοίωτη Μαύρου Ύψους Έχουμε πει ότι ο αριθμός των μαύρων κόμβων στο αριστερό υπόδεντρο οποιουδήποτε κόμβου πρέπει να είναι ίσος με τον αντίστοιχο αριθμό στο δεξί υπόδεντρο. Αυτό το εκφράζουμε μέσω του measure $isBH$ όπου bh είναι

```
{-@ measure isBH @-}
{-@ isBH :: RBTree k v -> Bool @-}
isBH :: RBTree k v -> Bool
isBH Nil              = True
isBH (Node c _ _ l r) = bh l == bh r
                     && isBH l
                     && isBH r
```

το measure που μετρά το μαύρο ύψος ενός δέντρου και ορίζεται ως εξής.

```
{-@ measure bh    @-}
{-@ bh :: t : RBTree k v -> Nat @-}
bh :: RBTree k v -> Int
bh (Nil)            = 0
bh (Node c k v l r) = bh l + if (c == R) then 0 else 1
```

Σημειώνουμε ότι, πράγματι το measure bh μετρά το ύψος μόνο για το αριστερό υπόδεντρο, ωστόσο αυτό αρκεί, εφόσον η $isBH$ εξασφαλίζει ότι το δεξί υπόδεντρο έχει το ίδιο μαύρο ύψος με το αριστερό.

Αναλλοίωτη Χρώματος Η αναλλοίωτη χρώματος ορίζεται μέσω του measure $isRB$ ως εξής.

```
{-@ measure isRB   @-}
isRB :: RBTree k v -> Bool
isRB (Nil)            = True
isRB (Node c k v l r) = isRB l && isRB r
                     && if c == R then (col l == B) && (col r == B)
                        else True
```

όπου col είναι η συνάρτηση - measure που απλώς επιστρέφει το χρώμα του κόμβου

Οι συναρτήσεις set και delete που θα ορίσουμε στην συνέχεια μπορεί να δημιουργήσουν σε κάποιο ενδιάμεσο στάδιο της εκτέλεσης ένα *almost red black tree*. Στα almost red-black δέντρα η αναλλοίωτη χρώματος μπορεί να παραβιάζεται στην ρίζα του δέντρου

```
{-@ measure col   @-}
col :: RBTree k v -> Color
col (Node c k v l r) = c
col (Nil)            = B
```

και μόνον εκεί. Δηλαδή μπορεί να έχουμε ένα ενδιάμεσο δέντρο με κόκκινη ρίζα και 1 ή 2 κόκκινα παιδιά. Αντί να δημιουργήσουμε νέους κατασκευαστές δεδομένων για αυτήν την περίπτωση, μπορούμε να ορίσουμε τα almost red black δέντρα μέσω ενός measure που απαιτεί να ισχύει η αναλλοίωτη χρώματος σε όλο το δέντρο εκτός της ρίζας.

```
{-@ measure isARB  @-}
isARB :: RBTree k v -> Bool
isARB (Nil)            = True
isARB (Node c k v l r) = isRB l && isRB r
```

Αναλλοίωτη Χρώματος Ρίζας Βάζοντας το κατηγόρημα $IsB\ T$ πλάι στον τύπο ενός δέντρου T, μπορούμε να εξασφαλίσουμε ότι το χρώμα της ρίζας του T είναι μαύρο.

```
{-@ predicate IsB T = not (col T == R) @-}
```

Σύνθεση Αναλλοίωτων Συνθέτοντας τις αναλλοίωτες που έχουμε περιγράψει ως εδώ σε ένα ψευδώνυμο τύπου μπορούμε να ορίσουμε ένα ασφαλές red-black δέντρο ως εξής.

```
{-@ type RBT k v = { t : ORBT k v | isRB t && isBH t && IsB t }  @-}
```

Όμως, όπως αναφέραμε και προηγουμένως, στα ενδιάμεσα στάδια των συναρτήσεων *set* και *delete* μπορεί να παραχθούν τόσο δέντρα που παραβιάζουν κάποιες ιδιότητες, ενώ διατηρούν κάποιες άλλες, όσο και δέντρα που είναι έγκυρα red-black. Γι αυτό, μας εξυπηρετεί να ορίσουμε αυτούς τους ενδιάμεσους τύπους και μέσω αυτών να ορίσουμε τους πιο αυστηρούς τύπους.

Almost Red-Black Trees

```
{-@ type ARBT k v     = {t: ORBT k v | isARB t && isBH t} @-}
{-@ type ARBTN k v N = {t: ARBT k v | bh t = N }          @-}
```

Ο πρώτος τύπος ορίζει το σύνολο των almost red-black δέντρων ως το σύνολο των διατεταγμένων δέντρων που ικανοποιούν επιπλέον την αναλλοίωτη χρώματος εξαιρουμένης της ρίζας του δέντρου που δεν απαιτείται να την ικανοποιεί.

Ο δεύτερος τύπος, ο $ARBTN\ k\ v\ N$, εκφράζει το σύνολο των almost red-black δέντρων που έχουν μαύρο ύψος ίσο με N και θα τον χρειαστούμε για να εξασφαλίσουμε ότι

ισχύει η αναλλοίωτη μαύρου ύψους για ένα δέντρο μετά από την κλήση μιας συνάρτησης εξισορρόπησης σε ένα υπόδεντρό του.

Red-Black Trees

```
{-@ type RBT k v     = {t: ARBT k v | isRB t}           @-}
{-@ type RBTN k v N = {t: RBT k v  | bh t = N}          @-}
```

Ο πρώτος τύπος ορίζει το σύνολο των red-black δέντρων ως το σύνολο των almost red-black δέντρων που ικανοποιούν επιπλέον την αναλλοίωτη χρώματος για τη ρίζα του δέντρου.

Ο δεύτερος τύπος, ο $RBTN\ k\ v\ N$, εκφράζει το σύνολο των red-black δέντρων που έχουν μαύρο ύψος ίσο με N και θα τον χρειαστούμε και αυτόν για να εξασφαλίσουμε ότι ισχύει η αναλλοίωτη μαύρου ύψους για ένα δέντρο μετά από την κλήση μιας συνάρτησης εξισορρόπησης σε ένα υπόδεντρό του.

Black Rooted Red-Black Trees

```
{-@ type BlackRBT k v  = {t: RBT k v | IsB T && bh T > 0 } @-}
```

Αυτός ο τύπος, εξασφαλίζει ότι η ρίζα του red-black δέντρου είναι μαύρη και ότι το δέντρο δεν είναι κενό. Αυτή είναι μια σύμβαση που κάνουμε για να αποδείξουμε ότι ύψος του δέντρου έχει λογαριθμική πολυπλοκότητα.

6.2.2 Εισαγωγή και Διαγραφή με Εξισορρόπηση

Η αναζήτηση ενός κλειδιού σε ένα red-black δέντρο είναι όμοια με την αναζήτηση σε ένα απλό δυαδικό δέντρο αναζήτησης. Τα πράγματα ξεκινούν να αποκτούν ενδιαφέρον, όταν χρειάζεται να προσθέσουμε ή να αφαιρέσουμε έναν κόμβο από ένα red-black δέντρο, χωρίς ωστόσο να παραβιάζουμε τις αναλλοίωτες χρώματος και μαύρου ύψους. Ακολουθούμε την μέθοδο εξισορρόπησης μέσω περιστροφών - με μικρές παραλλαγές - που παρουσίασε ο Okasaki στο [Okas99]. Αυτή η μέθοδος ανέδειξε την χρησιμότητα και την αναγκαιότητα των συναρτησιακών γλωσσών προγραμματισμού, αφού μία διαδικασία η οποία σε μία imperative γλώσσα απαιτεί ιδιαίτερη κατανόηση και αρκετές γραμμές κώδικα, εδώ υλοποιείται σε μόλις 5 γραμμές κώδικα, καθεμία για μια ξεχωριστή περίπτωση ενός red-black δέντρου που πρέπει να ”διορθωθεί”.

6.2.2.1 Εισαγωγή

Ας ξεκινήσουμε λοιπόν με την μέθοδο της εισαγωγής σε ένα red-black δέντρο. Η μέθοδος της εισαγωγής κι εδώ, ξεκινά με την αναδρομική αναζήτηση του υπόδεντρου όπου πρόκειται να τοποθετηθεί το νέο ζεύγος κλειδιού-τιμής. Ο καινούριος κόμβος που κρατά αυτό το ζεύγος χρωματίζεται πάντοτε κόκκινος. Αυτή η διαδικασία μπορεί να παραβιάσει

την αναλλοίωτη ενός red-black δέντρου με 2 τρόπους. Αφενός μπορεί να δημιουργήσει ένα δέντρο με κόκκινη ρίζα - όταν εισάγει έναν κόμβο σε κενό δέντρο. Αφετέρου, μπορεί να δημιουργήσει έναν κόκκινο κόμβο με ένα κόκκινο παιδί - όταν εισάγει έναν κόμβο σε ένα υπόδεντρο . Αυτό σημαίνει ότι διατηρείται η καθολική αναλλοίωτη μαύρου ύψους εν αντιθέσει με την τοπική αναλλοίωτη χρώματος. Για να επαναφέρουμε την αναλλοίωτη χρώματος, προτού επιστρέψει η αναδρομή, εξισορροπούμε το δέντρο με τις συναρτήσεις $balanceL$ και $balanceR$. Στο τέλος της αναδρομής, καλούμε τη συνάρτηση $makeBlack$ η οποία απλώς χρωματίζει μαύρη τη ρίζα.

```
{-@ reflect set @-}
{-@ set :: (Ord k) => k -> v
        -> t : RBT k v
        -> { t' : Tick (BlackRBT k v)
                | tcost t' <= height t}
@-}
set k v s = fmap makeBlack (insert k v s)

{-@ reflect makeBlack @-}
{-@ makeBlack :: {t : RBT k v | size t > 0} -> BlackRBT k v @-}
makeBlack (Node _ k v l r) = Node B k v l r

{-@ reflect insert @-}
{-@ insert ::  (Ord k) => k -> v
            -> t : RBT k v
            -> {t' : Tick { ts : (RBTN k v {bh t}) | size ts > 0}
                        | tcost t' <= height t }
@-}
insert :: (Ord k) => k -> v -> RBTree k v -> Tick (RBTree k v)
insert k v Nil  = pure (Node R k v Nil Nil)
insert k v (Node B key val l r)
    | k < key   = pure (\l' -> balanceL key val l' r) </> (insert k v l)
    | k > key   = pure (\r' -> balanceR key val l r') </> (insert k v r)
    | otherwise = wait (Node B key v l r)
insert k v (Node R key val l r)
    | k < key   = pure (\l' -> Node R key val l' r) </> (insert k v l)
    | k > key   = pure (\r' -> Node R key val l r') </> (insert k v r)
    | otherwise = wait (Node R key v l r)
```

Σχήμα 6.1: Μέθοδος εισαγωγής μέσα στο Tick Monad για ένα red-black δέντρο

Ας ξεκινήσουμε με τον τύπο της συνάρτησης *insert*. Η *insert* δέχεται ένα red-black δέντρο. Πιθανώς σε κάποια αναδρομική κλήση το δέντρο έχει κόκκινη ρίζα - αυτή είναι η μόνη ιδιότητα που παραβιάζει - οπότε ο τύπος $BlackRBT\ k\ v$ δεν είναι αποδεκτός παρά μόνο για την αρχική κλήση. Η *insert* επιστρέφει ένα red-black δέντρο με μαύρο ύψος ίδιο με το αρχικό. Αυτή η εκλέπτυνση είναι απαραίτητη για να αποδείξει ο SMT ότι διατηρείται η ιδιότητα μαύρου ύψους σε κάθε αναδρομική κλήση. Όσον αφορά το κόστος, κάθε φορά που εκτελείται μία πράξη σύγκρισης αυξάνουμε το αθροιστικό κόστος των υποεκφράσεων κατά 1. Ο SMT αποδεικνύει αυτόματα ότι το κόστος είναι μικρότερο ή ίσο του δέντρου της εισόδου, αλλά δεν μπορεί να αποδείξει ότι είναι λογαριθμικό. Αυτό θα το κάνουμε στη συνέχεια.

Η συνάρτηση *set* είναι η συνάρτηση που καλείται αρχικά, με είσοδο ένα δέντρο τύπου $BlackRBT\ k\ v$, δηλαδή ένα έγκυρο red-black δέντρο ή το κενό δέντρο. Κι επιστρέφει επίσης ένα έγκυρο red-black δέντρο αφου καλεί τη συνάρτηση *makeBlack* με είσοδο το δέντρο που έχει επιστρέψει η *insert* για να διορθώσει το χρώμα της ρίζας.

6.2.2.2 Εξισορρόπηση

Διαισθητικά, οι συναρτήσεις εξισορρόπησης καλούνται για να διορθώσουν ένα δέντρο στο οποίο εμφανίζεται ένας κόκκινος κόμβος με κόκκινο παιδί. Αυτό μπορεί στην πραγματικότητα να διορθωθεί χρωματίζοντας μαύρη τη ρίζα, εφόσον όμως η παραβίαση εμφανίζεται στον κόμβο της ρίζας. Το πρόβλημα γίνεται πιο ιδιαίτερο όταν έχουμε μία διαδοχή κόμβων χρώματος μαύρου-κόκκινου-κόκκινου σε ένα μονοπάτι από τη ρίζα όπου δεν αρκεί η κλήση της *makeBlack*. Εφόσον, η ρίζα έχει 2 παιδιά και 4 εγγόνια, υπάρχουν 4 περιπτώσεις όπου μπορεί να εμφανιστεί το μονοπάτι κόμβων χρώματος μαύρου-κόκκινου-κόκκινου.

Οι 4 περιπτώσεις στις οποίες παραβιάζεται η αναλλοίωτη χρώματος παρουσιάζονται στο Σχήμα 6.7. Αυτές τις περιπτώσεις σύμφωνα με το [Okas99] τις χειρίζεται η συνάρτηση balance. Εμείς χωρίζουμε την συνάρτηση *balance* σε *balanceL* και *balanceR* ανάλογα με το αν το δέντρο που εξισσοροπείται είναι αριστερό ή δεξί υπόδεντρο ενός κόμβου, για να μην κάνουμε περιττούς ελέγχους pattern matching. Οι άνω και αριστερή περιπτώσεις του σχήματος διορθώνονται από την *balanceL* ενώ οι κάτω και δεξιά περιπτώσεις από την *balanceR*. Το αποτέλεσμα της εξισορρόπησης και στις 2 συναρτήσεις διατηρεί το μαύρο ύψος, μετατρέποντας το μονοπάτι κόμβων σε κόκκινο-μαύρο-μαύρο.

Ας δούμε τις συναρτήσεις *balanceL* και *balanceR* και τί ιδιότητες μπορούμε να αποδείξουμε μέσω των εκλεπτυσμένων τύπων τους.

Η balanceL πρέπει να παίρνει ως παραμέτρους εισόδου ένα κλειδί k και μία τιμή που αντιστοιχούν στη ρίζα του δέντρο που εξετάζει η αναδρομή στο συγκεκριμένο στάδιο, ένα αριστερό υπόδεντρο - που πιθανώς παραβιάζει την ιδιότητα χρώματος - ένα δεξί

υπόδεντρο που ικανοποεί και της δύο ιδιότητες των red-black δέντρων και αυτά τα δύο υπόδεντρα είναι παιδιά του κόμβου με κλειδί k. Θέλουμε στην έξοδο να πάρουμε ένα νέο εξισορροπημένο δέντρο.

```
{-@ reflect balanceL @-}
{-@ balanceL :: k:k -> v
             -> {l : ARBT {key:k | key < k} v | size l > 0}
             -> r : RBTN {key:k | k < key} v {bh l}
             -> {t : (RBTN k v {1 + bh l}) | size t > 0}
@-}
balanceL z zv (Node R y yv (Node R x xv a b) c) d
    = Node R y yv (Node B x xv a b) (Node B z zv c d)
balanceL z zv (Node R x xv a (Node R y yv b c)) d =
    = Node R y yv (Node B x xv a b) (Node B z zv c d)
balanceL k v l r                                  =
    = Node B k v l r
```

Σχήμα 6.2: Η συνάρτηση balanceL

Με τον εκλεπτυσμένο τύπο της $balanceL$ (και της $balanceR$) επαληθεύουμε τα εξής:

- Τα δέντρα εισόδου l και r είναι σε σχέση διάταξης με το κλειδί k.
- Τα δέντρα εισόδου l και r έχουν το ίδιο μαύρο ύψος, έστω n.
- Τουλάχιστον ένα από τα l και r είναι έγκυρο red-black δέντρο.
- Το αποτέλεσμα t είναι ένα έγκυρο red-black δέντρο.
- Το δέντρο t έχει ύψος $n + 1$.

Αυτές οι συνθήκες που γράφουμε για τις παραμέτρους εισόδου και εξόδου είναι αναγκαίες ώστε να αποδείξει ο SMT ότι το t είναι έγκυρο red-black. Ως έξοδο λοιπόν η balanceL δίνει ένα red-black δέντρο. Αυτό το δέντρο έχει τουλάχιστον ένα στοιχείο και το μαύρο ύψος του είναι αυξημένο κατά 1 από το μαύρο ύψος των υπόδεντρων της εισόδου. Στην ουσία με αυτόν τον τύπο εκφράζουμε ότι το δέντρο της εξόδου ικανοποιεί τις 2 αναλλοίωτες, χρώματος και ύψους.

Η μορφή και οι ιδιότητες της $balanceR$ είναι ακριβώς αντίστοιχες με αυτές της $balanceL$, μόνο που εδώ εξισορροπούμε το δεξί υπόδεντρο.

Οι συναρτήσεις $balanceL$ και $balanceR$ θεωρούμε ότι έχουν μηδενικό κόστος εφόσον δεν εκτελούν καμία αριθμητική πράξη ή πράξη σύγκρισης.

```
{-@ reflect balanceR @-}
{-@ balanceR :: k:k -> v
             -> l : RBT {key:k | key < k} v
             -> {r : ARBTN {key:k | k < key} v {bh l} | size r > 0}
             -> {t : (RBTN k v {1 + bh l}) | size t > 0}
@-}
balanceR x xv a (Node R y yv b (Node R z zv c d))
    = Node R y yv (Node B x xv a b) (Node B z zv c d)
balanceR x xv a (Node R z zv (Node R y yv b c) d)
    = Node R y yv (Node B x xv a b) (Node B z zv c d)
balanceR x xv a b
    = Node B x xv a b
```

Σχήμα 6.3: Η συνάρτηση balanceR

6.2.2.3 Διαγραφή

Αρχικά η διαγραφή ενός στοιχείου του δέντρου, σημαίνει ότι και εδώ πρέπει το δέντρο να εξισορροπηθεί. Αυτό συνεπάγεται ότι μπορεί να υπάρξει ένας κόκκινος κόμβος στη ρίζα, οπότε χρησιμοποιούμε τη συνάρτηση $makeBlackD$, όμοια με τη συνάρτηση $makeBlack$ που έχουμε ήδη ορίσει, πάνω σε μία βοηθητική συνάρτηση del με την οποία εκτελούμε τη διαγραφή και εξισορροπούμε το δέντρο. Η διαφορά με τη συνάρτηση $makeBlack$ είναι ότι εδώ μπορεί η είσοδος και η έξοδος να είναι το κενό δέντρο.

```
{-@ reflect makeBlackD @-}
{-@ makeBlackD :: t  : ARBT k v
           -> {t' : RBT k v | (size t' = 0 || IsBlackRBT t' ) }
@-}
makeBlackD Nil            = Nil
makeBlackD (Node _ k v l r) = Node B k v l r
```

Έπειτα, στην περίπτωση της συνάρτησης $insert$, χρησιμοποιούσαμε τις συναρτήσεις εξισορρόπησης μόνο όταν η ρίζα του δέντρου που εξετάζαμε ήταν μαύρη, μιας και οι 4 κατηγορίες λανθανόντων δέντρων ανήκαν σε αυτήν την περίπτωση. Το ίδιο ισχύει και εδώ. Μόνο που οι συναρτήσεις εξισορρόπησης $balL, balR$ που θα δούμε παρακάτω δεν δίνουν με όλες τις εισόδους την ίδια έξοδο, όπως η $balanceL, balanceR$, αλλά δίνουν συμμετρικές μεταξύ τους εξόδους. Επίσης, πέρα από τις περιπτώσεις διαδοχής κόμβων μαύρου-κόκκινου-κόκκινου χρώματος, μας ενδιαφέρουν και οι περιπτώσεις που το αριστερό υπόδεντρο έχει μικρότερο μαύρο ύψος από το δεξί ή το αντίστροφο. Στην περί-

πτωση που βρίσκουμε το στοιχείο που θέλουμε να διαγράψουμε, αφαιρούμε τον κόμβο αυτόν και έπειτα συγχωνεύουμε το δεξί και το αριστερό του υπόδεντρο σε ένα δέντρο με τη συνάρτηση *merge*.

```
{-@ reflect delete @-}
{-@ delete :: Ord k => k
                    -> t : RBT k v
                    -> {ti : Tick  {t' : RBT k v
                                        | (size t' = 0 || IsBlackRBT t')}
                           | tcost ti <= height t}
@-}
delete k t = fmap makeBlackD (del k t)

{-@ reflect del @-}
{-@ del :: Ord k => k
        -> t : RBT k v
        -> { ti : Tick {t' : (ARBT k v)
                        | (if (IsB t &&& size  t>0)
                           then bh t' = bh t - 1
                           else bh t' = bh t )
                         &&& (not (IsB t) => isRB t') }
                | tcost ti <= height t }
       / [height t]
@-}
del k Nil        = pure Nil
del k (Node col key v l r)
    | k < key    = case l of
        Nil             -> wait (Node R key v Nil r)
        Node B _ _ _ _  -> pure (\l' -> balL key v l' r) </> (del k l)
        _               -> pure (\l' -> Node R key v l' r) </> (del k l)
    | k > key    = case r of
        Nil             -> wait (Node R key v l Nil)
        Node B _ _ _ _  -> pure (\r' -> balR key v l r') </> (del k r)
        _               -> pure (\r' -> Node R key v l r') </> (del k r)
    | otherwise = step 1 (merge k l r)
```

Σχήμα 6.4: Η μέθοδος delete εντός του Tick Monad για ένα red-black δέντρο

Η συνάρτηση *del* (Σχήμα 6.4) δέχεται ως παραμετρους εισόδου ένα κλειδί, ένα red-

black tree με πιθανώς κόκκινη ρίζα και επιστρέφει ένα almost red-black tree. Εδώ μετά την αφαίρεση ενός κόμβου μπορεί να παραβιαστούν και οι 2 ιδιότητες των red-black trees, διαδοχικών κόκκινων κόμβων και μαύρου ύψους. Θέλουμε να εξασφαλίσουμε ότι στην έξοδο θα πάρουμε ένα έγκυρο almost red-black δέντρο. Έτσι προσθέτουμε τις εξής εκλεπτύνσεις στον τύπο της εξόδου για να μπορεί να επαληθεύσει ο SMT ότι το δέντρο της εξόδου είναι έγκυρο. Σημειώνουμε ότι, μπορεί αυτές οι προσθήκες να φαίνονται προφανείς, ωστόσο δίχως αυτές ο SMT δεν μπορεί να επαληθεύσει το ζητούμενο. Πρέπει κάποιες φορές να ενισχύουμε την λογική, μη αυτόματα, με τέτοιες προτάσεις. Με δοκιμές και γνώμονα τις ιδιότητες των red-black δέντρων που έχουμε ορίσει, καταλήγουμε στις ελάχιστες εκλεπτύνσεις που πρέπει να προστεθούν για να αυτοματοποιηθεί η απόδειξη. Έστω t το δέντρο της εισόδου και t' το δέντρο της εξόδου εντός του $Tick\ Monad$.

- Το t' είναι almost red-black. Αυτό εξαρτάται από τις συναρτήσεις εξισορρόπησης $balL$ και $balR$ και τη συνάρτηση $merge$ που θα δούμε έπειτα. Αρκεί να μελετήσουμε τα σχήματα Σχήμα 6.8 και Σχήμα 6.9.

- Το t' διατηρεί το μαύρο ύψος του είτε αν το t είναι κενό (οπότε δεν έχουμε διαγράψει προφανώς κανένα στοιχείο) είτε αν η ρίζα του t είναι κόκκινη. Αν η ρίζα του t είναι μαύρη και δεν είναι κενό τότε το μαύρο ύψος του t' μειώνεται κατά ένα.

- Το t' ικανοποιεί την ιδιότητα $isRB$, δηλαδή δεν υπάρχουν δύο διαδοχικοί κόκκινοι κόμβοι ξεκινώντας από τη ρίζα αν η ρίζα του t είναι κόκκινη.

- Η διαγραφή ενός στοιχείου από το t κοστίζει λίγοτερο ή ίσο από το ύψος του t.

Αυτές οι προτάσεις θα γίνουν πιο κατανοητές παρακάτω που θα δούμε της συναρτήσεις εξισορρόπησης και συγχώνευσης, ξεκινώντας με τις $balL$ και $balR$ που είναι συμμετρικές μεταξύ τους.

6.2.2.4 Εξισορρόπηση

Όμοια με τις συναρτήσεις εξισορρόπησης για την $insert$ μας νοιάζουν τα εξής:

- Οι $balL, balR$ έχουν μηδενικό κόστος.

- τα δέντρα l και r της εισόδου είναι σε σχέση διάταξης με το κλειδί k.

- Το δέντρο l από το οποίο έχουμε διαγράψει το στοιχείο στην περίπτωση της balL έχει μικρότερο ύψος κατά 1 από το r και αντίστοιχα, το δέντρο r από το οποίο έχουμε διαγράψει το στοιχείο στην περίπτωση της balR έχει μικρότερο ύψος κατά 1 από το l.

- Το δέντρο t έχει έγκυρο μαύρο ύψος και στις 2 περιπτώσεις, $bh\ l + 1$ και $bh\ r + 1$ στην $balL$ και $balR$ αντίστοιχα, το οποίο προκύπτει από τον τρόπο που γίνεται η εξισορρόπηση.

- Αν το δέντρο r είναι μαύρο τότε το t που επιστρέφει η $balL$ ικανοποιεί την ιδιότητα $isRB\ t$ και αντίστοιχα, αν το δέντρο l είναι μαύρο τότε το t που επιστρέφει η $balR$ ικανοποιεί την ιδιότητα $isRB\ t$.

Για καλύτερη κατανόηση των παραπάνω σημείων παραπέμπουμε ξανά στα σχήματα 6.8 και 6.9.

```
{-@ reflect balL @-}
{-@ balL :: Ord k => k:k -> v
         -> l : ARBT {key:k | key < k} v
         -> r : RBTN {key:k | key > k} v {bh l+1}
         -> {t : (ARBTN k v {bh l + 1}) | IsB r => isRB t}
@-}
balL y yv (Node R x xv a b) c
    = Node R y yv (Node B x xv a b) c
balL x xv bl r@(Node B _ _ _ _)
    = balanceR x xv bl (makeRed r)
balL x xv bl (Node R z zv (Node B y yv a b) c)
    = Node R y yv (Node B x xv bl a) (balanceR z zv b (makeRed c))

{-@ reflect balR @-}
{-@ balR :: Ord k => k:k -> v
         -> l : RBT {key:k | key < k} v
         -> r : ARBTN {key:k | key > k} v {bh l - 1}
         -> {t : (ARBTN k v {bh l}) | IsB l => isRB t }
@-}
balR x xv a (Node R y yv b c)
    = Node R x xv a (Node B y yv b c) )
balR y yv l@(Node B _ _ _ _) bl
    = balanceL y yv (makeRed l) bl
balR z zv (Node R x xv a (Node B y yv b c)) bl
    = Node R y yv (balanceL x xv (makeRed a) b) (Node B z zv c bl)
```

Σχήμα 6.5: Η συναρτήσεις balL, balR

6.2.2.5 Συγχώνευση

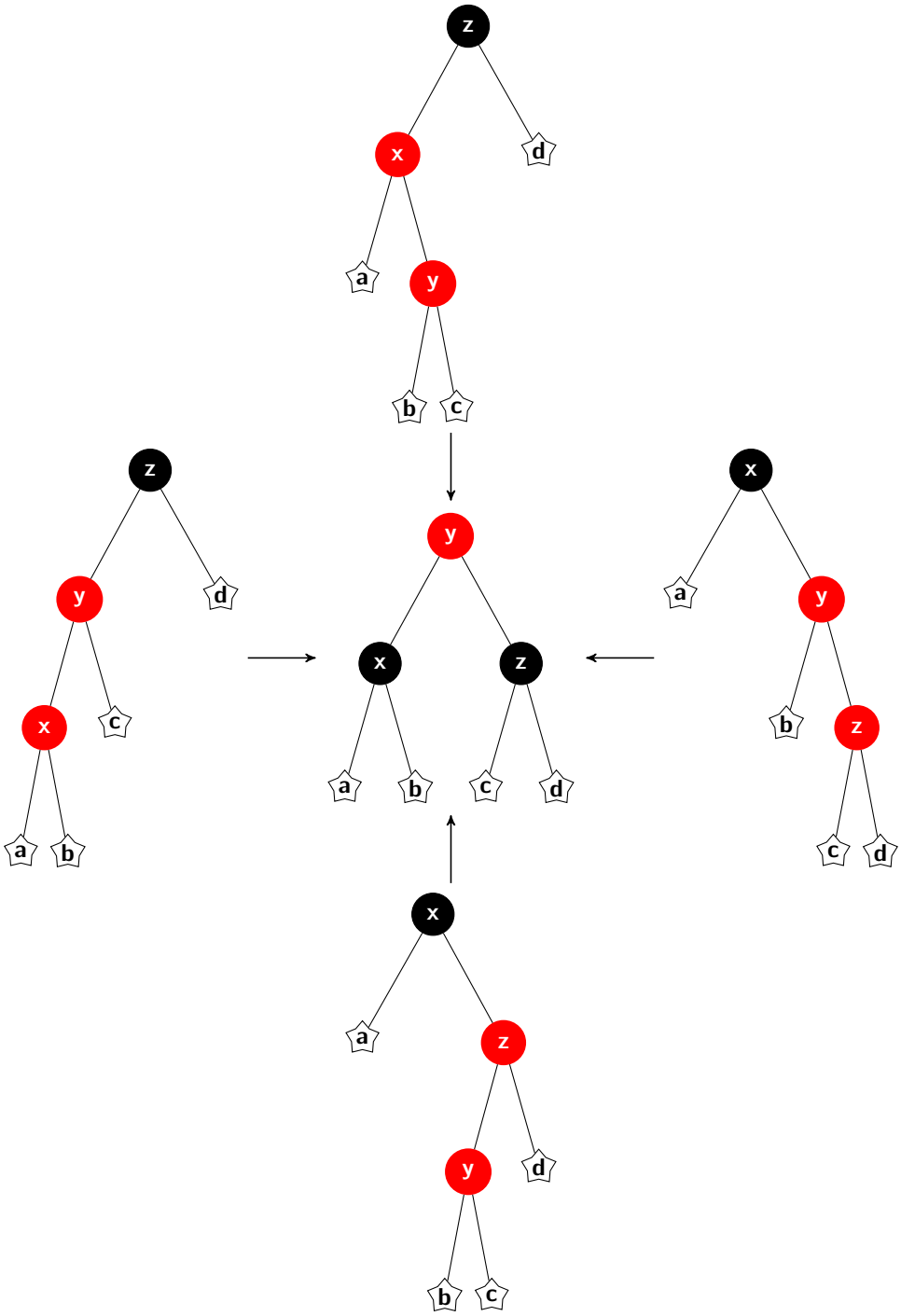
Διαισθητικά, η συνάρτηση $merge$ συγχωνεύει 2 υπόδεντρα l, r που ικανοποιούν την ιδιότητα δυαδικού δέντρου αναζήτησης, την ιδιότητα $isRB$ και την ιδιότητα $isBH$, παραβιάζοντας αναδρομικά την αναλλοίωτη χρώματος ενώ διατηρεί την αναλλοίωτη μαύρου ύψους. Καθώς επιστρέφει η αναδρομή διορθώνει την αναλλοίωτη χρώματος. Μόνο η ρίζα του δέντρου t που επιστρέφει η merge μπορεί να παραβιάζει την ιδιότητα χρώματος αλλά αυτό το αναλαμβάνει έπειτα η $delete$, μέσω της $makeBlackD$.

Πιο συγκεκριμένα, με αναφορά στον κώδικα του σχήματος 6.6 έχουμε τα εξής:

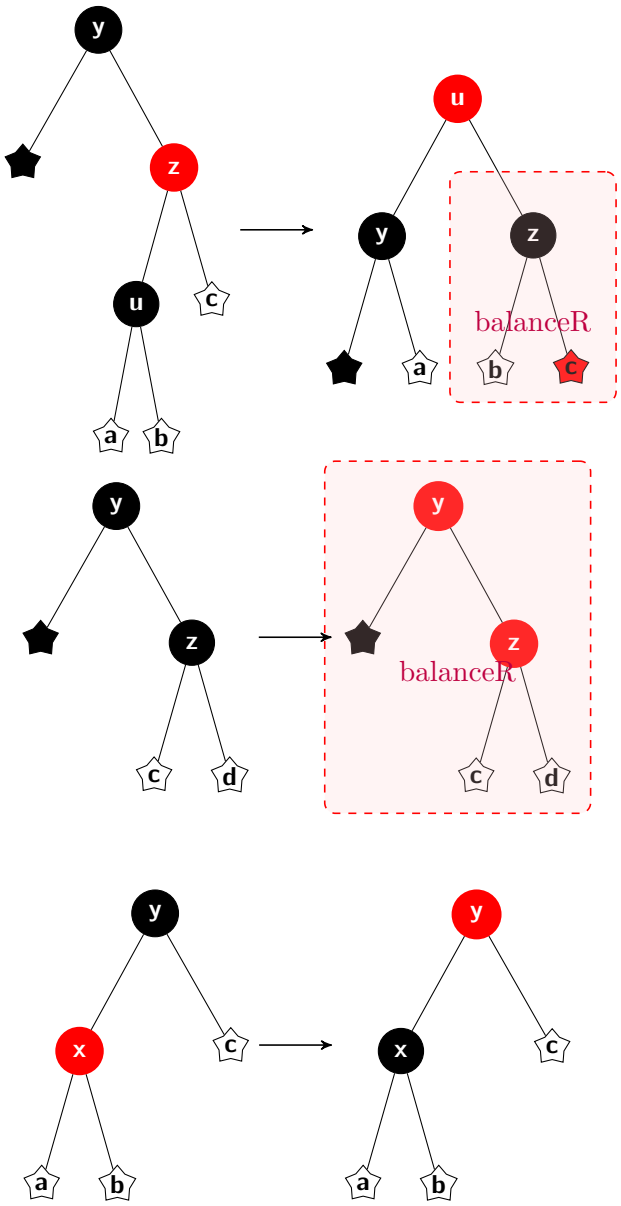
- Η ιδιότητα δυαδικού δέντρου αναζήτησης διατηρείται εφόσον τα l και r την ικανοποιούν και επίσης ισχύει ότι $root\ l < k < root\ r$.
- Η ιδιότητα χρώματος διατηρείται - εφόσον τα l και r την ικανοποιούν - αλλά πιθανώς να παραβιάζεται μονάχα στη ρίζα. Στις γραμμές 17,19 σίγουρα παραβιάζεται, στις γραμμές 24,26 σίγουρα είναι έγκυρη και στις γραμμές 29,32 γνωρίζουμε επίσης ότι είναι έγκυρη διότι τα a, b και b, c αντίστοιχα είναι σίγουρα μαύρα.
- Η ιδιότητα χρώματος διατηρείται σε όλο το δέντρο t αν τα l και r είναι μαύρα.
- Τα l και r έχουν το ίδιο μαύρο ύψος, έστω n.
- Το t έχει μαύρο ύψος n.
- Το κόστος της συγχώνευσης 2 red-black δέντρων είναι μικρότερο ή ίσο από το μικρότερο εξ αυτών σε ύψος δέντρο.

```
{-@ merge :: Ord k => k:k
          -> l : RBT {key:k | key < k} v
          -> r : RBTN {key:k | key > k} v {bh l}
          -> {ti : Tick { t : (ARBTN k v {bh l})
                              | IsB l && IsB r => isRB t}
                        | tcost ti <= min (height l) (height r) }
@-}
merge :: Ord k => k -> RBTree k v -> RBTree k v -> Tick (RBTree k v)
merge _ Nil x                                  =  pure x
merge _ x Nil                                  =  pure x
merge k (Node R x xv a b) (Node R y yv c d)
    =  pure (\m -> mergeR m) </> (merge k b c)
    where
        mergeR (Node R z zv b' c')
            = Node R z zv (Node R x xv a b') (Node R y yv c' d)
        mergeR bc
            = Node R x xv a (Node R y yv bc d)
merge k (Node B x xv a b) (Node B y yv c d)
    =  pure (\m -> mergeB m) </> (merge k b c)
    where
        mergeB (Node R z zv b' c')
            = Node R z zv (Node B x xv a b') (Node B y yv c' d)
        mergeB bc
            = balL x xv a (Node B y yv c d)
merge k a (Node R x xv b c)
    =  pure (\l' -> Node R x xv l' c) </> (merge k a b)
    -- IsB l && IsB r => isRB t
merge k (Node R x xv a b) c
    =  pure (\r' -> Node R x xv a r') </> (merge k b c)
    -- IsB l && IsB r => isRB t
```

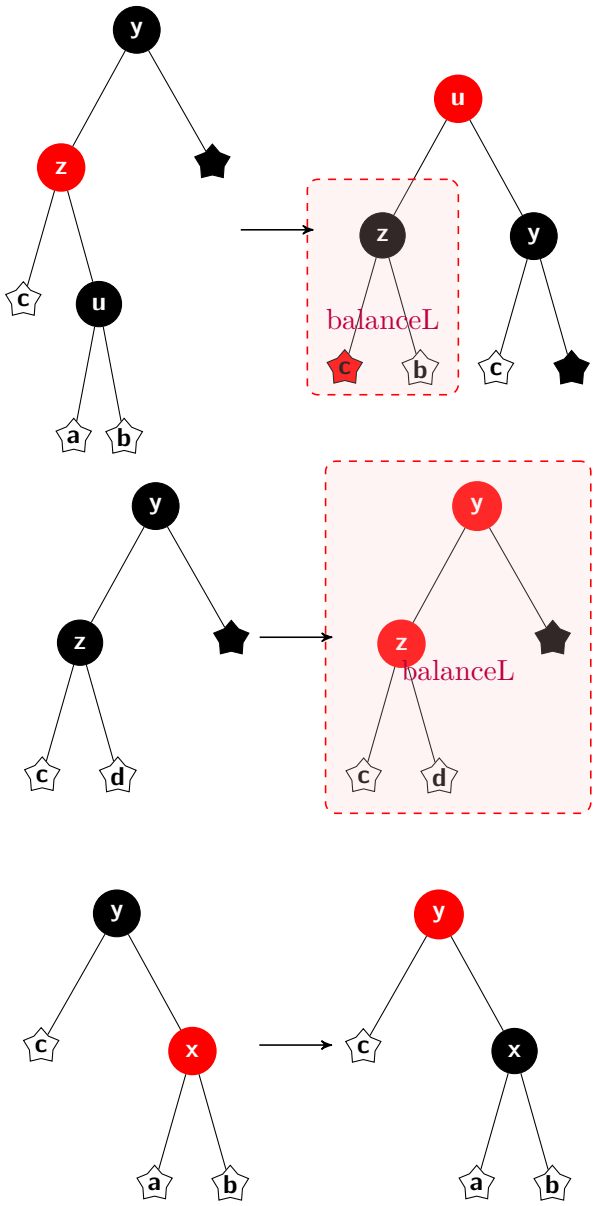
Σχήμα 6.6: Η συνάρτηση merge



Σχήμα 6.7: Εξάλειψη κόκκινων κόμβων με κόκκινα παιδιά με τις συναρτήσεις balanceL, balanceR



Σχήμα 6.8: Εξάλειψη κόκκινων κόμβων με κόκκινα παιδιά και διόρθωση μαύρου ύψους με τη συνάρτηση balL



Σχήμα 6.9: Εξάλειψη κόκκινων κόμβων με κόκκινα παιδιά και διόρθωση μαύρου ύψους με τη συνάρτηση balR

6.3 Απόδειξη Άνω Ορίου Ύψους Οποιουδήποτε Red-Black Δέντρου

Για να αποδείξουμε ότι το ύψος του δέντρου είναι $\mathcal{O}(\log n)$ πρέπει να αποδείξουμε το Λήμμα 6.1.2. Επίσης πρέπει να κωδικοποιήσουμε σε Liquid Haskell την ιδιότητα $rh\ t <= bh\ t$, που προκύπτει άμεσα από την αναλλοίωτη χρώματος(για κάθε κόκκινο κόμβο, αν έχει παιδιά, αυτά είναι οπωσδήποτε μαύρα), όπου $rh\ t$ το $measure$ που υπολογίζει το κόκκινο ύψος του δέντρου. Μέσω του εκλεπτυσμένου τύπου του $measureheight$ εξασφαλίζουμε ότι $height\ t <= bh\ t + rh\ t$. Άρα και $height\ t <= 2 * bh\ t$.

Ας ξεκινήσουμε από το δεύτερο. Προσθέτουμε την εξής υπόθεση

```
{-@ assume rh_bh :: t:BlackLLRBT k v-> { rh t <= bh t } @-}
rh_bh :: RBTree k v -> Proof
rh_bh _ = assumption
```

Επίσης στον τύπο του $measure\ height$ έχουμε προσθέσει τον εξής εκλεπτυσμένο τύπο.

```
{-@ height ::   t:RBTree k v
            -> {u:Nat | (isBH t => u <= rh t + bh t)
                     && (u >= bh t) }
@-}
```

Αυτά τα δύο εξασφαλίζουν ότι $height\ t <= 2 * bh\ t$

Τώρα όσον αφορά την απόδειξη του λήμματος 6.1.2, κατασκευάζουμε την εξής επαγωγική απόδειξη με μία συνάρτηση γραμμένη σε Haskell, χρησιμοποιώντας τους συνδυαστές απόδειξης.

Ο εκλεπτυσμένος τύπος της εξόδου της συνάρτησης-απόδειξης εκφράζει το λήμμα που αποδεικνύουμε.

```
{-@ lemma1 :: Ord k => t : RBT k v
           -> { (twoToPower (bh t)) <= size t + 1 }
    / [bh t]
@-}
```

Στο σώμα της συνάρτησης (Σχήμα 6.10) χρησιμοποιούμε την εκθετική συνάρτηση $twoToPower\ n$ η οποία είναι, όσον αφορά την τιμή της, ίση με την συνάρτηση 2^n της Haskell, αλλά την έχουμε εισάγει στην εκλεπτυσμένη λογική.

Με αυτά τα 2 είμαστε έτοιμες να αποδείξουμε το θεώρημα για το ύψος ένος red-black δέντρου. Η απόδειξη είναι σχετικά απλή, καθότι δεν είναι επαγωγική. Απλώς παραθέτουμε το λήμμα και καταλήγουμε στο ζητούμενο Σχήμα 6.11.

```
lemma1 :: Ord k => RBTree k v -> Proof
lemma1 t@Nil
    =.  twoToPower 0
    ==. 1
    ==. 0 + 1
    ==. size t + 1
    *** QED

lemma1 t@(Node R k v l r)
    =.  twoToPower (bh t)
    <=. 2*twoToPower (bh t)
    ==. 2*twoToPower (bh l)
    ==. twoToPower (bh l) + twoToPower (bh l)
    ==. twoToPower (bh l)  + twoToPower (bh r)
        ? lemma1 l
        ? lemma1 r
    <=. size l + 1 + size r + 1
    ==. size t + 1
    *** QED

lemma1 t@(Node B k v l r)
    =.  twoToPower (bh t)
    ==. 2*twoToPower (bh t - 1)
    ==. 2*twoToPower (bh l)
    ==. twoToPower (bh l) + twoToPower (bh l)
    ==. twoToPower (bh l)  + twoToPower (bh r)
        ? lemma1 l
        ? lemma1 r
    <=. size l + 1 + size r + 1
    ==. size t + 1
    *** QED
```

Σχήμα 6.10: Απόδειξη του λήμματος 6.1.2

```
{-@ height_costUB
    :: Ord k
    => t : BlackRBT k v
    -> { height t <= 2 * log (size t + 1) }
    / [height t]
@-}
height_costUB :: Ord k => RBTree k v -> Proof
height_costUB t
    =   height t
    <=. rh t + bh t
      ? toProof (rh_bh t)
    ==. 2 * bh t
      ? toProof (logTwotoPower (bh t))
    ==. 2 * log (twoToPower (bh t))
      ? lemma1 t
      ? toProof (logComp (twoToPower (bh t)) (size t + 1))
    <=. 2 * log (size t + 1)
    *** QED
```

Σχήμα 6.11: Απόδειξη του θεωρήματος 6.1.3

Με αυτό το θεώρημα μπορούμε να αποδείξουμε ότι το κόστος των συναρτήσεων get,set,delete είναι λογαριθμικό. Για συντομία παραθέτουμε την απόδειξη για το άνω όριο κόστους της συνάρτησης set μονάχα. Οι υπόλοιπες αποδείξεις είναι εντελώς αντίστοιχες με αυτήν.

```
{-@ set_costUB :: Ord k => k : k -> v:v -> t : BlackRBT k v
               -> { tcost (set k v t) <= 2 * log (size t + 1) }
    / [height t]
@-}
set_costUB :: Ord k => k -> v -> RBTree k v -> Proof
set_costUB k v t
    =   tcost (set k v t)
    <=. height t
      ? height_costUB t
    <=. 2 * log (size t + 1)
    *** QED
```

Σχήμα 6.12: Απόδειξη του άνω ορίου κόστους της συνάρτησης set

6.4 Left-Leaning Red-Black Trees

Ο R. Sedgewick, 3 δεκαετίες έπειτα από την δημοσίευση με τίτλο "A dichromatic framework for balanced trees" με τον L. J. Guibas [Guib78], που αποτελεί την βάση της δομής που ονομάζουμε σήμερα red-black tree, δημοσίευσε μία παραλλαγή αυτής της δομής, τη δομή left-leaning red-black tree [Sedg]. Η παραλλαγή αυτή, εγγυάται τον ίδιο ασυμπτωτικό χρόνο με την κλασσική και σχεδιάστηκε αρχικά με σκοπό να προσφέρει μία πιο εύκολη υλοποίηση. Σημειώνουμε ότι, σχεδιάστηκε για μία imperative γλώσσα. Σε μία συναρτησιακή γλώσσα η υλοποίηση του Okasaki είναι δύσκολο να απλοποιηθεί σημαντικά.

Ένα left-leaning red-black δέντρο, είναι ένα δυαδικό δέντρο αναζήτησης που ικανοποιεί τις εξής ιδιότητες.

- **Αριστερή κλίση:** Οποιοσδήποτε κόκκινος κόμβος πρέπει να είναι αριστερό παιδί ενός κόμβου και όχι δεξί.
- **Χρώμα:** Οποιοσδήποτε κόκκινος κόμβος έχει δύο μαύρα παιδιά, όπου οι κόμβοι *Nil* θεωρούμε ότι είναι μαύροι εξ ορισμού.
- **Μαύρο Ύψος:** Όλα τα μονοπάτια από την ρίζα του δέντρου - η οποία θεωρούμε ότι είναι μαύρη - σε οποιονδήποτε κόμβο Nil, περιέχουν τον ίδιο αριθμό μαύρων κόμβων.

Ο ορισμός, τα θεωρήματα και τα λήμματα που περιγράψαμε στην ενότητα 6.1 ισχύουν και εδώ.

6.5 Υλοποίηση της Δομής Left-Leaning Red-Black Δέντρου σε LH

Μπορούμε να υλοποιήσουμε τη δομή left-leaning red-black δέντρου, πολύ εύκολα, απλώς προσθέτοντας το *measure* $isLeftLean$ για να εξασφαλίσουμε την ιδιότητα αριστερής κλίσης.

```
{-@ measure isLeftLean @-}
{-@ isLeftLean :: RBTree k v -> Bool @-}
isLeftLean Nil              = True
isLeftLean (Node c _ _ l r) = isLeftLean l && isLeftLean r
                           && (col r == B)
```

Με βάση αυτό το measure και τους τύπους που ορίσαμε στην ενότητα 6.2 ορίζουμε τους τύπους που θα χρειαστούμε για την επαλήθευση των συναρτήσεων αναζήτησης

και εισαγωγής και της απόδειξης άνω ορίου ύψους. Οι παρακάτω τύποι είναι ίδιοι με εκείνους που ορίσαμε στην ενότητα 6.2, μόνο που εκλεπτύνονται επιπλέον με την ιδιότητα αριστερής κλίσης.

```
{-@ type LLRBT k v       = {t: RBT k v       | isLeftLean t}   @-}
{-@ type LLRBTN k v N    = {t: RBTN k v N    | isLeftLean t}   @-}
{-@ type LLARBT k v      = {t: ARBT k v      | isLeftLean t}   @-}
{-@ type LLARBTN k v N   = {t: ARBTN k v N   | isLeftLean t }  @-}
{-@ type BlackLLRBT k v = {t: BlackRBT k v | isLeftLean t }  @-}
```

Η συνάρτηση get' για την αναζήτηση ενός κλειδιού σε ένα left-leaning red-black δέντρο είναι προφανώς εντελώς όμοια με τη συνάρτηση get για την αναζήτηση σε ένα απλό red-black δέντρο. Αυτή που έχει ενδιαφέρον είναι η συνάρτηση set' διότι πραγματοποιούμε κάποιες διαφορετικές περιστροφές για την εξισορρόπηση του δέντρου. Ας δούμε τη συνάρτηση set' λοιπόν.

6.5.1 Εισαγωγή με Εξισορρόπηση

Για τη συνάρτηση set' γνωρίζουμε ότι:

- Δέχεται ένα left-leaning red-black δέντρο.
- Επιστρέφει ένα left-leaning red-black δέντρο με μαύρη ρίζα.
- Το κόστος εισαγωγής ενός στοιχείου στο δέντρο είναι μικρότερο ή ίσο από το ύψος του.
- Τον εκλεπτυσμένο τύπο της εξόδου εγγυώνται οι συναρτήσεις $makeBlack$ και $insert'$.

Το σώμα της συνάρτησης $makeBlack'$ είναι ίδιο με αυτό της συνάρτησης makeBlack, ενώ ο τύπος της είναι ο εξής:

```
{-@ makeBlack' :: {t : LLARBT k v | size t > 0} -> BlackLLRBT k v @-}
```

Ο εκλεπτυσμένος τύπος της συνάρτηση $insert'$ μας εξασφαλίζει ότι:

- Η συνάρτηση $insert$ δέχεται ένα κλειδί τύπου k, μία τιμή τύπου v και left-leaning red-black δέντρο, έστω t αυτό.
- Επιστρέφει ένα μη κενό, left-leaning almost red-black δέντρο, έστω t' αυτό, εντός του Tick Monad.
- Το μαύρο ύψος του t' είναι ίσο με το μαύρο ύψος του t.

- Αν η ρίζα του t είναι μαύρη τότε, το t' είναι έγκυρο red-black, δηλαδή δεν παραβιάζει την ιδιότητα χρώματος στη ρίζα.

- Το άνω όριο κόστους της εισαγωγής του ζεύγους k, v στο t είναι το πολύ ίσο με το ύψος του t.

```
{-@ reflect set' @-}
{-@ set' ::  (Ord k) => k -> v
            -> t : LLRBT k v
            -> {t' : Tick (BlackLLRBT k v) | tcost t' <= height t}

@-}
set' k v s = fmap makeBlack' (insert' k v s)

{-@ reflect insert' @-}
{-@ insert' ::   (Ord k) => k -> v
                -> t : LLRBT k v
                -> {ti : Tick {t' : (LLARBTN k v {bh t})
                                  | (IsB t => isRB t')
                                  && size t' > 0}
                      | tcost ti <= height t}
@-}
insert' :: Ord k => k -> v -> RBTree k v -> Tick (RBTree k v)
insert' k v Nil = pure (Node R k v Nil Nil)
insert' k v (Node B key val l r)
    | k < key   = pure (\l' -> balanceL' key val l' r) </> (insert' k v l)
    | k > key   = pure (\r' -> balanceR' B key val l r') </> (insert' k v r)
    | otherwise = wait (Node B key v l r)
insert' k v (Node R key val l r)
    | k < key   = pure (\l' -> Node R key val l' r) </> (insert' k v l)
    | k > key   = pure (\r' -> balanceR' R key val l r') </> (insert' k v r)
    | otherwise = wait (Node R key v l r)
```

Σχήμα 6.13: Η συνάρτηση set' εντός του Tick Monad για ένα left-leaning red-black δέντρο

6.5.1.1 Εξισορρόπηση

Όμοια με την υποενότητα 6.2.2.2, χωρίζουμε την διαδικασία εξισορρόπησης του δέντρου μετά από την εισαγωγή ενός στοιχείου, σε αριστερή και δεξιά εξισορρόπηση για να μην κάνουμε περιττούς ελέγχους. Παρατηρούμε βάσει του σχήματος 6.16 ότι οι περιπτώσεις δέντρων που χρήζουν εξισορρόπησης - αν συγχωνεύσουμε τις 2 τελευταίες - είναι 3 και όχι 4, αλλά εδώ δεν αφορούν μόνο τα δέντρα με διαδοχή κόμβων, ξεκινώντας από τη ρίζα, χρώματος μαύρου-κόκκινου-κόκκινου. Επίσης δεν αναγάγονται και οι 3 περιπτώσεις στο ίδιο δέντρο.

Οι συναρτήσεις που υλοποιούν τη διαδικασία που περιγράφεται στο Σχήμα 6.16 είναι οι $balanceL'$ και $balanceR'$.

```
{-@ reflect balanceL' @-}
{-@ balanceL' :: k:k -> v
                -> {l : LLARBT {key:k | key < k} v |  size l >0 }
                -> {r : LLRBTN {key:k | k < key} v {bh l} | IsB r}
                -> {t : (LLRBTN k v {bh l+1}) | size t > 0}
@-}
balanceL' z zv (Node R y yv (Node R x xv a b) c) d
    = Node R y yv (Node B x xv a b) (Node B z zv c d)
balanceL' x xv a b
    = Node B x xv a b
```

Σχήμα 6.14: Η συνάρτηση balanceL' για ένα left-leaning red-black δέντρο

Με τον εκλεπτυσμένο τύπο της balanceL' επαληθεύουμε ότι:

- Η balanceL' δέχεται ένα κλειδί k τύπου k, μία τιμή τύπου v, ένα αριστερό υπόδεντρο l κι ένα δεξί υπόδεντρο r.
- Το l είναι almost red-black, μη κενό, σε σχέση διάταξης με το k.
- Το r είναι έγκυρο red-black με μαύρη ρίζα, μη κενό, σε σχέση διάταξης με το k.
- Το r έχει το ίδιο μαύρο ύψος με το l, έστω n.
- Η balanceL' επιστρέφει ένα έγκυρο μη κενό red-black δέντρο, με μαύρο ύψος $n+1$.

Με τον εκλεπτυσμένο τύπο της balanceR' επαληθεύουμε ότι:

- Η balanceR' δέχεται ένα χρώμα c τύπου Color, ένα κλειδί k τύπου k, μία τιμή τύπου v, ένα αριστερό υπόδεντρο l κι ένα δεξί υπόδεντρο r.
- Το l είναι red-black, σε σχέση διάταξης με το k. Δηλαδή, $root\ l < k$.

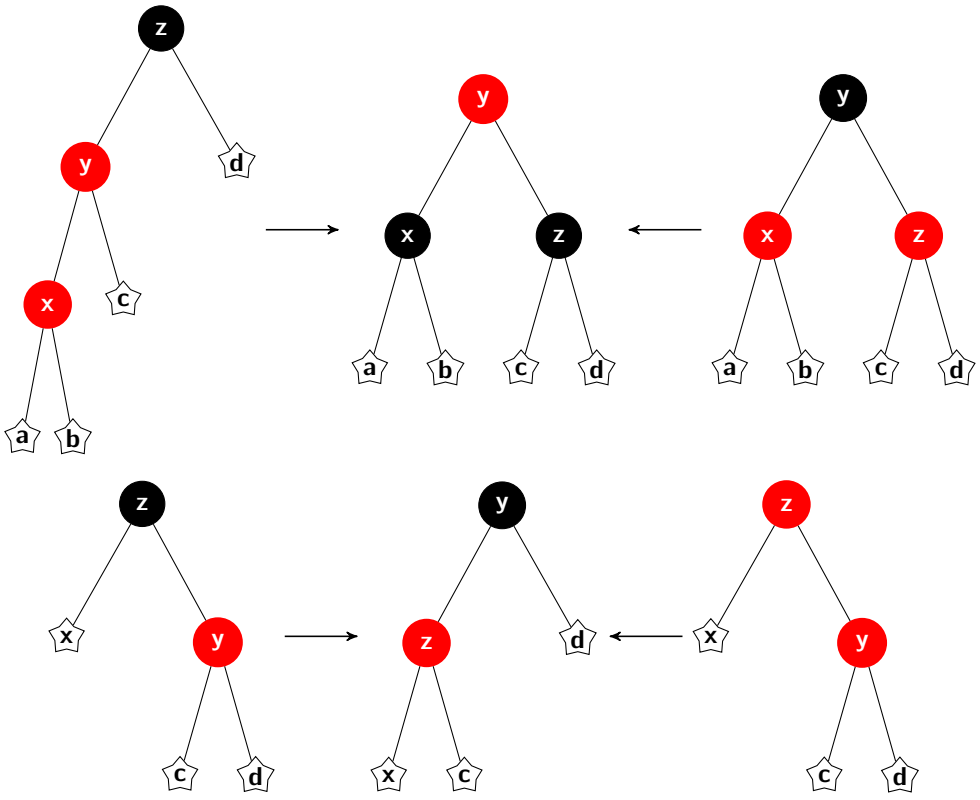
```
{-@ reflect balanceR' @-}
{-@ balanceR' :: c:Color -> k:k -> v
                -> {l : LLRBT {key:k | key < k} v | c == R =>  IsB l }
                -> {r : LLRBTN {key:k | k < key} v {bh l}
                      |  (c == R => isRB r) && size r > 0 }
                -> {t : (LLARBT k v )
                                  | (if (c==B ) then (bh t = bh l + 1)
                                     else (bh t = bh l))
                                   && ((c == B) => isRB t)
                                   && size t > 0}
@-}
balanceR' B y yv (Node R x xv a b) (Node R z zv c d)
    = Node R y yv (Node B x xv a b) (Node B z zv c d)
balanceR' col y yv x (Node R z zv c d)
    = Node col z zv (Node R y yv x c) d
balanceR' col x xv a b
    = Node col x xv a b
```

Σχήμα 6.15: Η συνάρτηση balanceR' για ένα left-leaning red-black δέντρο

- Αν το c είναι κόκκινο τότε το l έχει μαύρη ρίζα.
- Το r είναι almost red-black , μη κενό, σε σχέση διάταξης με το k. Δηλαδή, $root\ r > k$.
- Αν το c είναι κόκκινο τότε το r είναι έγκυρο red-black.
- Το r έχει το ίδιο μαύρο ύψος με το l, έστω n.
- Η balanceR' επιστρέφει ένα έγκυρο μη κενό red-black δέντρο, με μαύρο ύψος $n+1$, αν το c είναι μαύρο.
- Η balanceR' επιστρέφει ένα μη κενό almost red-black δέντρο, με μαύρο ύψος n, αν το c είναι κόκκινο.

Παρατηρούμε ότι και στις δύο συναρτήσεις και ιδιαίτερα στην $balanceR'$, χρειαζόμαστε περισσότερες εκλεπτύνσεις στα δεδομένα εισόδου για να αποδείξουμε τα κατηγορήματα στον τύπο της εξόδου. Αυτό γιατί, η Liquid Haskell προφανώς δεν μπορεί να αποδείξει αυτόματα την μορφή των παραμέτρων εισόδου, συσχετίζοντας τες με την κλήση της $insert$. Πρέπει να περιορίσουμε εμείς το σύνολο των πιθανών εισόδων, σκεπτόμενες ότι η $insert$ καλεί τη $balanceR$ με συγκεκριμένα σύνολα παραμέτρων. Για παράδειγμα,

γνωρίζουμε ότι, στη γραμμή 15, τα δεδομένα col, y, yv, x έχουν προκύψει από pattern matching στο δέντρο (Node col y yv x r) που είναι έγκυρο red-black. Όπότε αν το col έχει την τιμή "R" τότε η ρίζα του x είναι οπωσδήποτε μαύρη.



Σχήμα 6.16: Εξάλειψη κόκκινων κόμβων με κόκκινα παιδιά, με τις συναρτήσεις balanceL', balanceR', σε ένα left-leaning red-black δέντρο

6.6 Απόδειξη Άνω Ορίου Ύψους Οποιουδήποτε Left-Leaning Red-Black Δέντρου

Για να αποδείξουμε ότι το ύψος ενός left-leaning red-black δέντρου t είναι $\mathcal{O}(\log n)$, όμοια με την ενότητα 6.3 πρέπει να αποδείξουμε το Λήμμα 6.1.2. Επίσης πρέπει να κωδικοποιήσουμε σε Liquid Haskell την ιδιότητα $rh\ t <= bh\ t$ για αυτήν την ειδική περίπτωση των red-black δέντρων - και εδώ αυτή η ιδιότητα προκύπτει άμεσα από την αναλλοίωτη χρώματος, δηλαδή την ιδιότητα $isRB\ t$ (για κάθε κόκκινο κόμβο, αν έχει παιδιά, αυτά είναι οπωσδήποτε μαύρα), όπου $rh\ t$ το $measure$ που υπολογίζει το κόκκινο ύψος του δέντρου.

Όσον αφορά το πρώτο, κωδικοποιούμε την ιδιότητα $rh\ t <= bh\ t$, μέσω της εξής υπόθεσης:

```
{-@ assume rh_bh :: t:BlackLLRBT k v-> { rh t <= bh t } @-}
rh_bh :: RBTree k v -> Proof
rh_bh _ = assumption
```

Το σώμα της συνάρτησης που αποδεικνύει το λήμμα είναι ακριβώς ίδιο με το σώμα της συνάρτησης του σχήματος 6.10.

Ο εκλεπτυσμένος τύπος της συνάρτησης εξασφαλίζει ότι το λήμμα ισχύει για οποιοδήποτε left-leaning red-black δέντρο.

```
{-@ lemma1 :: Ord k => t:LLRBT k v
           -> { (twoToPower (bh t)) <= size t + 1 }
    / [bh t]
@-}
```

Αντίστοιχα, το σώμα της συνάρτησης που αποδεικνύει το θεώρημα άνω ορίου ύψους για οποιοδήποτε left-leaning red-black δέντρο, είναι ίδιο με το σώμα της συνάρτησης του σχήματος 6.11, ενώ ο εκλεπτυσμένος τύπος της είναι ο εξής:

```
{-@ height_costUB :: Ord k => t : BlackLLRBT k v
                  -> { height t <= 2 * log (size t + 1) }
    / [height t]
@-}
```

Με το θεώρημα $height\_costUB\ t$ μπορούμε να αποδείξουμε ότι το κόστος των συναρτήσεων get', set' είναι λογαριθμικό, ορίζοντας τις εξής συναρτήσεις get'_costUB και set'_costUB με τύπους:

```
{-@ get'_costUB :: Ord k => k : k -> t : BlackLLRBT k v
                -> { tcost (get' k t) <= 2 * log (size t + 1) }
    / [height t]
@-}

{-@ set'_costUB :: Ord k => k : k -> v : v -> t : BlackLLRBT k v
                -> { tcost (set' k v t) <= 2 * log (size t + 1) }
    / [height t]
@-}
```

6.6.1 Μέσο Όριο Ύψους

Όπως και στην περίπτωση των τυχαίων κατασκευασμένων δυαδικών δέντρων αναζήτησης, έτσι και εδώ, η εκτίμηση με βάση το *άνω όριο* ύψους, δηλαδή η εκτίμηση με βάση τη χειρότερη περίπτωση, είναι κάπως απαισιόδοξη. Πειράματα σε αρχικά κενά δέντρα, όπου τα κλειδιά εισάγονται με τυχαία σειρά, έχουν δείξει ότι το αναμενόμενο ύψος ενός left - leaning red - black δέντρου με n εσωτερικούς κόμβους δεν προσεγγίζει την εκτίμηση με βάση τη χειρότερη περίπτωση, δηλαδή την τιμή $2 \cdot log(n+1)$, αλλά την τιμή $\sim 1.00 \cdot logn$ [Sedg15]. Με βάση αυτό, η αναζήτηση σε ένα τυχαία κατασκευασμένο left - leaning red - black δέντρο με n κλειδιά απαιτεί $\sim 1.00 \cdot logn$ συγκρίσεις. Η ακριβής σταθερά δεν είναι γνωστή.

6.7 Βοηθητικές Αναλλοίωτες

Μέσω διαφόρων δοκιμών, παρατηρήσαμε πως, για να επαληθεύσουμε με τη Liquid Haskell τους τύπους των συναρτήσεων είναι απαραίτητο να ενισχύσουμε την εκλεπτυσμένη λογική. Γι' αυτό το σκοπό, προσθέτουμε βοηθητικές τοπικές αναλλοίωτες για τους τύπους οι οποίες ελέγχονται για κάθε εμφάνιση κατασκευαστή τύπου στο πρόγραμμα.

Για παράδειγμα, η παρακάτω αναλλοίωτη επαληθεύεται για κάθε εμφάνιση του κατα-

```
{-@ using (RBTree k v) as {t: RBTree k v | isRB t => isARB t}  @-}
```

σκευαστή Nil και του κατασκευαστη $Node$ στο πρόγραμμα.

Στα προγράμματα που περιέχεται ο κώδικας αυτού του κεφαλαίου, έχουμε προσθέσει τις εξής αναλλοίωτες:

```
{-@ using (Color) as {v: Color | v = R || v = B}      @-}
{-@ using (RBTree k v) as {t: RBTree k v | Invs t }  @-}
```

Όπου $Invs$ είναι ένα κατηγόρημα και ορίζεται ως εξής:

```
{-@ predicate Invs T = Inva T && Invb T && Invc T   @-}

{-@ predicate Inva T = isRB T => isARB T            @-}
{-@ predicate Invb T = (isARB T && IsB T) => isRB T @-}
{-@ predicate Invc V = IsBlackRBT V => rh V <= bh V @-}
```

και το κατηγόρημα $IsBlackRBT$ ορίζεται ως εξής:

```
{-@ predicate IsBlackRBT T = (isRB T && isBH T
                              && IsB T  && bh T > 0)
@-}
```

Κεφάλαιο 7

Συμπεράσματα, Σκέψεις και Μελλοντική Έρευνα

Σε αυτή τη διπλωματική εργασία, διερευνήσαμε τη χρηστικότητα καθώς και τη χρησιμότητα της Liquid Haskell σε διαφορετικά πλαίσια. Σε αυτό το κεφάλαιο, συνοψίζουμε και αξιολογούμε τα αποτελέσματα που έχουμε από τις μελέτες περίπτωσης. Συμπεριλαμβάνονται επίσης και οι δικές μας σκέψεις. Σκέψεις που προέκυψαν τόσο κατά τη διαδικασία ανάπτυξης του κώδικα όσο και κατά τη συγγραφή της παρούσας εργασίας. Στο τέλος αυτού του κεφαλαίου παρουσιάζουμε επίσης μερικές ιδέες για μελλοντική εργασία.

7.1 Συμπεράσματα

Στόχος μας ήταν να εξερευνήσουμε τις δυνατότητες της Liquid Haskell και να συνεισφέρουμε στη βιβλιοθήκη της. Αυτός ο στόχος πρέπει να θεωρηθεί επιτυχημένος, γιατί καταφέραμε να χρησιμοποιήσουμε επιτυχώς τη Liquid Haskell σε όλες τις μελέτες περίπτωσης.

Όσον αφορά τη χρηστικότητα της Liquid Haskell, παρατηρήσαμε ότι ο συνδυασμός των προδιαγραφών της LH - ιδιαιτέρως των measures και των reflected συναρτήσεων - με τον SMT-solver απλοποίησαν τρομερά την στατική ανάλυση του κώδικα, δηλαδή την ανάλυση του κατά τη διαδικασία μεταγλώττισης. Παρατηρήσαμε επίσης, πως οι τύποι δεδομένων που χρησιμοποιήσαμε ήταν ταυτόχρονα αρκετά εκφραστικοί για να επαληθεύσουμε την ορθότητα της υλοποίησής μας αλλά και αρκετά απλοί για να τους δοκιμάσουμε και να τους χρησιμοποιήσουμε καταλλήλως. Όταν επαληθεύσαμε τη λογαριθμική πολυπλοκότητα του ύψους οποιουδήποτε red-black δέντρου, παρατηρήσαμε πως ήταν κάθε άλλο παρά δύσκολο να γράψουμε επαγωγικές αποδείξεις και αποδείξεις εξισωτικής λογικής σε LH.

Από την άλλη, όσο εύκολη είναι η προσθήκη και η επαλήθευση των ιδιοτήτων πολυπλοκότητας και ορθότητας μέσω των προδιαγραφών της LH, τόσο δύσκολη είναι η

διαδικασία εντοπισμού των σφαλμάτων σε αυτές. Για παράδειγμα, αν ξεχάσουμε μία περίπτωση σε μία απόδειξη, δεν θα πάρουμε το σφάλμα που παίρνουμε όταν γίνεται κάτι αντίστοιχο στη Haskell, δηλαδή, το εξής μήνυμα: *Pattern match(es) are non-exhaustive*, αλλά θα πάρουμε το εξής: *Inferred type VV : [..] is not a subtype of Required type VV : VV : GHC.Prim.Addr | 5 < 4*, το οποίο αν δεν το έχουμε ξανασυναντήσει και επιλύσει δεν θα καταλάβουμε περί τίνος πρόκειται. Κρίνουμε απαραίτητη την επέκταση της Liquid Haskell ώστε να παρέχει πιο ακριβή μηνύματα σε σφάλματα. Ένα άλλο λιγότερο μείζων πρόβλημα είναι ότι δεν υπάρχει κάποιο ολοκληρωμένο documentation, φιλικό προς την προγραμματίστρια, για τη συγγραφή κώδικα σε Liquid Haskell, με αποτέλεσμα πολλές φορές να χρειάζεται να κάνουμε αρκετές δοκιμές έχοντας στο μυαλό μας το συντακτικό της Haskell παρόλο που τα δύο συντακτικά δεν ταυτίζονται πλήρως.

Σε αυτήν την διπλωματική εργασία, δείξαμε πως η LH είναι όντως ένα χρήσιμο εργαλείο τυπικής επαλήθευσης. Το αν η LH είναι το κατάλληλο εργαλείο για την επαλήθευση του εκάστοτε συστήματος ή της εκάστοτε βιβλιοθήκης, εξαρτάται από διάφορους παράγοντες. Πιστεύουμε έντονα ότι, εάν μπορούμε να εκφράσουμε τις ιδιότητες που θέλουμε μέσω measures και απλών αφηρημένων εκλεπτυσμένων τύπων, τότε η επαλήθευση του συστήματος θα είναι πιθανώς μία εύκολη διαδικασία. Αυτό γιατί, η LH μπορεί να εκμεταλλευτεί όλες τις δυνατότητες του SMT-solver και να αυτοματοποιήσει την απόδειξη, όταν χρησιμοποιεί κατηγορήματα που ανήκουν στην SMT-αποφασίσιμη λογική. Τα measures και οι προτάσεις με αφηρημένες εκλεπτύνσεις ανήκουν σε αυτήν την κατηγορία.

7.2 Κάποιες Σκέψεις

Η συγγραφή κώδικα σε Liquid Haskell ήταν, τις περισσότερες φορές, μια ενδιαφέρουσα και εκπαιδευτική διαδικασία. Σε αυτό σίγουρα βοήθησε και η πρότερη εξοικείωση με συναρτησιακές γλώσσες προγραμματισμού, όπως η ML και η Haskell. Όταν ξεκίνησα να πειραματίζομαι με την LH, μου έκανε εξαιρετική εντύπωση πως μπορούσα να έχω έναν σωστό κώδικα σε πολύ λίγο χρόνο. Επίσης οι προγραμματίστριες της LH είναι πολύ βοηθητικές στο Github και στο αντίστοιχο κανάλι στο Slack.

Παρόλα αυτά υπήρξαν κάποια σημεία που μας ταλαιπώρησαν αρκετά. Σε περίπτωση σφάλματος, τα μηνύματα δεν είναι πολύ βοηθητικά. Όταν έχουμε αμοιβαία αναδρομικές συναρτήσεις, όπου δεν μπορούμε να ξεχωρίσουμε τα κομμάτια κώδικα, η διαδικασία διόρθωσης σφαλμάτων είναι πολύ δύσκολη. Το ίδιο ισχύει όταν προσθέτουμε πιο περίπλοκα αφηρημένα κατηγορήματα στους τύπους. Επίσης, οι πολλοί περιορισμοί στους τύπους των συναρτήσεων που πρέπει να ελεγχθούν από τη LH, κοστίζουν σε χρόνο εκτέλεσης. Τα παραπάνω βέβαια, μπορούν να βελτιωθούν. Ήδη, παρακολουθώντας τη συζήτηση στο Github, βλέπω πως οι προγραμματίστριες τα έχουν υπόψη τους και προσπαθούν να τα

διορθώσουν.

Σκέφτομαι έντονα, πως ένα τέτοιο εργαλείο θα μπορούσε να ενταχθεί στην εκπαιδευτική διαδικασία ως μια πρώτη εξοικείωση των σπουδαστριών με το πεδίο της επαλήθευσης των προγραμμάτων και να συναγωνιστεί άλλα πιο καθιερωμένα, όπως το QuickCheck και το Frama-C. Ήδη, βλέπω πως έχει ενταχθεί στο πρόγραμμα διαλέξεων κάποιων μαθημάτων [1].

7.3 Μελλοντική Έρευνα

Σε αυτήν την διπλωματική εργασία καταφέραμε να αποδείξουμε το κόστος των συναρτήσεων get, set, delete που υλοποιήσαμε για τα δυαδικά δέντρα αναζήτησης και για τα red-black δέντρα. Για τα left-leaning red-black δέντρα, υλοποιήσαμε τις συναρτήσεις get,set και αποδείξαμε το κόστος τους, ενώ δεν υλοποιήσαμε την συνάρτηση delete. Θα ήταν φυσικό να ολοκληρώσουμε την βιβλιοθήκη μας με την υλοποίηση της delete για τα left-leaning red-black δέντρα και επιπλέον με την υλοποίηση των μεθόδων toList, fromList, union, intersection, difference, join, merge, split, minimum, maximum για όλες τις δομές δυαδικών δέντρων που περιγράψαμε στα προηγούμενα κεφάλαια.

Επίσης θα είχε ενδιαφέρον να εξερευνήσουμε τις δυνατότητες και την χρηστικότητα της Liquid Haskell όσον αφορά πιο περίπλοκες και εκτενείς αποδείξεις από αυτές που κατασκευάσαμε σε αυτήν την διπλωματική εργασία, όπως η απόδειξη του θεωρήματος 5.3.1.

[1] παραδείγματος χάρη εδώ: `https://www.seas.upenn.edu/~cis552/20fa/lectures/stub/lh-demo/LiquidHaskell.html`

Βιβλιογραφία

[Agui19] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg and Pierre-Yves Strub, "A Relational Logic for Higher-Order Programs", *Journal of Functional Programming*, vol. 29, p. e16, 2019. arXiv: 1703.05042.

[Alme11] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto and Simão Melo de Sousa, *Rigorous Software Development: An Introduction to Program Verification*, Undergraduate Topics in Computer Science, Springer-Verlag, London, 2011.

[Appe77] K. Appel and W. Haken, "Every planar map is four colorable. Part I: Discharging", *Illinois Journal of Mathematics*, vol. 21, no. 3, pp. 429–490, September 1977. Publisher: Duke University Press.

[Aspi07] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl and Alberto Momigliano, "A program logic for resources", *Theoretical Computer Science*, vol. 389, no. 3, pp. 411–445, December 2007.

[Atke11] Robert Atkey, "Amortised Resource Analysis with Separation Logic", *Logical Methods in Computer Science*, vol. 7, no. 2, p. 17, June 2011. arXiv: 1104.1998.

[Avan15] Martin Avanzini, Ugo Dal Lago and Georg Moser, "Analysing the complexity of functional programs: higher-order meets first-order", in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pp. 152–164, New York, NY, USA, August 2015, Association for Computing Machinery.

[Avan17] Martin Avanzini and Ugo Dal Lago, "Automated Sized-Type Inference and Complexity Analysis", *Electronic Proceedings in Theoretical Computer Science*, vol. 248, pp. 7–16, April 2017. arXiv: 1704.05585.

[Bert04] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program*

development. Coq'Art: The Calculus of inductive constructions., January 2004.

[Bonf11] Guillaume Bonfante, Jean-Yves Marion and Jean-Yves Moyen, "Quasi-interpretations a way to control resources", *Theoretical Computer Science*, vol. 412, pp. 2776–2796, June 2011.

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

[Dall13] Ugo Dal lago and Barbara Petit, "The geometry of types", in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pp. 167–178, New York, NY, USA, January 2013, Association for Computing Machinery.

[Dani08] Nils Anders Danielsson, "Lightweight semiformal time complexity analysis for purely functional data structures", in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pp. 133–144, New York, NY, USA, January 2008, Association for Computing Machinery.

[Free91] Timothy S. Freeman and F. Pfenning, "Refinement types for ML", in *PLDI '91*, 1991.

[Gont07] Georges Gonthier, "The Four Colour Theorem: Engineering of a Formal Proof", p. 333, January 2007.

[Grob01] Bernd Grobauer, "Cost recurrences for DML programs", in *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pp. 253–264, New York, NY, USA, October 2001, Association for Computing Machinery.

[Guib78] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees", in *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pp. 8–21, October 1978. ISSN: 0272-5428.

[Halm85] P. R. Halmos, *I Want to be a Mathematician: An Automathography*, Springer-Verlag, New York, 1985.

[Hand19] Martin A. T. Handley, Niki Vazou and Graham Hutton, "Liquidate your assets: reasoning about resource usage in liquid Haskell", *Proceedings of*

the ACM on Programming Languages, vol. 4, no. POPL, pp. 24:1–24:27, December 2019.

[Hoff12a] Jan Hoffmann, Klaus Aehlig and Martin Hofmann, "Multivariate amortized resource analysis", *ACM Transactions on Programming Languages and Systems*, vol. 34, no. 3, pp. 14:1–14:62, November 2012.

[Hoff12b] Jan Hoffmann, Klaus Aehlig and Martin Hofmann, "Resource aware ML", in *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pp. 781–786, Berlin, Heidelberg, July 2012, Springer-Verlag.

[Hoff15] Jan Hoffmann and Zhong Shao, "Automatic Static Cost Analysis for Parallel Programs", in Jan Vitek, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pp. 132–157, Berlin, Heidelberg, 2015, Springer.

[Hofm03a] Martin Hofmann and Steffen Jost, "Static prediction of heap space usage for first-order functional programs", *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 185–197, January 2003.

[Hofm03b] Martin Hofmann and Steffen Jost, "Static prediction of heap space usage for first-order functional programs", in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pp. 185–197, New York, NY, USA, January 2003, Association for Computing Machinery.

[Hold10] Stefan Holdermans and Jurriaan Hage, "Polyvariant Flow Analysis with Higher-ranked Polymorphic Types and Higher-order Effect Operators", vol. 45, September 2010.

[Hugh96] John Hughes, Lars Pareto and Amr Sabry, "Proving the correctness of reactive systems using sized types", in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pp. 410–423, New York, NY, USA, January 1996, Association for Computing Machinery.

[Jost10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl and Martin Hofmann, "Static determination of quantitative resource usage for higher-order programs", in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pp. 223–

236, New York, NY, USA, January 2010, Association for Computing Machinery.

[Jost17] Steffen Jost, Pedro Vasconcelos, Mário Florido and Kevin Hammond, "Type-Based Cost Analysis for Lazy Functional Languages", *Journal of Automated Reasoning*, vol. 59, no. 1, pp. 87–120, June 2017.

[Kahr01] Stefan Kahrs, "Red-black trees with types", *Journal of Functional Programming*, vol. 11, no. 4, pp. 425–432, July 2001.

[Lago12] Ugo Dal Lago and Marco Gaboardi, "Linear Dependent Types and Relative Completeness", *Logical Methods in Computer Science*, vol. 8, no. 4, p. 11, October 2012. arXiv: 1104.0193.

[Lein10] K. Rustan M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness", in Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pp. 348–370, Berlin, Heidelberg, 2010, Springer.

[McCa17] Jay McCarthy, Burke Fetscher, Max New, Daniel Feltey and Robert Findler, "A Coq library for internal verification of running-times", *Science of Computer Programming*, vol. 164, May 2017.

[Mead11] Catherine Meadows, "Program Verification and Security", in Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pp. 981–983, Springer US, Boston, MA, 2011.

[Mogg91] Eugenio Moggi, "Notions of computation and monads", *Information and Computation*, vol. 93, no. 1, pp. 55–92, July 1991.

[Mora99] Andrew Moran and David Sands, "Improvement in a lazy context: an operational theory for call-by-need", in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pp. 43–56, New York, NY, USA, January 1999, Association for Computing Machinery.

[Okas97] Chris Okasaki, "Three algorithms on Braun trees", *Journal of Functional Programming*, vol. 7, no. 6, pp. 661–666, November 1997. Publisher: Cambridge University Press.

[Okas99] Chris Okasaki, "Red-black trees in a functional setting", *Journal of Functional Programming*, vol. 9, no. 4, pp. 471–477, July 1999.

[Radi17] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg and Florian Zuleger, "Monadic refinements for relational cost analysis", *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 36:1–36:32, December 2017.

[Reis94] Brian Reistad and David K. Gifford, "Static dependent costs for estimating execution time", in *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pp. 65–78, New York, NY, USA, July 1994, Association for Computing Machinery.

[Robe96] Neil Robertson, Daniel Sanders, Paul Seymour and Communicated Graham, "A New Proof Of The Four-Colour Theorem", *Electron. Res. Announc. Amer. Math. Soc.*, vol. 2, October 1996.

[Rond08] Patrick M. Rondon, Ming Kawaguci and Ranjit Jhala, "Liquid types", *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 159–169, June 2008.

[Rush97] John Rushby, "Formal Methods and their Role in the Certification of Critical Systems", in Roger Shaw, editor, *Safety and Reliability of Software Based Systems*, pp. 1–42, London, 1997, Springer.

[Sedg] Robert Sedgewick, *Left-leaning Red-Black Trees*.

[Sedg15] Robert Sedgewick and Kevin Wayne, *Algorithms, Fourth Edition (Deluxe): Book and 24-Part Lecture Series*, Addison-Wesley Professional, 1st edition, 2015.

[Tarj85] Robert Endre Tarjan, "Amortized Computational Complexity", *SIAM Journal on Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306–318, April 1985. Publisher: Society for Industrial and Applied Mathematics.

[Tymo79] Thomas Tymoczko, "The Four-color Problem and Its Philosophical Significance", *undefined*, 1979.

[Vasc08] Pedro B. Vasconcelos, *Space cost analysis using sized types*, Thesis, University of St Andrews, November 2008. Accepted: 2008-12-01T14:05:11Z.

[Vazo14a] Niki Vazou, Eric L. Seidel and Ranjit Jhala, "LiquidHaskell: experience with refinement types in the real world", *ACM SIGPLAN Notices*, vol. 49, no. 12, pp. 39–51, September 2014.

[Vazo14b] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis and Simon Peyton-Jones, "Refinement types for Haskell", in *Proceedings of the*

19th ACM SIGPLAN international conference on Functional programming, ICFP '14, pp. 269–282, New York, NY, USA, August 2014, Association for Computing Machinery.

[Vazo16] Niki Vazou and Ranjit Jhala, "Refinement Reflection (or, how to turn your favorite language into a proof assistant using SMT)", *arXiv:1610.04641 [cs]*, October 2016. arXiv: 1610.04641.

[Wadl92] Philip Wadler, "The essence of functional programming", in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pp. 1–14, New York, NY, USA, February 1992, Association for Computing Machinery.

[Wang17] Peng Wang, Di Wang and Adam Chlipala, "TiML: a functional language for practical complexity analysis with invariants", *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 79:1–79:26, October 2017.

[Xi99] Hongwei Xi and Frank Pfenning, "Dependent types in practical programming", in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pp. 214–227, New York, NY, USA, January 1999, Association for Computing Machinery.

[Zeng97] Christoph Zenger, "Indexed types", *Theoretical Computer Science*, vol. 187, no. 1, pp. 147–165, November 1997.

Παράρτημα Α

Γλωσσάρι Θεωρίας των Γλωσσών Προγραμματισμού και Θεωρίας Τύπων

Όταν ξεκίνησα να γράφω αυτήν την διπλωματική στα ελληνικά, συνάντησα αρκετές δυσκολίες στην εύρεση των αντίστοιχων ελληνικών όρων. Το πρώτο πλήγμα ήταν όταν έγραψα τον όρο *εκλεπτυσμένοι τύποι* για να αποδώσω τον αντίστοιχο αγγλικό *refinement types*. Δεν θεωρούσα ότι αποδίδει ακριβώς την έννοια του αγγλικού όρου, αλλά ταυτόχρονα ήταν και το καλύτερο που μπορούσα να κάνω. Στις περισσότερες περιπτώσεις ανέτρεξα στο *Αγγλοελληνικόν Λεξικόν των Θεωρητικών και Εφηρμοσμένων Μαθηματικών* του Μέμα Κολαΐτη. Ωστόσο όντας ένα παλαιό λεξικό (εκδόθηκε το 1975) δεν μπορεί πάντα να συμβαδίζει με το νόημα που έχουν αποκτήσει οι λέξεις σήμερα. Για παράδειγμα ο όρος *reasoning* μεταφράζεται ως *διαλογισμός*. Θεωρώ ότι αυτός ο ελληνικός όρος δεν θα απέδιδε το κατάλληλο νόημα στην αναγνώστρια. Για αυτό επιλέχθηκε ένας εναλλακτικός, ο όρος *συλλογιστική*. Σε άλλες περιπτώσεις, κατέφυγα στο *Αγγλοελληνικό Λεξικό Μαθηματικής Ορολογίας* του Γιώργου Γεωργίου είτε για να διασταυρώσω μία μετάφραση είτε για να αναζητήσω μία εναλλακτική. Τέλος, αναγκάστηκα πολλές φορές να αυτοσχεδιάσω.

A.1 Γλωσσάρι Θεωρίας Γλωσσών Προγραμματισμού

Ελληνικός Όρος	**Αγγλικός Όρος**
άλγεβρα πολλαπλών ταξινομήσεων	multi-sorted algebra
αλγεβρικές προδιαγραφές	algebraic specification
αμιγής	pure
αναλλοίωτη	invariant
ανερμήνευτη (συνάρτηση)	uninterpretated (function)
(συναρτήσεις) ανώτερης τάξης	higer-order (functions)
αυτόματη ανάλυση πόρων απόσβεσης	automatic amortized resource analysis
αφαίρεση κατηγορήματος	predicate abstraction
αφηρημένη ερμηνεία	abstract interpretation
βοηθός απόδειξης	proof assistant
δηλωτική μοντελοποίηση	declarative modelling
δηλωτική σημασιολογία	denotational semantics
ελεγκτής απόδειξης	proof-checker
επισημείωση υπαιτιότητας	blame label
εφαρμοστής	applicative
κατηγόρημα	predicate
λογική διαχωρισμού	separation logic
μετα-συνθήκη	post-condition
μονάδα	monad
μονάδα κόστους	cost monad
μονοειδές	monoid
οκνηρή	lazy
προδιαγραφή	specification
προυπόθεση	pre-condition
(συναρτήσεις) πρώτης τάξης	first class (functions)
συλλογιστική	reasoning
συμβολικοί αλγόριθμοι	symbolic algorithms
συναρτητής	functor
συνδυαστής απόδειξης	proof combinator
σχολιασμός προγράμματος	program annotation
φυσικός μετασχηματισμός	natural transformation
χρονική λογική	temporal logic
χώρος καταστάσεων	state space

A.2 Γλωσσάρι Θεωρίας Τύπων

Ελληνικός Όρος	**Αγγλικός Όρος**
αφηρημένος εκλεπτυσμένος τύπος	abstract refinement type
εκλεπτυσμένη λογική	refinement logic
εκλεπτυσμένος τύπος	refinement type
εξαρτημένος τύπος	dependent type
εξαρτημένο ζεύγος	dependent pair
θεωρία τύπων	type theory
λ-αφαίρεση	lambda abstraction
λ-έκφραση	lambda expression
λ-λογισμός με τύπους	typed lambda calculi
όρος	term
περιβάλλον	context
indexed types	τύποι με δείκτες
sized types	τύποι με μέγεθος
τύπος	type
;;;	sort