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

Μηχανική Επαλήθευση Προστακτικών Προγραμμάτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΒΑΣΙΛΕΙΟΣ ΠΑΠΑΒΑΣΙΛΕΙΟΥ

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

Αθήνα, Ιούλιος 2009

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

Μηχανική Επαλήθευση Προστακτικών Προγραμμάτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΒΑΣΙΛΕΙΟΣ ΠΑΠΑΒΑΣΙΛΕΙΟΥ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιουλίου 2009.

Νικόλαος Παπασπύρου Επίκ. Καθηγητής Ε.Μ.Π.	Κωστής Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Ευστάθιος Ζάχος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2009

Βασίλειος Παπαβασιλείου

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

Περίληψη

Σκοπός της διπλωματικής είναι ο ορισμός μιας γλώσσας προγραμματισμού που επιτρέπει την επαλήθευση προγραμμάτων και η υλοποίηση ενός εργαλείου επαλήθευσης. Το σύστημα επαλήθευσης βασίζεται στο εργαλείο υποστήριξης αποδείξεων (proof assistant) Coq.

Τα προγράμματα που δέχεται ως είσοδο το σύστημά μας είναι γραμμένα σε μια απλή γλώσσα προστακτικού προγραμματισμού, την Tony. Λόγω του προστακτικού της ύφους η Tony είναι οικεία στους περισσότερους προγραμματιστές, παρέχει όμως και χαρακτηριστικά αναγκαία για την επαλήθευση προγραμμάτων. Οι προδιαγραφές που πρέπει να ικανοποιούν τα προγράμματα ορίζονται με τη μορφή προσυνθηκών (preconditions) και μετασυνθηκών (postconditions), τις οποίες ο προγραμματιστής συμπεριλαμβάνει στο πρόγραμμα ως επισημειώσεις (annotations). Στόχος ήταν να παρέχουμε μια εκφραστική γλώσσα για τη διατύπωση των συνθηκών αυτών. Επιλέξαμε ως εκ τούτου να γράφονται απ' ευθείας στη γλώσσα προδιαγραφών του Coq (Gallina). Ο κώδικας των προγραμμάτων περιλαμβάνει και άλλα στοιχεία που μπορούν να βοηθήσουν στη διαδικασία απόδειξης της ορθότητας, όπως θεωρήματα και ορισμούς που θα χρειαστούν και αναλλοίωτες βρόχου (loop invariants).

Το εργαλείο επαλήθευσης διαβάζει ένα πρόγραμμα σε Tony και παράγει ένα θεώρημα του Coq, το οποίο αποτελεί διατύπωση της μερικής ορθότητας κατά Hoare. Στη συνέχεια εφαρμόζει αυτοματοποιημένες τακτικές για να κατασκευάσει μια απόδειξη του εν λόγω θεωρήματος. Σε περίπτωση που αποτύχει, ενημερώνει τον προγραμματιστή ποια θεωρήματα πρέπει να αποδείξει για να ολοκληρωθεί η διαδικασία επαλήθευσης, με τη μορφή ενός προτύπου (template) με κώδικα Coq που πρέπει να συμπληρωθεί.

Λέξεις κλειδιά

Αξιωματική σημασιολογία, λογική Hoare, επαλήθευση προγραμμάτων, μερική ορθότητα, Coq.

Abstract

The purpose of this diploma project was to define a programming language that allows program verification and to implement a verification tool. Our verification system is based on the Coq proof assistant.

Our system accepts programs written in a simple imperative language that we call Tony. Due to its imperative style, Tony should be familiar to most programmers. Nevertheless, it provides features that are necessary for program verification. The specifications that programs must meet are expressed in terms of preconditions and postconditions, written by the programmer as annotations in the program. Our goal was to provide an expressive language for these conditions. We chose to write them directly in Coq's specification language (Gallina). In addition, the programs' code contains other elements that are helpful in proving correctness, such as theorems and definitions as well as loop invariants.

The program verification tool reads a program in Tony and produces a theorem in Coq, which formalizes the program's partial correctness according to Hoare. Subsequently, it applies automated techniques for constructing a proof of this theorem. In case it fails, it reports to the programmer which theorems must be proved to complete the verification process, in the form of a template.

Key words

Axiomatic semantics, Hoare logic, program verification, partial correctness, Coq.

Ευχαριστίες

Ευχαριστώ τους γονείς μου για την στήριξη που μου παρείχαν καθ'όλη τη διάρκεια των σπουδών μου και τον καθηγητή μου κ. Παπασπύρου για την αγάπη που μου μετέδωσε για την Επιστήμη των Υπολογιστών και το χρόνο που διέθεσε για τη συγκεκριμένη εργασία.

Βασίλης Παπαβασιλείου,

Αθήνα, 14η Ιουλίου 2009

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-3-09, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2009.

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

Περιεχόμενα

Σχήματα

Κεφάλαιο 1

Εισαγωγή

1.1 Εισαγωγή στις Έννοιες

Σκοπός της εργασίας είναι η υλοποίηση ενός συστήματος τυπικής επαλήθευσης προγραμμάτων σε μια προστακτική γλώσσα προγραμματισμού. Το σύστημα θα βασιστεί στο εργαλείο υποστήριξης αποδείξεων Coq.

Με τον όρο *επαλήθευση*, εννοούμε σε αυτή την εργασία τη διαδικασία απόδειξης της ορθότητας ενός προγράμματος με αναφορά σε έναν τυπικό ορισμό των προδιαγραφών του και με χρήση τυπικών μεθόδων. Οι προδιαγραφές αποτελούν μια διατύπωση της επιθυμητής συμπεριφοράς του προγράμματος. Στην εργασία αυτή ασχολούμαστε με τη *μερική ορθότητα* (partial correctness) των προγραμμάτων. Ένα μερικώς ορθό πρόγραμμα εφόσον τερματίζεται θα επιστρέφει σωστό αποτέλεσμα, δεν μπορούμε όμως να γνωρίζουμε αν θα τερματιστεί. Η *ολική ορθότητα* (total correctness) αντίθετα εγγυάται και τερματισμό.

Η γλώσσα προγραμματισμού που σχεδιάσαμε έχει *προστακτικό ύφος* (imperative paradigm), ο προγραμματιστής δηλαδή διατυπώνει τον υπολογισμό που θα πραγματοποιηθεί με εντολές που αλλάζουν την κατάσταση εκτέλεσης. Είναι το πρότυπο που ακολουθούν διαδεδομένες γλώσσες όπως η C και η Pascal. Μια *επισημείωση* (annotation) στο πρόγραμμα είναι ένα κομμάτι κώδικα το οποίο δεν εκτελείται (δεν το διαβάζει ο διερμηνέας ή ο μεταγλωττιστής της γλώσσας), είναι όμως χρήσιμο για το εργαλείο επαλήθευσης. Τα προγράμματα της γλώσσας μας είναι επισημειωμένα με *προσυνθήκες* (preconditions), *μετασυνθήκες* (postconditions) και *αναλλοίωτες βρόχου* (loop invariants). Προσυνθήκη είναι μια πρόταση που πρέπει να ισχύει πριν εκτελεστεί ένα πρόγραμμα, μετασυνθήκη κάποια πρόταση που θα ισχύει αφού τερματιστεί το πρόγραμμα και αναλλοίωτη μια πρόταση που είναι αληθής καθ' όλη τη διάρκεια εκτέλεσης ενός βρόχου.

Σημασιολογία (semantics) μιας γλώσσας προγραμματισμού είναι ένα μαθηματικό μοντέλο που περιγράφει τους πιθανούς υπολογισμούς σε αυτή. Στη βιβλιογραφία συναντά κανείς διαφορετικές προσεγγίσεις στη σημασιολογία των γλωσσών προγραμματισμού. Τα συστήματα σημασιολογίας κατηγοριοποιούνται συνήθως στη *λειτουργική* (operational), τη *δηλωτική* (denotational) και την *αξιωματική σημασιολογία* (axiomatic semantics). Η αξιωματική σημασιολογία συνδέεται στενότερα με την επαλήθευση των προγραμμάτων και είναι το μοντέλο που ακολουθούμε στην παρούσα εργασία. Η αξιωματική σημασιολογία ορίζει τη σημασία των εντολών με βάση την επίδρασή τους σε ισχυρισμούς (assertions) σχετικά με την κατάσταση εκτέλεσης του προγράμματος.

Το *Coq* είναι *ένα εργαλείο υποστήριξης αποδείξεων* (proof assistant), ένα πρόγραμμα δηλαδή που μας επιτρέπει να εκφράσουμε μαθηματικά θεωρήματα και να αποδείξουμε ότι ισχύουν. Τα μαθηματικά θεωρήματα διατυπώνονται σε μια τυπική γλώσσα. Η διαδικασία της απόδειξης είναι διαδραστική. Αυτοματοποιημένες τακτικές μπορούν να αναλάβουν μέρος της απόδειξης, είναι όμως σημαντικός και ο ρόλος του χρήστη. Η βιβλιοθήκη που συνοδεύει το Coq παρέχει αποδείξεις θεωρημάτων σε πολλά πεδία εφαρμογών, και τακτικές που διευκολύνουν την αποδεικτική διαδικασία σε συγκεκριμένο πεδίο. Η θεωρία στην οποία βασίζεται το Coq είναι το *Calculus of Inductive Constructions* (CIC), ένας υψηλότερης τάξης λάμδα λογισμός με τύπους.

1.2 Παρακίνηση για την Εργασία

Στην εποχή της κοινωνίας της πληροφορίας και της γνώσης, το ζήτημα της αξιοπιστίας του λογισμικού έχει μεγάλη βαρύτητα. Τα σύγχρονα συστήματα υπολογιστών χειρίζονται μεγάλο όγκο ευαίσθητων δεδομένων. Προγραμματιστικά λάθη μπορεί να θέσουν σε κίνδυνο την ακεραιότητα και την εμπιστευτικότητα των δεδομένων και να έχουν σημαντικές οικονομικές επιπτώσεις.

Ορισμένες γλώσσες προγραμματισμού όπως η Eiffel υποστηρίζουν *συμβόλαια* (contracts). Ο προγραμματιστής μπορεί να διατυπώσει προδιαγραφές ορθότητας των προγραμμάτων με τη μορφή προσυνθηκών, μετασυνθηκών και αναλλοίωτων σε επίπεδο βρόχου ή αντικειμένου σε μια τυπική γλώσσα. Ο έλεγχος των συμβολαίων γίνεται όμως στον χρόνο εκτέλεσης του προγράμματος. Οι υλοποιήσεις των γλωσσών δεν κατασκευάζουν μια μαθηματική απόδειξη ότι επαληθεύονται οι προδιαγραφές που ορίζουν τα συμβόλαια. Δεν μπορούμε συνεπώς να αποφανθούμε με τυπικό τρόπο για την ορθότητα των προγραμμάτων και οι δοκιμές του προγράμματος είναι δύσκολο ή αδύνατο να καλύπτουν όλο το εύρος των δυνατών εισόδων. Επιπλέον, οι έλεγχοι κατά τον χρόνο εκτέλεσης έχουν κόστος στην απόδοση. Θα ήταν χρήσιμο να μιμηθούμε συντακτικά τα συμβόλαια, καθώς η λειτουργία τους γίνεται εύκολα αντιληπτή διαισθητικά, να εισάγουμε όμως παράλληλα μηχανισμούς επαλήθευσης.

Τα τελευταία χρόνια έχουν αναπτυχθεί ισχυρά συστήματα αυτόματης ή διαδραστικής απόδειξης θεωρημάτων. Εργαλεία υποστήριξης αποδείξεων βασίζονται σε προηγμένα συστήματα λογικής και κατά συνέπεια παρέχουν μια εκφραστική γλώσσα για τη διατύπωση προδιαγραφών. Επιπλέον, η έρευνα στη σημασιολογία των γλωσσών προγραμματισμού έχει αποδώσει καρπούς και μπορούμε να ορίσουμε τυπικά και τις πλέον πολύπλοκες δομές τους. Εκμεταλλευόμενοι το μαθηματικό υπόβαθρο της σημασιολογίας και την τεχνολογία που έχει αναπτυχθεί για την απόδειξη θεωρημάτων, είναι δυνατό να κατασκευάσουμε εργαλεία που διευκολύνουν σημαντικά την επαλήθευση λογισμικού.

Πιστεύουμε ότι η διαδικασία της επαλήθευσης δεν πρέπει να γίνεται αντιληπτή ως ένα ανεξάρτητο στάδιο που ακολουθεί τον προγραμματισμό καθ' αυτόν. Κάθε προγραμματιστής έχει στο μυαλό του τα επιθυμητά αποτελέσματα του προγράμματος εφόσον η είσοδος καλύπτει κάποια κριτήρια. Με άλλα λόγια, κάθε πρόγραμμα υπονοεί ένα θεώρημα ορθότητας. Για την ανάπτυξη ενός σωστού προγράμματος είναι επιθυμητή η λογική ανάλυση της επιθυμητής συμπεριφοράς, ήδη έχει γίνει επομένως ένα βήμα προς την κατεύθυνση της επαλήθευσης. Είναι ευκολότερο να επαληθεύσουμε τμηματικά τα προγράμματα αποδεικνύοντας θεωρήματα για τη συμπεριφορά μικρών τμημάτων τους ταυτόχρονα με τη συγγραφή. Στόχος μας είναι λοιπόν να παρέχουμε μια γλώσσα προγραμματισμού και σχετικά εργαλεία που θα ενθαρρύνουν ένα ιδίωμα προγραμματισμού, κατά το οποίο η συγγραφή και η επαλήθευση είναι αλληλένδετες και πραγματοποιούνται ταυτόχρονα.

Η γλώσσα Tony και το εργαλείο επαλήθευσης που τη συνοδεύει αποτελούν ένα βήμα προς αυτή την κατεύθυνση. Το σύστημα επαλήθευσης έχει υλοποιηθεί ως επέκταση του εργαλείου υποστήριξης αποδείξεων του Coq και εκμεταλλεύεται την εκφραστικότητα της γλώσσας διατύπωσης προδιαγραφών του τελευταίου. Η Tony ακολουθεί ένα ύφος οικείο στους προγραμματιστές, καθώς είναι προστακτική, ενθαρρύνει όμως και την επαλήθευση ως διαδικασία παράλληλη με τη συγγραφή του κώδικα.

1.3 Σύνοψη της εργασίας

Δεύτερο Κεφάλαιο Γίνεται μια εισαγωγή στις διαφορετικές μεθόδους για την επαλήθευση προγραμμάτων και τις προσεγγίσεις στη σημασιολογία των γλωσσών προγραμματισμού. Δίνεται έμφαση στην αξιωματική σημασιολογία.

Τρίτο Κεφάλαιο Στο κεφάλαιο αυτό παρουσιάζονται εργαλεία επαλήθευσης προγραμμάτων που ήδη έχουν υλοποιηθεί και το εργαλείο υποστήριξης αποδείξεων Coq.

Τέταρτο Κεφάλαιο Παρουσιάζουμε τη γλώσσα προγραμματισμού Tony. Ορίζεται η σύνταξη και η σημασιολογία της και δίνονται ενδεικτικά προγράμματα.

Πέμπτο Κεφάλαιο Περιγράφουμε τον τρόπο λειτουργίας του εργαλείου επαλήθευσης προγραμμάτων της Tony. Παρουσιάζεται η κωδικοποίηση της αξιωματικής σημασιολογίας στο Coq και οι τακτικές επαλήθευσης που ακολουθούμε.

Έκτο Κεφάλαιο Αποτιμούμε τα επιτεύγματα και τους περιορισμούς της εργασίας, και προτείνουμε κατευθύνσεις για σχετική έρευνα στο μέλλον.

Κεφάλαιο 2

Επαλήθευση και Αξιωματική Σημασιολογία

2.1 Τυπική Επαλήθευση

Τυπική επαλήθευση (formal verification) είναι η διαδικασία απόδειξης της ορθότητας των αλγορίθμων στους οποίους βασίζεται ένα σύστημα υλικού ή λογισμικού, με αναφορά σε κάποια τυπική προδιαγραφή και με χρήση τυπικών μεθόδων. Θα αναφερθούμε στη συνέχεια στις δύο βασικές προσεγγίσεις στην τυπική επαλήθευση, τον *έλεγχο μοντέλων* και την επαλήθευση με *αποδείξεις θεωρημάτων*.

2.1.1 Έλεγχος Μοντέλων

Ο έλεγχος μοντέλων (model checking) αποτελεί την πρώτη προσέγγιση στην επαλήθευση. Με δεδομένο ένα απλοποιημένο μοντέλο ενός συστήματος, η προσέγγιση αυτή διερευνά αν το μοντέλο ικανοποιεί μια προδιαγραφή. Το υπό εξέταση σύστημα είναι στις περισσότερες περιπτώσεις ένα σύστημα υλικού (hardware), ένα σύστημα λογισμικού ή ένα πρωτόκολλο επικοινωνίας. Η προδιαγραφή αποτελεί διατύπωση απαιτήσεων ασφαλείας όπως η απουσία αδιέξοδων (deadlocks) ή άλλων καταστάσεων που μπορούν να οδηγήσουν το σύστημα σε κατάρρευση.

Μια σημαντική κατηγορία μεθόδων ελέγχου μοντέλων αφορά την επαλήθευση προδιαγραφών διατυπωμένων σε *χρονική λογική* (temporal logic). Ένα σύστημα χρονικής λογικής επιτρέπει τη διατύπωση προτάσεων με ποσοτικοποίηση ως προς τον χρόνο και την επαλήθευση τους. Σε χρονική λογική, η τιμή αλήθειας μιας πρότασης μπορεί να μεταβάλλεται με το χρόνο. Μπορούμε για παράδειγμα να διατυπώσουμε πως μια ιδιότητα ισχύει πάντα, πως τελικά θα καταλήξουμε σε ισχύ της ιδιότητας ή πως η ιδιότητα θα ισχύει έως ότου συμβεί κάποιο γεγονός. Όσον αφορά τα συστήματα λογισμικού και υλικού, μια πρόταση σε χρονική λογική μπορεί να ορίζει πως αν μια διεργασία ζητήσει κάποιον πόρο τελικά θα τον λάβει και επομένως η εκτέλεσή της δεν θα ανασταλεί για πάντα. Για την πρωτοποριακή τους έρευνα στο πεδίο του ελέγχου μοντέλων χρονικής λογικής, οι E. M. Clarke, E. A. Emerson και Ιωσήφ Σηφάκης έλαβαν το βραβείο Turing του 2007.

Συνήθως μοντελοποιούμε το σύστημα προς επαλήθευση ως μια *δομή Kripke* (Kripke structure). Μια δομή Kripke είναι στην ουσία ένας κατευθυνόμενος γράφος, οι κόμβοι του οποίου είναι επισημειωμένοι με σύνολα ατομικών προτάσεων. Οι κόμβοι αντιστοιχούν σε *καταστάσεις*, ενώ οι ακμές σε *μεταβάσεις* μεταξύ αυτών. Ένας ή περισσότεροι κόμβοι αποτελούν αρχικές καταστάσεις. Μπορούμε να αντιληφθούμε τη δομή Kripke ως ένα είδος αυτόματου πεπερασμένων καταστάσεων. Αν το σύστημα έχει αναπαρασταθεί ως δομή Kripke, το πρόβλημα του ελέγχου μοντέλων ισοδυναμεί με αναζήτηση σε γράφο.

Η προσέγγιση του ελέγχου μοντέλων παρουσιάζει αρκετά πλεονεκτήματα, με σημαντικότερο τον πλήρως αυτόματο χαρακτήρα του. Ο χρήστης εισάγει μια υψηλού επιπέδου αναπαράσταση του μοντέλου και της προδιαγραφής που θα ελεγχθεί. Ο αλγόριθμος του ελέγχου επαλήθευσης επιστρέφει ως απάντηση την τιμή «αληθής» αν το μοντέλο ικανοποιεί την προδιαγραφή, ενώ αν αυτό δεν ισχύει παρέχει ένα αντιπαράδειγμα. Τα αντιπαραδείγματα αυτά είναι ιδιαίτερα χρήσιμα στην αντιμετώπιση σφαλμάτων σε συστήματα με περίπλοκες μεταβάσεις. Η διαδικασία του ελέγχου μοντέλων είναι γενικά γρήγορη. Επιπλέον, επιτρέπει έλεγχο τμηματικών προδιαγραφών και κατά συνέπεια μπορούν να εξαχθούν χρήσιμες πληροφορίες για την ορθότητα του συστήματος πριν αυτό οριστεί πλήρως. Τέλος, τα συστήματα λογικής που χρησιμοποιούνται μπορούν να εκφράσουν άμεσα πολλές από τις

επιθυμητές ιδιότητες των συστημάτων με ταυτοχρονισμό.

Κύριο πρόβλημα στην τεχνική του ελέγχου μοντέλων είναι η κατάσταση που περιγράφεται ως *έκρηξη καταστάσεων* (state explosion). Το πρόβλημα εμφανίζεται σε συστήματα με πολλά συνιστώντα μέρη που αλληλεπιδρούν, ή με δομές δεδομένων που μπορούν να λάβουν πολλές διαφορετικές τιμές. Ο αριθμός των καταστάσεων σε πολλές περιπτώσεις αυξάνεται εκθετικά με τον αριθμό των αλληλεπιδρώντων διεργασιών και γρήγορα γίνεται απαγορευτικός. Η αντιμετώπιση του προβλήματος της έκρηξης καταστάσεων είναι η βασική κατεύθυνση έρευνας στον έλεγχο μοντέλων.

2.1.2 Επαλήθευση με Αποδείξεις Θεωρημάτων

Η δεύτερη προσέγγιση στην τυπική επαλήθευση είναι η χρήση *εργαλείων απόδειξης θεωρημάτων*. Χρησιμοποιούνται τυπικά συστήματα που παρέχουν λογικούς κανόνες για την επαλήθευση προγραμμάτων, όπως η *λογική Hoare* που θα παρουσιάσουμε εκτενώς στη συνέχεια. Τα εργαλεία επαλήθευσης που βασίζονται σε απόδειξη θεωρημάτων είναι μόνο εν μέρει αυτοματοποιημένα. Απαιτούν σημαντική εξοικείωση του χρήστη με το πρόγραμμα που επαληθεύει, αλλά και τεχνογνωσία σχετικά με το ίδιο το εργαλείο απόδειξης.

Διαφορά-κλειδί σε σχέση με τον έλεγχο μοντέλων είναι ότι στην περίπτωση της επαλήθευσης με αποδείξεις θεωρημάτων δεν χρειάζεται να επισκευθούμε εξαντλητικά τον χώρο καταστάσεων του προγράμματος. Αντίθετα, εφαρμόζουμε κανόνες της λογικής για να καταλήξουμε στο συμπέρασμα ότι η προδιαγραφή ικανοποιείται. Σαν αποτέλεσμα, ένας άπειρος χώρος καταστάσεων ή περίπλοκες δομές δεδομένων που απαιτούν αναδρομή δεν εμποδίζουν την επαλήθευση. Η προσέγγιση του ελέγχου μοντέλων είναι σε γενικές γραμμές καταλληλότερη για υλικό ή λογισμικό με περίπλοκη ροή ελέγχου αλλά απλά δεδομένα. Το άρθρο [Ouim08] αναφέρεται εκτενέστερα στις διαφορές μεταξύ των δύο προσεγγίσεων με αναφορές στη βιβλιογραφία.

2.2 Προσεγγίσεις στη Σημασιολογία

Στη μελέτη των γλωσσών προγραμματισμού, κυρίαρχες είναι οι έννοιες της σύνταξης και της σημασιολογίας. Η σύνταξη αναφέρεται στη δομή των έγκυρων προτάσεων στη γλώσσα. Η σημασιολογία αναφέρεται στη σημασία των προτάσεων αυτών, την οποία οφείλουν να υλοποιούν οι μεταγλωττιστές και διερμηνείς της γλώσσας.

Η σύνταξη των γλωσσών προγραμματισμού περιγράφεται συνήθως με τυπικό τρόπο. Χρησιμοποιείται εκτενώς η μορφή Backus-Naur (BNF) και οι παραλλαγές της για να εκφράσουμε γραμματικές χωρίς συμφραζόμενα. Από την άλλη πλευρά, η τυπική σημασιολογία μιας γλώσσας αποτελεί ένα μαθηματικό μοντέλο που περιγράφει τους πιθανούς υπολογισμούς στη γλώσσα. Η σημασιολογία μιας γλώσσας προγραμματισμού συνήθως διατυπώνεται με άτυπο τρόπο, για παράδειγμα με παραδείγματα ή με περιγραφή σε φυσική γλώσσα. Μπορούν εύκολα να προκύψουν ασάφειες ή αμφισημίες.

Για τον ορισμό της τυπικής σημασιολογίας των γλωσσών προγραμματισμού χρησιμοποιούνται τρεις βασικές προσεγγίσεις:

- Η *δηλωτική σημασιολογία* (denotational semantics) χρησιμοποιεί κάποιο μαθηματικό φορμαλισμό για τον ορισμό της σημασίας των προγραμμάτων. Μπορούν, για παράδειγμα, να χρησιμοποιηθούν συναρτήσεις από εισόδους σε εξόδους.

- Η *λειτουργική σημασιολογία* (operational semantics) περιγράφει τη σημασία των προγραμμάτων με σχέσεις μετάβασης μεταξύ των καταστάσεων μιας αφηρημένης μηχανής.

- Η *αξιωματική σημασιολογία* (axiomatic semantics) προσδίδει σημασία στα προγράμματα περιγράφοντας τα αξιώματα της λογικής τα οποία ικανοποιούν. Κατά την αξιωματική σημασιολογία, σημασία του προγράμματος είναι οτιδήποτε μπορεί να αποδειχθεί για αυτό σε ένα σύστημα λογικής.

$S \rightarrow$ skip
| *id* = *E*
| if *C* then *S* else *S*
| while *C* do *S*
| *S* ; *S*

Σχήμα 2.1: Η προστακτική γλώσσα της λογικής Hoare

Οι παραπάνω προσεγγίσεις εξυπηρετούν διαφορετικούς σκοπούς και δεν πρέπει να τις αντιλαμβανόμαστε ως ανταγωνιστικές μεταξύ τους. Η δηλωτική και η λειτουργική σημασιολογία μπορούν να χρησιμοποιηθούν για την υλοποίηση ενός διερμηνέα για μια νέα γλώσσα προγραμματισμού και με αυτόν τον τρόπο να διευκολύνουν την εξέλιξή της. Η αξιωματική σημασιολογία περιγράφει τη σημασία των προγραμμάτων με λογικές προτάσεις που δηλώνουν ιδιότητες τους, υποστηρίζει επομένως άμεσα την επαλήθευση προγραμμάτων. Σε αυτό το κεφάλαιο θα αναφερθούμε εκτενώς στην αξιωματική σημασιολογία, καθώς αποτελεί το μοντέλο με το οποίο πραγματοποιείται η επαλήθευση προγραμμάτων στη συγκεκριμένη εργασία.

2.3 Λογική Hoare

Η *λογική Hoare* είναι ένα τυπικό σύστημα που μας επιτρέπει να αποφανθούμε για την ορθότητα προγραμμάτων με την αυστηρότητα της μαθηματικής λογικής. Αναπτύχθηκε από τον Βρετανό C. A. R. Hoare [Hoar83], ενώ σχετίζεται με παλαιότερη εργασία του Robert Floyd [Floy67] η οποία περιέγραφε ένα ανάλογο σύστημα για διαγράμματα ροής. Η λογική Hoare αποτελεί το κανονικό παράδειγμα αξιωματικής σημασιολογίας.

Βασική έννοια στην λογική Hoare είναι η τριάδα Hoare. Οι τριάδες Hoare είναι της μορφής $\{P\}\ \mathtt{S}\ \{Q\}$, όπου P η προσυνθήκη, Q η μετασυνθήκη και $\mathtt{S}$ ένα πρόγραμμα ή τμήμα προγράμματος. Η προσυνθήκη και η μετασυνθήκη αποτελούν κατηγορήματα στην κατάσταση του προγράμματος, και μπορούν να εμφανίζονται σε αυτές οι μεταβλητές του. Μια τριάδα Hoare αποτελεί μια προδιαγραφή για τη λειτουργία του προγράμματος. Εφόσον η τριάδα είναι αληθής, δηλαδή η προδιαγραφή ικανοποιείται, αν η αρχική κατάσταση ικανοποιεί την προσυνθήκη τότε μετά την εκτέλεση του προγράμματος η κατάσταση εκτέλεσης θα ικανοποιεί τη μετασυνθήκη.

Η λογική Hoare μας επιτρέπει να αποφανθούμε μόνο για τη μερική ορθότητα των προγραμμάτων. Αν το πρόγραμμα εκτελεστεί με αρχική κατάσταση που ικανοποιεί την προσυνθήκη και τερματιστεί, τότε θα ικανοποιείται η μετασυνθήκη. Η αλήθεια μιας τριάδας Hoare δεν συνεπάγεται όμως τερματισμό του προγράμματος και επομένως ολική ορθότητά του.

Στην εργασία του ο Hoare χρησιμοποιεί μια απλή προστακτική γλώσσα. Η σύνταξή της δίνεται στο σχήμα 2.1. Η λογική Hoare παρέχει αξιώματα και κανόνες συμπερασμού για όλες τις δομές της.

$$\{P\}\ \mathtt{skip}\ \{P\} \tag{2.1}$$

Ο τύπος 2.1 είναι το *αξίωμα της κενής εντολής*. Η εντολή `skip` δεν μεταβάλλει την κατάσταση εκτέλεσης του προγράμματος, επομένως η προσυνθήκη είναι ίδια με τη μετασυνθήκη.

$$\{P[E/V]\}\ \mathtt{V} := \mathtt{E}\ \{P\} \tag{2.2}$$

Σύμφωνα με τον *κανόνα ανάθεσης* (τύπος 2.2), αν ίσχυε η συνθήκη $\{P[E/V]\}$ πριν την ανάθεση θα ισχύει η συνθήκη $\{P\}$ μετά την ανάθεση, όπου ο συμβολισμός $\{P[E/V]\}$ δηλώνει αντικατάσταση των ελεύθερων εμφανίσεων της μεταβλητής V με την έκφραση E στη συνθήκη P. Για παράδειγμα, είναι έγκυρη η τριάδα $\{x = 18\}\ \mathtt{x} = 24 + \mathtt{x}\ \{x = 42\}$, καθώς με αντικατάσταση του x με $24 + x$

στη μετασυνθήκη προκύπτει η ισότητα $24 + x = 42$, η οποία είναι ισοδύναμη με την προσυνθήκη.

$$\frac{\{P\}\ \mathtt{S}_1\ \{R\},\ \ \{R\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \mathtt{S}_1;\ \mathtt{S}_2\ \{Q\}} \tag{2.3}$$

Ο *κανόνας της σύνθεσης* (τύπος 2.3) εφαρμόζεται σε προγράμματα (εντολές) που εκτελούνται το ένα αμέσως μετά το άλλο. Σύμφωνα με την προϋπόθεση του κανόνα αυτού, η μετασυνθήκη του πρώτου προγράμματος πρέπει να είναι κοινή με την προσυνθήκη του δεύτερου, με άλλα λόγια το πρώτο πρόγραμμα πρέπει να παρέχει την κατάσταση εκτέλεσης που αναμένει το δεύτερο. Σε αυτή την περίπτωση είναι αληθής η τριάδα Hoare με προσυνθήκη αυτή του προγράμματος $\mathtt{S}_1$, μετασυνθήκη αυτή του $\mathtt{S}_2$ και πρόγραμμα την ακολουθία των δύο.

$$\frac{\{P \wedge C\}\ \mathtt{S}_1\ \{Q\},\quad \{P \wedge \neg C\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \mathtt{if\ C\ then\ S}_1\ \mathtt{else\ S}_2\ \{Q\}} \tag{2.4}$$

Για να εφαρμόσουμε τον *κανόνα του if* (τύπος 2.4), πρέπει να αποδείξουμε ότι καταλήγουμε στη μετασυνθήκη Q οποιοδήποτε από τα δύο σκέλη και αν εκτελεστεί. Στην περίπτωση που εκτελεστεί το σκέλος `then` γνωρίζουμε ότι ισχύει η συνθήκη C. Η προσυνθήκη της αντίστοιχης τριάδας Hoare είναι επομένως η σύζευξη της C με την προσυνθήκη P της τριάδας - στόχου. Προσυνθήκη της τριάδας που αντιστοιχεί στο σκέλος `else` είναι η σύζευξη της P με την άρνηση της C.

$$\frac{\{P \wedge C\}\ \mathtt{S}\ \{P\}}{\{P\}\ \mathtt{while\ C\ do\ S}\ \{P \wedge \neg C\}} \tag{2.5}$$

Ο τύπος 2.5 καλείται *κανόνας του* `while`. Η συνθήκη P στον κανόνα αυτόν ονομάζεται *αναλλοίωτη βρόχου* γιατί ισχύει πριν και μετά από κάθε επανάληψη. Υποχρέωσή μας είναι να αποδείξουμε ότι η εντολή `S` στο εσωτερικό του βρόχου διατηρεί πράγματι την αναλλοίωτη. Στην προσυνθήκη της αντίστοιχης τριάδας Hoare η P εμφανίζεται σε σύζευξη με την συνθήκη C, καθώς αυτό το τμήμα κώδικα δεν θα εκτελεστεί σε περίπτωση που η C δεν ισχύει. Στη μετασυνθήκη της εντολής `while` εμφανίζεται εκτός από την αναλλοίωτη P και η άρνηση της C, διότι αν ίσχυε η C ο βρόχος δεν θα είχε τερματιστεί. Η επιλογή κατάλληλης αναλλοίωτης βρόχου δεν είναι πάντα προφανής. Τα εργαλεία επαλήθευσης αναμένουν συνήθως από τον προγραμματιστή να παρέχει την αναλλοίωτη με τη μορφή επισημείωσης στο πρόγραμμα, ενώ σε κάποιες περιπτώσεις μπορούν να εφαρμόσουν ευριστικές μεθόδους για εξαγωγή κατάλληλης αναλλοίωτης.

$$\frac{P \Rightarrow R,\quad \{R\}\ \mathtt{C}\ \{Q\}}{\{P\}\ \mathtt{C}\ \{Q\}} \tag{2.6}$$

Όπως θα αναμέναμε και διαισθητικά, ο *κανόνας ενδυνάμωσης της προσυνθήκης* (τύπος 2.6) μας επιτρέπει να αντικαταστήσουμε την προσυνθήκη μιας τριάδας Hoare με κάποια ισχυρότερη. Εφόσον η συνθήκη P συνεπάγεται την προσυνθήκη R, η πρώτη παρέχει την κατάσταση εκτέλεσης που αναμένει η εντολή.

$$\frac{\{P\}\ \mathtt{C}\ \{R\},\quad R \Rightarrow Q}{\{P\}\ \mathtt{C}\ \{Q\}} \tag{2.7}$$

Ο *κανόνας αποδυνάμωσης της μετασυνθήκης* (τύπος 2.7) είναι συμπληρωματικός του παραπάνω και μας επιτρέπει να αντικαταστήσουμε την μετασυνθήκη με κάποια ασθενέστερη. Γνωρίζουμε ότι ισχύει η τριάδα Hoare με προσυνθήκη P και μετασυνθήκη R. Επομένως, αν το πρόγραμμα C εκτελεστεί με προσυνθήκη P και τερματιστεί, θα ισχύει η μετασυνθήκη R, η οποία όμως συνεπάγεται ισχύ της μετασυνθήκης Q. Είναι επομένως έγκυρη η τριάδα Hoare με μετασυνθήκη Q.

$$\frac{\{P_1\}\ \mathtt{C}\ \{Q_1\}\quad \{P_2\}\ \mathtt{C}\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \mathtt{C}\ \{Q_1 \wedge Q_2\}} \tag{2.8}$$

Ο τύπος 2.8 είναι ο *κανόνας σύζευξης*. Αν πριν από την εκτέλεση του προγράμματος C ισχύει η συνθήκη $P_1 \wedge P_2$, θα ισχύουν οι συνθήκες P_1 και P_2. Σύμφωνα με την πρώτη τριάδα της υπόθεσης αν

τερματιστεί το πρόγραμμα C θα ισχύει η συνθήκη Q_1. Ομοίως σύμφωνα με τη δεύτερη τριάδα θα ισχύει η συνθήκη Q_2. Καταλήγουμε σε ισχύ της σύζευξης των δύο. Η τριάδα-στόχος είναι έγκυρη.

$$\frac{\{P_1\}\ \mathtt{C}\ \{Q_1\} \quad \{P_2\}\ \mathtt{C}\ \{Q_2\}}{\{P_1 \vee P_2\}\ \mathtt{C}\ \{Q_1 \vee Q_2\}} \tag{2.9}$$

Ο τύπος 2.9 καλείται *κανόνας διάζευξης*. Σύμφωνα με την προσυνθήκη της τριάδας-στόχου, πριν την εκτέλεση του προγράμματος C ισχύει μια τουλάχιστον από τις P_1, P_2. Αν ισχύει η P_1, με βάση την πρώτη τριάδα της υπόθεσης καταλήγουμε σε ισχύ της Q_1 σε περίπτωση τερματισμού. Αν ισχύει η P_2, καταλήγουμε σε ισχύ της Q_2. Μια τουλάχιστον από τις Q_1, Q_2 ισχύει μετά την εκτέλεση του C, άρα η μετασυνθήκη με διάζευξη των δύο ικανοποιείται.

2.4 Μια Απόδειξη με Λογική Hoare

Ως παράδειγμα εφαρμογής της λογικής Hoare για την επαλήθευση προγραμμάτων, θα αποδείξουμε τη μερική ορθότητα του ακόλουθου προγράμματος υπολογισμού της δύναμης a^n:

```
r := 1;
i := 0;
while i < n do (
    r = r * a;
    i = i + 1
)
```

Η μετασυνθήκη που εκφράζει τον επιθυμητό υπολογισμό είναι $r = a^n$. Η προσυνθήκη πρέπει να εξασφαλίζει ότι $n \geq 0$, γιατί το πρόγραμμα δεν λαμβάνει υπ' όψιν την περίπτωση που χρειάζονται διαιρέσεις αντί για πολλαπλασιασμούς. Υποθέτουμε $a > 0$ για να αποφύγουμε τον υπολογισμό 0^0. Επιλέγουμε λοιπόν την προσυνθήκη $n \geq 0 \wedge a > 0$.

Σημαντική δυσκολία στις αποδείξεις είναι η επιλογή κατάλληλων αναλλοίωτων βρόχου. Σε κάθε επανάληψη ισχύει $r = a^i$. Αν εφαρμόσουμε τον κανόνα του `while` (2.5) προκύπτει για τον βρόχο η μετασυνθήκη $r = a^i \wedge \neg(i < n)$. Η συνθήκη αυτή όμως δεν είναι αρκετή γιατί δεν συνεπάγεται την επιθυμητή μετασυνθήκη $r = a^n$. Για να συμβεί αυτό, πρέπει να μπορούμε να αποδείξουμε ότι μετά την τελευταία επανάληψη θα ισχύει $i = n$. Προσθέτουμε στην αναλλοίωτη βρόχου τη συνθήκη $i \leq n$. Τώρα από τη σύζευξη $\neg(i < n) \wedge i \leq n$ προκύπτει $i = n$. Θέλουμε επιπλέον $a > 0$ και $i \geq 0$ για τους λόγους που προαναφέραμε. Η αναλλοίωτη βρόχου είναι τελικά

$$r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \tag{2.10}$$

Η προσυνθήκη που αναμένει η `while` για να μπορούμε να εφαρμόσουμε τον αντίστοιχο κανόνα ταυτίζεται με την αναλλοίωτη βρόχου. Πρέπει, κατά συνέπεια, να επαληθεύσουμε την τριάδα

$$\{n \geq 0 \wedge a > 0\}\ \mathtt{r := 1; i := 0}\ \{r = a^i \wedge 0 \leq i \leq n \wedge a > 0\} \tag{2.11}$$

Για να εφαρμόσουμε τον κανόνα της σύνθεσης, πρέπει να επαληθεύσουμε τις τριάδες Hoare

$$\begin{array}{rl} \{\ n \geq 0 \wedge a > 0\ \} & \mathtt{r := 1}\ \ \{\ P\ \} \\ \{\ P\ \} & \mathtt{i := 0}\ \ \{\ r = a^i \wedge 0 \leq i \leq n \wedge a > 0\ \} \end{array} \tag{2.12}$$

Η συνθήκη P μας είναι προς το παρόν άγνωστη. Επιλέγουμε ως P τη μετασυνθήκη της ακολουθίας των δύο εντολών, με αντικατάσταση του i με 0. Οι τριάδες γίνονται

$$\begin{array}{rl} \{\ n \geq 0 \wedge a > 0\ \} & \mathtt{r := 1}\ \ \{\ r = a^0 \wedge 0 \leq 0 \leq n \wedge a > 0\ \} \\ \{\ r = a^0 \wedge 0 \leq 0 \leq n \wedge a > 0\ \} & \mathtt{i := 0}\ \ \{\ r = a^i \wedge 0 \leq i \leq n \wedge a > 0\ \} \end{array} \tag{2.13}$$

Η δεύτερη τριάδα είναι αληθής σύμφωνα με τον κανόνα της ανάθεσης, καθώς επιλέξαμε κατάλληλη προσυνθήκη. Αντικαθιστούμε στη μετασυνθήκη της πρώτης τριάδας το r με 1. Κατά το αξίωμα της ανάθεσης, θα είναι αληθής η τριάδα

$$\{1 = a^0 \wedge 0 \leq 0 \leq n \wedge a > 0\}\ \texttt{r := 1}\ \{r = a^0 \wedge 0 \leq 0 \leq n \wedge a > 0\} \tag{2.14}$$

Η προσυνθήκη της παραπάνω τριάδας δεν ταυτίζεται με την προσυνθήκη της πρώτης τριάδας στην εξίσωση 2.13. Πρέπει επομένως να εφαρμόσουμε τον κανόνα ενίσχυσης της προσυνθήκης. Χρειάζεται να αποδείξουμε τη συνεπαγωγή

$$n \geq 0 \wedge a > 0 \Rightarrow 1 = a^0 \wedge 0 \leq 0 \leq n \wedge a > 0 \tag{2.15}$$

Πράγματι,

$n \geq 0 \wedge a > 0$	σύμφωνα με την υπόθεση
$1 = a^0$	γιατί η ισότητα αυτή ισχύει για κάθε $a > 0$, με $a > 0$ γνωστό
$0 \leq 0$	από τον ορισμό του τελεστή $\leq$
$0 \leq n$	σύμφωνα με την υπόθεση
$a > 0$	σύμφωνα με την υπόθεση
$1 = a^0 \wedge 0 \leq 0 \leq n \wedge a > 0$	με λογική σύζευξη των τεσσάρων παραπάνω προτάσεων

Επαληθεύσαμε επομένως τις τριάδες 2.13 και, με εφαρμογή του κανόνα της σύνθεσης, την τριάδα 2.11.

Πρέπει να αποδείξουμε ότι η αναλλοίωτη βρόχου διατηρείται, δηλαδή να επαληθεύσουμε την τριάδα Hoare

$$\begin{array}{ll} \{\, r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \wedge i < n \,\} & \texttt{r := r * a;} \\ & \texttt{i := i + 1} \quad \{\, r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \,\} \end{array} \tag{2.16}$$

Εφαρμόζουμε τον κανόνα σύνθεσης, με ενδιάμεση συνθήκη η οποία προκύπτει με αντικατάσταση του i με $i + 1$ στη μετασυνθήκη. Η αλήθεια της δεύτερης τριάδας προκύπτει άμεσα από τον κανόνα της ανάθεσης, ενώ θα πρέπει να αποδείξουμε την αλήθεια της πρώτης με κανόνα ενίσχυσης της προσυνθήκης, όπως νωρίτερα:

$$\begin{array}{lll} \{\, r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \wedge i < n \,\} & \texttt{r := r * a} & \{\, r = a^{i+1} \wedge 0 \leq i + 1 \leq n \wedge a > 0 \,\} \\ \{\, r = a^{i+1} \wedge 0 \leq i + 1 \leq n \wedge a > 0 \,\} & \texttt{i := i + 1} & \{\, r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \,\} \end{array} \tag{2.17}$$

Πρέπει να αποδείξουμε ότι η προσυνθήκη συνεπάγεται την ενδιάμεση συνθήκη με αντικατάσταση του r με $r * a$, δηλαδή

$$r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \wedge i < n \Rightarrow r * a = a^{i+1} \wedge 0 \leq i + 1 \leq n \wedge a > 0 \tag{2.18}$$

Πράγματι,

$r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \wedge i < n$	σύμφωνα με την υπόθεση
$r * a = a^i * a$	με πολλαπλασιασμό κατά μέλη της υπόθεσης $r = a^i$
$r * a = a^{i+1}$	με την αντικατάσταση $a^{i+1} = a^i * a$ στην προηγούμενη
$0 \leq i + 1$	διότι σύμφωνα με την υπόθεση $0 \leq i$
$i + 1 \leq n$	διότι σύμφωνα με την υπόθεση $i < n$
$a > 0$	σύμφωνα με την υπόθεση
$r * a = a^{i+1} \wedge 0 \leq i + 1 \leq n \wedge a > 0$	με σύζευξη των παραπάνω

Εφόσον η αναλλοίωτη βρόχου διατηρείται, με εφαρμογή του κανόνα του `while` καταλήγουμε σε ισχύ της τριάδας Hoare

$$\begin{array}{l}\{\, r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \,\} \\ \texttt{while i < n do (r = r * a; i = i + 1)} \\ \{\, r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \wedge \neg(i < n) \,\}\end{array} \tag{2.19}$$

Η επιθυμητή μετασυνθήκη είναι όμως $r = a^n$. Πρέπει λοιπόν να εφαρμόσουμε κανόνα αποδυνάμωσης της μετασυνθήκης. Αποδεικνύουμε πως

$$r = a^i \wedge 0 \leq i \leq n \wedge a > 0 \wedge \neg(i < n) \Rightarrow r = a^n \tag{2.20}$$

Σύμφωνα με την υπόθεση $i \leq n \wedge \neg(i < n)$, επομένως $i = n$. Επίσης $r = a^i$, επομένως λόγω της ισότητας $i = n$ έχουμε $r = a^n$.

Εφαρμόσουμε τέλος των κανόνα σύνθεσης για τις τριάδες 2.11 και 2.20. Αποδείξαμε πως αν το πρόγραμμα εκτελεστεί με προσυνθήκη $n \geq 0 \wedge a > 0$ και τερματιστεί, θα ισχύει $r = a^n$.

2.5 Σημασιολογία Μετασχηματισμού Κατηγορημάτων

Η *σημασιολογία μετασχηματισμού κατηγορημάτων* (predicate transformer semantics) είναι μια προσέγγιση στη σημασιολογία που σχετίζεται στενά με τη λογική Hoare. Την προσέγγιση του μετασχηματισμού κατηγορημάτων ακολούθησε για πρώτη φορά ο Δανός Edsger Wybe Dijkstra το 1975 [Dijk75]. Η προσέγγιση αυτή ορίζει τη σημασιολογία μιας προστακτικής γλώσσας προγραμματισμού ορίζοντας για κάθε εντολή έναν *μετασχηματιστή κατηγορημάτων*. Ο μετασχηματιστής κατηγορημάτων είναι μια πλήρης συνάρτηση (total function) μεταξύ κατηγορημάτων στην κατάσταση του προγράμματος.

Στην αρχική δημοσίευση του Dijkstra, ο μετασχηματιστής κατηγορημάτων είναι μια συνάρτηση που επιστρέφει την *ασθενέστερη προσυνθήκη* (weakest precondition) και συνήθως συμβολίζεται με wp. Αν S είναι μια εντολή ή πρόγραμμα και Q μια μετασυνθήκη, η $wp(S, Q)$ αποτελεί την πλέον ασθενή προσυνθήκη με την οποία εκτέλεση του προγράμματος S καταλήγει σε αλήθεια της μετασυνθήκης Q. Αντίθετα με τις προσυνθήκες του Hoare, η $wp(S, Q)$ δεν οδηγεί μόνο σε σωστό αποτέλεσμα αλλά εγγυάται και τερματισμό, έχουμε δηλαδή ολική ορθότητα.

$$\begin{array}{rcll} wp(\texttt{skip}, Q) & = & Q & \\ wp(\texttt{V := E}, Q) & = & Q[E/V] & \\ wp(\texttt{S}_1\texttt{; S}_2, Q) & = & wp(\texttt{S}_1, wp(\texttt{S}_2, Q)) & \\ wp(\texttt{if B then S}_1\texttt{ else S}_2, Q) & = & (B \Rightarrow wp(\texttt{S}_1, Q)) \wedge (\neg B \Rightarrow wp(\texttt{S}_2, Q)) & \\ wp(\texttt{while B do S}, Q) & = & \exists k.(k \geq 0 \wedge P_k), & \text{όπου} \\ P_0 & = & \neg B \wedge Q & \\ P_{k+1} & = & B \wedge wp(S, P_k) & \end{array}$$

Σχήμα 2.2: Συνάρτηση ασθενέστερης προσυνθήκης για μια απλή προστακτική γλώσσα

Στο σχήμα 2.2 δίνεται η συνάρτηση wp για την προστακτική γλώσσα της λογικής Hoare. Η γλώσσα την οποία χρησιμοποιεί ο Dijkstra διαφέρει [1]. Στον ορισμό της wp για την εντολή `while`, P_k είναι η ασθενέστερη προσυνθήκη που πρέπει να είναι αληθής πριν τον βρόχο, έτσι ώστε αυτός να τερματιστεί μετά από k ακριβώς επαναλήψεις σε κατάσταση που ικανοποιεί την μετασυνθήκη Q. Καθώς σύμφωνα με τον ορισμό της συνάρτησης wp για την εντολή `while` υπάρχει k τέτοιο ώστε το P_k να είναι αληθές η ασθενέστερη προσυνθήκη οδηγεί σε τερματισμό.

Οι ασθενέστερες προσυνθήκες δεν είναι η μοναδική περίπτωση μετασχηματισμού κατηγορημάτων, αν και είναι το κανονικό παράδειγμα. Ενδεικτικά, μια επέκταση της ιδέας των ασθενέστερης προσυνθήκης είναι η *ασθενέστερη φιλελεύθερη προσυνθήκη* (weakest liberal precondition) η οποία δεν

[1] Οι δομές `if` και `while` περιέχουν σειρά εντολών με φρουρούς (guarded commands). Σύμφωνα με τον ορισμό της wp, η εκτέλεση είναι μη ντετερμινιστική αν οι φρουροί δεν αλληλοαποκλείονται

εξασφαλίζει τερματισμό, ενώ οι μετασχηματιστές win και wlp έχουν προταθεί για την επαλήθευση προγραμμάτων με ταυτοχρονισμό.

Σε αντίθεση με άλλους σημασιολογικούς φορμαλισμούς, η σημασιολογία μετασχηματισμού κατηγορημάτων δεν ήταν προσπάθεια μαθηματικής θεμελίωσης των υπολογισμών που πραγματοποιούνται από τις γλώσσες προγραμματισμού. Αντίθετα, σκοπός του Dijkstra ήταν να παρέχει ένα σύνολο κανόνων (έναν λογισμό) για την παραγωγή ορθών προγραμμάτων από τυπικές προδιαγραφές τους. Ακόμα και αν δεν επιδιώκουμε παραγωγή προγραμμάτων αλλά επαλήθευση τους, οι ασθενέστερες προσυνθήκες αποδεικνύονται χρήσιμο εργαλείο. Για να επαληθεύσουμε μια τριάδα $\{P\}$ S $\{Q\}$ υπολογίζουμε την ασθενέστερη προσυνθήκη $wp(S, Q)$ και αποδεικνύουμε ότι $P \Rightarrow wp(S, Q)$. Ενδέχεται αυτή να μην είναι η μοναδική *συνθήκη επαλήθευσης* και να έχουμε επιπλέον υποχρεώσεις απόδειξης, για παράδειγμα τη διατήρηση των αναλλοίωτων στους βρόχους.

2.6 Επεκτάσεις της Λογικής Hoare

2.6.1 Ολική Ορθότητα

Η μόνη εντολή της προστακτικής γλώσσας του σχήματος 2.1 που μπορεί να οδηγήσει σε μη τερματισμό είναι `while`. Με τροποποίηση του σχετικού κανόνα (τύπος 2.5) ώστε να αποδεικνύεται πως θα συμβεί πεπερασμένος αριθμός επαναλήψεων μπορούμε να αποφανθούμε για την ολική ορθότητα των προγραμμάτων.

Στα μαθηματικά, μια σχέση R καλείται *ορθώς ορισμένη* (well-founded) σε μια κλάση X αν και μόνο αν κάθε μη κενό υποσύνολο της X έχει ένα ελάχιστο στοιχείο όσον αφορά την R. Πιο τυπικά, σε κάθε μη κενό υποσύνολο S του X υπάρχει στοιχείο m τέτοιο ώστε για κάθε στοιχείο s του S, το ζεύγος (s, m) να μην ανήκει στην R:

$$\forall S \subseteq X (S \neq \emptyset \rightarrow \exists m \in S \, \forall s \in S \, (s, m) \notin R) \tag{2.21}$$

Για παράδειγμα, είναι ορθώς ορισμένη η σχέση μικρότερου ($<$) στο σύνολο των θετικών ακέραιων $\{1, 2, 3...\}$.

Αν $(, <)$ είναι ορθώς ορισμένη σχέση και το x ανήκει στο σύνολο X, τότε όλες οι αλυσίδες που ξεκινούν από το x είναι πεπερασμένες. Στην ιδιότητα αυτή βασίζεται η απόδειξη ολικής ορθότητας στη λογική Hoare. Αν ένας όρος (ο οποίος αποκαλείται *loop variant*) μειώνεται αυστηρά αναφορικά με κάποια ορθώς ορισμένη σχέση σε κάθε επανάληψη, τότε αποδεικνύεται τερματισμός. Αυτό συμβαίνει διότι οι τιμές του όρου ανήκουν σε ένα πεπερασμένο, ολικά διατεταγμένο σύνολο (αλυσίδα), άρα και ο αριθμός των επαναλήψεων θα είναι πεπερασμένος. Προκύπτει ο ακόλουθος κανόνας του `while` για ολική ορθότητα:

$$\frac{< \text{ορθώς ορισμένη}, \quad [P \wedge C \wedge t = z] \; \texttt{S} \; [P \wedge t < z]}{[P] \; \texttt{while C do S} \; [P \wedge \neg C]} \tag{2.22}$$

Στον παραπάνω κανόνα χρησιμοποιούμε αγκύλες αντί για τα συνήθη άγκιστρα της λογικής Hoare για να δηλώσουμε ολική ορθότητα.

2.6.2 Χειρισμός των Συνωνύμων

Ο τρόπος με τον οποίο αντιμετωπίζει η λογική Hoare τις αναθέσεις είναι σωστός μόνο όταν γνωρίζουμε ότι διαφορετικά ονόματα μεταβλητών αναφέρονται σε διαφορετικές θέσεις της μνήμης. Στις πρακτικές γλώσσες προγραμματισμού εμφανίζονται όμως *συνώνυμα* (aliases), δηλαδή δύο διαφορετικά ονόματα αναφέρονται στην ίδια θέση μνήμης. Τα συνώνυμα μπορούν να εμφανιστούν σε διαφορετικές εκδοχές. Η παρακάτω λίστα δεν είναι εξαντλητική:

- Συνώνυμα λόγω παραμέτρων (parameter aliasing) εμφανίζονται σε γλώσσες όπως η Pascal που επιτρέπουν πέρασμα παραμέτρων κατ' αναφορά (call-by-reference). Το πρόβλημα εμφανίζεται όταν μια παράμετρος κατ' αναφορά αντιστοιχεί σε μια θέση μνήμης έξω από τη διαδικασία (procedure), αλλά μπορούμε να αναφερθούμε σε αυτή με άλλο τρόπο μέσα από τη διαδικασία.

- Συνώνυμα λόγω πινάκων (subscript aliasing) εμφανίζονται γιατί μπορούμε να αναφερθούμε στο ίδιο στοιχείο του πίνακα με διαφορετικές μεταβλητές δείκτη. Οι $b[i]$ και $b[j]$ είναι η ίδια θέση μνήμης όταν $i = j$.

- Συνώνυμα λόγω δεικτών (pointer aliasing) εμφανίζονται όταν μπορούμε να αναφερθούμε σε μια οντότητα έμμεσα μέσω ενός δείκτη στη μνήμη.

Δυσκολότερα στον χειρισμό τους είναι τα συνώνυμα λόγω παραμέτρων. Ευκολότερο είναι να αντιμετωπίσουμε τα συνώνυμα λόγω πινάκων. Μπορούμε να αντιμετωπίσουμε έναν πίνακα ως μια μοναδική μεταβλητή που περιγράφει μια αντιστοίχηση ανάμεσα σε δείκτες και τιμές. Η εντολή $a[i] = E$ αναθέτει μια νέα αντιστοίχιση $a \oplus i \mapsto E$ στον a. Η $Q[a \oplus i \mapsto E/a]$ είναι η αναγκαία προσυνθήκη ώστε η εντολή να τερματιστεί με κατάσταση που ικανοποιεί τη μετασυνθήκη Q. Η αλλαγή της αντιστοίχισης ορίζεται ως $(a \oplus i \mapsto E)[j] = (\texttt{if } i = j \texttt{ then } E \texttt{ else } a[j])$. Το βήμα αναγωγής (reduction step) που επιβάλλει η συνθήκη $i = j$ επιλύει πλήρως το πρόβλημα των συνωνύμων σε πίνακες.

Σε μια γλώσσα προγραμματισμού όπως η C, ο προγραμματιστής έχει τη δυνατότητα να δεσμεύει και να απελευθερώνει δυναμικά θέσεις στη μνήμη, με συναρτήσεις βιβλιοθήκης όπως η `malloc(3)` και η `free(3)`. Η περιοχή της μνήμης την οποία διαχειρίζονται οι συναρτήσεις αυτές καλείται σωρός. Μπορούμε να αναφερθούμε στον σωρό μέσω δεικτών. Δύο δείκτες είναι δυνατόν να καταλήγουν στην ίδια θέση μνήμης, προκύπτουν δηλαδή συνώνυμα (pointer aliases). Θα μπορούσαμε να μοντελοποιήσουμε τον σωρό ως έναν πίνακα, διαιρώντας τον ενδεχομένως σε υπο-πίνακες για τους διαφορετικούς τύπους δεδομένων. Αυτό μας επιτρέπει να αντιμετωπίσουμε το πρόβλημα όπως αντιμετωπίσαμε τα συνώνυμα λόγω πινάκων, παρουσιάζει όμως πρακτικές δυσκολίες. Το βασικότερο είναι ότι μας οδηγεί σε καθολικό συμπερασμό (global reasoning): οποιαδήποτε ανάθεση σε στοιχείο που βρίσκεται στον σωρό επηρεάζει κάθε ισχυρισμό που αναφέρεται με κάποιο τρόπο σε αυτόν. Αυτό δεν συμβαδίζει με τον τρόπο που διαχειρίζεται η λογική Hoare τις μεταβλητές, καθώς η ανάθεση σε μεταβλητή επηρεάζει μόνο τους ισχυρισμούς που αναφέρονται σε αυτή.

Η *λογική διαχωρισμού* (separation logic) [Reyn02] είναι μια επέκταση της λογικής Hoare που μας επιτρέπει να αποφανθούμε για την ορθότητα προγραμμάτων χαμηλού επιπέδου. Αποτελεί μια προσπάθεια να αποφευχθεί ο καθολικός συμπερασμός. Η λογική διαχωρισμού προσθέτει χωρικούς τελεστές στη γλώσσα προδιαγραφών της λογικής Hoare. Η πράξη $P \star Q$ καλείται *διαχωριστική*, *ανεξάρτητη* ή *χωρική σύζευξη* (separating, independent or spatial conjunction) και εξασφαλίζει ότι οι P και Q ισχύουν για ανεξάρτητα τμήματα της μνήμης που μπορεί να διευθυνσιοδοτηθεί. Στους κανόνες της λογικής Hoare προστίθεται ο *κανόνας πλαισίου* (frame rule):

$$\frac{\{P\}\ \mathtt{C}\ \{Q\}}{\{R \star P\}\ \mathtt{C}\ \{R \star Q\}} \tag{2.23}$$

Μπορούμε σύμφωνα με τον τύπο 2.23 να ενισχύσουμε την προσυνθήκη και την μετασυνθήκη με κάποια R η οποία δεν επηρεάζεται από την εντολή C (η εντολή αναμένει προσυνθήκη P και καταλήγει σε μετασυνθήκη Q, οι οποίες εμφανίζονται σε διαχωριστική σύζευξη με την R). Με αυτόν τον τρόπο, πρέπει κατά τη διαδικασία της επαλήθευσης να μας απασχολεί μόνο το κομμάτι της μνήμης στο οποίο επεμβαίνει. Η λογική διαχωρισμού εισάγει με αυτό τον τρόπο τη δυνατότητα τοπικού συμπερασμού (local reasoning).

Κεφάλαιο 3

Σχετική Εργασία

Στο κεφάλαιο αυτό παρουσιάζουμε εργαλεία που αυτοματοποιούν εν μέρει τη διαδικασία της επαλήθευσης προγραμμάτων. Γίνεται επίσης μια εισαγωγή στο σύστημα υποστήριξης αποδείξεων Coq στο οποίο βασίζεται η δική μας εργασία.

3.1 Why

Το Why [Fill03] είναι μια πλατφόρμα για την επαλήθευση λογισμικού, η οποία έχει αναπτυχθεί από τον Jean-Christophe Filliâtre και άλλους ερευνητές στο γαλλικό Laboratoire de Recherche en Informatique. Το Why δέχεται ως είσοδο ένα πρόγραμμα με επισημειώσεις και επιστρέφει ως έξοδο *συνθήκες επαλήθευσης* (verification conditions), δηλαδή προτάσεις της λογικής η ισχύς των οποίων συνεπάγεται την ορθότητα του προγράμματος. Η πλατφόρμα Why επιτρέπει επαλήθευση προγραμμάτων σε διαφορετικές γλώσσες, ενώ ο έλεγχος των συνθηκών επαλήθευσης πραγματοποιείται από εξωτερικά εργαλεία απόδειξης θεωρημάτων.

3.1.1 Η Γλώσσα WL

Το Why δεν περιορίζεται σε μια γλώσσα προγραμματισμού. Αντίθετα, διαθέτει τη δική του γλώσσα, την *WL*, στην οποία μπορούν να μεταγλωττιστούν προγράμματα γραμμένα σε γλώσσες όπως η C και η Java. Η WL αποτελεί διάλεκτο της ML. Η σύνταξή της παρουσιάζεται στο σχήμα 3.1. Σε συμφωνία με την παράδοση των γλωσσών της ML οικογένειας, η WL ενοποιεί συντακτικά τις εκφράσεις, τις εντολές, τις τοπικές μεταβλητές και τις συναρτήσεις (με το μη τερματικό σύμβολο e στο σχήμα 3.1). Αυτό διευκολύνει τη συμβολική επεξεργασία και έχει ως αποτέλεσμα λιγότερες περιπτώσεις προς διερεύνηση κατά τον υπολογισμό ασθενέστερων προσυνθηκών και συνθηκών επαλήθευσης. Η WL διαθέτει στοιχεία προστακτικών γλωσσών όπως αναφορές (references) και πίνακες που επιτρέπουν ανάθεση (arrays). Παρέχει επίσης εξαιρέσεις (exceptions), αν και δεν εμφανίζονται στη γραμματική 3.1 για λόγους απλότητας. Μπορούμε να μεταγλωττίσουμε εντολές μη ομαλού τερματισμού όπως οι `return`, `break` και `continue` της C και της Java σε εξαιρέσεις, με αποτέλεσμα να μην απαιτείται άμεση υποστήριξη αυτών. Η WL δεν επιτρέπει συνώνυμα στις μεταβλητές, χαρακτηριστικό που διευκολύνει την εύρεση συνθηκών επαλήθευσης.

Όπως θα αναμέναμε, η WL επιτρέπει την επισημείωση των προγραμμάτων με προσυνθήκες, μετασυνθήκες, αναλλοίωτες και μεταβλητές βρόχου. Το σύστημα λογικής του Why είναι πρώτης τάξης, όπως φαίνεται και από τους ισχυρισμούς (τερματικό σύμβολο p στο σχήμα 3.1). Το μη τερματικό σύμβολο β αναφέρεται στους βασικούς τύπους (η μεταβλητή x αναφέρεται σε αφηρημένους τύπους που ορίζει ο χρήστης στην πλευρά της λογικής). Η WL διαχωρίζει τους τύπους για τις τιμές (τ) από τους τύπους των υπολογισμών (κ). Οι τελευταίοι περιέχουν επιπλέον μια προσυνθήκη, μια μετασυνθήκη και ένα αποτέλεσμα (effect) που δηλώνει τις μεταβλητές που τροποποιούν. Η WL υποστηρίζει ετικέτες πριν από την αποτίμηση κάποιας έκφρασης ($L : E$). Στους ισχυρισμούς μια μεταβλητή μπορεί να επισημειωθεί με ετικέτα ($e@L$), για να αναφερθούμε στην τιμή της σε κάποιο σημείο του προγράμματος. Αναφερόμαστε τέλος στο αποτέλεσμα μιας έκφρασης στην αντίστοιχη μετασυνθήκη με τη μεταβλητή $result$.

$t \rightarrow constant \mid x \mid x\ @\ L \mid f\ (t, ..., t)$

$p \rightarrow x \mid x\ (t, ..., t) \mid \text{true} \mid \text{false} \mid \text{not } p \mid p \text{ and } p \mid p \text{ or } p \mid \text{if } t \text{ then } p \text{ else } p$
$\quad \mid \text{forall } x : \beta\ .\ p \mid \text{exists } x : \beta\ .\ p$

$\beta \rightarrow \text{unit} \mid \text{bool} \mid \text{int} \mid \text{float} \mid x$

$\tau \rightarrow \beta \mid \beta \text{ ref} \mid \beta \text{ array} \mid x : \tau \text{ -> } \kappa$

$\kappa \rightarrow \{\ p\ \}\ \tau\ \varepsilon\ \{\ p\ \}$

$\varepsilon \rightarrow \text{reads } x, ..., x \text{ writes } x, ..., x$

$e \rightarrow \{\ p\ \}\ e\ \{\ p\ \}$
$\quad \mid t \mid !\,x \mid x := e \mid \text{ref } e \mid x\ [\ e\ ] \mid x\ [\ e\ ] := e \mid e\ ;\ e \mid L : e$
$\quad \mid \text{if } e \text{ then } e \text{ else } e \mid \text{let } x = e \text{ in } e \mid \text{fun } (x : \tau) \text{ -> } e \mid (e\ e)$
$\quad \mid \text{rec } x : \tau\ \{ \text{ variant } t\ \} = e \mid \text{while } e \text{ do } \{ \text{ invariant } p \text{ variant } t\ \}\ e \text{ done}$

Σχήμα 3.1: Η γλώσσα WL

```
(*
 * calculating square root
 *)

let sqrt = fun (x : int) ->
  { x >= 0 }
  begin
    if x = 0 then
      0
    else if x <= 3 then
      1
    else
      let y = ref x in
      let z = ref (x + 1) / 2 in
      begin
        while !z < !y do
        {
          invariant
            z > 0 and
            y > 0 and
            z = (x / y + y) / 2 and
            x < (y + 1) * (y + 1) and
            x < (z + 1) * (z + 1)
          variant y
        }
          y := !z;
          z := (x / !z + !z) / 2
        done;
        !y
      end
  end
  { result * result <= x and x < (result+1)*(result+1) }
```

Σχήμα 3.2: Υπολογισμός τετραγωνικής ρίζας με τη WL

Ένα παράδειγμα προγράμματος σε WL δίνεται στο σχήμα 3.2. Η συνάρτηση `sqrt` υπολογίζει την ακέραιη τετραγωνική ρίζα του ορίσματος `x`. Η προσυνθήκη επιβάλλει ο αριθμός `x` να είναι μη αρνητικός. Σύμφωνα με τη μετασυνθήκη, για το αποτέλεσμα `result` της συνάρτησης ισχύει $result^2 \leq x \wedge x < (result+1)^2$, το `result` ικανοποιεί δηλαδή τον ορισμό της ακέραιας τετραγωνικής ρίζας. Η συνάρτηση `sqrt` είναι γραμμένη σε προστακτικό ύφος και μια υλοποίηση σε κάποια προστακτική γλώσσα όπως η C δεν θα διέφερε πολύ. Τα `x` και `y` αποτελούν αναφορές σε θέσεις μνήμης των οποίων το περιεχόμενο αλλάζουμε σε κάθε επανάληψη. Το πρόγραμμα είναι επισημειωμένο με μια αναλλοίωτη αλλά και μια μεταβλητή βρόχου. Οι υποχρεώσεις απόδειξης που παράγει το Why εξασφαλίζουν μεταξύ άλλων ότι το νέο `y` μετά από κάθε επανάληψη είναι αυστηρά μικρότερο του παλιού, κάτι που συνεπάγεται τερματισμό του προγράμματος. Μπορούμε επομένως με το Why να αποφανθούμε για την ολική ορθότητα των προγραμμάτων. Η ομάδα ανάπτυξης του Why παρέχει αποδείξεις σε Coq για τις συνθήκες επαλήθευσης του συγκεκριμένου προγράμματος.

3.1.2 Λογισμός Ασθενέστερης Προσυνθήκης στο Why

$$
\begin{array}{rcl}
wp(\{p\}\ \mathtt{e}\ \{q'\}, q) & = & p' \vee \forall result.\forall \omega.q' \Rightarrow q \\
wp(t, q) & = & q[result \leftarrow t] \\
wp(!x, q) & = & q[result \leftarrow x] \\
wp(x := e, q) & = & wp(e, q[result \leftarrow void; x \leftarrow result]) \\
wp(t[e], q) & = & wp(e, q[result \leftarrow (access\ t\ result)]) \\
wp(t[e_1] := e_2, q) & = & wp(e_1, wp(e_2, q_1)[v_1 \leftarrow result]) \\
 & \text{όπου} & q_1 = q[result \leftarrow void;\ t \leftarrow (update\ t\ v_1\ result)] \\
wp(e_1;\ e_2, q) & = & wp(e_1, wp(e_2, q)) \\
wp(L : e, q) & = & wp(e, q)[x@L \leftarrow x] \\
wp(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3, q) & = & wp(e_1, \mathtt{if}\ result\ \mathtt{then}\ wp(e_2, q)\ \mathtt{else}\ wp(e_3, q)) \\
wp(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, q) & = & wp(e_1, wp(e_2, q)[x \leftarrow result])
\end{array}
$$

Σχήμα 3.3: Συνάρτηση wp για τις δομές της WL

Το Why παράγει συνθήκες επαλήθευσης με βάση τη συνάρτηση ασθενέστερης προσυνθήκης wp που αντιστοιχεί στη γλώσσα WL. Η συνάρτηση αυτή δίνεται στο σχήμα 3.3. Ιδιαίτερο χαρακτηριστικό του λογισμού ασθενέστερης προσυνθήκης που ακολουθεί το Why είναι ο τρόπος που χειρίζεται τις υποεκφράσεις με επισημειώσεις (πρώτη περίπτωση στο 3.3). Η τριάδα $\{p'\}\ \mathtt{e}\ \{q'\}$ αντιμετωπίζεται ως ένα μαύρο κουτί με προδιαγραφή που ορίζεται από την προσυνθήκη p' και τη μετασυνθήκη q'. Δεν εξετάζει την `e` για να υπολογίσει ασθενέστερη προσυνθήκη με μετασυνθήκη την q. Αντίθετα, εφαρμόζει ανεξάρτητα τον λογισμό ασθενέστερης προσυνθήκης για την e με μετασυνθήκη q'. Αυτό εισάγει μια μορφή τμηματοποίησης (modularity). $access$ και $update$ είναι οι αντίστοιχες λειτουργίες για τους πίνακες όπως αυτές έχουν μοντελοποιηθεί στην πλευρά της λογικής.

Το Why δεν διαθέτει δικό του σύστημα απόδειξης θεωρημάτων. Οι συνθήκες επαλήθευσης που παράγονται με τη συνάρτηση wp δίνονται ως είσοδοι σε πολλά εξωτερικά εργαλεία απόδειξης. Υποστηρίζονται συστήματα υποστήριξης αποδείξεων (proof assistants) όπως τα Coq, Isabelle, PVS, HOL4, HOL Light, Mizar και διαδικασίες απόφασης (decision procedures) όπως οι CRC3, Simplify, Alt-Ergo, YICES.

3.1.3 Caduceus και Σχετικά Εργαλεία

Μπορεί κανείς να γράψει προγράμματα προς επαλήθευση απευθείας σε WL. Τα χαρακτηριστικά της όμως επιτρέπουν και μεταγλώττιση προγραμμάτων από διαδεδομένες γλώσσες σε αυτή. Το εργαλείο *Caduceus* [Fill04] επιτρέπει την επαλήθευση προγραμμάτων γραμμένων σε C με κατάλληλες επισημειώσεις, μεταγλωττίζοντας τα σε WL. Υποστηρίζεται ένα μεγάλο μέρος των χαρακτηριστικών της

ANSI C. Η μόνη δομή ελέγχου που δεν υποστηρίζεται είναι η εντολή `goto`. Αριθμητική δεικτών υποστηρίζεται επίσης, με εξαίρεση τις προσαρμογές (casts). Οι επισημειώσεις δίνονται μέσα σε σχόλια που ξεκινούν από `/*@`, ώστε να παραμένει ο κώδικας έγκυρη C. Κάθε πεδίο μιας δομής (struct) της C αντιστοιχεί σε διαφορετική μεταβλητή της WL. Ο σωρός και οι λειτουργίες που σχετίζονται με αυτόν μοντελοποιούνται με αφηρημένες δομές και πράξεις πάνω σε αυτές.

Παρόμοια λειτουργία με το Caduceus έχει η επέκταση (plugin) Jessie της πλατφόρμας για την ανάλυση πηγαίου κώδικα C Frama-C. Στην περίπτωση του Jessie οι προδιαγραφές γράφονται στη γλώσσα ANSI/ISO C Specification Language (ACSL), η οποία έχει προκύψει από τη γλώσσα προδιαγραφών του Caduceus. Η πλατφόρμα Why επιτρέπει την επαλήθευση προγραμμάτων γραμμένων σε Java με επισημειώσεις σε JML (Java Modeling Language) μέσω του εργαλείου Krakatoa. Και σε αυτή την περίπτωση τα προς επαλήθευση προγράμματα μεταφράζονται στην WL.

```
/*@ requires
  @   \valid_range(t,0,n-1)
  @ ensures
  @   (0 <= \result < n => t[\result] == v) &&
  @   (\result == n => \forall int i; 0 <= i < n => t[i] != v)
  @*/
int index(int t[], int n, int v)
{
  int i = 0;
  /*@ invariant
    @   0 <= i &&
    @   \forall int k; 0 <= k < i => t[k] != v
    @ variant
    @   n - i
    @*/
  while (i < n) {
    if (t[i] == v)
      break;
    i++;
  }
  return i;
}
```

Σχήμα 3.4: Συνάρτηση αναζήτησης σε λίστα με επισημειώσεις για το Caduceus

Ένα παράδειγμα συνάρτησης σε C με τις επισημειώσεις που απαιτεί το εργαλείο Caduceus παρουσιάζεται στο σχήμα 3.4. Η συνάρτηση `index` αναζητά μια τιμή σε έναν πίνακα και επιστρέφει την πρώτη θέση στην οποία την βρίσκει. Το σκέλος `requires` των επισημειώσεων είναι η προσυνθήκη και το `ensures` η μετασυνθήκη. Παρατηρούμε ότι μπορούν να οριστούν κατηγορήματα για τη διατύπωση των συνθηκών. Όπως και στα προγράμματα της WL οι βρόχοι επισημειώνονται με αναλλοίωτες αλλά και μεταβλητές που θα χρησιμοποιηθούν για αποδείξεις τερματισμού.

3.2 Coq

Το *Coq* [Bert04] είναι ένα σύστημα υποστήριξης αποδείξεων (*proof assistant*). Το Coq μας επιτρέπει να εκφράσουμε θεωρήματα και να αναπτύξουμε αποδείξεις σε ένα εκφραστικό σύστημα λογικής. Οι αποδείξεις αυτές αναπτύσσονται διαδραστικά, με τη βοήθεια αυτόματων τακτικών όταν αυτό είναι δυνατό. Το Coq είναι γραμμένο στη γλώσσα προγραμματισμού OCaml και αναπτύσσεται κυρίως από το Ινστιτούτο INRIA της Γαλλίας.

3.2.1 Κατασκευαστική Λογική και Ισομορφισμός Curry-Howard

Η προσέγγιση της λογικής που ακολουθεί το Coq ονομάζεται *κατασκευαστική* (constructive) ή *ιντουισιονιστική* (intuitionistic) και διαφέρει από την κλασική λογική. Για να περιγράψουμε τις διαφορές μεταξύ των δύο προσεγγίσεων χρησιμοποιούμε ένα παράδειγμα σε ένα απλό σύστημα λογικής, την ελάχιστη προτασιακή λογική (minimal propositional logic), στην οποία οι προτάσεις κατασκευάζονται με τη χρήση μόνο προτασιακών μεταβλητών και συνεπαγωγής. Έστω P, Q και R προτάσεις. Επιδιώκουμε να αποδείξουμε την πρόταση

$$(P \Rightarrow Q) \Rightarrow (Q \Rightarrow R) \Rightarrow (P \Rightarrow R) \tag{3.1}$$

Αντιμετωπίσουμε το παραπάνω πρόβλημα αρχικά με την κατασκευή ενός πίνακα αλήθειας. Αναθέτουμε μια τιμή t (αληθής) ή f (ψευδής) σε κάθε μεταβλητή και υπολογίζουμε την αντίστοιχη τιμή της πρότασης 3.1. Όπως φαίνεται στον πίνακα 3.5, το αποτέλεσμα είναι t για κάθε πιθανό συνδυασμό τιμών των P, Q και R, επομένως η πρόταση 3.1 ισχύει.

P	Q	R	$P \Rightarrow Q$	$Q \Rightarrow R$	$P \Rightarrow R$	$(Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$	$(P \Rightarrow Q) \Rightarrow (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$
f	f	f	t	t	t	t	t
f	f	t	t	t	t	t	t
f	t	f	t	f	t	t	t
f	t	t	t	t	t	t	t
t	f	f	f	t	f	f	t
t	f	t	f	t	t	t	t
t	t	f	t	f	f	t	t
t	t	t	t	t	t	t	t

Σχήμα 3.5: Πίνακας αλήθειας της πρότασης $(P \Rightarrow Q) \Rightarrow (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$

Κατά τον παραπάνω τρόπο αντιμετώπισης του προβλήματος, αναρωτηθήκαμε αν "η πρόταση 3.1 είναι αληθής" και όχι "ποιες είναι οι αποδείξεις της 3.1, αν υπάρχουν". Η πρώτη προσέγγιση αντιστοιχεί στην κλασική λογική, ενώ η δεύτερη στην ιντουισιονιστική. Στην κλασική λογική ισχύει ο κανόνας του αποκλεισμένου μέσου ($A \vee \neg A$), μια πρόταση είναι δηλαδή είτε αληθής είτε ψευδής. Αυτό δικαιολογεί τη χρήση πινάκων αλήθειας. Ο κανόνας αυτός δεν ισχύει στην ιντουισιονιστική λογική.

Καθώς το Coq ακολουθεί την ιντουισιονιστική προσέγγιση και μας ενδιαφέρει η ύπαρξη απόδειξης, είναι αναγκαίο να μπορούμε με κάποιο τρόπο να αναπαραστήσουμε όχι μόνο τις προτάσεις αλλά και τις αποδείξεις τους. Την ανάγκη αυτή την καλύπτουν οι παραλλαγές του λάμδα λογισμού με τύπους. Ο *Ισομορφισμός Curry-Howard* περιγράφει μια συντακτική αναλογία μεταξύ συστημάτων τύπων και συστημάτων λογικής. Σύμφωνα με τον ισομορφισμό, τα θεωρήματα είναι ισοδύναμα με τύπους και οι αποδείξεις με λάμδα όρους. Απόδειξη ενός θεωρήματος είναι ένας όρος που κατοικεί τον αντίστοιχο τύπο.

Στο σχήμα 3.6 δίνονται οι αντιστοιχίες μεταξύ στοιχείων της λογικής και χαρακτηριστικών των συστημάτων τύπων. Οι συναρτήσεις στην πλευρά των συστημάτων υπολογισμών αντιστοιχούν στην συνεπαγωγή στην πλευρά της λογικής. Διαισθητικά, αν έχουμε μια συνάρτηση f από τον τύπο που αντιστοιχεί σε ένα θεώρημα της λογικής (έστω A) προς τον τύπο ενός άλλου θεωρήματος (έστω B) και καλέσουμε τη συνάρτηση με όρισμα x τύπου A (απόδειξη του αντίστοιχου θεωρήματος), προκύπτει ένας όρος που κατοικεί το B ($fx : B$). Επομένως το A συνεπάγεται το B. Η λογική σύζευξη αντιστοιχεί σε τύπο γινομένου, ενώ ο τύπος αθροίσματος (έστω $A * B$), καθώς για να κατοικηθεί ο τύπος του γινομένου πρέπει να έχουμε κατασκευάσει όρους (αποδείξεις θεωρημάτων) τόσο για το A όσο και για το B. Η λογική διάζευξη αντιστοιχεί σε τύπο αθροίσματος (έστω $A + B$) διότι σε

Λογική	Συστήματα Τύπων	
	κατά Curry	κατά Church
καθολική ποσοτικοποίηση	τύπος εξαρτώμενου γινομένου	τύπος τομής
υπαρξιακή ποσοτικοποίηση	τύπος εξαρτώμενου αθροίσματος	τύπος ένωσης
συνεπαγωγή	τύπος συνάρτησης	
σύζευξη	τύπος γινομένου	
διάζευξη	τύπος αθροίσματος	
αληθής πρόταση	μοναδιαίος τύπος	
ψευδής πρόταση	κενός τύπος	

Σχήμα 3.6: Αντιστοιχίες σύμφωνα με τον Curry-Howard ισομορφισμό

αυτή την περίπτωση μας αρκεί όρος για ένα από τα A, B για να κατοικήσουμε τον τύπο. Η αληθής πρόταση (True) αντιστοιχεί στον μοναδιαίο τύπο, έναν τύπο ο οποίος κατοικείται από έναν μοναδικό όρο $() : unit$. Η ψευδής πρόταση αντιστοιχεί στον κενό τύπο, ο οποίος δεν κατοικείται από κανέναν όρο. Η αναπαραστάσεις της καθολικής ποσοτικοποίησης ($\forall$) και της υπαρξιακής ποσοτικοποίησης ($\exists$) διαφέρουν ανάμεσα στην εκδοχή του λάμδα λογισμού του Church και την εκδοχή του Curry.

3.2.2 Calculus of Inductive Constructions

Η θεωρία στην οποία βασιζόταν αρχικά το Coq είναι το *Calculus of Constructions* (CoC). Σήμερα βασίζεται στο *Calculus of Inductive Constructions* (CIC), το οποίο αποτελεί επέκταση του CoC με επαγωγικούς τύπους (inductive types).

Το CIC, όπως και το CoC, είναι ένας υψηλότερης τάξης λάμδα λογισμός. Οι τύποι είναι πρώτης τάξης τιμές (first-class values). Αυτό σημαίνει ότι μπορούμε να ορίσουμε συναρτήσεις από ακέραιους (λόγου χάρη) σε ακέραιους, από τύπους σε τύπους, από τύπους σε ακέραιους και φυσικά από ακέραιους σε ακέραιους. Το CIC είναι συγγενές με το σύστημα $\lambda P\omega$ στον Λάμδα Κύβο (Lambda Cube) του Henk Barendregt [Bare92]. Όπως και όλες οι εκδοχές του Λάμδα Λογισμού που παρουσιάζει ο Barendregt, το CIC είναι αυστηρά κανονικοποιήσιμο (strongly normalizing), δηλαδή κάθε ακολουθία αναγωγών (reductions) καταλήγει σε μια κανονική μορφή. Κατά συνέπεια, οι υπολογισμοί τερματίζονται πάντα και το CIC δεν είναι πλήρες κατά Turing (Turing complete). Κατά τον Curry-Howard ισομορφισμό το CIC αντιστοιχεί σε ένα σύστημα λογικής υψηλότερης τάξης.

3.2.3 Ένα παράδειγμα

```
Inductive le (n: nat): nat → Prop :=
  | le_n: le n n
  | le_S: ∀ m: nat, le n m → le n (S m).

Theorem le_trans : ∀ n m p, le n m → le m p → le n p.
Proof.
  induction 2;
  [ apply H | apply le_S; apply IHle ].
Qed.
```

Σχήμα 3.7: Επαγωγικός ορισμός της σχέσης μικρότερου ή ίσου στο Coq, και απόδειξη της μεταβατικής ιδιότητας

Στο σχήμα 3.7 ορίζουμε με επαγωγικό τρόπο το κατηγόρημα le για τη σχέση μικρότερου ή ίσου ($\leq$) μεταξύ δύο αριθμών. Ο *κατασκευαστής* (constructor) len ορίζει ότι ένας αριθμός είναι μικρότερος ή

ίσος από τον εαυτό του. Σύμφωνα με το σκέλος που αντιστοιχεί στον κατασκευαστή le_S, η ανισότητα $n \leq m$ συνεπάγεται την $n \leq m + 1$. Κατά τον Curry Howard ισομορφισμό, για να αποδείξουμε ότι ο *n* είναι μικρότερος του *m* πρέπει να βρούμε έναν όρο που κατοικεί τον τύπο le *n m*. Ένας τέτοιος όρος μπορεί να προκύψει μόνο από εφαρμογή ενός από τους κατασκευαστές, αντίστοιχα από εφαρμογή ενός από τους δύο κανόνες στην πλευρά της λογικής.

Το θεώρημα le_trans στο σχήμα 3.7 κωδικοποιεί τη μεταβατική ιδιότητα για τη σχέση μικρότερου ή ίσου. Αν $n \leq m$ (πρόταση le *n m*) και $m \leq p$ (πρόταση le *m p*), τότε θα ισχύει $n \leq p$ (πρόταση le *n p*). Οι εντολές ανάμεσα στις λέξεις - κλειδιά `Proof` και `Qed` αποτελούν ένα σενάριο απόδειξης (proof script), είναι δηλαδή οι τακτικές που ακολουθήσαμε για την απόδειξη του θεωρήματος. Για την εν λόγω απόδειξη χρησιμοποιήσαμε αναδρομή πάνω στη δεύτερη υπόθεση. Ένας όρος τύπου le *m p* θα έχει προκύψει από έναν εκ των δύο κατασκευαστών. Στην περίπτωση του κατασκευαστή len θα ισχύει *m* = *p*, αλλιώς θα υπάρχει *m0*, τέτοιο ώστε S *m0* = *m*. Θα πρέπει να αποδείξουμε πως le *m p* σε κάθε μια από τις δύο περιπτώσεις. Σε αυτές τις υποχρεώσεις αντιστοιχούν οι δύο υπο-στόχοι (σχήμα 3.8) που αφήνει ανοιχτούς η `induction` 2. Στην πρώτη περίπτωση, αρκει να εφαρμόσουμε την υπόθεση *H*. Στη δεύτερη περίπτωση, εφαρμόζουμε τον κατασκευαστή le_S ώστε να προκύψει νέος υπο-στόχος le *n m0*, ο οποίος γνωρίζουμε ότι ισχύει από την υπόθεση *IHle*.

Ο όρος που κατασκευάσαμε με αυτη την ακολουθία βημάτων είναι

```
fun (n m p : nat) (H : le n m) (H0 : le m p) ⇒
le_ind m (fun p0 : nat ⇒ le n p0) H
  (fun (m0 : nat) (_ : le m m0) (IHle : le n m0) ⇒ le_S n m0 IHle) p H0
```

Επισημαίνουμε ότι οι όροι - αποδείξεις και τα σενάρια απόδειξης είναι διαφορετικές έννοιες. Τα σενάρια απόδειξης μπορούμε να τα αντιληφθούμε ως γεννήτορες κώδικα που παράγουν όρους οι οποίοι κατοικούν τους τύπους που μας ενδιαφέρουν, δεν είναι όμως τα ίδια τέτοιοι όροι.

n	:	nat
m	:	nat
H	:	le *n m*
le *n m*		

n	:	nat
m	:	nat
H	:	le *n m*
m0	:	nat
H0	:	le *m m0*
IHle	:	le *n m0*
le *n* (S *m0*)		

Σχήμα 3.8: Ανοιχτοί υπο-στόχοι μετά από εφαρμογή της `induction` για το θεώρημα le_trans

Κεφάλαιο 4

Η γλώσσα Tony

Στο κεφάλαιο αυτό παρουσιάζουμε τη γλώσσα προγραμματισμού Tony. Η Tony ακολουθεί το προστακτικό πρότυπο, με αποτέλεσμα ο προγραμματισμός σε αυτή να φαίνεται οικείος στους περισσότερους προγραμματιστές. Η σύνταξη της γλώσσας Tony όμως υποστηρίζει άμεσα τη διατύπωση προδιαγραφών ορθότητας των προγραμμάτων, με τη μορφή προσυνθηκών και μετασυνθηκών. Το συντακτικό δέντρο του προγράμματος και οι συνθήκες αυτές σε ελάχιστες μόνο περιπτώσεις επαρκούν για να κατασκευάσει το εργαλείο επαλήθευσης μια απόδειξη μερικής ορθότητας. Ως εκ τούτου, η γλώσσα Tony επιτρέπει την ενσωμάτωση στα προγράμματα βοηθητικών θεωρημάτων που θα χρησιμοποιηθούν στη διαδικασία απόδειξης και επισημειώσεων με στοιχεία που το εργαλείο απόδειξης δεν μπορεί να μαντέψει (αναλλοίωτες βρόχου).

4.1 Σύνταξη

⟨letter⟩ ::= “**a**”- “**z**” | “**A**”- “**Z**”
⟨digit⟩ ::= “**0**”- “**9**”
⟨id⟩ ::= (⟨letter⟩ | ⟨digit⟩ | “_”)$^+$
⟨numconst⟩ ::= ⟨digit⟩$^+$
⟨binop⟩ ::= “+” | “−” | “∗” | “/” | “%”
⟨cmpop⟩ ::= “<” | “>” | “=” | “<=” | “>=”
⟨expr⟩ ::= ⟨id⟩
| ⟨numconst⟩
| ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
| “(” ⟨expr⟩ “)”
⟨cond⟩ ::= ⟨expr⟩ ⟨cmpop⟩ ⟨expr⟩
⟨stmt⟩ ::= “**skip**”
| ⟨id⟩ “=” ⟨expr⟩
| “**if**” ⟨cond⟩ ⟨stmt⟩ “**fi**”
| “**if**” ⟨cond⟩ ⟨stmt⟩ “**else**” ⟨stmt⟩ “**fi**”
| “**while**” ⟨cond⟩ ⟨stmt⟩ “**end**”
| ⟨stmt⟩ ⟨stmt⟩

Σχήμα 4.1: Η γλώσσα Tony

Στο σχήμα 4.1 δίνεται σε μορφή EBNF η σύνταξη του τμήματος υπολογισμών της γλώσσας Tony. Η γραμματική αυτή είναι διφορούμενη, οι αμφισημίες όμως μπορούν να λυθούν αν λάβουμε υπ’ όψιν πως οι αριθμητικοί τελεστές έχουν αριστερή προσεταιριστικότητα και πως οι πολλαπλασιαστικοί τελεστές (“∗”, “/”, “%”) έχουν μεγαλύτερη προτεραιότητα από τους προσθετικούς (“+”, “−”).

Όλες οι μεταβλητές της γλώσσας Tony είναι ακέραιες. Η γλώσσα παρέχει τελεστές για τις βασικές πράξεις με ακεραίους. Η εντολή `skip` δεν έχει καμία επίδραση. Για ανάθεση έκφρασης σε μεταβλητή χρησιμοποιούμε όπως στη C το σύμβολο =. Οι συνθήκες που εμφανίζονται στις εντολές `if` και `while` αφορούν σύγκριση μεταξύ εκφράσεων. Η εντολή `if` μπορεί να έχει ή όχι σκέλος `else`. Για να είναι πιο ευανάγνωστα τα προγράμματα και καθώς δεν υπάρχει συντακτικά ανάγκη για διαχωρισμό μεταξύ εντολών που εκτελούνται η μια μετά την άλλη, δεν χρησιμοποιούμε κάποιο σύμβολο όπως το ελληνικό ερωτηματικό. Σε κάθε σημείο του προγράμματος μπορούν να εμφανίζονται σχόλια με τη σύνταξη του Coq και της OCaml, δηλαδή μέσα σε παρενθέσεις με αστερίσκο: `(*` και `*)`. Τα σχόλια μπορούν να είναι φωλιασμένα.

4.2 Αξιωματική Σημασιολογία

Στο σχήμα 4.2 δίνεται η αξιωματική σημασιολογία της γλώσσας Tony. Οι δομές της γλώσσας είναι όμοιες με αυτές της προστακτικής γλώσσας της λογικής Hoare όπως παρουσιάστηκαν στο κεφάλαιο 2, ως εκ τούτου δεν τις περιγράφουμε αναλυτικά σε αυτό το σημείο. Η Tony παρέχει ως διευκόλυνση σύνταξη για εντολή `if` χωρίς σκέλος `else`. Το εργαλείο επαλήθευσης όμως απλά προσθέτει `skip` στο σκέλος `else`.

$$\{P\}\ \texttt{skip}\ \{P\} \qquad \text{κανόνας του skip}$$

$$\{P[E/V]\}\ \mathtt{V} = \mathtt{E}\ \{P\} \qquad \text{κανόνας ανάθεσης}$$

$$\frac{\{P\}\ \mathtt{S}_1\ \{R\},\ \ \{R\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \mathtt{S}_1\ \mathtt{S}_2\ \{Q\}} \qquad \text{κανόνας σύνθεσης}$$

$$\frac{\{P \wedge C\}\ \mathtt{S}_1\ \{Q\},\ \ \{P \wedge \neg C\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \texttt{if}\ \mathtt{C}\ \mathtt{S}_1\ \texttt{else}\ \mathtt{S}_2\ \texttt{fi}\ \{Q\}} \qquad \text{κανόνας του if}$$

$$\frac{\{P \wedge C\}\ \mathtt{S}_1\ \{Q\},\ \ P \wedge \neg C \Rightarrow Q}{\{P\}\ \texttt{if}\ \mathtt{C}\ \mathtt{S}_1\ \texttt{fi}\ \{Q\}} \qquad \text{κανόνας του if, χωρίς σκέλος } \texttt{else}$$

$$\frac{\{P \wedge C\}\ \mathtt{S}\ \{P\}}{\{P\}\ \texttt{while}\ \mathtt{C}\ \mathtt{S}\ \texttt{end}\ \{P \wedge \neg C\}} \qquad \text{κανόνας του while}$$

$$\frac{P \Rightarrow R,\ \ \{R\}\ \mathtt{C}\ \{Q\}}{\{P\}\ \mathtt{C}\ \{Q\}} \qquad \text{κανόνας ενδυνάμωσης προσυνθήκης}$$

$$\frac{\{P\}\ \mathtt{C}\ \{R\},\ \ R \Rightarrow Q}{\{P\}\ \mathtt{C}\ \{Q\}} \qquad \text{κανόνας αποδυνάμωσης μετασυνθήκης}$$

$$\frac{\{P_1\}\ \mathtt{C}\ \{Q_1\} \quad \{P_2\}\ \mathtt{C}\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \mathtt{C}\ \{Q_1 \wedge Q_2\}} \qquad \text{κανόνας σύζευξης}$$

$$\frac{\{P_1\}\ \mathtt{C}\ \{Q_1\} \quad \{P_2\}\ \mathtt{C}\ \{Q_2\}}{\{P_1 \vee P_2\}\ \mathtt{C}\ \{Q_1 \vee Q_2\}} \qquad \text{κανόνας διάζευξης}$$

Σχήμα 4.2: Η αξιωματική σημασιολογία της Tony

4.3 Επισημειώσεις

Στο σχήμα 4.1 παρουσιάσαμε το υπολογιστικό κομμάτι της γλώσσας Tony. Τώρα δίνουμε τις αναγκαίες επεκτάσεις της ώστε να είναι δυνατή η επαλήθευση των προγραμμάτων.

⟨assert⟩ ::= “{” ⟨constr⟩ “}”
⟨stmt⟩ ::= “**skip**”
| ⟨id⟩ “=” ⟨expr⟩
| “**if**” ⟨cond⟩ ⟨stmt⟩ “**fi**”
| “**if**” ⟨cond⟩ ⟨stmt⟩ “**else**” ⟨stmt⟩ “**fi**”
| “**while**” ⟨cond⟩ ⟨assert⟩ ⟨stmt⟩ “**end**”
| ⟨stmt⟩ ⟨stmt⟩
⟨triple⟩ ::= ⟨assert⟩ ⟨stmt⟩ ⟨assert⟩
⟨header⟩ ::= “**Program**” ⟨id⟩
⟨prog⟩ ::= ⟨vncmd⟩* “**Tactic**” ⟨id⟩ ⟨header⟩ ⟨triple⟩

Σχήμα 4.3: Σύνταξη της Tony με επισημειώσεις

Η σύνταξη των εκφράσεων (μη τερματικό σύμβολο ⟨expr⟩) και των συνθηκών (μη τερματικό σύμβολο ⟨cond⟩) δεν αλλάζει σε σχέση με το σχήμα 4.2. Το μη τερματικό σύμβολο ⟨constr⟩ αναφέρεται σε όρους της *Gallina*, της γλώσσας προδιαγραφών (specification language) του Coq. Το μη τερματικό σύμβολο ⟨vncmd⟩ αναφέρεται σε προτάσεις του *Vernacular*, της γλώσσας εντολών του Coq.

Κάθε πρόγραμμα της Tony είναι επισημειωμένο με μια προσυνθήκη και μια μετασυνθήκη, η σημασία των οποίων είναι αυτή της λογικής Hoare. Η προσυνθήκη δηλώνει τις υποχρεώσεις του χρήστη του προγράμματος και εκφράζει την κατάσταση εκτέλεσης που αυτό αναμένει. Η μετασυνθήκη αποτελεί διατύπωση της επιθυμητής κατάστασης εκτέλεσης μετά την εκτέλεση του προγράμματος και εφόσον αυτό τερματιστεί. Οι βρόχοι στη Tony είναι υποχρεωτικά επισημειωμένοι με μια αναλλοίωτη, μια συνθήκη δηλαδή που θα ισχύει μετά από κάθε επανάληψη. Θα ήταν πολύ δύσκολο ή αδύνατο το πρόγραμμα επαλήθευσης να ανακαλύπτει κατάλληλες αναλλοίωτες χωρίς τη βοήθεια του προγραμματιστή. Οι προσυνθήκες, μετασυνθήκες και αναλλοίωτες βρόχου γράφονται μέσα σε άγκιστρα με τη σύνταξη της Gallina [1]. Μέσα στα άγκιστρα ο προγραμματιστής αναφέρεται στις τιμές των μεταβλητών του προγράμματος με τη συνάρτηση *e* τύπου *string* → *Z*.

Το τμήμα του προγράμματος που προηγείται του τερματικού συμβόλου “**Tactic**” περιλαμβάνει αυθαίρετο κώδικα Coq. Ο χρήστης καλείται να συμπεριλάβει θεωρήματα τα οποία δεν μπορούν να αποδειχτούν αυτόματα, είναι όμως απαραίτητα για να επαληθευτεί το πρόγραμμα[2]. Σε αυτό το τμήμα είναι δυνατό να οριστούν επαγωγικοί τύποι ή κατηγορήματα που εμφανίζονται στην προσυνθήκη, τη μετασυνθήκη και τις αναλλοίωτες βρόχου. Το αναγνωριστικό που ακολουθεί το τερματικό σύμβολο “**Tactic**” είναι η τακτική με την οποία θα επιχειρήσει το εργαλείο επαλήθευσης να επιλύσει τις υποχρεώσεις απόδειξης που θα προκύψουν. Αυτή στα περισσότερα προγράμματα είναι η *verify* που συνοδεύει το εργαλείο επαλήθευσης. Δυνητικά ο χρήστης μπορεί να ορίσει δική του τακτική που αυτοματοποιεί την διαδικασία απόδειξης θεωρημάτων σε συγκεκριμένο πεδίο εφαρμογών, και κατά συνέπεια διευκολύνει την επαλήθευση των αντίστοιχων προγραμμάτων. Το Coq παρέχει τη γλώσσα *Ltac* για την υλοποίηση τακτικών.

4.4 Παραδείγματα Προγραμμάτων

4.4.1 Υπολογισμός Μεγίστου

Στο σχήμα 4.4 δίνεται ένα πρόγραμμα σε Tony για τον υπολογισμό του μεγίστου των A, B. Δεν έχουμε κάποια απαίτηση για τις τιμές των A, B και επομένως η προσυνθήκη έχει την τιμή *True*. Αρχικά οι μεταβλητές a και b αρχικοποιούνται με τις τιμές των A και B. Εκχωρούμε στη μεταβλητή

[1] με τις συντακτικές διευκολύνσεις που παρέχονται στο scope *Z_scope*

[2] το εργαλείο επαλήθευσης θα προσπαθήσει να εκμεταλλευτεί τα θεωρήματα που έχουν εισαχθεί στη βάση συμβουλών *myhints*.

```
Tactic verify

Program max
{ True }
  a = A
  b = B
  if a < b
    max = b
  else
    max = a
  fi
{ (e "max" = e "A" /\ e "max" >= e "B") \/
  (e "max" = e "B" /\ e "max" >= e "A")
}
```

Σχήμα 4.4: Υπολογισμός μεγίστου στη γλώσσα Tony

max την τιμή της b, αν ισχύει $a < b$. Σε αντίθετη περίπτωση η max λαμβάνει την τιμή της a. Η μετασυνθήκη αποτελεί διατύπωση της επιθυμητής λειτουργίας του προγράμματος. Σύμφωνα με αυτή, η μεταβλητή max θα έχει την τιμή μιας εκ των δύο μεταβλητών και θα είναι μεγαλύτερη ή ίση της άλλης. Το πρόγραμμα αυτό είναι πολύ απλό, με αποτέλεσμα η επαλήθευσή του να επιτυγχάνει χωρίς να χρειάζεται ο προγραμματιστής να εισάγει κάποιο βοηθητικό θεώρημα.

4.4.2 Μέγιστος Κοινός Διαιρέτης

Στο σχήμα 4.5 δίνεται πρόγραμμα για τον υπολογισμό του μέγιστου κοινού διαιρέτη των μη αρνητικών αριθμών A και B. Το πρόγραμμα υλοποιεί την εκδοχή του αλγόριθμου του Ευκλείδη με υπόλοιπο διαίρεσης αντί για πολλαπλές αφαιρέσεις. Ο αλγόριθμος βασίζεται στην ιδέα ότι

$$gcd(a, b) = gcd(a \ mod \ b, b), \text{όπου } b > 0. \tag{4.1}$$

Το θεώρημα *gcd_euclid_mod* που εμφανίζεται στον κώδικα του σχήματος 4.5 αποτελεί διατύπωση αυτής ακριβώς της ισότητας. Το κατηγόρημα *Zis_gcd* ορίζεται στην ενότητα *Znum_theory* της βιβλιοθήκης του Coq· η πρόταση *Zis_gcd a b c* εκφράζει ότι ο αριθμός c είναι ο μέγιστος κοινός διαιρέτης των a και b. Για την απόδειξη του *gcd_euclid_mod* εφαρμόσαμε το *Zis_gcd_for_euclid* της *Znum_theory*, σύμφωνα με το οποίο $gcd(a, b) = gcd(a - q * b, b)$ με q ακέραιο.

Οι μεταβλητές a, b αρχικοποιούνται με τις τιμές των A και B αντίστοιχα. Οι τιμές των a και b μετά την ν-οστή επανάληψη, έστω a_ν και b_ν, είναι ίσες με $a_{\nu-1} \ mod \ b_{\nu-1}$ και $b_{\nu-1}$. Ο μέγιστος κοινός διαιρέτης των νέων τιμών είναι ίδιος με των παλαιών σύμφωνα με την 4.1. Η αναλλοίωτη βρόχου δηλώνει ότι ο μέγιστος κοινός διαιρέτης των a, b παραμένει ίσος με των A, B, ενώ οι a και b παραμένουν μεγαλύτερες ή ίσες του μηδενός. Το εργαλείο επαλήθευσης αποδεικνύει ότι διατηρείται το πρώτο σκέλος της αναλλοίωτης εφαρμόζοντας το *gcd_euclid_mod*. Για να αποδειχθεί η διατήρηση του δεύτερου σκέλους ($b \geq 0 \vee a \geq 0$) χρειάζεται να εφαρμοστεί το θεώρημα *mod_ge0*, σύμφωνα με το οποίο για $b > 0$ ισχύει $a \ mod \ b \geq 0$.

Ο βρόχος τερματίζεται όταν το b γίνει μηδέν. Τότε ο μέγιστος κοινός διαιρέτης είναι ίσος με a, όπως δηλώνει και η μετασυνθήκη. Αυτό μπορεί να αποδειχθεί καθώς $gcd(a, b) = a$ αν $b = 0$. Το αντίστοιχο θεώρημα στο Coq είναι το *gcd_b_0*. Τα τρια θεωρήματα που έχει αποδείξει ο προγραμματιστής αρκούν για να κατασκευάσει το εργαλείο επαλήθευσης μια πλήρη απόδειξη μερικής ορθότητας.

```
Require Import Coq.ZArith.Znumtheory.
Open Scope string_scope.

Theorem gcd_euclid_mod: forall a b c,
  b > 0 -> Zis_gcd b (a mod b) c -> Zis_gcd a b c.
Proof.
  intros.
  apply Zis_gcd_for_euclid with (q := a / b).
  rewrite Zmult_comm.
  pattern a at 1.
  rewrite Z_div_mod_eq with (b := b).
  replace (b * (a / b) + a mod b - b * (a / b)) with
          (a mod b).
  apply H0.
  ring.
  apply H.
Qed.

Theorem mod_ge0: forall a b : Z, b > 0 -> a mod b >= 0.
Proof.
  intros.
  apply Zle_ge.
  assert (0 <= a mod b < b).
  apply Z_mod_lt; apply H.
  elim H0; intros; apply H1.
Qed.

Theorem gcd_b_0: forall a b, b = 0 -> Zis_gcd a b a.
Proof.
  intros a b H; rewrite H; apply Zis_gcd_0.
Qed.

Hint Resolve gcd_euclid_mod mod_ge0 gcd_b_0: myhints.

Tactic verify

Program gcd
{ e "A" >= 0 /\ e "B" >= 0 }
  a = A
  b = B
  while b > 0 { ( forall x,
                    Zis_gcd (e "a") (e "b") x ->
                    Zis_gcd (e "A") (e "B") x
                )
                /\ e "b" >= 0 /\ e "a" >= 0 }
    r = a % b
    a = b
    b = r
  end
{ Zis_gcd (e "A") (e "B") (e "a") }
```

Σχήμα 4.5: Υπολογισμός ελάχιστου κοινού διαιρέτη στη γλώσσα Tony

```
Tactic verify

Program power_trivial
{ e "n" >= 0 /\ e "a" > 0 }
  r = 1
  i = 0
  while i < n { e "r" = (e "a") ^ (e "i") /\
                0 <= e "i" <= e "n" /\
                e "a" > 0
              }
      r = r * a
      i = i + 1
  end
{ e "r" = (e "a") ^ (e "n") }
```

Σχήμα 4.6: Ημιτελές πρόγραμμα ύψωσης σε δύναμη στην Tony

4.4.3 Ύψωση σε Δύναμη

Στο σχήμα 4.6 δίνεται πρόγραμμα για τον υπολογισμό της δύναμης a^n στη γλώσσα Tony. Η μέθοδος που ακολουθούμε είναι απλοϊκή, και περιλαμβάνει n πολλαπλασιασμούς του ενδιάμεσου αποτελέσματος r με a. Το r έχει αρχικοποιηθεί με 1, ενώ η μεταβλητή i χρησιμοποιείται ως μετρητής επαναλήψεων. Αναμένουμε, σύμφωνα με την προσυνθήκη, το n να είναι μη αρνητικό και το a μεγαλύτερο του μηδενός. Σύμφωνα με την αναλλοίωτη του βρόχου, το r θα έχει μετά από κάθε επανάληψη την τιμή a^i, το i θα παραμένει στο διάστημα $[0...n]$ και το a θα παραμένει μεγαλύτερο του 0. Η μετασυνθήκη δηλώνει πως το r θα είναι ίσο με το επιθυμητό αποτέλεσμα. Έχουμε παρουσιάσει μια απόδειξη ορθότητας του ίδιου προγράμματος στην ενότητα 2.4.

Ο κώδικας δεν περιλαμβάνει κανένα θεώρημα του Coq. Το εργαλείο επαλήθευσης, όπως είναι αναμενόμενο, δεν καταφέρνει να κατασκευάσει μια απόδειξη μερικής ορθότητας, καθώς δεν καταφέρνει να επιλύσει *υποχρεώσεις απόδειξης* που εμφανίζονται στο σχήμα 4.7. Για τη διευκόλυνση του προγραμματιστή, το εργαλείο επαλήθευσης μπορεί να παράγει ένα πρότυπο (template) με κώδικα Vernac προς συμπλήρωση. Αν ο προγραμματιστής συμπληρώσει σε αυτό το πρότυπο τα βήματα των αποδείξεων (proof scripts) και τις συμπεριλάβει στον κώδικά του, η διαδικασία της επαλήθευσης ολοκληρώνεται με επιτυχία. Για το πρόγραμμα 4.6, το συμπληρωμένο πρότυπο δίνεται στο σχήμα 4.8.

f	:	*env_f*	f	:	*env_f*
H1	:	f "i" < f "n"	*H1*	:	f "i" < f "n" → *False*
H	:	f "r" = f "a" ^ f "i"	*H*	:	f "r" = f "a" ^ f "i"
H3	:	f "a" > 0	*H3*	:	f "a" > 0
H2	:	$0 \leq f$ "i"	*H2*	:	$0 \leq f$ "i"
H4	:	f "i" $\leq f$ "n"	*H4*	:	f "i" $\leq f$ "n"
f "r" * f "a" = f "a" ^ (f "i" + 1)			f "r" = f "a" ^ f "n"		

Σχήμα 4.7: Ανεπίλυτες υποχρεώσεις απόδειξης για το πρόγραμμα 4.6

```
Require Import Zpow_facts.
Theorem h1 :
∀f : env_f,
(f "r" = f "a" ^ f "i" ∧ 0 ≤ f "i" ≤ f "n" ∧ f "a" > 0) ∧ f "i" < f "n" →
f "r" × f "a" = f "a" ^ (f "i" + 1) ∧ 0 ≤ f "i" + 1 ≤ f "n" ∧ f "a" > 0.
Proof.
  (* human input begin *)
  intuition.
  rewrite H.
  replace (f "i" + 1) with (Zsucc (f "i")).
  rewrite Zpower_Zsucc.
  auto with zarith.
  apply H2.
  auto with zarith.
  (* human input end *)
Qed.
Theorem h2 :
∀f : env_f,
(f "r" = f "a" ^ f "i" ∧ 0 ≤ f "i" ≤ f "n" ∧ f "a" > 0) ∧
¬ f "i" < f "n" → f "r" = f "a" ^ f "n".
Proof.
  (* human input begin *)
  intuition.
  replace (f "n") with (f "i").
  apply H.
  auto with zarith.
  (* human input end *)
Qed.
```

Σχήμα 4.8: Κώδικας Vernac προς συμπλήρωση για το πρόγραμμα 4.6

Κεφάλαιο 5

Μηχανή Αποδείξεων

5.1 Το συντακτικό δέντρο της Tony στο Coq

Πριν εκφράσουμε στο Coq την αξιωματική σημασιολογία της γλώσσας Tony, είναι απαραίτητο να κωδικοποιήσουμε τις συντακτικές δομές της γλώσσας μας σε αυτό. Η κωδικοποίηση αυτή είναι παρόμοια με την αναπαράσταση του αφηρημένου συντακτικού δέντρου σε έναν μεταγλωττιστή. Ο επαγωγικός τύπος expr αναπαριστά εκφράσεις της Tony στο Coq. Ο κατασκευαστής e_var με όρισμα μια ακολουθία χαρακτήρων αντιστοιχεί σε μεταβλητές της Tony. Ο κατασκευαστής e_num αντιστοιχεί στις ακέραιες σταθερές, για την αναπαράσταση των οποίων χρησιμοποιούμε τον τύπο Z του Coq όπως αυτός ορίζεται στην ενότητα *Coq.ZArith* της βιβλιοθήκης. Η βιβλιοθήκη του Coq παρέχει πλήθος αξιωμάτων για αριθμητική στο $\mathbb{Z}$, τα οποία μπορεί να εκμεταλλευτεί ο προγραμματιστής της Tony για την επαλήθευση προγραμμάτων. Οι υπόλοιποι κατασκευαστές αφορούν πρόσθεση, αφαίρεση, πολλαπλασιασμό, ακέραια διαίρεση και ακέραιο υπόλοιπο διαίρεσης:

```
Inductive expr: Type :=
  | e_var (s: string)
  | e_num (z: Z)
  | e_plus (e1 e2: expr)
  | e_minus (e1 e2: expr)
  | e_mult (e1 e2: expr)
  | e_div (e1 e2: expr)
  | e_mod (e1 e2: expr).
```

Ο επαγωγικός τύπος cond αναπαριστά τις συνθήκες που εμφανίζονται στις δομές `if` και `while` της Tony. Ορίζουμε κατασκευαστές για τις συνήθεις συγκρίσεις μεταξύ αριθμητικών εκφράσεων. Οι δύο παράμετροι κάθε κατασκευαστή έχουν τύπο expr.

```
Inductive cond: Type :=
  | c_eq (e1 e2: expr)
  | c_lt (e1 e2: expr)
  | c_gt (e1 e2: expr)
  | c_leq (e1 e2: expr)
  | c_geq (e1 e2: expr).
```

Ορίζουμε τον τύπο των συναρτήσεων κατάστασης env_f ως string → Z. Όπως έχει αναφερθεί σε προηγούμενο κεφάλαιο, οι προσυνθήκες, μετασυνθήκες και αναλλοίωτες βρόχου αποτελούν κατηγορήματα στην κατάσταση του προγράμματος. Ο τύπος assert για τους ισχυρισμούς αυτούς ορίζεται ως env_f → `Prop`.

Ο επαγωγικός τύπος instr αντιστοιχεί στις εντολές της γλώσσας Tony. Ο κατασκευαστής i_skip αναπαριστά την εντολή `skip`. Ο κατασκευαστής i_let αναπαριστά τις εντολές ανάθεσης και αναμένει ως ορίσματα το string του ονόματος της μεταβλητής στην οποία εκχωρούμε και την έκφραση (τύπου expr) στα δεξιά του συμβόλου ανάθεσης. Ο κατασκευαστής i_seq αναπαριστά ακολουθία δύο εντολών. Ο κατασκευαστής i_if αντιστοιχεί στην εντολή `if`, με ορίσματα τη συνθήκη τύπου cond και τις εντολές των περιπτώσεων `true` και `false` (αν η εντολή στο αρχικό πρόγραμμα δεν έχει

τμήμα `else` η δεύτερη εντολή είναι i_skip). Τέλος, ο κατασκευαστής i_while αναπαριστά της εντολές επανάληψης και έχει ως ορίσματα τη συνθήκη, την αναλλοίωτη βρόχου (τύπου assert) και την εντολή στο εσωτερικό.

```
Inductive instr: Type :=
  | i_skip
  | i_let (s: string) (e: expr)
  | i_seq (i1 i2: instr)
  | i_if (c: cond) (i1 i2: instr)
  | i_while (c: cond) (a: assert) (i: instr).
```

5.2 Αξιωματική Σημασιολογία στο Coq

5.2.1 Απαραίτητοι Ορισμοί

Σε κάποιους από τους κανόνες της λογικής Hoare, όπως για παράδειγμα ο κανόνας ενίσχυσης της προσυνθήκης (εξίσωση 2.6), εμφανίζεται σχέση συνεπαγωγής μεταξύ δύο συνθηκών. Στο Coq την ορίζουμε ως εξής:

```
Definition impl_a (a1 a2: assert) := ∀ f, a1 f → a2 f.
```

Στον ορισμό της σχέσης impl_a έχουμε καθολική ποσοτικοποίηση για τη συνάρτηση κατάστασης, πρέπει δηλαδή η πρώτη συνθήκη να συνεπάγεται τη δεύτερη ανεξάρτητα από τις τιμές των μεταβλητών του προγράμματος.

Ορίζουμε συναρτήσεις (fixpoint) αποτίμησης για τις εκφράσεις (expr) και τις συνθήκες (cond). Η συνάρτηση eval_e για την αποτίμηση των εκφράσεων επιστρέφει ως αποτέλεσμα έναν ακέραιο, ενώ η eval_c επιστρέφει μια πρόταση (`Prop`). Ακολουθεί η υλοποίησή τους.

```
Fixpoint eval_e (e: expr) (f: env_f): Z :=
  match e with
  | e_var v ⇒ f v
  | e_num n ⇒ n
  | e_plus e1 e2 ⇒ (eval_e e1 f) + (eval_e e2 f)
  | e_minus e1 e2 ⇒ (eval_e e1 f) - (eval_e e2 f)
  | e_mult e1 e2 ⇒ (eval_e e1 f) × (eval_e e2 f)
  | e_div e1 e2 ⇒ (eval_e e1 f) / (eval_e e2 f)
  | e_mod e1 e2 ⇒ (eval_e e1 f) mod (eval_e e2 f)
  end.
Fixpoint eval_c (c: cond) (f: env_f): Prop :=
  match c with
  | c_lt e1 e2 ⇒ (eval_e e1 f) < (eval_e e2 f)
  | c_gt e1 e2 ⇒ (eval_e e1 f) > (eval_e e2 f)
  | c_leq e1 e2 ⇒ (eval_e e1 f) ≤ (eval_e e2 f)
  | c_geq e1 e2 ⇒ (eval_e e1 f) ≥ (eval_e e2 f)
  | c_eq e1 e2 ⇒ (eval_e e1 f) = (eval_e e2 f)
  end.
```

Ο κανόνας της ανάθεσης (εξίσωση 2.2) απαιτεί να ορίσουμε την αντικατάσταση μιας μεταβλητής, έστω i, με μια έκφραση e σε συνθήκη a (re_a a i e). Η αντικατάσταση αυτή δεν μπορεί να πραγματοποιηθεί σε συντακτικό επίπεδο, διασχίζοντας δηλαδή το δέντρο της συνθήκης και αντικαθιστώντας τα κλαδιά που αντιστοιχούν στη μεταβλητή με το συντακτικό δέντρο της έκφρασης. Αντίθετα, η αντικατάσταση γίνεται με επέμβαση στη συνάρτηση τύπου env_f που αναμένει ως όρισμα ο ισχυρισμός a (τύπου assert). Έστω g (συνάρτηση κατάστασης τύπου string → Z) το όρισμα του κατηγορήματος που επιστρέφουμε ως αποτέλεσμα. Η αποτίμηση της συνθήκης a δεν γίνεται με εφαρμογή της g, αλλά με

την εφαρμογή συνάρτησης που επιστρέφει το αποτέλεσμα της e για μεταβλητή ονόματος i, την τιμή $g\ i$ σε αντίθετη περίπτωση. Η σύγκριση των string για να διαπιστώσουμε την ταύτιση των ονομάτων γίνεται με χρήση της string_dec που παρέχει η βιβλιοθήκη του Coq. Ο αντίστοιχος κώδικας είναι

```
Definition re_a (a: assert) (i: string) (e: expr): assert :=
  (fun g ⇒ a (fun j ⇒ if string_dec j i then (eval_e e g) else g j)).
```

Οι αντικαταστάσεις με την re_a δημιουργούν αρκετά περίπλοκους όρους για τους ισχυρισμούς (assert). Οι περιττές λάμδα αφαιρέσεις (abstractions) εξαφανίζονται με την τακτική `simpl` που απλοποιεί τους όρους πραγματοποιώντας τους δυνατούς υπολογισμούς (αναγωγές του λάμδα λογισμού).

5.2.2 Το κατηγόρημα `axiom`

```
Inductive axiom: assert → instr → assert → Prop :=
  | x_skip: ∀ p,
      axiom p i_skip p
  | x_seq: ∀ p q r i1 i2,
      axiom p i1 q →
      axiom q i2 r →
      axiom p (i_seq i1 i2) r
  | x_let: ∀ p v e,
      axiom (re_a p v e) (i_let v e) p
  | x_while: ∀ a c i,
      axiom (fun f ⇒ a f ∧ eval_c c f) i a →
      axiom a (i_while c a i) (fun f ⇒ a f ∧ ¬ eval_c c f)
  | x_if: ∀ p q i1 i2 c,
      axiom (fun f ⇒ p f ∧ eval_c c f) i1 q →
      axiom (fun f ⇒ p f ∧ ¬ eval_c c f) i2 q →
      axiom p (i_if c i1 i2) q
  | x_pre: ∀ (p q r: assert) c,
      impl_a p r → axiom r c q → axiom p c q
  | x_post: ∀ (p r q: assert) c,
      impl_a r q → axiom p c r → axiom p c q
  | x_conj: ∀ p1 p2 q1 q2 c,
      axiom p1 c q1 → axiom p2 c q2 →
      axiom (fun f ⇒ p1 f ∧ p2 f) c (fun f ⇒ q1 f ∧ q2 f)
  | x_disj: ∀ p1 p2 q1 q2 c,
      axiom p1 c q1 → axiom p2 c q2 →
      axiom (fun f ⇒ p1 f ∨ p2 f) c (fun f ⇒ q1 f ∨ q2 f).
```

Σχήμα 5.1: Τα αξιώματα και η κανόνες της λογικής Hoare στο Coq

Ο επαγωγικός ορισμός του σχήματος 5.1 κωδικοποιεί τους κανόνες της λογικής Hoare στο Coq. Η πρόταση axiom $p\ c\ q$, με p, q ισχυρισμούς (assert) και c εντολή (instr) αποτελεί διατύπωση της τριάδας Hoare με προσυνθήκη p, πρόγραμμα c και μετασυνθήκη q. Το axiom $p\ c\ q$ είναι ένας τύπος που ανήκει στο είδος (kind) `Prop`. Σύμφωνα με όσα έχουν ειπωθεί στην ενότητα 3.2.1, η απόδειξη της πρότασης ισοδυναμεί με κατασκευή ενός όρου που κατοικεί τον αντίστοιχο τύπο. Ένας τέτοιος όρος θα προκύψει με εφαρμογή κάποιου από τους κατασκευαστές x_seq κ.ο.κ., καθένας εκ των οποίων αντιστοιχεί σε έναν από τους κανόνες της λογικής Hoare (τύποι 2.1 ως 2.9).

Ο κατασκευαστής x_skip δηλώνει ότι ισχύει η πρόταση axiom p i_skip p με καθολική ποσοτικοποίηση για τον ισχυρισμό p. Δεν προκύπτουν επιπλέον υποχρεώσεις απόδειξης από την εφαρμογή του.

Το σκέλος που αντιστοιχεί στον κατασκευαστή x_seq δηλώνει ότι η ισχύς των προτάσεων axiom *p i1 q* και axiom *q i2 r* συνεπάγεται ισχύ της πρότασης axiom *p* (i_seq *i1 i2*) *r*, για κάθε τιμή των εντολών (*i1* και *i2*) και των ισχυρισμών (*p*, *q*, και *r*). Εφαρμογή του κανόνα οδηγεί σε δύο υποχρεώσεις απόδειξης. Με άγνωστη την ενδιάμεση συνθήκη *q* δεν μπορούμε να εφαρμόσουμε άμεσα τον κατασκευαστή x_seq, με την τακτική `apply`. Η εντολή `eapply` x_seq μας δίνει τη δυνατότητα να προχωρήσουμε την απόδειξη με μια υπαρξιακή μεταβλητή για την *q*. Μπορούμε λοιπόν πρώτα να προσπαθήσουμε να αντιμετωπίσουμε την υποχρέωση απόδειξης που αντιστοιχεί στη δεύτερη εντολή της ακολουθίας, και η στρατηγική που θα εφαρμόσουμε να οδηγήσει σε συγκεκριμένη τιμή της *q*.

Το σκέλος του κατασκευαστή x_let αποτελεί διατύπωση του κανόνα της ανάθεσης. Για κάθε ισχυρισμό *p*, μεταβλητή *v* και έκφραση *e* ισχύει axiom (re_a *p v e*) (i_let *v e*) *p*. Με βάση τη μορφή του κανόνα, αν προσπαθούμε να αποδείξουμε μια πρόταση της μορφής axiom *?q* (i_let *v e*) *p*, με γνωστή δηλαδή μετασυνθήκη αλλά υπαρξιακή μεταβλητή *?q* για την προσυνθήκη, είναι εύκολο να μαντέψουμε κατάλληλη τιμή της προσυνθήκης. Σε περίπτωση που η προσυνθήκη είναι γνωστή, είναι μάλλον απίθανο να ταυτίζεται δομικά με τον όρο re_a *p v e*. Θα χρειαστεί να εφαρμόσουμε κανόνα ενδυνάμωσης της προσυνθήκης και να αποδείξουμε ότι η προσυνθήκη συνεπάγεται την re_a *p v e*.

Ο κατασκευαστής x_while κωδικοποιεί τον κανόνα του `while`. Η πρόταση στα αριστερά της συνεπαγωγής (→) εγγυάται διατήρηση της αναλλοιωτης βρόχου. Δεν έχουμε καθολική ποσοτικοποίηση για τη μετασυνθήκη της `while` κατά τον κανόνα αυτόν, αλλά αυτή προκύπτει από σύζευξη της αναλλοίωτης βρόχου με την άρνηση της συνθήκης του βρόχου (`fun` $f \Rightarrow a\,f \wedge \neg$ eval_c $c\,f$). Ενδεχομένως να πρέπει να αποδείξουμε ότι η συνθήκη αυτή συνεπάγεται κάποια γνωστή μετασυνθήκη (με εφαρμογή του κανόνα αποδυνάμωσης μετασυνθήκης της λογικής Hoare). Από τη δομή του κανόνα προκύπτει επίσης συγκεκριμένη προσυνθήκη, αν αυτή δεν είναι γνωστή.

Κατά τον κατασκευαστή x_if, ισχύς των προτάσεων axiom (`fun` $f \Rightarrow p\,f \wedge$ eval_c $c\,f$) *i1 q* και axiom (`fun` $f \Rightarrow p\,f \wedge \neg$ eval_c $c\,f$) *i2 q* συνεπάγεται ισχύ της axiom *p c q* . Οι δύο υποχρεώσεις απόδειξης εξασφαλίζουν πως θα καταλήξουμε σε αλήθεια της *q* είτε είναι αληθής η *c* και εκτελεστεί η *i1*, είτε είναι ψευδής και εκτελεστεί η *i2*. Για να εφαρμόσουμε τον κανόνα πρέπει να ανακαλύψουμε κατάλληλη προσυνθήκη *p*. Αν καταλήξουμε σε προσυνθήκες *p1* και *p2* για τις εντολές *i1* και *i2* αντίστοιχα, κατάλληλη προσυνθήκη είναι η (`fun` $f \Rightarrow$ (eval_c $c\,f \rightarrow p1\,f$) $\wedge$ (˜eval_c $c\,f \rightarrow p2\,f$)) .

Το σκέλος που αντιστοιχεί στον κατασκευαστή x_pre κωδικοποιεί τον κανόνα ενδυνάμωσης της προσυνθήκης της λογικής Hoare. Μπορούμε να αντικαταστήσουμε την προσυνθήκη *p* με κάποια *r* αρκεί να κατασκευάσουμε μια απόδειξη πως η δεύτερη συνεπάγεται την πρώτη (impl_a *r p*). Ο κανόνας αυτός εφαρμόζεται συχνότατα αν ακολουθούμε μια στρατηγική απόδειξης που με βάση τις μετασυνθήκες και τις εντολές μαντεύει κατάλληλες προσυνθήκες (σαν αυτές που θα προέκυπταν από μια συνάρτηση ασθενέστερης προσυνθήκης). Χρειάζεται σε τέτοιες περιπτώσεις να εφαρμόσουμε κανόνα ενδυνάμωσης και να αποδείξουμε ότι η προσυνθήκη που δίνεται ως επισημείωση συνεπάγεται αυτή που έχει προκύψει από τη στρατηγική επαλήθευσης.

Αντίστοιχα με τον κατασκευαστή x_pre για την περίπτωση ενδυνάμωσης της προσυνθήκης, ο x_post κωδικοποιεί τον κανόνα αποδυνάμωσης της μετασυνθήκης. Θα εφαρμόζαμε πολύ συχνά τον κανόνα με μια στρατηγική απόδειξης που από γνωστές προσυνθήκες υπολογίζει ισχυρότερες μετασυνθήκες (strongest postconditions). Με τη διαδικασία επαλήθευσης που ακολουθούμε στην εργασία αυτή, κανόνας αποδυνάμωσης της μετασυνθήκης θα χρησιμοποιηθεί στην περίπτωση των εντολών `while`.

Ο κατασκευαστής x_conj αποτελεί διατύπωση του κανόνα της σύζευξης της λογικής Hoare. Ισχύς των προτάσεων axiom *p1 c q1* και axiom *p2 c q2* συνεπάγεται ισχύ της πρότασης axiom (`fun` $f \Rightarrow p1\,f \wedge p2\,f$) *c* (`fun` $f \Rightarrow q1\,f \wedge q2\,f$). Προσυνθήκη είναι η σύζευξη των *p1*, *p2* και μετασυνθήκη η σύζευξη των *q1*, *q2*. Αντίστοιχα, ο κατασκευαστής x_disj κωδικοποιεί τον κανόνα διάζευξης. Οι κανόνες αυτοί δεν έχουν ιδιαίτερη χρησιμότητα στη διαδικασία της μηχανιστικής επαλήθευσης προγραμμάτων. Είναι δύσκολο να διαχωρίσουμε μια γνωστή μετασυνθήκη (ή προσυνθήκη) σε δύο κατάλληλες συνθήκες σε σύζευξη ή διάζευξη, και να προχωρήσουμε τη διαδικασία επαλήθευσης χωριστά για κάθε μια από αυτές.

5.2.3 Βοηθητικά θεωρήματα

```
Lemma self_impl_a: ∀ a, impl_a a a.

Lemma impl_seq: ∀ p1 p2 q1 q2 i1 i2,
  axiom p2 i2 q2 → impl_a q1 p2 → axiom p1 i1 q1 →
  axiom p1 (i_seq i1 i2) q2.

Lemma rev_seq: ∀ p q r i1 i2,
  axiom q i2 r → axiom p i1 q → axiom p (i_seq i1 i2) r.

Lemma impl_postc_while: ∀ c a i q ,
  axiom (fun f ⇒ a f ∧ eval_c c f) i a →
  impl_a (fun f ⇒ a f ∧ ¬ eval_c c f) q →
  axiom a (i_while c a i) q.

Lemma impl_prec_while: ∀ c a i q r,
  axiom (fun f ⇒ a f ∧ eval_c c f) i a →
  impl_a (fun f ⇒ a f ∧ ¬ eval_c c f) q →
  impl_a r a →
  axiom r (i_while c a i) q.

Lemma impl_let: ∀ p r v e,
  impl_a p (re_a r v e) → axiom p (i_let v e) r.

Lemma impl_skip: ∀ p q,
  impl_a p q → axiom p i_skip q.

Lemma if_pre: ∀ p1 p2 p q c i1 i2,
  axiom p1 i1 q →
  axiom p2 i2 q →
  p = (fun f ⇒ (eval_c c f → p1 f) ∧ (~eval_c c f → p2 f)) →
  axiom p (i_if c i1 i2) q.
```

Σχήμα 5.2: Λήμματα τα οποία χρησιμοποιούνται κατά τη διαδικασία επαλήθευσης

Στο σχήμα 5.2 δίνονται λήμματα τα οποία εφαρμόζει η τακτική επαλήθευσης που θα παρουσιάσουμε στη συνέχεια. Παραλείπονται τα βήματα που ακολουθήσαμε για την κατασκευή των αποδείξεών τους (proof scripts) καθώς είναι δύσκολο να διαβαστούν χωρίς να γνωρίζουμε τους ανοιχτούς στόχους απόδειξης κάθε στιγμή, όπως συμβαίνει στο κέλυφος (toplevel) του Coq.

- Το λήμμα self_impl_a κωδικοποιεί την ανακλαστική ιδιότητα της συνεπαγωγής (⇒) για τους ισχυρισμούς (assert). Κάθε ισχυρισμός συνεπάγεται τον εαυτό του.

- Το λήμμα impl_seq συνδυάζει τον κανόνα αποδυνάμωσης της μετασυνθήκης με τον κανόνα της σύνθεσης. Η ακολουθία των εντολών είναι ορθή αν η μετασυνθήκη της πρώτης εντολής συνεπάγεται την προσυνθήκη της δεύτερης.

- Στη στρατηγική επαλήθευσης που ακολουθούμε, διατρέχουμε το πρόγραμμα από το τέλος προς την αρχή. Η γλώσσα Ltac δεν μας επιτρέπει να αντιμετωπίσουμε τους υπό-στόχους με τη σειρά που επιθυμούμε. Για να το επιτύχουμε αυτό, εφαρμόζουμε στα προς απόδειξη θεωρήματα της μορφής axiom *p* (i_seq *i1 i2*) *q* το λήμμα rev_seq, το οποίο έχει τις ίδιες υποθέσεις με το τον κατασκευαστή i_seq σε αντίστροφη σειρά.

- Το λήμμα impl_postc_while συνδυάζει τον κανόνα αποδυνάμωσης της μετασυνθήκης με τον κανόνα του `while`. Ο κανόνας του `while` όπως έχουμε δει μας δεσμεύει ως προς τη μετασυνθήκη, επομένως αν η επιθυμητή είναι διαφορετική πρέπει να αποδείξουμε συνεπαγωγή.

- Το λήμμα impl_prec_while εφαρμόζει κανόνα αποδυνάμωσης της μετασυνθήκης (για τον ίδιο λόγο με το impl_postc_while). Επιπλέον, εφαρμόζει κανόνα ενδυνάμωσης της προσυνθήκης, προκύπτει επομένως υποχρέωση απόδειξης πως η επιθυμητή προσυνθήκη συνεπάγεται την αναλλοίωτη βρόχου.

- Το λήμμα impl_let συνδυάζει τον κανόνα της ανάθεσης με τον κανόνα ενδυνάμωσης της προσυνθήκης. Για να είναι ορθή η ανάθεση *v* = *e* με προσυνθήκη *p* και μετασυνθήκη *r*, πρέπει η *p* να συνεπάγεται την (*re_a r v e*).

- Το λήμμα impl_skip συνδυάζει τον κανόνα της κενής εντολής με τον κανόνα ενδυνάμωσης της προσυνθήκης. Πρέπει η επιθυμητή προσυνθήκη να συνεπάγεται την επιθυμητή μετασυνθήκη.

- Το λήμμα if_pre συγκεκριμενοποιεί την προσυνθήκη του κανόνα του `if`. Αν οι εντολές των δύο σκελών αναμένουν προσυνθήκες *p1* και *p2*, η

  (`fun` $f \Rightarrow$ (eval_c $c\ f \rightarrow p1\ f$) $\wedge$ ($\tilde{}$eval_c $c\ f \rightarrow p2\ f$))

  είναι μια έγκυρη προσυνθήκη για τη δομή `if`.

5.3 Τακτικές Επαλήθευσης

Η διαδικασία της επαλήθευσης πραγματοποιείται σε δύο στάδια, από δύο ανεξάρτητες τακτικές που έχουν υλοποιηθεί με τη γλώσσα Ltac:

- Εφαρμόζουμε την τακτική *axiom_tac* με ανοιχτό στόχο (goal) ένα θεώρημα της μορφής *axiom p c q*. Η τακτική αυτή αναλαμβάνει το κομμάτι της απόδειξης που σχετίζεται άμεσα με τους κανόνες της λογικής Hoare. Εφαρμόζει τους κατασκευαστές του επαγωγικού ορισμού axiom ή βοηθητικά αξιώματα με βάση την υπό εξέταση εντολή.

- Η τακτική *impl_tac* αποδεικνύει υπο-στόχους (subgoals) που άφησε ανοιχτούς η *axiom_tac*. Στο σημείο αυτό έχουν αφαιρεθεί οι αναφορές στη σύνταξη του προγράμματος και δεν ασχολούμαστε πλέον με τους κανόνες της λογικής Hoare. Τα θεωρήματα που αποδεικνύει η `impl_tac` είναι αυτά που θα είχε δημιουργήσει μια συνάρτηση wp (ασθενέστερης προσυνθήκης) σε ένα σύστημα σαν το Why.

Η τακτική *verify* που χρησιμοποιούμε για την επαλήθευση των περισσότερων προγραμμάτων (και εμφανίζεται στα παραδείγματα του κεφαλαίου 4) ορίζεται ως

`Ltac` *verify* := *axiom_tac*; *impl_tac*.

Η τακτική μετά το ερωτηματικό (*impl_tac*) εφαρμόζεται σε όλους τους υπό-στόχους (subgoals) που αφήνει ανοιχτούς η τακτική πριν το ερωτηματικό (*axiom_tac*).

5.3.1 Η τακτική *axiom_tac*

Στο σχήμα 5.3 δίνεται η τακτική `axiom_tac`, η οποία εφαρμόζει τους κανόνες της λογικής Hoare για να αποδείξει προτάσεις της μορφής axiom *p i q*.

Η τακτική λειτουργεί με ταίριασμα προτύπων (pattern matching) στον ανοιχτό υποστόχο. Ανάλογα με την εντολή στο δεύτερο όρισμα του κατηγορήματος axiom, εφαρμόζουμε κατάλληλο κατασκευαστή ή κάποιο από τα λήμματα του σχήματος 5.2. Χρησιμοποιούμε εκτενώς την `eapply`, η οποία αντίθετα με την `apply` δεν αποτυγχάνει αν δεν μπορεί να δώσει τιμές σε όλες τις μεταβλητές. Αντίθετα, μετατρέπει τις μεταβλητές για τις οποίες δε γνωρίζει συγκεκριμένη τιμή σε υπαρξιακές (existential). Οι υπαρξιακές μεταβλητές εμφανίζονται στο κέλυφος (toplevel) με *?n*, όπου n κάποιος αριθμός. Στην περίπτωσή μας, υπαρξιακές μεταβλητές εμφανίζονται για προσυνθήκες εντολών που δεν γνωρίζουμε ακόμα. Η μετασυνθήκη μιας εντολής είναι αντίθετα πάντοτε γνωστή. Η τακτική

```
Ltac axiom_tac :=
  match goal with
  | [ ⊢ axiom _ (i_let _ _) _ ] ⇒
    (* if it succeeds we don't know precondition but we can guess it *)
    apply x_let ||
    (* we know precondition and have to prove it implies (re_a q v e) *)
    (eapply impl_let; idtac)
  | [ ⊢ axiom _ (i_seq _ _) _ ] ⇒
    (* no other way to solve second subgoal first with ltac? *)
    eapply rev_seq; [ axiom_tac | axiom_tac ]
  | [ ⊢ axiom _ (i_if _ _ _) _ ] ⇒
    (eapply if_pre; [ axiom_tac | axiom_tac | solve [ trivial ] ]) ||
    (* if impl_if fails we know the precondition *)
    (apply x_if; [ axiom_tac | axiom_tac ])
  | [ ⊢ axiom _ (i_while _ _ _) _ ] ⇒
    (eapply impl_postc_while; [ axiom_tac | idtac ]) ||
    (* if impl_postc_while fails we know the precondition *)
    (eapply impl_prec_while; [ axiom_tac | idtac | idtac ])
  | [ ⊢ axiom _ i_skip _ ] ⇒
    eapply x_skip ||
    (* we know precondition and prove it implies postcondition *)
    (apply impl_skip; idtac)
  end.
```

Σχήμα 5.3: Η τακτική *axiom_tac* για την εφαρμογή των κανόνων της αξιωματικής σημασιολογίας

`idtac` αφήνει τους στόχους για τους οποίους καλείται ανεπηρέαστους. Οι στόχοι αυτοί είναι οι υποχρεώσεις απόδειξης που θα αντιμετωπίσει αργότερα η *impl_tac*.

Στην περίπτωση που η εντολή είναι ανάθεση, δοκιμάζουμε αρχικά να εφαρμόσουμε τον κατασκευαστή x_let. Σε περίπτωση που επιτύχει η ως τότε υπαρξιακή μεταβλητή για την προσυνθήκη έχει πλέον απτή τιμή (κατά τα γνωστά *re_a e q v*, όπου *q* η μετασυνθήκη, *v* η μεταβλητή στην οποία αναθέτουμε και *e* η έκφραση στα δεξιά του συμβόλου ανάθεσης). Σε αντίθετη περίπτωση μια προσυνθήκη είναι ήδη γνωστή (διότι, για παράδειγμα, πρόκειται για την πρώτη εντολή του προγράμματος). Τότε εφαρμόζουμε το λήμμα impl_let. Από την εφαρμογή αυτή προκύπτει υποχρέωση απόδειξης μιας συνεπαγωγής, την οποία αφήνουμε για την *impl_tac*.

Όταν το θεώρημα προς απόδειξη αφορά ακολουθία εντολών, εφαρμόζουμε το λήμμα rev_seq ώστε να αντιμετωπίσουμε πρώτα τη δεύτερη εντολή. Προκύπτουν δύο υποστόχοι για τις δύο εντολές, τους οποίους αποδεικνύουμε αναδρομικά με την *axiom_tac*. Αρχικά η προσυνθήκη της δεύτερης εντολής και μετασυνθήκη της πρώτης είναι μια υπαρξιακή μεταβλητή, θα έχει όμως αποκτήσει τιμή μέχρι να αντιμετωπίσουμε τον υποστόχο που αφορά την πρώτη.

Αν έχουμε εντολή `if`, εφαρμόζουμε αρχικά το λήμμα if_pre υποθέτοντας ότι δεν γνωρίζουμε την προσυνθήκη. Οι δύο πρώτες υποθέσεις του λήμματος αφωρούν την μερική ορθότητα των εντολών στα δύο σκέλη, με μετασυνθήκη κοινή με το `if`. Με επίλυση αυτών των υποχρεώσεων (με αναδρομική κλήση της *axiom_tac*) προκύπτουν απτές τιμές για τις προσυνθήκες *p1*, *p2* των δύο σκελών. Τρίτη υπόθεση του if_pre και κατά συνέπεια ανοιχτός υποστόχος για εμάς είναι η ισότητα $p = (\texttt{fun}\, f \Rightarrow (\text{eval\_c}\ c\, f \rightarrow p1\, f) \wedge (\tilde{}\,\text{eval\_c}\ c\, f \rightarrow p2\, f))$. Αν κάποια προσυνθήκη δεν ήταν ήδη γνωστή, η τακτική `trivial` επιτυγχάνει και η p ενοποιείται με την έκφραση στα δεξιά. Αν αποτύχει η `trivial` γιατί γνωρίζαμε την προσυνθήκη, εφαρμόζουμε τον κατασκευαστή x_if. Για τους δύο υποστόχους που προκύπτουν από αυτόν καλούμε αναδρομικά την *axiom_tac*.

Στην περίπτωση εντολής `while`, εφαρμόζουμε αρχικά το λήμμα impl_postc_while. Αν αυτό πε-

τύχει, η προσυνθήκη ήταν άγνωστη αλλά έχει τώρα ενοποιηθεί με την αναλλοίωτη βρόχου. Καλούμε αναδρομικά την *axiom_tac* για να αποδείξουμε ότι η αναλλοίωτη βρόχου διατηρείται, και αφήνουμε στην *impl_tac* να αποδείξει πως η αναλλοίωτη σε σύζευξη με την άρνηση της συνθήκης του `while` συνεπάγεται την επιθυμητή μετασυνθήκη. Σε περίπτωση γνωστής προσυνθήκης (αποτυχίας εφαρμογής του impl_postc_while) εφαρμόζουμε το impl_prec_while. Προκύπτει ένας επιπλέον ανοιχτός υποστόχος για τη συνεπαγωγή μεταξύ προσυνθήκης και αναλλοίωτης βρόχου, τον οποίο θα αντιμετωπίσει η *impl_tac*.

Αν, τέλος, έχουμε εντολή `skip` προσπαθούμε να εφαρμόσουμε τον κατασκευαστή x_skip. Αν επιτύχει, η προσυνθήκη ήταν άγνωστη και ενοποιείται με τη μετασυνθήκη. Αλλιώς, εφαρμόζουμε το λήμμα impl_skip και αφήνουμε στην *impl_tac* να αποδείξει πως η προσυνθήκη συνεπάγεται τη μετασυνθήκη.

5.3.2 Η τακτική *impl_tac*

```
Ltac impl_tac :=
  try (apply self_impl_a);
  unfold impl_a;
  unfold re_a;
  simpl;
  try ( solve [
    eauto with exact_hints |
    try (
      intuition;
      solve [
        eauto with myhints zarith |
        omega |
        ring
      ]
    )
  ] ).
```

Σχήμα 5.4: Η τακτική *impl_tac* για την επίλυση των υποχρεώσεων απόδειξης

Η τακτική *impl_tac* καλείται για να επιλύσει στόχους της μορφής *impl_a p q*, όπου *p* και *q* κατηγορήματα του τύπου *assert*. Οι στόχοι αυτοί έχουν προκύψει, όπως είδαμε, κατά την εκτέλεση της τακτικής *axiom_tac*.

Αρχικά εξετάζουμε την περίπτωση οι δύο ισχυρισμοί να είναι ίδιοι. Τότε επιτυγχάνει η εφαρμογή του λήμματος self_impl_a και δεν χρειάζονται περαιτέρω ενέργειες. Αυτό δεν ισχύει στις περισσότερες περιπτώσεις, και προχωράμε «ξεδιπλώνοντας» τους ορισμούς των *impl_a* και *re_a* με την τακτική `unfold`. Λόγω του ορισμού της *re_a* που παρουσιάσαμε στην ενότητα 5.2.1, εφαρμογή του κανόνα της ανάθεσης οδηγεί σε περίπλοκους όρους. Οι όροι αυτοί μπορούν να απλοποιηθούν με απλές ι-αναγωγές. Αυτό γίνεται με χρήση της τακτικής `simpl`.

Η τακτική `try` δοκιμάζει την τακτική που δίνεται ως όρισμα. Αν αυτή αποτύχει, η `try` αφήνει την απόδειξη στην προηγούμενη κατάσταση. Η τακτική `solve` δοκιμάζει μια σειρά από τακτικές και αποτυγχάνει αν καμία από αυτές δεν επιλύσει τον στόχο. Μια τακτική δεν αποτυγχάνει απαραίτητα αν δεν επιλύσει τον στόχο· μπορεί να τον αφήσει ανεπηρέαστο ή να προχωρήσει κατά μερικά βήματα.

Μετά την απλοποίηση με `simpl`, δοκιμάζουμε να επιλύσουμε τον στόχο με:

- Την `eauto with` *exact_hints*. Η *exact_hints* είναι μια βάση συμβουλών (hint database) που περιλαμβάνει λήμματα τα οποία προηγούμενη εκτέλεση του εργαλείου απόδειξης επέστρεψε ως υποχρεώσεις. Αν ο προγραμματιστής τις έχει συμπληρώσει όλες και τα έχει συμπεριλάβει

στον κώδικά του, η τακτική αυτή θα επιτύχει για όλες τις υποχρεώσεις που θα κληθεί να αντιμετωπίσει και η διαδικασία της επαλήθευσης θα ολοκληρωθεί.

- Αν δεν έχει συμπληρωθεί ακόμα η βάση συμβουλών *exact_hints*, εφαρμόζουμε τη δομή

```
try (
  intuition
    (solve [
      eauto with myhints zarith |
      ring
    ])
  )
])
```

Η τακτική `intuition` βασίζεται σε μια διαδικασία απόφασης για τον ιντουισιονιστικό προτασιακό λογισμό (intuitionistic propositional calculus). Λύνει από μόνη της κάθε ταυτολογία του ιντουισιονιστικού προτασιακού λογισμού και εφαρμόζει την τακτική την οποία δέχεται ως όρισμα στους υποστόχους που δεν μπορεί να αντιμετωπίσει. Εμείς χρησιμοποιούμε ως τακτική-όρισμα μια δομή *solve*, η οποία εφαρμόζει:

  - Την `eauto` με τις βάσεις συμβουλών *myhints* και *zarith*. Η πρώτη περιέχει βοηθητικά θεωρήματα του προγραμματιστή, όχι για τις ακριβείς υποχρεώσεις που έχει παράγει προηγούμενη εκτέλεση του προγράμματος. Η δεύτερη βάση συμβουλών αφορά την αριθμητική στο σύνολο των ακεραίων (Z) και ορίζεται στη βιβλιοθήκη του Coq.
  - Την `ring`, η οποία αποδεικνύει ισότητες εφαρμόζοντας την προσεταιριστική (associative) και την αντιμεταθετική ιδιότητα. Δεν μπορεί να χειριστεί μη πολυωνυμικές εκφράσεις, για παράδειγμα εκθετικές.

Αν τα παραπάνω δεν δώσουν αποτέλεσμα, δίνουμε ως έξοδο τις υποχρεώσεις απόδειξης με τη μορφή ενός προτύπου που ο προγραμματιστής καλείται να συμπληρώσει. Οι υποχρεώσεις απόδειξης εμφανίζονται όπως της άφησε η τακτική `simpl` στην *impl_tac*.

5.4 Υλοποίηση του Εργαλείου Επαλήθευσης

Το εργαλείο επαλήθευσης που έχουμε υλοποιήσει, συγκεκριμένα το τμήμα του που είναι γραμμένο σε OCaml μετατρέπει το προστακτικό πρόγραμμα που δίνει ως είσοδο ο προγραμματιστής σε έναν όρο του τύπου instr. Αυτό δεν γίνεται με εκτύπωση πηγαίου κώδικα Coq αλλά με απευθείας κατασκευή όρων στις δομές δεδομένων που χρησιμοποιεί το Coq για την αναπαράσταση του αφηρημένου συντακτικού δέντρου του.

Η αντιστοιχία των συντακτικών δομών της Tony με τους επαγωγικούς τύπους που περιγράψαμε στην ενότητα 5.1 είναι προφανής. Χρειάζεται όμως να διευκρινίσουμε πως μετατρέπονται οι επισημειώσεις του προγραμματιστή σε κατηγορήματα του τύπου assert. Αναμένουμε ο προγραμματιστής να χρησιμοποιεί συνάρτηση κατάστασης με το προκαθορισμένο όνομα e για να αναφερθεί στις μεταβλητές του προγράμματος. Ο συντακτικός αναλυτής του Coq διαβάζει έναν όρο με ελεύθερη τη μεταβλητή e, έστω $P[e]$, ως επισημειώση μέσα σε άγκυστρα. Το εργαλείο επαλήθευσης περικλείει τον όρο $P[e]$ με μια λάμδα αφαίρεση μεταβλητής e. Προκύπτει ο όρος (`fun` $e \Rightarrow P[e]$) τύπου assert.

Αφού μεταγλωττίσουμε το πρόγραμμα σε όρο του Coq (έστω c) σύμφωνα με τα παραπάνω, κατασκευάζουμε ένα θεώρημα axiom p c q, με προσυνθήκη p και μετασυνθήκη q αυτές που έχει δώσει ο προγραμματιστής. Τον όρο του θεωρήματος τον εισάγουμε στο Coq και πάλι ως συντακτικό δέντρο. Καλούμε την τακτική *verify* για την απόδειξη του θεωρήματος, και δίνουμε ως έξοδο τους υπό-στόχους που αφήνει ανοιχτούς.

```
Tactic verify

Program power
{ e "X" >= 0 /\ e "N" > 0 /\ (e "X" > 0 \/ e "N" > 0) }
x = X
n = N
r = 1
while 0 < n { e "x" ^ (e "n") * (e "r") = (e "X") ^ (e "N")
                /\ e "n" >= 0
            }
  if n % 2 = 1
    r = r * x
  fi
  n = n / 2
  x = x * x
end
{ e "r" = e "X" ^ e "N" }
```

Σχήμα 5.5: Πρόγραμμα ύψωσης σε δύναμη με διαδοχικούς τετραγωνισμούς στην Tony

```
axiom
  (fun e : env_f ⇒ e "X" ≥ 0 ∧ e "N" ≥ 0)
  (i_seq (i_let "x" (e_var "X"))
    (i_seq (i_let "n" (e_var "N"))
      (i_seq (i_let "r" (e_num 1))
        (i_while (c_lt (e_num 0) (e_var "n"))
          (fun e : env_f ⇒
            e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0)
          (i_seq
            (i_if (c_eq (e_mod (e_var "n") (e_num 2)) (e_num 1))
              (i_let "r" (e_mult (e_var "r") (e_var "x"))) i_skip)
            (i_seq (i_let "n" (e_div (e_var "n") (e_num 2)))
              (i_let "x" (e_mult (e_var "x") (e_var "x")))))))))
  (fun e : env_f ⇒ e "r" = e "X" ^ e "N").
```

Σχήμα 5.6: Το προς απόδειξη θεώρημα μερικής ορθότητας για το πρόγραμμα 5.5

5.5 Βήματα Επαλήθευσης για ένα Παράδειγμα

Στο σχήμα 5.5 δίνεται πρόγραμμα για τον υπολογισμό της δύναμης X^N. Όπως δηλώνει η προσυνθήκη, το πρόγραμμα αναμένει X και N μεγαλύτερα ή ίσα του μηδενός και ένα εκ των δύο διάφορο του 0, για να μην προσπαθήσουμε να υπολογίσουμε τη δύναμη 0^0. Το πρόγραμμα ακολουθεί τη *μέθοδο των διαδοχικών τετραγωνισμών* (repeated squaring).

Η μέθοδος τετραγωνίζει διαδοχικά το X (X^2, $\{X^2\}^2$ και ούτω καθ' εξής) και πολλαπλασιάζει μεταξύ τους τα «χρήσιμα» τετράγωνα. Έστω ότι θέλουμε να υπολογίσουμε τη δύναμη 3^{13}. Το 13 σε δυαδική μορφή είναι $(1101)_2$, δηλαδή $13 = 2^3 + 2^2 + 2^0$, επομένως $3^{13} = 3^{2^3+2^2+2^0} = 3^{2^3}3^{2^2}3^{2^0} = ((3^2)^2)^2(3^2)^23$. Από το παράδειγμα φαίνεται ότι το i-οστό τετράγωνο είναι χρήσιμο αν το i-τάξης δυαδικό ψηφίο είναι 1. Το πρόγραμμά μας κρίνει αν αυτό ισχύει με βάση τη συνθήκη[1] $n \; mod \; 2 = 1$.

[1] Το n αρχικοποιείται με την τιμή N και σε κάθε επανάληψη διαιρείται δια 2, κατά τη στιγμή του ελέγχου επομένως έχουμε $n = N/2^i$, όπου i ο αριθμός της επανάληψης ξεκινώντας από 0. Αν ο αριθμός αυτός είναι περιττός, το i-οστό

Σύμφωνα με την αναλλοίωτη του βρόχου, ισχύει μετά από κάθε επανάληψη $x^n r = X^N$. Μπορούμε εύκολα να διαπιστώσουμε ότι η αναλλοίωτη ισχύει κατά τον υπολογισμό του παραδείγματος 3^{13}.

Το εργαλείο επαλήθευσης μεταφράζει το πρόγραμμα στο θεώρημα του σχήματος 5.6 και εφαρμόζει σε αυτό την τακτική *verify*, η οποία ορίζεται ως *axiom_tac*; *impl_tac*. Η *axiom_tac* εφαρμόζει αρχικά ταίριασμα προτύπων στον στόχο 5.6. Το όρισμα της εντολής είναι ακολουθία (κατασκευαστής i_seq) με πρώτη εντολή την ανάθεση `x = X` και δεύτερη το υπόλοιπο πρόγραμμα. Εφαρμόζεται το rev_seq και δημιουργούνται δύο υποστόχοι με τον δεύτερο να αφορά την ανάθεση. Με αναδρομές της *axiom_tac* καταλήγουμε σε εφαρμογή του rev_seq, με πρώτη εντολή την ανάθεση `r = 1` και δεύτερη την `while`. Καλούμαστε τώρα να επιλύσουμε έναν στόχο της μορφής

```
axiom
  ?P
  (i_while (c_lt (e_num 0) (e_var "n"))
     (fun e : env_f ⇒
      e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0)
     (i_seq
        (i_if (c_eq (e_mod (e_var "n") (e_num 2)) (e_num 1))
           (i_let "r" (e_mult (e_var "r") (e_var "x"))) i_skip)
        (i_seq (i_let "n" (e_div (e_var "n") (e_num 2)))
           (i_let "x" (e_mult (e_var "x") (e_var "x"))))))
  (fun e : env_f ⇒ e "r" = e "X" ^ e "N").
```

Για την άγνωστη προσυνθήκη έχει δημιουργηθεί η υπαρξιακή μεταβλητή *?P*. Η εντολή είναι `while`, άρα η *axiom_tac* εφαρμόζει το λήμμα impl_postc_while. Η εφαρμογή πετυχαίνει γιατί η προσυνθήκη ήταν η υπαρξιακή μεταβλητή *?P* και μπορεί να ενοποιηθεί με την αναλλοίωτη του βρόχου:

```
(fun e : env_f ⇒ e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0)
```

Μετά την εφαρμογή του impl_postc_while έχουν προκύψει δύο νέοι υποστόχοι. Ο ένας αφορά τη διατήρηση της αναλλοίωτης βρόχου. Είναι

```
axiom
  (fun f : env_f ⇒
   (f "x" ^ f "n" × f "r" = f "X" ^ f "N" ∧ f "n" ≥ 0) ∧
   eval_c (c_lt (e_num 0) (e_var "n")) f)
  (i_seq
     (i_if (c_eq (e_mod (e_var "n") (e_num 2)) (e_num 1))
        (i_let "r" (e_mult (e_var "r") (e_var "x"))) i_skip)
     (i_seq (i_let "n" (e_div (e_var "n") (e_num 2)))
        (i_let "x" (e_mult (e_var "x") (e_var "x")))))
  (fun e : env_f ⇒ e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0).
```

Πρέπει επίσης η επιθυμητή μετασυνθήκη να προκύπτει από τη σύζευξη της αναλλοίωτης βρόχου με την άρνηση της συνθήκης του βρόχου. Αυτός είναι ο δεύτερος υποστόχος:

```
impl_a
  (fun f : env_f ⇒
     (f "x" ^ f "n" × f "r" =
      f "X" ^ f "N" ∧
      f "n" ≥ 0) ∧
     ~eval_c (c_lt (e_num 0) (e_var "n")) f)
  (fun e : env_f ⇒ e "r" = e "X" ^ e "N").
```

Για την ακολουθία εντολών στο εσωτερικό του βρόχο πραγματοποιούνται και πάλι αναδρομικές κλήσεις της *axiom_tac* και εφαρμογές του rev_seq. Όταν φτάσουμε στην τελευταία ανάθεση (`x = x * x`) αντιμετωπίζουμε τον υποστόχο

δυαδικό ψηφίο είναι 1.

```
axiom
  ?Q
  (i_let "x" (e_mult (e_var "x") (e_var "x")))
  (fun e : env_f ⇒ e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0).
```

Η *axiom_tac*, εφόσον πρόκειται για ανάθεση, δοκιμάζει τον κατασκευαστή x_let. Η εφαρμογή πετυχαίνει καθώς η προσυνθήκη είναι υπαρξιακή μεταβλητή, και η προσυνθήκη ενοποιείται με την αντικατάσταση της μεταβλητής με την έκφραση στην μετασυνθήκη:

```
re_a
  (fun e : env_f ⇒ e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0)
  "x"
  (e_mult (e_var "x") (e_var "x")).
```

Η πιο πρόσφατη εφαρμογή του rev_seq αφορούσε την ακολουθία των αναθέσεων `n = n / 2, x = x * x`. Πλέον έχουμε μια απτή τιμή για την μετασυνθήκη της πρώτης και προσυνθήκη της δεύτερης. Η *axiom_tac* καλείται με εντολή την `n = n / 2`, μετασυνθήκη την παραπάνω και υπαρξιακή μεταβλητή για την προσυνθήκη. Εφαρμόζεται και πάλι ο x_let και η προσυνθήκη ενοποιείται με την αντικατάσταση της *n* με (e_div (e_var "n") (e_num "2")) στην μετασυνθήκη. Πρώτη εντολή στον βρόχο είναι η `if`. Η προσυνθήκη της είναι γνωστή, όπως και η μετασυνθήκη. Η *axiom_tac* προσπαθεί να εφαρμόσει το λήμμα και επιτυγχάνει, αποτυγχάνει όμως η επίλυση του τρίτου υποστόχου που ανοίγει με την *solve* [`trivial`] γιατί ήταν ήδη γνωστή μια προσυνθήκη. Εφαρμόζεται επομένως ο κατασκευαστής x_if. Και οι δύο υποστόχοι που η εφαρμογή αυτή δημιουργεί αντιμετωπίζονται με την *axiom_tac* και έχουν γνωστές προσυνθήκες και μετασυνθήκες. Θα εφαρμοστούν τα αξιώματα impl_let και impl_skip αφού οι ενοποιήσεις με τους αντίστοιχους κατασκευαστές θα αποτύχουν. Προκύπτουν οι υποχρεώσεις απόδειξης

```
impl_a
  (fun f : env_f ⇒
     ((f "x" ^ f "n" × f "r" = f "X" ^ f "N" ∧ f "n" ≥ 0) ∧
      eval_c (c_lt (e_num 0) (e_var "n")) f) ∧
     eval_c (c_eq (e_mod (e_var "n") (e_num 2)) (e_num 1)) f)
  (re_a
     (re_a
        (re_a
           (fun e : env_f ⇒
              e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0)
           "x"
           (e_mult (e_var "x") (e_var "x")))
        "n"
        (e_div (e_var "n") (e_num 2)))
     "r"
     (e_mult (e_var "r") (e_var "x"))).


impl_a
  (fun f : env_f ⇒
     ((f "x" ^ f "n" × f "r" = f "X" ^ f "N" ∧ f "n" ≥ 0) ∧
      eval_c (c_lt (e_num 0) (e_var "n")) f) ∧
     ~eval_c (c_eq (e_mod (e_var "n") (e_num 2)) (e_num 1)) f)
  (re_a
     (re_a
        (fun e : env_f ⇒ e "x" ^ e "n" × e "r" = e "X" ^ e "N" ∧ e "n" ≥ 0)
        "x"
        (e_mult (e_var "x") (e_var "x")))
```

```
    ”n”
    (e_div (e_var ”n”) (e_num 2))).
```

Η προσυνθήκη του `while` (και μετασυνθήκη της ανάθεσης `r = 1` σύμφωνα με το λήμμα rev_seq που εφαρμόσαμε) έχει ενοποιηθεί με την αναλλοίωτη βρόχου. Για την ανάθεση η προσυνθήκη είναι άγνωστη, οπότε η *axiom_tac* εφαρμόζει τον κατασκευαστή x_let. Η προσυνθήκη ενοποιείται με την αντικατάσταση της μεταβλητής `r` με την έκφραση 1 στην μετασυνθήκη. Η διαδικασία επαναλαμβάνεται για την ανάθεση `n = N`. Για την ανάθεση `x = X` είναι γνωστή και η προσυνθήκη, οπότε εφαρμόζεται το λήμμα impl_let. Προκύπτει υποχρέωση απόδειξης του

```
impl_a
  (fun e : env_f ⇒ e ”X” ≥ 0 ∧ e ”N” ≥ 0 ∧ (e ”X” > 0 ∨ e ”N” > 0))
  (re_a
     (re_a
        (re_a
           (fun e : env_f ⇒
              e ”x” ^ e ”n” × e ”r” = e ”X” ^ e ”N” ∧ e ”n” ≥ 0)
        ”r”
        (e_num 1))
     ”n”
     (e_var ”N”))
  ”x”
  (e_var ”X”)).
```

Αν η *impl_tac* καταφέρει να επιλύσει τους τέσσερις στόχους της μορφής impl_a *p q* που περιγράψαμε παραπάνω, η διαδικασία της επαλήθευσης ολοκληρώνεται με επιτυχία. Χωρίς βοηθητικά θεωρήματα αυτό συμβαίνει μόνο για μια από τις τέσσερις, οπότε και τυπώνονται οι τρεις υποχρεώσεις με τη μορφή κώδικα Vernac προς συμπλήρωση. Η έξοδος που λαμβάνει ο χρήστης είναι οι τρεις στόχοι μετά από ξεδίπλωμα (unfolding) των impl_a, re_a και απλοποίηση με το `simpl`.

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

Μπορούμε να συνοψίσουμε τη συνεισφορά της εργασίας ως εξής:

- Προτάθηκε η Tony, μια προστακτική γλώσσα προγραμματισμού που επιτρέπει επαλήθευση των προγραμμάτων. Τα προγράμματα της Tony περιέχουν επισημειώσεις με προσυνθήκες, μετασυνθήκες και αναλλοίωτες βρόχων. Βασική επιδίωξη ήταν να παρέχουμε μια εκφραστική γλώσσα για τη διατύπωση προδιαγραφών ορθότητας των προγραμμάτων. Ως εκ τούτου, επιλέξαμε οι επισημειώσεις να γράφονται στην Gallina του Coq.

- Υλοποιήθηκε ένα σύστημα επαλήθευσης προγραμμάτων για την Tony, ως επέκταση του Coq. Οι τακτικές που ακολουθεί το σύστημα αυτό αυτοματοποιούν κατά το δυνατό τη διαδικασία επαλήθευσης. Στην περίπτωση που η επαλήθευση αποτύχει, το σύστημα επιστρέφει μια σειρά από υποχρεώσεις απόδειξης με τη μορφή ενός ημιτελούς Coq αρχείου που πρέπει να συμπληρώσει ο προγραμματιστής.

- Υλοποιήθηκε στη γλώσσα Tony μια σειρά προγραμμάτων με πλήρη απόδειξη μερικής ορθότητας. Τα προγράμματα εκμεταλλεύονται θεωρήματα της βιβλιοθήκης του Coq όταν αυτό είναι δυνατό ή συμπεριλαμβάνουν νέα.

6.2 Περιορισμοί

Πρώτος περιορισμός της εργασίας είναι η εκφραστικότητα της γλώσσας προγραμματισμού. Έχουμε περιοριστεί στις απαραίτητες δομές μιας προστακτικής γλώσσας προγραμματισμού και σε πράξεις μεταξύ ακεραίων αριθμών. Η γλώσσα προγραμματισμού δεν έχει πίνακες (arrays), οι οποίοι θα μας επέτρεπαν να επιχειρήσουμε να επαληθεύσουμε πιο σύνθετα και ενδιαφέροντα προγράμματα, για παράδειγμα υλοποιήσεις αλγόριθμων ταξινόμησης. Η γλώσσα δεν υποστηρίζει επίσης συναρτήσεις, δεν μπορούν επομένως να γραφτούν αναδρομικά προγράμματα.

Δεύτερος περιορισμός της εργασίας είναι η σημαντική σε πολλές περιπτώσεις δυσκολία κατασκευής αποδείξεων. Συγκεκριμένα, για να αποδείξει ισότητες ή ανισότητες εκφράσεων χρειάζεται να παρέμβει ο προγραμματιστής σε χαμηλού επιπέδου σημεία της απόδειξης, να εφαρμόσει για παράδειγμα την αντιμεταθετική ή την προσεταιριστική ιδιότητα. Τακτικές που έχουν αναπτυχθεί για να αυτοματοποιήσουν τέτοιου είδους αποδείξεις υποστηρίζουν πολυωνυμικές εκφράσεις, αποτυγχάνουν επομένως όταν στις προδιαγραφές μας εμφανίζεται κάποια πιο σύνθετη έκφραση, όπως ύψωση σε μεταβλητή.

6.3 Μελλοντικές κατευθύνσεις

Μια μελλοντική επαναπροσέγγιση της γλώσσας Tony θα μπορούσε να συμπεριλάβει δομές που δεν υποστηρίζονται στην τρέχουσα έκδοση. Σημαντική προσθήκη θα ήταν οι πίνακες. Έχουμε παρουσιάσει στην ενότητα 2.6.2 πως μπορούμε να τους αντιμετωπίσουμε στην αξιωματική σημασιολογία.

Δευτερευόντως, η γλώσσα μπορεί να επεκταθεί με συναρτήσεις. Αυτό θα μας επιτρέψει να αναπτύξουμε μεγαλύτερα προγράμματα, αλλά και να διατυπώσουμε αναδρομικούς αλγόριθμους.

Για να διαδοθεί εκτενώς η επαλήθευση προγραμμάτων πρέπει να μειωθεί ο χρόνος που απαιτεί η κατασκευή των αποδείξεων. Είναι αναγκαία λοιπόν η ανάπτυξη τακτικών που αυτοματοποιούν το είδος των αποδείξεων που η διαδικασία της επαλήθευσης απαιτεί. Κατά την επαλήθευση προγραμμάτων με το εργαλείο που κατασκευάσαμε, ιδιαίτερη δυσκολία συναντούμε σε αποδείξεις που απαιτούν αντικαταστάσεις σε μαθηματικές εξισώσεις και εφαρμογή ιδιοτήτων όπως η προσεταιριστική και η αντιμεταθετική. Η ανάπτυξη μιας τακτικής σαν την *ring*, η οποία όμως δεν θα περιορίζεται σε πολυωνυμικές εκφράσεις θα διευκόλυνε σημαντικά τη διαδικασία της επαλήθευσης προγραμμάτων της Tony. Γενικότερα, αλλαγές προς την κατεύθυνση της αυτοματοποίησης της απόδειξης θεωρημάτων θα έχουν άμεσα οφέλη στην τυπική επαλήθευση.

Μια ενδιαφέρουσα ερευνητική κατεύθυνση είναι, τέλος, ο πιστοποιημένος εκτελέσιμος κώδικας (proof carrying code). Το εργαλείο επαλήθευσης της συγκεκριμένης εργασίας και άλλα εργαλεία παρόμοιας φιλοσοφίας κατασκευάζουν μια απόδειξη για την ορθότητα του πηγαίου κώδικα που δέχονται ως είσοδο. Η απόδειξη αυτή δεν συνοδεύει ωστόσο τον εκτελέσιμο κώδικα και ο τελικός χρήστης του προγράμματος δεν μπορεί να διαπιστώσει αν το εκτελέσιμο πρόγραμμα ικανοποιεί κάποιες προδιαγραφές. Στα συστήματα πιστοποιημένων εκτελέσιμων όπως το NFlint [Shao05] ο εκτελέσιμος κώδικας συνοδεύεται από μια διατύπωση των προδιαγραφών και από αποδείξεις πως αυτές οι προδιαγραφές ικανοποιούνται. Θα μπορούσε ένα εργαλείο επαλήθευσης όπως αυτό της Tony αφού κατασκευάσει μια απόδειξη ορθότητας να την μεταφέρει, μαζί με το ίδιο το πρόγραμμα, σε ένα σύστημα πιστοποιημένων εκτελέσιμων.

Βιβλιογραφία

[Bare92] Henk Barendregt, "Lambda Calculi with Types", in *Handbook of Logic in Computer Science*, pp. 117–309, Oxford University Press, 1992.

[Bert04] Yves Bertot and Pierre Casteran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Springer, 2004.

[Dijk75] Edsger W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs", *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[Fill03] J.-C. Filliâtre, "Why: a multi-language multi-prover verification tool", Research Report 1366, LRI, Université Paris Sud, March 2003.

[Fill04] Jean-Christophe Filliâtre and Claude Marché, "Multi-Prover Verification of C Programs", in *Sixth International Conference on Formal Engineering Methods (ICFEM)*, vol. 3308 of *Lecture Notes in Computer Science*, pp. 15–29, Seattle, November 2004, Springer-Verlag.

[Floy67] Robert W. Floyd, "Assigning Meanings to Programs", in *Proceedings of a Symposium on Applied Mathematics*, vol. 19 of *Mathematical Aspects of Computer Science*, pp. 19–31, Providence, 1967.

[Hoar83] C. A. R. Hoare, "An axiomatic basis for computer programming", *Commun. ACM*, vol. 26, no. 1, pp. 53–56, 1983.

[Ouim08] Martin Ouimet, "Formal Software Verification: Model Checking and Theorem Proving", Technical report, Embedded Systems Laboratory, Massachusetts Institute of Technology, 2008.

[Reyn02] John C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures", in *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, Washington, DC, USA, 2002, IEEE Computer Society.

[Shao05] Zhong Shao, Valery Trifonov, Bratin Saha and Nikolaos Papaspyrou, "A type system for certified binaries", *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 1, pp. 1–45, 2005.