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

Σχεδίαση και Υλοποίηση Γλώσσας Υψηλού Επιπέδου για Προγραμματισμό με Αποδείξεις

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΝΤΩΝΗΣ Μ. ΣΤΑΜΠΟΥΛΗΣ

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

Αθήνα, Ιούλιος 2006

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

Σχεδίαση και Υλοποίηση Γλώσσας Υψηλού Επιπέδου για Προγραμματισμό με Αποδείξεις

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΝΤΩΝΗΣ Μ. ΣΤΑΜΠΟΥΛΗΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Ιουλίου 2006.

....	....	....
Νικόλαος Παπασπύρου Λέκτορας Ε.Μ.Π.	Ευστάθιος Ζάχος Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2006

....

Αντώνης Μ. Σταμπούλης

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

Περίληψη

Σκοπός της παρούσας εργασίας είναι αφενός η σχεδίαση μίας απλής γλώσσας υψηλού επιπέδου με υποστήριξη για προγραμματισμό με αποδείξεις, αφετέρου η υλοποίηση ενός μεταγλωττιστή για τη γλώσσα αυτή που θα παράγει κώδικα για μία γλώσσα ενδιάμεσου επιπέδου κατάλληλη για δημιουργία πιστοποιημένων εκτελέσιμων.

Στη σημερινή εποχή, η ανάγκη για αξιόπιστο και πιστοποιημένα ασφαλή κώδικα γίνεται διαρκώς ευρύτερα αντιληπτή. Τόσο κατά το παρελθόν όσο και πρόσφατα έχουν γίνει γνωστά προβλήματα ασφάλειας και συμβατότητας προγραμμάτων που είχαν ως αποτέλεσμα προβλήματα στην λειτουργία μεγάλων συστημάτων και συνεπώς οικονομικές επιπτώσεις στους οργανισμούς που τα χρησιμοποιούσαν. Τα προβλήματα αυτά οφείλονται σε μεγάλο βαθμό στην έλλειψη δυνατότητας προδιαγραφής και απόδειξης της ορθότητας των προγραμμάτων που χαρακτηρίζει τις σύγχρονες γλώσσες προγραμματισμού. Για το σκοπό αυτό, έχουν προταθεί συστήματα πιστοποιημένων εκτελέσιμων, στα οποία έχουμε τη δυνατότητα να προδιαγράφουμε την ορθότητα των προγραμμάτων, και να παρέχουμε μία τυπική απόδειξη αυτής, η οποία μπορεί να ελεγχθεί μηχανιστικά πριν το χρόνο εκτέλεσης.

Τα συστήματα που έχουν προταθεί είναι ενδιάμεσου επιπέδου οπότε η διαδικασία προγραμματισμού σε αυτά είναι ιδιαίτερα πολύπλοκη. Οι γλώσσες υψηλού επιπέδου που συνοδεύουν αυτά τα συστήματα, ενώ είναι ιδιαίτερα εκφραστικές, παραμένουν δύσκολες στον προγραμματισμό. Μία απλούστερη γλώσσα υψηλού επιπέδου, όπως αυτή που προτείνουμε σε αυτή την εργασία, θα επέτρεπε ευρύτερη εξάπλωση του συγκεκριμένου ιδιώματος προγραμματισμού.

Στη γλώσσα που προτείνουμε, ο προγραμματιστής προδιαγράφει τη μερική ορθότητα του προγράμματος, δίνοντας προσυνθήκες και μετασυνθήκες για τις παραμέτρους και τα αποτελέσματα των συναρτήσεων που ορίζει. Επίσης δίνει ένα σύνολο θεωρημάτων βάσει του οποίου κατασκευάζονται αποδείξεις της ορθής υλοποίησης και χρήσης των συναρτήσεων αυτών. Ως μέρος της εργασίας, έχουμε υλοποιήσει σε γλώσσα OCaml ένα μεταφραστή αυτής της γλώσσας στο σύστημα πιστοποιημένων εκτελέσιμων NFLINT.

Επιτύχαμε να διατηρήσουμε τη γλώσσα κοντά στο ύφος των ευρέως διαδεδομένων συναρτησιακών γλωσσών, καθώς και να διαχωρίσουμε τη φάση προγραμματισμού από τη φάση απόδειξης της ορθότητας των προγραμμάτων. Έτσι ένας μέσος προγραμματιστής μπορεί εύκολα να προγραμματίζει στη γλώσσα που προτείνουμε με τον τρόπο που ήδη γνωρίζει, και ένας γνώστης μαθηματικής λογικής να αποδεικνύει σε επόμενη φάση την μερική ορθότητα των προγραμμάτων. Ως απόδειξη της πρακτικότητας της προσέγγισης αυτής, παραθέτουμε ένα σύνολο παραδειγμάτων στη γλώσσα με απόδειξη μερικής ορθότητας.

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, Προγραμματισμός με αποδείξεις, Ασφαλείς γλώσσες προγραμματισμού, Πιστοποιημένος κώδικας

Abstract

The purpose of this diploma dissertation is on one hand the design of a simple high-level language that supports programming with proofs, and on the other hand the implementation of a compiler for this language. This compiler will produce code for an intermediate-level language suitable for creating certified binaries.

The need for reliable and certifiably secure code is even more pressing today than it was in the past. In many cases, security and software compatibility issues put in danger the operation of large systems, with substantial financial consequences. The lack of a formal way of specifying and proving the correctness of programs that characterizes current programming languages is one of the main reasons why these issues exist. In order to address this problem, a number of frameworks with support for certified binaries have recently been proposed. These frameworks offer the possibility of specifying and providing a formal proof of the correctness of programs. Such a proof can easily be checked for validity before running the program.

The frameworks that have been proposed are intermediate-level in nature, thus the process of programming in these is rather cumbersome. The high-level languages that accompany some of these frameworks, while very expressive, are hard to use. A simpler high-level language, like the one proposed in this dissertation, would enable further use of this programming idiom.

In the language we propose, the programmer specifies the partial correctness of a program by annotating function definitions with pre- and post-conditions that must hold for their parameters and results. The programmer also provides a set of theorems, based on which proofs of the proper implementation and use of the functions are constructed. An implementation in OCaml of a compiler from this language to the NFLINT certified binaries framework was also completed as part of this dissertation.

We managed to keep the language close to the feel of the current widespread functional languages, and also to fully separate the programming stage from the correctness-proving stage. Thus an average programmer can program in a familiar way in our language, and later an expert on formal logic can prove the semi-correctness of a program. As evidence of the practicality of our design, we provide a number of examples in our language with full semi-correctness proofs.

Key words

Programming languages, Programming with proofs, Secure programming languages, Certified code

Ευχαριστίες

Κατ'αρχάς θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου κο. Νικόλαο Παπασπύρου, για την καθοδήγησή του και τη βοήθειά του σε κάθε φάση της δημιουργίας της. Επίσης τα υπόλοιπα μέλη του εργαστηρίου Τεχνολογίας Λογισμικού, ιδιαίτερα δε τον Μιχάλη Παπακυριάκου και τον Χρήστο Δημουλά, η συνεισφορά των οποίων στην υλοποίηση του διερμηνέα του συστήματος NFLINT ήταν ιδιαίτερα σημαντική για την επιτυχή ολοκλήρωση της εργασίας αυτής. Θέλω να εκφράσω την ευγνωμοσύνη μου στους γονείς μου για την διαρκή τους υποστήριξη, που επέτρεψε την επιτυχή διεκπεραίωση των σπουδών μου. Τέλος θέλω να ευχαριστήσω τους φίλους και συναδέλφους μου για τα όμορφα φοιτητικά χρόνια που περάσαμε μαζί.

Αντώνης Μ. Σταμπούλης,

Αθήνα, 27η Ιουνίου 2006

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-06, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2006.

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

Περιεχόμενα

Σχήματα

Πίνακες

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός της εργασίας

Σκοπός της παρούσας εργασίας είναι αφενός η σχεδίαση μίας απλής γλώσσας υψηλού επιπέδου καθαρού συναρτησιακού προγραμματισμού με υποστήριξη για προγραμματισμό με αποδείξεις, και αφετέρου η υλοποίηση ενός μεταγλωττιστή για τη γλώσσα αυτή που θα παράγει κώδικα για μία γλώσσα ενδιάμεσου επιπέδου κατάλληλη για δημιουργία πιστοποιημένων εκτελέσιμων.

Με τον όρο *προγραμματισμός με αποδείξεις*, εννοούμε στη συγκεκριμένη εργασία την δημιουργία κώδικα που περιλαμβάνει μία περιγραφή μέσω λογικών προτάσεων των συνθηκών για τη μερική ορθότητα αυτού, δηλαδή μία *προδιαγραφή* για τον κώδικα, καθώς και κατάλληλων βοηθητικών στοιχείων ώστε να είναι δυνατή η δημιουργία μίας πλήρους λογικής απόδειξης της ισχύος των καθορισμένων συνθηκών. Ως *πιστοποιημένο εκτελέσιμο* ορίζουμε ένα εκτελέσιμο πρόγραμμα, που συνοδεύεται από ένα σύνολο ιδιοτήτων καθώς και την πλήρη απόδειξη των ιδιοτήτων αυτών. Επομένως, εάν ταυτίσουμε το σύνολο ιδιοτήτων με τις προδιαγραφές του αρχικού προγράμματος, το πιστοποιημένο εκτελέσιμο που μπορεί να προκύψει από μία γλώσσα που υποστηρίζει τον προγραμματισμό με αποδείξεις, θα περιλαμβάνει, πέρα από τον εκτελέσιμο κώδικα, τόσο την περιγραφή όσο και την απόδειξη της μερικής ορθότητας του κώδικα αυτού. Έτσι πριν από την εκτέλεση του κώδικα, μπορεί να διαπιστωθεί με μηχανιστικό και αποδοτικό τρόπο η μερική ορθότητα αυτού.

Η έμφαση στη σχεδίαση της γλώσσας υψηλού επιπέδου δίνεται στην ευχρηστία και φιλικότητα προς τον μέσο προγραμματιστή, παρά στην εκφραστική δυνατότητα της γλώσσας. Σκοπός είναι η γλώσσα να είναι όσο το δυνατόν πιο κοντά στις υπάρχουσες συναρτησιακές γλώσσες, με εύκολα κατανοητές επεκτάσεις για την ενσωμάτωση των προδιαγραφών του κώδικα μέσα σε αυτόν, και τα στοιχεία που πρέπει να δώσει ο προγραμματιστής για την διαδικασία της παραγωγής των αποδείξεων των προδιαγραφών να είναι τα ελάχιστα δυνατά.

1.2 Παρακίνηση για την εργασία

Στη σημερινή εποχή, η ανάγκη για αξιόπιστο και πιστοποιημένα ασφαλή κώδικα γίνεται διαρκώς ευρύτερα αντιληπτή. Τόσο κατά το παρελθόν όσο και πρόσφατα έχουν γίνει γνωστά προβλήματα ασφάλειας και συμβατότητας προγραμμάτων που είχαν ως αποτέλεσμα προβλήματα στην λειτουργία μεγάλων συστημάτων και συνεπώς οικονομικές επιπτώσεις στους οργανισμούς που τα χρησιμοποιούσαν.

Η έννοια της ασφάλειας είναι αλληλένδετη με την έννοια της ορθότητας του κώδικα, αφού κώδικας που είναι ορθός βάσει ενός κατάλληλα ορισμένου συνόλου επιτρεπτών λειτουργιών, δεν μπορεί να εκτελέσει λειτουργίες που δεν είναι ασφαλείς. Για παράδειγμα, η λήψη στοιχείου από πίνακα είναι ένα τυπικό παράδειγμα ανασφαλούς λειτουργίας, διότι υπάρχει η πιθανότητα να ζητείται στοιχείο με δείκτη μεγαλύτερο από το μέγεθος του πίνακα. Εάν η μοναδική στοιχειώδης λειτουργία για τη λήψη στοιχείου από πίνακα μπορεί να εκτελεστεί μόνον εάν ο δείκτης του στοιχείου που ζητείται είναι μικρότερος από το μέγεθος του πίνακα, τότε ένα τμήμα ορθού κώδικα θα είναι και ασφαλές ως προς αυτή τη λειτουργία, αφού η πιθανότητα για ανασφαλή λειτουργία αποκλείεται.

Μπορούμε να δούμε τον κώδικα σε μία συμβατική γλώσσα προγραμματισμού ως ένα σύνολο από μικρά ανεξάρτητα τμήματα (συναρτήσεις, κλάσεις, συνθετήματα, κ.λπ.), τα οποία συνεργάζονται μεταξύ τους μέσω διαπροσωπειών που ορίζουν το πρωτόκολλο διαλειτουργίας. Η σημασιολογία αυτών των τμημάτων καθορίζεται συνήθως με άτυπο τρόπο και έτσι δεν μπορεί να διαπιστωθεί με κάποια μηχανιστική διαδικασία εάν γίνεται ορθή χρήση αυτών και κατά πόσον αυτά συμφωνούν με την επιθυμητή σημασιολογία. Και τα δύο θέματα αυτά εναπόκεινται στον εκάστοτε προγραμματιστή.

Επομένως εάν μαζί με τα ανεξάρτητα τμήματα δίνουμε με κάποιον τρόπο μία προδιαγραφή της ορθής χρήσης και των επιθυμητών αποτελεσμάτων τους και εάν υπάρχει μία μηχανιστική διαδικασία που πιστοποιεί ότι και τα τμήματα αυτά είναι σύμφωνα με την προδιαγραφή τους και ότι η χρήση τους γίνεται σύμφωνα με τις προδιαγραφές αυτές, έχουμε επιτύχει την έννοια του ασφαλούς κώδικα. Μάλιστα υπάρχουν ήδη γλώσσες προγραμματισμού που ενσωματώνουν αυτή την ιδέα, με τη μορφή του *'σχεδιασμού με συμβόλαια'* (design by contract), που περιλαμβάνει τις έννοιες των προ- και μετα-συνθηκών στην κλήση συναρτήσεων, των αναλλοίωτων συνθηκών βρόχων και των τοπικών μεταβλητών, και άλλες. Η πιο διαδεδομένη από τις γλώσσες αυτές είναι η Eiffel. Το αρνητικό των γλωσσών αυτών είναι ότι ο έλεγχος των προδιαγραφών γίνεται κατά το χρόνο εκτέλεσης, με αποτέλεσμα να υπάρχει σημαντικό κόστος στην απόδοση του τελικού εκτελέσιμου κώδικα.

Βάσει πρόσφατης έρευνας προκύπτει, ότι, αν και αποτελεί αρκετά φιλόδοξο σχέδιο, υπάρχει μία εναλλακτική προσέγγιση για τη λύση του συγκεκριμένου προβλήματος, η οποία δεν έχει καμία αρνητική επίπτωση στην απόδοση του τελικού εκτελέσιμου κώδικα. Αυτή είναι η προσέγγιση που ονομάζουμε προγραμματισμός με αποδείξεις. Συγκεκριμένα, έχουν προκύψει μία πλειάδα από τυπο-θεωρητικά γενικά πλαίσια, με τη χρήση των οποίων γίνεται δυνατή η αναπαράσταση με τυπικό τρόπο πολύπλοκων προτάσεων και των αποδείξεών τους σε γλώσσες χαμηλού επιπέδου (ενδιάμεσες ή συμβολικές) με ισχυρά συστήματα τύπων. Σε μια τέτοια γλώσσα, ένα αρχείο πιστοποιημένου κώδικα είναι απλά ένα πρόγραμμα, του οποίου ο τύπος παρέχει μια συλλογή από ιδιότητες που το πρόγραμμα ικανοποιεί. Ο ελεγκτής τύπων της γλώσσας μπορεί στατικά και σχετικά εύκολα να αποκρίνεται για το αν ένα δεδομένο αρχείο είναι συνεπές και, στην περίπτωση που είναι, το πρόγραμμα μπορεί εν συνεχεία να εκτελεστεί χωρίς επιπλέον επιβάρυνση στην απόδοση. Ένα από τα γενικά πλαίσια αυτά είναι το FLINT (βλ. Shao *et al.* (2002)), καθώς και η παραλλαγή του NFLINT που έχει αναπτυχθεί στο Εργαστήριο Τεχνολογίας Λογισμικού του ΕΜΠ. Σε σύνδεση με τα παραπάνω, οι προδιαγραφές μπορούν να οριστούν ως σύνολο λογικών προτάσεων ενός τέτοιου πλαισίου, ενώ η απόδειξη της τήρησης της προδιαγραφής πρέπει να δίνεται μαζί με το πρόγραμμα. Η πιστοποίησή της δε, και άρα η πιστοποίηση της ορθότητας του προγράμματος, είναι μία μηχανιστική διαδικασία που δεν επιβαρύνει το χρόνο εκτέλεσης του προγράμματος.

Η παρακίνηση για τη συγκεκριμένη εργασία είναι να εξετάσουμε την προσέγγιση αυτή για τη δημιουργία πιστοποιημένα ορθού κώδικα. Μας ενδιαφέρει να διερευνήσουμε κατά πόσον είναι δυνατόν να σχεδιαστεί μία γλώσσα που ενώ είναι κοντά στη συνήθη πρακτική του προγραμματισμού και δεν αναγκάζει τον προγραμματιστή να ασχολείται με την δύσκολη διαδικασία της απόδειξης τυπικών προτάσεων, να έχει τη δυνατότητα να παράγει πιστοποιημένο κώδικα για ένα σύστημα όπως το NFLINT, σύμφωνο με τις προδιαγραφές που δίνονται.

1.3 Σύνοψη της εργασίας

Η δομή των κεφαλαίων που ακολουθούν είναι η εξής:

Κεφάλαιο 2. Στο κεφάλαιο αυτό παρουσιάζονται διάφορες γλώσσες που έχουν ήδη προταθεί και υποστηρίζουν τον προγραμματισμό με αποδείξεις.

Κεφάλαιο 3. Στο κεφάλαιο αυτό παρουσιάζεται η γλώσσα που σχεδιάσαμε, δίνεται η σύνταξή της, και παρέχονται παραδείγματα. Επισημαίνουμε διαφορές με τις γλώσσες του προηγούμενου κεφαλαίου.

Κεφάλαιο 4. Στο κεφάλαιο αυτό δίνονται οι κανόνες τύπων της γλώσσας καθώς και η σημασιολογία της.

Κεφάλαιο 5. Εδώ παρουσιάζουμε την γλώσσα τύπων καθώς και την γλώσσα υπολογισμών της ενδιάμεσης γλώσσας NFLINT για πιστοποιημένα εκτελέσιμα.

Κεφάλαιο 6. Εδώ ορίζουμε τη διαδικασία μετάφρασης από την υψηλού επιπέδου γλώσσα στην ενδιάμεση γλώσσα NFLINT, καθώς και τον τρόπο που λειτουργεί το υποσύστημα παραγωγής αποδείξεων που χρησιμοποιείται κατά τη διαδικασία μετάφρασης.

Κεφάλαιο 7. Συμπεράσματα που προκύπτουν από την εργασία και κατευθύνσεις για μελλοντική έρευνα.

Κεφάλαιο 2

Προγραμματισμός με αποδείξεις

Στο κεφάλαιο αυτό θα γίνει μία ανασκόπηση σε κάποιες υπάρχουσες γλώσσες υψηλού επιπέδου που υποστηρίζουν προγραμματισμό με αποδείξεις. Η παρουσίαση των γλωσσών είναι περιληπτική και υποθέτουμε ότι ο αναγνώστης έχει γνώση των εννοιών του συναρτησιακού προγραμματισμού, προτιμότερα μέσα από τις γλώσσες Haskell και ML.

Πριν προχωρήσουμε στην παρουσίαση των γλωσσών θα κάνουμε μία εισαγωγή στην έννοια των εξαρτώμενων τύπων (dependent types), διότι είναι βασική για την κατανόηση των γλωσσών αυτών και δεν είναι ευρέως διαδεδομένη στις συναρτησιακές γλώσσες που βρίσκονται σε χρήση.

2.1 Εξαρτώμενοι τύποι

Η έννοια του πολυμορφισμού είναι ευρέως διαδεδομένη τόσο στις συναρτησιακές όσο και στις αντικειμενοστραφείς γλώσσες. Μία πολυμορφική δήλωση τύπου δεδομένων ή συνάρτησης επιτρέπει την χρήση της για πολλαπλούς τύπους δεδομένων. Έτσι, είναι δυνατόν να ορίσουμε, για παράδειγμα, μόνον ένα τύπο δεδομένων για τις λίστες και μόνο μία φορά τις στοιχειώδεις πράξεις για αυτές, και να χρησιμοποιήσουμε τις δηλώσεις αυτές για λίστες κάθε τύπου – λίστες ακεραίων, λίστες χαρακτήρων, λίστες από λίστες ακεραίων, κ.ο.κ. Στις αντικειμενοστραφείς γλώσσες αυτή η δυνατότητα εμφανίζεται με τη μορφή των templates (C++) και των generics (Java, C#).

Μία τέτοια πολυμορφική δήλωση τύπου δεδομένων παράγει έναν κατασκευαστή τύπων, που δεδομένων των παραμέτρων τύπων κατασκευάζει ένα συγκεκριμένο τύπο δεδομένων της γλώσσας. Έχουμε έτσι ένα στοιχείο του επιπέδου των τύπων (τον κατασκευαστή) που εξαρτάται από τύπους. Η εξάρτηση τύπου από τύπους είναι επομένως μία γνωστή έννοια. Η έννοια των εξαρτώμενων τύπων (dependent types) είναι η επέκταση αυτής της εξάρτησης, ώστε να επιτρέψουμε εξάρτηση τύπου από όρους της γλώσσας. Έτσι μπορούμε να έχουμε κατασκευαστές τύπων που δέχονται ως παράμετρο τιμές της εκάστοτε γλώσσας.

Στην περίπτωση αυτή προκύπτει ένα σύστημα τύπων με μεγάλη εκφραστική δυνατότητα, κυρίως επειδή μπορούμε να δημιουργήσουμε πολύ πιο "εκλεπτυσμένους" τύπους. Μπορούμε για παράδειγμα να ορίσουμε τύπους για λίστες ακεραίων συγκεκριμένου μήκους, ή ξεχωριστούς τύπους ακεραίων για κάθε ακέραιο αριθμό – και έτσι να διαχωρίσουμε στο επίπεδο των τύπων δύο ακέραιες σταθερές. Η εκλέπτυνση των τύπων είναι δυνατή και για τις συναρτήσεις: για παράδειγμα μία συνάρτηση μπορεί να δέχεται ως παράμετρο μία λίστα μήκους ν και μία λίστα μήκους μ και να επιστρέφει ως αποτέλεσμα μία λίστα μήκους $\nu + \mu$. Ένας τέτοιος τύπος συνάρτησης θα ήταν κατάλληλος για τη συνάρτηση συνένωσης δύο λιστών, και μάλιστα καλύτερος (πιο λεπτομερής) από το συνηθισμένο τύπο της συγκεκριμένης συνάρτησης.

Επιτρέποντας την εξάρτηση τύπων από τιμές της εκάστοτε γλώσσας προγραμματισμού αδιακρίτως, προκύπτει ένα σημαντικό πρόβλημα: ο έλεγχος τύπων της γλώσσας προγραμματισμού παύει να είναι αποκρίσιμο πρόβλημα. Αυτό είναι φυσικό επακόλουθο του γεγονότος ότι στη γενική περίπτωση η αποτίμηση μίας έκφρασης μίας υπολογιστικά πλήρους γλώσσας δεν είναι απαραίτητο να τερματίζει. Έτσι, κατά τον έλεγχο τύπων, όταν προκύψει απαίτηση ισότητας μεταξύ δύο τύπων, εάν αυτοί εξαρτώνται από εκφράσεις, αυτές πρέπει να αποτιμηθούν πλήρως για να ελεγχθούν ως προς την ισότητα – και η αποτίμηση αυτή μπορεί να μην τερματίζει.

Στις γλώσσες που θα δούμε παρακάτω οι εξαρτώμενοι τύποι έχουν βασικό ρόλο. Σε κάποιες από αυτές το πρόβλημα της μη αποκρισιμότητας του ελέγχου τύπων έχει αντιμετωπιστεί με τρόπους που θα δούμε κατά την παρουσίαση της κάθε γλώσσας.

2.2 Cayenne

Η Cayenne έχει αναπτυχθεί από τον Lennart Augustsson στο Chalmers University of Technology στο Göteborg της Σουηδίας [Augu98]. Είναι μία καθαρή συναρτησιακή γλώσσα που υποστηρίζει εξαρτώμενους τύπους. Επειδή επιτρέπεται αδιάκριτη χρήση τιμών της γλώσσας στο επίπεδο των τύπων, ο έλεγχος τύπων της γλώσσας είναι όπως αναμένουμε, μη αποκρίσιμη διαδικασία.

Η Cayenne είναι πολύ κοντά στην Haskell. Οι τύποι αποτελούν αντικείμενα πρώτης τάξης, δηλαδή οι τύποι μπορούν να χρησιμοποιηθούν ως τιμές. Δεν υπάρχει διάκριση μεταξύ τύπων και τιμών στο συντακτικό επίπεδο της γλώσσας, και έτσι ο χειρισμός των τύπων και των τιμών είναι ο ίδιος: χρησιμοποιείται και στις δύο περιπτώσεις η αναδρομή και το ταίριασμα προτύπων (pattern matching). Ο τύπος των τύπων συμβολίζεται ως “#”.

Οι τύποι της Cayenne είναι αρκετά εκφραστικοί ώστε να μπορούμε να κωδικοποιήσουμε προτάσεις της κατασκευαστικής κατηγορηματικής λογικής ως τύπους της γλώσσας, μέσω του ισομορφισμού Curry-Howard. Επίσης ένας όρος της γλώσσας με έναν συγκεκριμένο τύπο, είναι κωδικοποίηση της λογικής απόδειξης της πρότασης στην οποία αυτός ο τύπος αντιστοιχεί. Ο ισομορφισμός αυτός, δηλαδή η ισοδυναμία μεταξύ τύπων και λογικών προτάσεων, καθώς και όρων και αποδείξεων, είναι ένα ευρέως γνωστό γενικό αποτέλεσμα για τις γλώσσες με εξαρτώμενους τύπους. Ας δούμε πώς αυτή η κωδικοποίηση είναι δυνατή στην Cayenne.

Η πάντα αληθής πρόταση (σταθερά $\top$) κωδικοποιείται ως οποιοσδήποτε τύπος έχει μόνο έναν κατασκευαστή χωρίς παραμέτρους. Το λογικό παράδοξο (σταθερά $\perp$) κωδικοποιείται ως ένας κενός τύπος (οπότε δεν μπορούν να δημιουργηθούν όροι με τύπο το λογικό παράδοξο – δεν υπάρχει απόδειξη ενός παραδόξου σε μία συνεπή λογική). Η λογική σύζευξη αναπαρίσταται ως ο τύπος των ζευγών (τύπος γινομένου), ενώ η λογική διάζευξη ως τύπος αθροίσματος – δηλαδή ένας τύπος που μπορεί να κατασκευαστεί από στοιχεία καθενός από δύο τύπους. Η λογική συνεπαγωγή “Α συνεπάγεται Β” ($A \to B$) αναπαρίσταται ως τύπος συνάρτησης από τον τύπο που αντιστοιχεί στο Α προς τον τύπο που αντιστοιχεί στο Β. Η λογική άρνηση του Α στην κατασκευαστική λογική είναι ισοδύναμη με τη λογική συνεπαγωγή $A \to \perp$. Ο καθολικός ποσοδείκτης αναπαρίσταται ως τύπος εξαρτώμενης συνάρτησης, όπου ο τύπος του αποτελέσματος της συνάρτησης εξαρτάται από τον τύπο του ορίσματος αυτής. Παρόμοια ο υπαρξιακός ποσοδείκτης αναπαρίσταται ως τύπος ισχυρού αθροίσματος (strong-sum type), ο οποίος χρησιμοποιείται για τις εγγραφές (records) της γλώσσας, και επιτρέπει εξάρτηση του τύπου ενός στοιχείου της εγγραφής από τον τύπο προηγουμένων στοιχείων αυτής. Η κωδικοποίηση αυτή φαίνεται στο σχήμα 2.1.

Βάσει αυτών είναι φανερό ότι η Cayenne έχει ένα σύστημα τύπων αρκετά εκφραστικό ώστε να αναπαραστήσουμε λογικές προτάσεις καθώς και τις αποδείξεις αυτών μέσα στη γλώσσα, κάτι που την κάνει κατάλληλη για προγραμματισμό με αποδείξεις. Μπορούμε έτσι για παράδειγμα να ορίσουμε το κατηγόρημα μεγαλύτερο-ίσο για ακέραιους αριθμούς, και να το χρησιμοποιήσουμε ώστε να απαιτήσουμε μία ακέραια παράμετρο μιας συνάρτησης να είναι μεγαλύτερη από μία ακέραια σταθερά:

```
example :: (x :: Int) -> (GreaterThan x 0) -> Int
```

Η παράμετρος τύπου `GreaterThan x 0` θα είναι ουσιαστικά μία απόδειξη της ισχύος της συνθήκης αυτής.

Ενώ η Cayenne μπορεί να χρησιμοποιηθεί για προγραμματισμό με αποδείξεις, κρίνεται ακατάλληλη για αυτό το σκοπό για τους εξής λόγους: κατ’αρχάς ο έλεγχος τύπων είναι μη

Υποθέτουμε τους ορισμούς τύπων:

```
data Truth = truth
data Absurd =
data Either x y = Left x | Right y
data Pair x y = MkPair x y
```

Κατηγορηματική λογική	Τύπος Cayenne
$\top$	`Truth`
$\bot$	`Absurd`
$x \vee y$	`Either x y`
$x \wedge y$	`Pair x y`
$\neg x$	`x -> Absurd`
$x \to y$	`x -> y`
$\forall x \in A.P(x)$	`(x :: A) -> P(x)`
$\exists x \in A.P(x)$	`{x :: A, y :: P(x)}`

Σχήμα 2.1: Κωδικοποίηση της κατηγορηματικής λογικής ως τύποι Cayenne.

αποκρίσιμος, οπότε η διαδικασία της μεταγλώττισης μπορεί να μην τερματίζεται. Κατά δεύτερον ο προγραμματιστής πρέπει να κατασκευάζει πλήρως τους όρους των αποδείξεων, κάτι που είναι ιδιαίτερα πολύπλοκο στις περισσότερες περιπτώσεις. Τρίτον οι αποδείξεις είναι τιμές της γλώσσας επομένως η κατασκευή και χρήση τους μέσα στις συναρτήσεις επιβαρύνει την απόδοση του τελικού προγράμματος στο χρόνο εκτέλεσης. Τα χαρακτηριστικά αυτά είναι συνεπαγόμενα του γεγονότος ότι ο προγραμματισμός με αποδείξεις δεν ήταν ένας από τους σχεδιαστικούς στόχους της συγκεκριμένης γλώσσας, αλλά φυσικό επακόλουθο της υποστήριξης για εξαρτώμενους τύπους.

2.3 Ωmega

Η Ωmega είναι μία συναρτησιακή γλώσσα που αποτελεί επέκταση της Haskell, με την εξαίρεση ότι δεν υποστηρίζει, όπως η Haskell, το σύστημα κλάσεων. Έχει αναπτυχθεί από τον Tim Sheard [Shea05a, Shea05b, Shea04]. Εδώ θα εξετάσουμε μόνον τα χαρακτηριστικά της γλώσσας που σχετίζονται με τον προγραμματισμό με αποδείξεις, και όχι άλλα ενδιαφέροντα χαρακτηριστικά της, όπως για παράδειγμα την υποστήριξη για πολλαπλά στάδια μετάφρασης.

Η πρώτη επέκταση που εισάγει η Ωmega αναφέρεται ως *επεκτάσιμο σύστημα ειδών*. Σε μία συναρτησιακή γλώσσα, οι όροι (οι τιμές) της γλώσσας κατηγοριοποιούνται από τύπους. Οι τύποι κατηγοριοποιούνται από τα **είδη** (kinds). Υπάρχει ένα προκαθορισμένο είδος, που είναι το είδος των τύπων της γλώσσας. Στην Ωmega το είδος αυτό συμβολίζεται ως $*^0$, άρα για παράδειγμα ο τύπος των ακεραίων έχει είδος $*^0$ ($\mathsf{Int} : *^0$). Η γλώσσα υποστηρίζει μία άπειρη ιεραρχία τέτοιων κατηγοριοποιήσεων, έτσι με τη σειρά του το είδος $*^0$ κατηγοριοποιείται από το $*^1$ κ.ο.κ. Η επεκτασιμότητα του συστήματος ειδών στην Ωmega σημαίνει ότι μπορεί ο προγραμματιστής να ορίσει δικά του είδη, που θα βρίσκονται έτσι στο ίδιο επίπεδο με το είδος των τύπων της γλώσσας – θα κατηγοριοποιούνται δηλαδή από το $*^1$. Τα είδη αυτά ορίζονται παρόμοια με τους αναδρομικούς τύπων δεδομένων. Οι κατασκευαστές που ορίζονται είναι και αυτοί τύποι (με είδη διαφορετικά του $*^0$). Επίσης υπάρχει υποστήριξη για ορισμό συναρτήσεων στο επίπεδο των τύπων. Ορίζονται παρόμοια με τις κανονικές συναρτήσεις της γλώσσας, δηλαδή με χρήση αναδρομής και ταιριάσματος προτύπων, αλλά οι συναρτήσεις αυτές πρέπει να είναι πλήρεις (total).

Η δεύτερη επέκταση που εισάγει η Ωmega είναι οι *γενικευμένοι αλγεβρικοί τύποι δεδομένων* (Generalized Algebraic Datatypes - GADTs), οι οποίοι έχουν μία λεπτή διαφορά σε σχέση με τους αλγεβρικούς τύπους δεδομένων που υποστηρίζονται από γλώσσες όπως η Haskell και η ML. Η διαφορά είναι ότι όταν οι αλγεβρικοί τύποι δεδομένων είναι πολυμορφικοί ως προς παράμετρο τύπου, τότε όλοι οι κατασκευαστές του τύπου παράγουν στιγμιότυπα του τύπου που ορίζονται με τις ίδιες παραμέτρους, ενώ οι γενικευμένοι, αναιρούν αυτόν τον περιορισμό. Σε συνδυασμό με το γεγονός ότι μπορούμε να έχουμε πολυμορφικούς τύπους με εξάρτηση από τύπους με είδος διαφορετικό του $*^0$, προκύπτει ότι στην Ωmega έχουμε υποστήριξη για εξαρτώμενους τύπους (με τον μηχανισμό των singleton τύπων που θα δούμε παρακάτω), χωρίς όμως να έχουμε εξάρτηση τύπου από όρο της γλώσσας. Έχουμε μόνο εξάρτηση τύπου από τύπο, αλλά τα χαρακτηριστικά που περιγράψαμε έως εδώ αρκούν για να έχουμε την ίδια εκφραστικότητα με ένα κανονικό σύστημα εξαρτώμενων τύπων. Επιπλέον, επιτυγχάνουμε να διατηρήσουμε την αποκρισιμότητα του ελέγχου τύπων, διότι η διαδικασία αποτίμησης εκφράσεων στο επίπεδο των τύπων της γλώσσας τερματίζει πάντα, αφού έχει αποκλειστεί η περίπτωση των συναρτήσεων στο επίπεδο των τύπων που δεν τερματίζουν για κάποιες εισόδους. Παρόμοια προσέγγιση χρησιμοποιείται στις περισσότερες γλώσσες που είναι κατάλληλες για προγραμματισμό με αποδείξεις.

Θα δούμε τώρα πώς αυτά τα χαρακτηριστικά της Ωmega μπορούν να χρησιμοποιηθούν για να έχουμε απόδειξη της μερικής ορθότητας του κώδικα, μέσα από παραδείγματα. Για την υποστήριξη εξαρτώμενων τύπων με τη συγκεκριμένη προσέγγιση, είναι βασική η έννοια των singleton τύπων (τύποι μοναδιαίου στιγμιοτύπου). Δεδομένης μίας τιμής στο επίπεδο των τιμών, θέλουμε συχνά να μιλάμε για την τιμή αυτή στο επίπεδο των τύπων. Για παράδειγμα, για να απαιτήσουμε σε μία συνάρτηση η τιμή μίας ακέραιας παραμέτρου της να είναι μεγαλύτερη του μηδενός, πρέπει να εκφράσουμε στον τύπο της συνάρτησης (δηλαδή στο επίπεδο των τύπων) την ανισοτική σχέση, και στη σχέση αυτή, χρειαζόμαστε μία αναπαράσταση της τιμής της παραμέτρου στο επίπεδο των τύπων. Έτσι μία συνήθης πρακτική είναι να ορίζουμε ένα πεδίο όπως οι φυσικοί αριθμοί τόσο στο επίπεδο των τιμών (με ορισμό ενός GADT), όσο και στο επίπεδο των τύπων (με αναδρομικό ορισμό ενός καινούριου είδους). Θέλουμε να υπάρχει σύνδεση μεταξύ των δύο επιπέδων, και για αυτό ορίζουμε τους φυσικούς αριθμούς στο επίπεδο των τιμών ως εξαρτώμενους από φυσικούς αριθμούς στο επίπεδο των τύπων. Η κατανόηση είναι ευκολότερη μέσω παραδείγματος, βλ. σχήμα 2.2.

```
kind Nat =
  TypeZero
| TypeSucc of Nat

data SNat :: Nat ~> *0 where
  ValueZero :: SNat TypeZero
  ValueSucc :: SNat x -> SNat (TypeSucc x)
```

Σχήμα 2.2: Ορισμός singleton τύπου φυσικών αριθμών στην Ωmega.

Στο παράδειγμα αυτό, το σύμβολο `~>` είναι ο κατασκευαστής συναρτήσεων στο επίπεδο των ειδών, ενώ το σύμβολο `->` είναι ο κατασκευαστής συναρτήσεων στο επίπεδο των τύπων της γλώσσας.

Μπορούμε επίσης να ορίσουμε τη σχέση "$\leq$" ως έναν τύπο που εξαρτάται από δύο φυσικούς αριθμούς του επιπέδου των τύπων και έχει ως κατασκευαστές τα αξιώματα της σχέσης μικρότερο-ίσο: $\forall \nu. 0 \leq \nu$ και $\forall \nu, \mu. n \leq m \rightarrow (\mathrm{Succ}\ n) \leq (\mathrm{Succ}\ m)$. Ένας όρος με τύπο $\alpha \leq \beta$ θα είναι ουσιαστικά απόδειξη της πρότασης αυτής, αφού θα είναι μία κατάλληλη σύνθεση των αξιωμάτων της σχέσης ώστε να προκύπτει τελικά ο συγκεκριμένος τύπος. Έτσι συναντάμε και πάλι τον ισομορφισμό Curry-Howard. Δεδομένης της σχέσης μικρότερο-ίσο, μπορούμε να απαιτήσουμε μία ακέραια παράμετρος μία συνάρτησης να είναι μικρότερη-ίση μιας άλλης ακέραιας τιμής, απαιτώντας μία κατάλληλη απόδειξη της ισχύος της σχέσης αυτής. Ομοίως μπορούμε να διαβεβαιώσουμε ότι η συνάρτηση επιστρέφει ακέραιο μικρότερο-ίσο μίας ακέραιας τιμής, παρέχοντας μία απόδειξη με τον αντίστοιχο τύπο. Στο σχήμα 2.3 φαίνεται ένα τέτοιο παράδειγμα, μαζί με τον ορισμό της σχέσης μικρότερο-ίσο.

```
-- ορισμός της σχέσης μικρότερο-ίσο μεταξύ φυσικών
data LessEqual :: Nat ~> Nat ~> *0 where
  LeBase :: LessEqual TypeZero x
  LeStep :: LessEqual x y -> LessEqual (TypeSucc x) (TypeSucc y)

-- παράδειγμα: απόδειξη ότι 1 μικρότερο ίσο του 2
le_1_2 :: LessEqual (TypeSucc (TypeZero)) (TypeSucc (TypeSucc (TypeZero)))
le_1_2 = LeStep (LeBase)

-- τύπος Covert: για να κρύβουμε το όρισμα ενός εξαρτώμενου
-- τύπου. Ουσιαστικά το χρησιμοποιούμε στην επόμενη συνάρτηση
-- στον τύπο επιστροφής, αντί για υπαρξιακό τύπο, αφού δεν
-- μπορούμε στο επίπεδο των τύπων να αναπαραστήσουμε το
-- αποτέλεσμα της συνάρτησης.
data Covert :: (Nat ~> *0) ~> *0 where
  Hide :: (t x) -> Covert t

-- συνάρτηση προηγούμενου : αφαιρεί 1 από το όρισμα, αλλά απαιτεί το
-- όρισμα να είναι μεγαλύτερο του μηδενός
pred :: SNat n -> (LessEqual (TypeSucc TypeZero) n) -> Covert SNat
-- pred ValueZero prf = αδύνατον, διότι δεν μπορεί να κατασκευαστεί
-- όρος με τύπο (LessEqual (TypeSucc TypeZero) (TypeZero))
pred (ValueSucc x) prf = x
```

Σχήμα 2.3: Παράδειγμα ορισμού και χρήσης της σχέσης μικρότερο-ίσο στην Ωmega.

Βάσει αυτών είναι φανερό ότι η Ωmega μπορεί να χρησιμοποιηθεί για προγραμματισμό με αποδείξεις. Οι αποδείξεις όπως παρατηρούμε είναι στο επίπεδο των τιμών, οπότε η κατασκευή και μεταχείρισή τους επιβαρύνει την απόδοση εκτέλεσης του προγράμματος. Επειδή δεν επιτρέπεται εξάρτηση είδους από είδος (υπό τη μορφή, για παράδειγμα, πολυμορφικών δηλώσεων ειδών), δεν μπορούμε να αποφύγουμε κάτι τέτοιο, ορίζοντας τις λογικές σχέσεις/προτάσεις όπως το μικρότερο-ίσο με κατηγοριοποίηση $*^1$ (οπότε οι όροι των αποδείξεων θα ήταν τύποι). Η γλώσσα ωστόσο έχει έναν μηχανισμό, ώστε οι λογικές προτάσεις να μην επιβαρύνουν το χρόνο εκτέλεσης, και αυτό γίνεται με την τρίτη επέκταση που εισάγει η γλώσσα, τις *στατικές προτάσεις*. Μέσω του μηχανισμού αυτού, μπορούμε να ορίσουμε στατικές προτάσεις, οι οποίες δεν επηρεάζουν το πρόγραμμα κατά το χρόνο εκτέλεσης, αλλάζοντας τη λέξη-κλειδί `data` στη δήλωση της πρότασης με τη λέξη-κλειδί `prop`. Τις στατικές προτάσεις μπορούμε να τις χρησιμοποιήσουμε για να θέσουμε περιορισμούς (constraints) στις δηλώσεις τύπων, οι οποίοι πρέπει να μπορούν να αποδειχθούν ότι ισχύουν κατά τον έλεγχο τύπων. Η απόδειξη των περιορισμών γίνεται μέσω ενός ενοποιητικού μηχανισμού λύσης περιορισμών (unification based constraint-solver). Οι συναρτήσεις μεταξύ στοιχείων με τύπους που είναι στατικές προτάσεις μπορούν να χρησιμοποιηθούν και αυτές από τον συγκεκριμένο μηχανισμό προς λύση των περιορισμών. Ο τύπος αυτών των συναρτήσεων είναι ουσιαστικά ένα θεώρημα, ενώ η υλοποίησή τους αποτελεί την απόδειξη του εκάστοτε θεωρήματος. Το παράδειγμα του σχήματος 2.3 με χρήση στατικών προτάσεων φαίνεται στο σχήμα 2.4, όπου επίσης φαίνεται και ένας ορισμός συνάρτησης - θεωρήματος για τη σχέση μικρότερο-ίσο.

Συνοψίζοντας, η Ωmega είναι μία σειρά επεκτάσεων στη Haskell που διατηρούν τη φιλοσοφία της αρχικής γλώσσας, οι οποίες επιτρέπουν να χρησιμοποιηθεί η γλώσσα για προγραμματισμό με αποδείξεις. Ο έλεγχος τύπων είναι αποκρίσιμη διαδικασία, και υπάρχει δυνατότητα οι

```
-- ορισμός της στατικής σχέσης μικρότερο-ίσο μεταξύ φυσικών
prop LessEqual :: Nat ~> Nat ~> *0 where
  LeBase :: LessEqual TypeZero x
  LeStep :: LessEqual x y -> LessEqual (TypeSucc x) (TypeSucc y)

-- ορισμός τύπου φυσικού ακεραίου με περιορισμό ο ακέραιος να
-- είναι μεγαλύτερος ίσος του 1
data NatLEone :: Nat ~> *0 where
  MkNatLEone :: LessEqual (TypeSucc TypeZero) n => SNat n -> NatLEone n

pred' :: NatLEone n -> Covert SNat
-- pred' MkNatLEone(ValueZero) = αδύνατον, διότι δεν μπορεί να ικανοποιηθεί
-- ο περιορισμός (LessEqual (TypeSucc TypeZero) (TypeZero))
pred' (MkNatLEone(ValueSucc x)) = Hide x

-- θεώρημα: εάν x <= y τότε x <= y+1
theorem1 :: LessEqual x y -> LessEqual x (TypeSucc y)
theorem1 LeBase = LeBase
theorem1 (LeStep prf) = LeStep (theorem1 prf)
```

Σχήμα 2.4: Χρήση στατικών προτάσεων στην Ωmega.

αποδείξεις να κατασκευάζονται αυτόματα από κατάλληλο μηχανισμό του διερμηνέα της γλώσσας κατά τον έλεγχο τύπων, χωρίς να επηρεάζουν την απόδοση του προγράμματος στο χρόνο εκτέλεσης. Τα χαρακτηριστικά αυτά είναι πολύ επιθυμητά για μία γλώσσα προγραμματισμού με αποδείξεις.

2.4 Dependent ML και ATS

Το Applied Type System ($\mathcal{ATS}$) είναι ένα πλαίσιο για τη σχεδίαση και φορμαλιστική περιγραφή προχωρημένων συστημάτων τύπων για γλώσσες προγραμματισμού, που έχει προταθεί από τον Hongwei Xi του Boston University [Xi04]. Η γλώσσα ATS είναι μία καθαρή συναρτησιακή γλώσσα στο στυλ της ML με ένα σύστημα τύπων που βασίζεται στο $\mathcal{ATS}$ [Chen05]. Μπορούμε να πούμε ότι η Dependent ML [Xi98] είναι ουσιαστικά ένα υποσύνολο της ATS, και ως τέτοιο θα την εξετάσουμε εδώ, αν και πρακτικά υπάρχουν διαφορές μεταξύ των δύο γλωσσών.

Ξεκινούμε από την Dependent ML. Στη γλώσσα αυτή, το είδος των τύπων της γλώσσας αναφέρεται ως *type*, ενώ υπάρχουν επίσης τα είδη *bool* και *int*. Οι όροι που κατηγοριοποιούνται με αυτά τα είδη αναφέρονται ως στατικοί δυαδικοί όροι και στατικοί ακέραιοι όροι αντίστοιχα. Τέτοιοι όροι είναι οι δυαδικές και ακέραιες σταθερές, οι γραμμικές πράξεις ακεραίων (πρόσθεση, αφαίρεση και συνάρτηση αντιθέτου), οι δυαδικές πράξεις (σύζευξη, διάζευξη, άρνηση), και οι σχέσεις διάταξης μεταξύ ακεραίων. Επειδή επιτρέπεται πολυμορφισμός ως προς όρους καθενός από τα τρία αυτά είδη, προκύπτει ότι έχουμε τιμές και τύπους της γλώσσας που εξαρτώνται από τύπους (όπως στις συνήθεις συναρτησιακές), αλλά και από στατικούς δυαδικούς και ακέραιους όρους. Με αυτόν τον τρόπο, έχουμε και στην Dependent ML τη μορφή των εξαρτώμενων τύπων που είδαμε στην Ωmega, όπου δηλαδή υπάρχει διαχωρισμός μεταξύ του επιπέδου των τύπων και του επιπέδου των τιμών. Η διαφορά με την Ωmega είναι ότι ο προγραμματιστής δεν μπορεί να ορίσει είδη εκτός των ήδη υπαρχόντων

Οι τύποι που υποστηρίζει η γλώσσα είναι οι εξής:

- τύπος ακεραίου εξαρτώμενος από στατικό ακέραιο όρο, με συμβολισμό $\mathbf{int}(I)$, όπου I οι στατικοί ακέραιοι όροι

- δυαδικός τύπος εξαρτώμενος από στατικό δυαδικό όρο, με συμβολισμό $\mathbf{bool}(B)$, όπου B οι στατικοί δυαδικοί όροι

- τύπος συνάρτησης ($T_1 \rightarrow T_2$)

- ο καθολικός τύπος $\forall\alpha : \sigma.T$, όπου σ το σύνολο των ειδών. Ο τύπος αυτός χαρακτηρίζει πολυμορφικές δηλώσεις συναρτήσεων και τύπων δεδομένων. Συμβολίζεται με άγκιστρα, δηλαδή ως $\{\alpha : \sigma\}\ T$.

- ο υπαρξιακός τύπος $\exists\alpha : \sigma.T$, ο οποίος χρησιμοποιείται για παράδειγμα για να αποφύγουμε να προσδιορίσουμε την παράμετρο ενός εξαρτώμενου τύπου, δημιουργώντας έτσι έναν μη-εξαρτώμενο τύπο από έναν εξαρτώμενο. Συμβολίζεται με αγκύλες. Για παράδειγμα ο μη εξαρτώμενος τύπος ακεραίων είναι ο $[n : \mathsf{int}]\ \mathbf{int}(n)$.

- γενικευμένοι αλγεβρικοί τύποι δεδομένων

- τύπος με φρουρό (guarded datatype), ο οποίος δέχεται μία παράμετρο είδους *bool* και έναν τύπο, και χρησιμοποιείται στην δήλωση συναρτήσεων. Ο τύπος αυτός υποδηλώνει ότι η συνάρτηση που ορίζεται μπορεί να εφαρμοστεί μόνον εάν ισχύει η λογική συνθήκη του τύπου. Συμβολίζεται ως συνθήκη μέσα σε καθολικό ποσοδείκτη. Για παράδειγμα ο τύπος με φρουρό ακεραίου μεγαλύτερου του 0 είναι $\{n : \mathsf{int} \mid n > 0\}\ \mathbf{int}(n)$.

- τύπος με διαβεβαίωση (asserting datatype), που είναι το αντίστροφο του τύπου με φρουρό. Υποδηλώνει ότι η συνάρτηση που ορίζεται επιστρέφει τιμή για την οποία ισχύει η συνθήκη του τύπου. Συμβολίζεται ως συνθήκη μέσα σε υπαρξιακό ποσοδείκτη. Για παράδειγμα ο τύπος με διαβεβαίωση ακεραίου μεγαλύτερου του 0 είναι $[n : \mathsf{int} \mid n > 0]\ \mathbf{int}(n)$.

Για παράδειγμα ο τύπος μίας συνάρτησης που παίρνει έναν θετικό ακέραιο και επιστρέφει έναν αρνητικό θα είναι: $\{n : \mathsf{int} \mid n \geq 0\}\ \mathbf{int}(n) \rightarrow [r : \mathsf{int} \mid r < 0]\ \mathbf{int}(r)$. Ο έλεγχος για το εάν οι περιορισμοί που επιβάλλουν οι τύποι με φρουρό και με διαβεβαίωση ισχύουν, μπορεί να γίνει με οποιαδήποτε μέθοδο γραμμικού προγραμματισμού, διότι αν παρατηρήσουμε τον ορισμό των στατικών όρων προκύπτει ότι οι περιορισμοί που προκύπτουν έχουν τη μορφή γραμμικών ανισώσεων ακεραίων. Έτσι ο έλεγχος τύπων της γλώσσας πρέπει να περιλαμβάνει και ένα μηχανισμό προς επίλυση συστημάτων τέτοιων ανισώσεων. Επειδή για τα συστήματα αυτά μπορεί είτε να βρεθεί λύση σε πολυωνυμικό, ή σχεδόν πολυωνυμικό, χρόνο, είτε να δειχθεί ότι είναι αδύνατα συστήματα, ο έλεγχος τύπων παραμένει μία πάντα αποκρίσιμη διαδικασία. Εάν βέβαια κάποιες από τις συνθήκες των τύπων με φρουρό ή διαβεβαίωση δεν τηρούνται (όπως για παράδειγμα εάν σε μία συνάρτηση με τον τύπο του παραδείγματος δώσουμε ως όρισμα έναν αρνητικό ακέραιο), ο έλεγχος τύπων θα αποφανθεί ότι το πρόγραμμα περιέχει σφάλμα τύπων.

Τα στοιχεία αυτά της γλώσσας αρκούν για να ορίσουμε πιο λεπτομερείς τύπους για ορισμένες συναρτήσεις, με αποτέλεσμα να αποκαλύπτονται περισσότερα πιθανά σφάλματα κατά τον έλεγχο τύπων. Για παράδειγμα, δεδομένου ενός τύπου λίστας εξαρτώμενου από το μήκος αυτής, μπορούμε να ορίσουμε τη συνάρτηση συνένωσης λιστών ως μία συνάρτηση που δέχεται δύο λίστες και επιστρέφει μία λίστα με μήκος ίσο με το άθροισμα των μηκών των δύο λιστών, και τη συνάρτηση αντιστροφής λίστας ως μία συνάρτηση που δέχεται μία λίστα κάποιου μήκους και επιστρέφει λίστα του ιδίου μήκους. Το παράδειγμα αυτό φαίνεται στο σχήμα 2.5.

Παρ'όλα αυτά, η εκφραστικότητα των τύπων της Dependent ML είναι περιορισμένη σε σχέση με την Ωmega. Για παράδειγμα, δεν μπορούμε με κάποιον τρόπο να δώσουμε έναν ακριβή τύπο για τη συνάρτηση πολλαπλασιασμού ακεραίων, αφού δεν μπορούμε να εκφράσουμε τον πολλαπλασιασμό μεταξύ στατικών ακέραιων όρων. Η ATS επεκτείνει το σύστημα που έχουμε περιγράψει έως εδώ με δυνατότητα για αναδρομικό ορισμό ειδών από τον προγραμματιστή, καθώς και με ένα νέο είδος, το *prop*, το οποίο χρησιμοποιείται για ορισμό λογικών

```
datatype list (type, int) =
   {a : type} nil(a, 0)
 | {a : type, n : int}
              cons(a,n+1) of a * list(a,n)

fun append {a: type, n:nat, m:nat}
     (x: list(a,m), y:list(a,n)) : list(a,m+n) =
   case x of
     | nil => y
     | cons(x, xs) => cons(x, append(xs,y))

fun reverse {a: type, n:nat} (l : list(a,n)) : list(a,n) =
   case l of
     | nil => nil
     | cons(x, xs) => append(reverse(xs), cons(x, nil))
```

Σχήμα 2.5: Παράδειγμα συναρτήσεων χειρισμού λίστας στην Dependent ML.

κατηγορημάτων (παρόμοια με τις στατικές προτάσεις της Ωmega). Μπορούμε να ορίσουμε ένα λογικό κατηγόρημα όπως ορίζουμε έναν αναδρομικό τύπο δεδομένων. Οι όροι που έχουν ως τύπο ένα τέτοιο κατηγόρημα αποτελούν ουσιαστικά απόδειξη του κατηγορήματος. Επίσης μπορούμε να ορίσουμε συναρτήσεις μεταξύ κατηγορημάτων, οι οποίες αποτελούν θεωρήματα, ενώ οι υλοποιήσεις τους αποτελούν αποδείξεις αυτών. Οι συναρτήσεις αυτές πρέπει να είναι πλήρεις, κάτι που διασφαλίζεται δίνοντας κατά τον ορισμό τους μία μετρική τερματισμού, που πρέπει να φθίνει κατά την αναδρομική κλήση της εκάστοτε συνάρτησης. Τέλος προστίθεται ένας καινούριος τύπος, ο τύπος ζεύγους λογικού κατηγορήματος και τύπου, που μπορεί να χρησιμοποιηθεί για να απαιτήσουμε την ύπαρξη απόδειξης για μία πρόταση ή να διαβεβαιώσουμε ότι υπάρχει απόδειξη για μία πρόταση. Να σημειώσουμε ότι οι όροι αποδείξεων μας ενδιαφέρουν μόνο κατά τον έλεγχο τύπων, και η αποτίμηση των εκφράσεων της γλώσσας γίνεται χωρίς να λαμβάνονται οι όροι αυτοί υπόψη, άρα και χωρίς κάποιο κόστος στον χρόνο εκτέλεσης. Τα χαρακτηριστικά αυτά της γλώσσας ATS φαίνονται στο παράδειγμα του σχήματος 2.6.

Έτσι, η ATS μπορεί να αποδεικνύει αυτόματα προτάσεις για ένα συγκεκριμένο πεδίο σχέσεων (γραμμικές ανισώσεις ακεραίων), ενώ για άλλα πεδία πρέπει να κατασκευάζονται μέσα στο πρόγραμμα πλήρεις αποδείξεις. Τα υπόλοιπα χαρακτηριστικά της γλώσσας είναι παρόμοια με την Ωmega.

2.5 Vero

Η Vero είναι μία συναρτησιακή γλώσσα προγραμματισμού που έχει προταθεί από τον Xinyu Feng [Feng04]. Έχει στηριχθεί κατά μεγάλο μέρος στο σύστημα τύπων για πιστοποιημένα εκτελέσιμα που περιγράφεται στο Shao *et al.* (2002), στο οποίο στηρίζεται και το σύστημα NFLINT για το οποίο παράγει κώδικα ο μεταγλωττιστής της γλώσσας που κατασκευάσαμε.

Η γλώσσα αυτή αποτελείται από δύο ξεχωριστά μέρη, το επίπεδο της γλώσσας αποδείξεων, το οποίο χρησιμοποιείται ως ένα πλαίσιο λογικής, και το επίπεδο της γλώσσας υπολογισμών. Η γλώσσα αποδείξεων είναι ουσιαστικά ταυτόσημη με τη γλώσσα τύπων που περιγράφεται στο TSCB [Shao02], η οποία είναι ένας λ-λογισμός με υποστήριξη για επαγωγικούς ορισμούς ειδών. (Η μόνη διαφορά μεταξύ των δύο γλωσσών είναι ότι το είδος των αποδείξεων στην Vero ονομάζεται Prop αντί για Kind). Προτείνουμε στον αναγνώστη να διαβάσει το σχετικό κείμενο για την κατανόηση της γλώσσας αυτής, μιας και η αναλυτική παρουσίασή της είναι εκτός των πλαισίων αυτής της εργασίας. Ωστόσο, στο κεφάλαιο 5 γίνεται περιληπτική περιγραφή της

```
(* ορισμός κατηγορήματος για τον πολλαπλασιασμό ακεραίων:
   υπάρχει απόδειξη για το MUL(n,m,p) αν p = n*m *)
dataprop MUL (int, int, int) =
  | {n : int} MULbas (0, n, 0)
  | {m : int, n : int, p : int | m > 0}
              MULind (m, n, p) of MUL(m-1, n, p-n)
  | {m : int, n : int, p : int | m > 0}
              MULneg (m, n, p) of MUL(-m,  n, -p)

(* υποθέτουμε την συνάρτηση mul για πολλαπλασιασμό ακεραίων:
mul : {a1 : int, a2 : int} (int(a1), int(a2)) ->
      {a3 : int} MUL(a1,a2,a3) * int(a3)
*)

(* θεώρημα: αν a1>=0 και a2>=0 τότε a1*a2>=0 *)
(* με '() συμβολίζεται ο μοναδικός όρος του μοναδιαίου
   τύπου (unit) *)
prfun lemma {a1:int, a2:int, a3:int | a1 >= 0, a2 >= 0}
   .<a1>. (* το a1 είναι η μετρική τερματισμού *)
   (prf : MUL(a1, a2, a3)) : [a3 >= 0] '() =
   case prf of
       MULbas => '()
     | MULind prf => lemma prf

(* συνάρτηση παραγοντικού, η οποία διαβεβαιώνει ότι
   το αποτέλεσμά της είναι μεγαλύτερο του 0 *)
fun factorial {a : int | a >= 0} (x : int(a)) : {r : int | r >= 0} int(r) =
   if x == 0 then
      1
   else
      let
        val '(pf | r) = x mul factorial (pred x)
        prval _ = lemma pf   (* απόδειξη ότι r >= 0: αφού προκύπτει από
                                μη γραμμική σχέση δεν μπορεί να εξαχθεί
                                αυτόματα, και έτσι πρέπει να δημιουργηθεί
                                χειροκίνητα σε αυτή τη θέση, μιας και η
                                ύπαρξή της επιβάλλεται από τον τύπο με
                                διαβεβαίωση της συνάρτησης *)
      in
        r
```

Σχήμα 2.6: Προγραμματισμός με αποδείξεις στην ATS.

```
Inductive Nat : Prop = O : Nat | S : Nat -> Nat;

Inductive Bool : Prop = True : Bool | False : Bool;

(* Η δήλωση Fix αντιστοιχεί στην εντολή Fixpoint του Coq *)
Fix plus [n1 : Nat] : Nat -> Nat =
    [n2 : Nat] <: Nat :> Cases n1 of O => n2
                                 | S n' => S (plus n' n2)
                         End;

(* Η δήλωση type αντιστοιχεί στην εντολή Definition του Coq *)
type ifez : Nat -> (k : Prop) k -> (Nat -> k) -> k =
    [n : Nat] [k:Prop, p1:k, p2:Nat->k]
              <: k :> Cases n of O => p1
                              | S n' => p2 n'
                      End;

Fix le [n1 : Nat] : Nat -> Bool =
    [n2 : Nat]
    <: Bool :> Cases n1 of O => True
                         | S n' => ifez n2 Bool False (le n')
               End;
```

Σχήμα 2.7: Παραδείγματα στη γλώσσα αποδείξεων της Vero.

γλώσσας τύπων του NFLINT, η οποία αποτελεί μικρή παραλλαγή αυτής του TSCB.

Η γλώσσα αποδείξεων της Vero είναι αρκετά κοντά στον Λογισμό των Επαγωγικών Κατασκευών (Calculus of Inductive Constructions – CIC) που χρησιμοποιείται στον βοηθό κατασκευής αποδείξεων Coq [CDev03]. Μάλιστα οι όροι της γλώσσας αποδείξεων γράφονται στην Vero με σύνταξη πολύ παρόμοια με αυτή που χρησιμοποιείται στο Coq. Περιληπτικά μπορούμε να πούμε ότι επιτρέπονται δηλώσεις επαγωγικών και μη τύπων, καθώς και δήλωση πλήρων συναρτήσεων με χρήση πρωταρχικής αναδρομής. Στο σχήμα 2.7 φαίνονται δηλώσεις στη γλώσσα αποδείξεων της Vero για τους φυσικούς ακέραιους, τις δυαδικές μεταβλητές, για τη συνάρτηση πρόσθεσης ακεραίων, καθώς και για τη συνάρτηση μικρότερο-ίσο μεταξύ ακεραίων.

Η γλώσσα υπολογισμών της Vero είναι μία επέκταση της γλώσσας υπολογισμών του TSCB και του NFLINT, με την κύρια διαφορά ότι υποστηρίζει αλγεβρικούς τύπους δεδομένων. Οι όροι της γλώσσας υπολογισμών κατηγοριοποιούνται από τύπους ξεχωριστούς από τη γλώσσα αποδείξεων, οι οποίοι έχουν είδος Kind. Επιτρέπεται εξάρτηση όρων της γλώσσας υπολογισμών από όρους της γλώσσας αποδείξεων αλλά όχι το αντίστροφο – έτσι έχουμε την προσέγγιση που έχουμε δει στην Ωmega και την ATS, με διατήρηση της αποκρισιμότητας του ελέγχου τύπων. Επίσης οι εξαρτώμενοι τύποι υποστηρίζονται και σε αυτή την περίπτωση μέσω του μηχανισμού των singleton τύπων: για παράδειγμα ο τύπος ενός φυσικού αριθμού εξαρτάται από έναν όρο αποδείξεων τύπου Nat (όπως αυτός δηλώθηκε στο σχήμα 2.7). Ας δούμε πιο αναλυτικά τους τύπους που υποστηρίζει η γλώσσα υπολογισμών της Vero και τις εκφράσεις που αντιστοιχούν σε αυτούς.

- βασικοί τύποι, όπως μοναδιαίος τύπος (Unit), singleton τύπος φυσικού αριθμού με εξάρτηση από όρο γλώσσας αποδείξεων τύπου Nat, singleton δυαδικός τύπος με εξάρτηση από όρο γλώσσας αποδείξεων τύπου Bool

- τύπος συνάρτησης

```
(* ορισμός του μη εξαρτώμενου τύπου φυσικού αριθμού *)
type nat : Kind = <n : Nat> snat n.

val add : nat * nat -> nat =
   fn ns : nat * nat =>
      let val n1 = #1 ns
          val n2 = #2 ns
          open <n1', sn1> = n1
          open <n2', sn2> = n2
      in
          pack n = plus n1' n2'
          with sn1 + sn2 : snat n
          end
      end;
```

Σχήμα 2.8: Συνάρτηση πρόσθεσης ακεραίων στην Vero.

```
(* Υποθέτουμε ότι έχει οριστεί στην γλώσσα αποδείξεων
   η σχέση ισοδυναμίας Eq μεταξύ δύο όρων της *)

type rec list : Kind -> Nat -> Kind =
[t: Kind, n : Nat]
    { nil  of <prf: Eq Nat 0 n> unit
     |cons of <n': Nat, prf: Eq Nat (S n') n> t * (list t n')
    }
```

Σχήμα 2.9: Ορισμός εξαρτώμενου τύπου λίστας στην Vero.

- καθολικός τύπος για πολυμορφισμό είτε ως προς τύπο της γλώσσας υπολογισμού, είτε ως προς όρο της γλώσσας αποδείξεων. Συμβολίζεται με αγκύλες.
- υπαρξιακός τύπος για απαίτηση ύπαρξης είτε τύπου γλώσσας υπολογισμού, είτε όρου της γλώσσας αποδείξεων, χωρίς να γίνεται φανερός στο επίπεδο των τύπων. Συμβολίζεται με γωνιακές αγκύλες. Για παράδειγμα ο μη εξαρτώμενος τύπος φυσικών είναι $\langle n : Nat\rangle\ snat(n)$.
- αλγεβρικοί τύποι δεδομένων, με δυνατότητα για ορισμό αμοιβαίως αναδρομικών τύπων δεδομένων

Οι εκφράσεις τις γλώσσας είναι οι εξής: σταθερές βασικών τύπων, πράξεις μεταξύ ακεραίων και δυαδικών μεταβλητών, ορισμός συνάρτησης (μέσω της λέξης-κλειδί fn), πολυμορφισμός ως προς τύπο ή όρο της γλώσσας αποδείξεων (συμβολισμός όπως και οι καθολικοί τύποι, με χρήση αγκύλων), εκφράσεις pack και open για κατασκευή και αποδόμηση υπαρξιακών τύπων, ταίριασμα προτύπων (δομή case), και τοπικοί ορισμοί ονομάτων για εκφράσεις (δομή let). Στο σχήμα 2.8 φαίνεται ένα παράδειγμα για τη συνάρτηση πρόσθεσης ακεραίων. Στο σχήμα 2.9 φαίνεται ένα παράδειγμα ορισμού τύπου λίστας με εξάρτηση από το μήκος της. Με χρήση της σχέσης ισοδυναμίας Eq που πρέπει να έχει οριστεί στο επίπεδο της γλώσσας αποδείξεων, αποδεικνύεται ότι μπορεί να αρθεί ο περιορισμός σε αλγεβρικούς τύπους δεδομένων που περιγράψαμε στην Ωmega, και να ορίσουμε έτσι γενικευμένους αλγεβρικούς τύπους δεδομένων.

Τέλος στο σχήμα 2.10 παρουσιάζεται ένα παράδειγμα σε Vero με εμφανή τη χρήση αποδείξεων. Ορίζουμε στην γλώσσα αποδείξεων τον πολλαπλασιασμό φυσικών καθώς και ένα κατηγόρημα για τη σχέση "μικρότερο" μεταξύ φυσικών. Έπειτα ορίζουμε στη γλώσσα υπολογισμών τη συνάρτηση παραγοντικού, η οποία εκτός του αποτελέσματος επιστρέφει και μία απόδειξη ότι αυτό είναι μεγαλύτερο του μηδενός. Η συνάρτηση άπαξ και περάσει από τον έλεγχο τύπων θα είναι μερικώς ορθή σύμφωνα πάντα με τη συγκεκριμένη προδιαγραφή – δεν έχουμε τρόπο να αποδείξουμε ότι η συνάρτηση σίγουρα θα τερματίσει, οπότε δεν μπορούμε παρά να μιλάμε για μερική ορθότητα μόνο. Να σημειώσουμε ότι το παράδειγμα αυτό υποθέτει μία συγκεκριμένη μορφή για το ταίριασμα προτύπων ακεραίων, την οποία η γλώσσα Vero δεν διαθέτει στη μορφή που παρουσιάζεται στο [Feng04]. Κάποια παρόμοια όμως δομή είναι απαραίτητη για την υλοποίηση των περισσότερων συναρτήσεων μεταξύ ακεραίων. Λαμβάνοντας υπόψη το γεγονός ότι δεν έχει υλοποιηθεί μεταγλωττιστής για τη γλώσσα, άρα η γλώσσα δεν είναι απαραίτητα ολοκληρωμένη και δεν έχει δοκιμαστεί, μπορούμε να υποθέσουμε ότι μία τέτοια δομή θα υπήρχε σε μία τελική έκδοση της γλώσσας.

Εν κατακλείδι, μπορούμε να πούμε ότι η Vero είναι μία γλώσσα που απλοποιεί αρκετά τον προγραμματισμό με αποδείξεις σε σχέση με ένα σύστημα ενδιάμεσου επιπέδου όπως το NFLINT, λόγω της υψηλότερου επιπέδου φύσης της. Επίσης, έχει ένα σύστημα τύπων εξίσου ισχυρό με την Ωmega και την ATS, αφού αρκεί για την δήλωση και χρήση γενικευμένων αλγεβρικών τύπων δεδομένων. Οι αποδείξεις δεν επηρεάζουν το χρόνο εκτέλεσης του προγράμματος, διότι αποτελούν στοιχεία ξεχωριστά από τους όρους υπολογισμούς της γλώσσας που χρησιμοποιούνται μόνο κατά τον έλεγχο τύπων. Ωστόσο, ο συνδυασμός των εξαρτώμενων τύπων και η μη ύπαρξη κάποιου μηχανισμού για αυτόματη κατασκευή (κάποιων) αποδείξεων, έχουν ως αποτέλεσμα ο προγραμματισμός στη γλώσσα αυτή να είναι πολύ δύσκολος για έναν μέσο προγραμματιστή που δεν έχει εξοικείωση με τις σχετικές έννοιες.

```
(* δήλωση πολλαπλασιασμού φυσικών αριθμών στη γλώσσα αποδείξεων *)
Fix mul [n1: Nat] : Nat -> Nat =
    [n2 : Nat] <: Nat :> Cases n1 of O => O
                                   | S n' => plus n2 (mul n' n2)
                         End;

(* δήλωση της σχέσης μικρότερο μεταξύ φυσικών αριθμών *)
Inductive LT : Nat -> Nat -> Prop =
    ltzs : (n : Nat) LT O (S n)
   |ltss : (n : Nat, m  : Nat) LT n m -> LT (S n) (S m);

(* θεώρημα: αν 0 < x και 0 < y τότε 0 < x * y.
   η απόδειξη του θεωρήματος δεν παρουσιάζεται εδώ, δεν είναι
   απλή υπόθεση ωστόσο! Σαφώς δεν μπορούμε να περιμένουμε κάποιον
   αυτόματο μηχανισμό κατασκευής αποδείξεων μίας γλώσσας να
   κατασκευάζει αποδείξεις για θεωρήματα όπως αυτό. *)
type theorem1 : (x : Nat, y : Nat) LT O x -> LT O y -> LT O (mul x y)
= (* ... *);

(* συνάρτηση παραγοντικού στη γλώσσα υπολογισμών της Vero *)
val rec factorial : (n : Nat) snat n -> <r: nat> <prf: LT O r> snat r =
    [nt : nat] fn n : snat nt =>
                  case n of
                    | 0 =>
                       pack r = (S O)
                       with pack prf = ltzs O
                            with 1 : snat (S O)
                            : <prf: LT O r> snat r
                    | 1 + n' (S nt') =>
                       let val resp = factorial [nt'] n'
                           open <rest, res'> = resp
                           open <resprf, res>= res'
                       in
                           pack r = mul nt rest
                           with pack prf = theorem1 nt rest (ltzs nt') resprf
                             (* είναι εύλογο να θεωρήσουμε ότι αυτή
                                η απόδειξη θα μπορούσε να δημιουργηθεί
                                από κάποιον αυτόματο μηχανισμό κατασκευής
                                αποδείξεων, δεδομένου του theorem1 *)
                                with n * res : snat (mul nt rest)
                                : <prf : LT O r> snat r
```

Σχήμα 2.10: Προγραμματισμός με αποδείξεις στην Vero.

Κεφάλαιο 3

Εισαγωγή στην Karooma

Στο κεφάλαιο 2 είδαμε μία σειρά συναρτησιακών γλωσσών που υποστηρίζουν προγραμματισμό με αποδείξεις. Η χρήση εξαρτώμενων τύπων είναι θεμελιώδους σημασίας προς αυτό τον σκοπό, στο σύνολο των γλωσσών που εξετάσαμε. Σε κάποιες από αυτές (Ωmega, ATS για συγκεκριμένες περιπτώσεις), υπάρχει δυνατότητα για αυτόματη κατασκευή κάποιων αποδείξεων, ενώ σε άλλες η κατασκευή αποδείξεων πρέπει να γίνεται χειροκίνητα από τον προγραμματιστή. Στην πλειοψηφία των γλωσσών οι αποδείξεις ανήκουν στο επίπεδο των τύπων και έτσι επηρεάζουν μόνο τον έλεγχο τύπων. Η κατασκευή και μεταχείρισή τους δεν επιβαρύνει το χρόνο εκτέλεσης.

Στο κεφάλαιο αυτό θα παρουσιάσουμε τη συναρτησιακή γλώσσα Karooma που σχεδιάσαμε. Στόχος της σχεδίασής της ήταν να επιτύχουμε ένα ιδίωμα προγραμματισμού που είναι όσο το δυνατόν πιο κοντά στις διαδεδομένες γλώσσες συναρτησιακού προγραμματισμού, διατηρώντας όμως τη δυνατότητα να προδιαγράφουμε τη μερική ορθότητα των προγραμμάτων που γράφουμε σε αυτή. Προς εκπλήρωση αυτού του στόχου, πήραμε δύο αποφάσεις καθοριστικές για τη μορφή της γλώσσας: η διαδικασία απόδειξης των προδιαγραφών είναι απόλυτα ξεχωριστή από τη διαδικασία υλοποίησης του προγράμματος, ενώ η γλώσσα δεν έχει εξαρτώμενους τύπους. Η προδιαγραφή της μερικής ορθότητας του προγράμματος γίνεται μέσω της διαδεδομένης πρακτικής των προσυνθηκών (preconditions) και μετασυνθηκών (postconditions). Έτσι κατά τον ορισμό μίας συνάρτησης ο προγραμματιστής πρέπει να προδιαγράφει τις συνθήκες που πρέπει να ισχύουν για τις παραμέτρους για να κληθεί η συνάρτηση, καθώς και τις συνθήκες που βεβαιώνει η συνάρτηση ότι θα ισχύουν για το αποτέλεσμα που επιστρέφει.

Μπορούμε να πούμε ότι αναγνωρίζουμε δύο ρόλους κατά τον προγραμματισμό στην Karooma: το ρόλο του προγραμματιστή και το ρόλο του αποδείκτη. Ο προγραμματιστής δεν χρειάζεται να έχει κατανόηση των βασικών εννοιών του προγραμματισμού με αποδείξεις, όπως αυτές έγιναν φανερές κατά την παρουσίαση των διαφόρων ανάλογων γλωσσών στο κεφάλαιο 2, ούτε εμπειρία στην μαθηματική λογική και στην κατασκευή αποδείξεων. Αναλαμβάνει να υλοποιήσει με φυσικό τρόπο τις διάφορες συναρτήσεις ενός προγράμματος, προσέχοντας μόνο να δίνει κατάλληλες προδιαγραφές για αυτές. Ο αποδείκτης αναλαμβάνει να αποδείξει την ισχύ των προδιαγραφών αυτών δημιουργώντας πλήρεις λογικές αποδείξεις διαφόρων προτάσεων. Θεωρούμε όμως ότι έχει εμπειρία στη μαθηματική λογική και στη χρήση εργαλείων όπως το Coq για διευκόλυνση στην κατασκευή αποδείξεων.

Όπως έχουμε ήδη αναφέρει, τελικός σκοπός είναι ο κώδικας που παράγεται από τον μεταγλωττιστή της Karooma να προορίζεται για το σύστημα NFLINT. Το σύστημα αυτό μοιάζει περισσότερο στις γλώσσες που είδαμε έως τώρα, αλλά είναι ενδιάμεσου επιπέδου. Ως εκ τούτου, δεν προορίζεται για προγραμματισμό απευθείας σε αυτό μιας και κάτι τέτοιο είναι πολύπλοκη διαδικασία. Οι αποδείξεις στο NFLINT αποτελούν στοιχείο του επιπέδου των τύπων, και έτσι δεν επιβαρύνουν το χρόνο εκτέλεσης. Συγκεκριμένα το σύστημα αυτό χαρακτηρίζεται από σημασιολογία διαγραφής τύπων, και έτσι η εκτέλεση κώδικα NFLINT είναι το ίδιο αποδοτική με την εκτέλεση του ίδιου κώδικα χωρίς τύπους. Επομένως, η τελική απόδοση ενός προγράμματος Karooma είναι θεωρητικά όμοια με την απόδοση του ίδιου προγράμματος σε μία τυπική συναρτησιακή γλώσσα (όπου δεν συμπεριλαμβάνουμε κάποια μορφή ελέγχου της τήρησης των προδιαγραφών του προγράμματος). Το γεγονός όμως ότι το τελικό πρόγραμμα περνάει τον έλεγχο τύπων του NFLINT σημαίνει ότι αυτό είναι μερικώς ορθό σύμφωνα με τις προδιαγραφές

Τελεστές	Περιγραφή	Αριθμός τελουμένων	Θέση και προσεταιρι-στικότητα
$!$	Λογική άρνηση	1	prefix
$*$ $/$ $\&\&$	Πολλαπλασιαστικοί τελεστές	2	infix, αριστερή
$+$ $-$ $\|\|$	Προσθετικοί τελεστές	2	infix, αριστερή
$==$ $>$ $<$ $<=$ $>=$ $!=$	Σχεσιακοί τελεστές	2	infix, καμία

Πίνακας 3.1: Προτεραιότητα και προσεταιριστικότητα των τελεστών της Karooma.

που έχουμε θέσει.

Να σημειώσουμε τέλος ότι το σύστημα τύπων της Karooma είναι προς το παρόν τετριμμένο, υποστηρίζοντας μόνο φυσικούς ακεραίους και λογικές μεταβλητές. Δεν υπάρχει υποστήριξη για τύπους συναρτήσεων, άρα δεν είναι δυνατόν να έχουμε συναρτήσεις υψηλότερης τάξης (συναρτήσεις που δέχονται ως παραμέτρους συναρτήσεις ή που επιστρέφουν ως αποτέλεσμα συναρτήσεις). Επίσης δεν υπάρχει υποστήριξη για αλγεβρικούς τύπους δεδομένων, οπότε δεν μπορούμε να ορίσουμε, για παράδειγμα, τύπο λίστας. Ο γενικός σχεδιασμός της γλώσσας δεν αποκλείει τέτοιους τύπους, αλλά ο συνδυασμός της αυξημένης πολυπλοκότητας στην υλοποίηση του μεταγλωττιστή και οι λεπτομέρειες της συμπερίληψής τους στη γλώσσα μας απέτρεψαν από να τους εισάγουμε σε αυτή τη φάση. Η υποστήριξη τέτοιων τύπων είναι ο επόμενός μας στόχος στη σχεδίαση της γλώσσας, και ευελπιστούμε ότι σύντομα θα υποστηρίζονται σε μία νέα έκδοση της.

Στο υπόλοιπο του κεφαλαίου θα παρουσιάσουμε τη γλώσσα αρχικά χωρίς να λαμβάνουμε υπόψη τα στοιχεία που έχουν να κάνουν με τον προγραμματισμό με αποδείξεις πέραν των προ- και μετα-συνθηκών, έπειτα περιγράφοντας τα στοιχεία αυτά, και τέλος παραθέτοντας παραδείγματα προγραμμάτων.

3.1 Δομή προγράμματος και σύνταξη της Karooma

Η δομή ενός προγράμματος Karooma είναι η εξής. Οι πρώτες δύο δηλώσεις ενός προγράμματος έχουν να κάνουν με τον προγραμματισμό με αποδείξεις και θα τις εξετάσουμε σε βάθος παρακάτω. Έπειτα έχουμε τους ορισμούς των συναρτήσεων του προγράμματος. Επειδή δεν υπάρχει κάποιος μηχανισμός εισόδου-εξόδου ώστε να γίνεται εκτύπωση αποτελεσμάτων, και επειδή έχουμε υλοποιήσει μεταγλωττιστή και όχι διερμηνέα της γλώσσας, ώστε να μπορούμε να έχουμε κάποιο διαδραστικό τρόπο υποβολής ερωτήσεων με τη μορφή κλήσης συνάρτησης για να λάβουμε αποτελέσματα, έχουμε επιλέξει να ορίζουμε το αποτέλεσμα του προγράμματος μέσα σε αυτό. Έτσι στο τέλος του προγράμματος πρέπει να ορίσουμε το αποτέλεσμα του προγράμματος. Σε κάθε σημείο του προγράμματος επιτρέπονται σχόλια με τη σύνταξη της OCaml, δηλαδή μέσα σε παρενθέσεις με αστερίσκο: `(*` και `*)`. Τα σχόλια μπορεί να είναι φωλιασμένα.

Στο σχήμα 3.1 περιγράφεται σε μορφή EBNF η σύνταξη της Karooma. Η γραμματική αυτή είναι διφορούμενη, οι αμφισημίες όμως αίρονται αν ληφθούν υπόψη οι αναμενόμενοι κανόνες προτεραιότητας και προσεταιριστικότητας των τελεστών, οι οποίοι φαίνονται στον πίνακα 3.1. Το αρχικό μη-τερματικό της γλώσσας είναι το *program*. Καλούμε εκφράσεις τους όρους που παράγονται από το μη-τερματικό *expr* και λογικές προτάσεις τους όρους του μη-τερματικού *pred*.

Περιληπτικά, οι εκφράσεις της γλώσσας αποτελούνται από ακέραιες και λογικές σταθερές, τις συνηθισμένες πράξεις αυτών, την αναφορά σε παραμέτρους της συνάρτησης, τη δομή if-then-else και την αναδρομική κλήση συναρτήσεων. Να σημειώσουμε ότι επιτρέπεται και αμοιβαία αναδρομή μεταξύ συναρτήσεων, ενώ δεν υποστηρίζεται currying. (Υπενθυμίζουμε ότι currying

$$
\begin{array}{lcl}
\langle\text{letter}\rangle & ::= & \text{``}\textbf{a}\text{''-``}\textbf{z}\text{''} \mid \text{``}\textbf{A}\text{''-``}\textbf{Z}\text{''} \\
\langle\text{digit}\rangle & ::= & \text{``}\textbf{0}\text{''-``}\textbf{9}\text{''} \\
\langle\text{id}\rangle & ::= & \langle\text{letter}\rangle \; (\, \langle\text{letter}\rangle \mid \langle\text{digit}\rangle \mid \text{``\_''})^{*} \\
\langle\text{numconst}\rangle & ::= & \langle\text{digit}\rangle^{+} \\
\langle\text{boolconst}\rangle & ::= & \text{``}\textbf{true}\text{''} \mid \text{``}\textbf{false}\text{''} \\
\langle\text{binop}\rangle & ::= & \text{``}+\text{''} \mid \text{``}-\text{''} \mid \text{``}*\text{''} \mid \text{``}/\text{''} \\
 & \mid & \text{``}<=\text{''} \mid \text{``}>=\text{''} \mid \text{``}<\text{''} \mid \text{``}>\text{''} \mid \text{``}==\text{''} \mid \text{``}!=\text{''} \\
 & \mid & \text{``\&\&''} \mid \text{``}||\text{''} \\
\langle\text{unop}\rangle & ::= & \text{``!''} \\
\langle\text{program}\rangle & ::= & \langle\text{repr}\rangle \; \langle\text{thrm}\rangle \; \langle\text{functiondef}\rangle^{+} \; \langle\text{res}\rangle \\
 & & \\
\langle\text{repr}\rangle & ::= & \text{``}\textbf{representation}\text{''} \; \langle\text{id}\rangle \; \text{``.''} \\
\langle\text{thrm}\rangle & ::= & \text{``}\textbf{theorems}\text{''} \; \langle\text{id}\rangle \; \text{``.''} \\
\langle\text{functiondef}\rangle & ::= & \text{``}\textbf{function}\text{''} \; \langle\text{id}\rangle \; \text{``:''} \; \langle\text{vars}\rangle \; \langle\text{optpreds}\rangle \; \text{``}\rightarrow\text{''} \; \langle\text{var}\rangle \; \langle\text{optpreds}\rangle \; \text{``}\textbf{is}\text{''} \; \langle\text{expr}\rangle \\
\langle\text{res}\rangle & ::= & \text{``}\textbf{result}\text{''} \; \text{``}\textbf{is}\text{''} \; \langle\text{expr}\rangle \; \text{``.''} \\
\langle\text{vars}\rangle & ::= & \langle\text{var}\rangle \mid \langle\text{vars}\rangle \; \text{``,''} \; \langle\text{var}\rangle \\
\langle\text{var}\rangle & ::= & \langle\text{id}\rangle \; \text{``:''} \; \langle\text{type}\rangle \\
\langle\text{type}\rangle & ::= & \text{``}\textbf{nat}\text{''} \mid \text{``}\textbf{bool}\text{''} \\
\langle\text{optpreds}\rangle & ::= & [\, (\text{``[''} \; \langle\text{preds}\rangle \; \text{``]''}) \,] \\
\langle\text{preds}\rangle & ::= & \langle\text{pred}\rangle \mid \langle\text{preds}\rangle \; \text{``,''} \; \langle\text{pred}\rangle \\
\langle\text{expr}\rangle & ::= & \langle\text{id}\rangle \\
 & \mid & \langle\text{numconst}\rangle \\
 & \mid & \langle\text{boolconst}\rangle \\
 & \mid & \langle\text{expr}\rangle_1 \; \langle\text{binop}\rangle \; \langle\text{expr}\rangle_2 \\
 & \mid & \langle\text{unop}\rangle \; \langle\text{expr}\rangle \\
 & \mid & \langle\text{id}\rangle \; \text{``(''} \; \langle\text{expr}\rangle_1 \; \text{``,''} \; \langle\text{expr}\rangle_2 \; \text{``,''} \cdots \text{``,''} \; \langle\text{expr}\rangle_n \; \text{``)''} \\
 & \mid & \text{``}\textbf{if}\text{''} \; \langle\text{expr}\rangle_1 \; \text{``}\textbf{then}\text{''} \; \langle\text{expr}\rangle_2 \; \text{``}\textbf{else}\text{''} \; \langle\text{expr}\rangle_3 \\
 & \mid & \text{``(''} \; \langle\text{expr}\rangle \; \text{``)''} \\
\langle\text{pred}\rangle & ::= & \langle\text{id}\rangle \mid \langle\text{numconst}\rangle \mid \langle\text{boolconst}\rangle \\
 & \mid & \langle\text{pred}\rangle_1 \; \langle\text{binop}\rangle \; \langle\text{pred}\rangle_2 \mid \langle\text{unop}\rangle \; \langle\text{pred}\rangle \\
 & \mid & \langle\text{id}\rangle \; \text{``(''} \; \langle\text{pred}\rangle_1 \; \text{``,''} \; \langle\text{pred}\rangle_2 \; \text{``,''} \cdots \text{``,''} \; \langle\text{pred}\rangle_n \; \text{``)''} \mid \text{``(''} \; \langle\text{pred}\rangle \; \text{``)''}
\end{array}
$$

Σχήμα 3.1: Σύνταξη της γλώσσας Karooma.

ονομάζεται η δυνατότητα να χειριζόμαστε μία συνάρτηση n ορισμάτων ως συνάρτηση ενός ορίσματος που επιστρέφει μία συνάρτηση $n-1$ ορισμάτων). Κατά τη δήλωση συναρτήσεων πρέπει να προσέχουμε να δίνουμε ονόματα και τύπους τόσο στις παραμέτρους, όσο και στην τιμή επιστροφής: εκτός του τύπου της τιμής που επιστρέφει η συνάρτηση, πρέπει να δίνουμε και όνομα σε αυτήν, ώστε να μπορούμε να αναφερόμαστε σε αυτή μέσα στις λογικές προτάσεις. Επίσης, σε περίπτωση που θέλουμε να διασφαλίσουμε τη μερική ορθότητα της συνάρτησης, πρέπει να δίνουμε μέσα σε αγκύλες τις λογικές προτάσεις που αποτελούν τις προ-συνθήκες και μετα-συνθήκες της χρήσης της συνάρτησης.

Παρατηρούμε ότι οι λογικές προτάσεις έχουν την ίδια σύνταξη με τις εκφράσεις, με την εξαίρεση ότι δεν επιτρέπεται if-then-else. Εκτός αυτού, όπως θα δούμε παρακάτω, ενώ οι μεταβλητές που χρησιμοποιούνται μέσα στις λογικές προτάσεις είναι οι ίδιες με αυτές των εκφράσεων, οι συναρτήσεις που χρησιμοποιούνται είναι διαφορετικές από αυτές που χρησι-

```
representation default.
theorems default.
function sqrt_aux :
    n : nat, aux : nat [ aux * aux ≤ n ] → r : nat [ r * r ≤ n, (r+1)*(r+1) > n ]
is
    if (aux+1)*(aux+1) > n then
        aux
    else
        sqrt_aux(n,aux+1)


function sqrt :
    n : nat → r : nat [ r * r ≤ n, (r+1) * (r+1) > n ]
is
    sqrt_aux(n, 0)


result is sqrt(160).
```

Σχήμα 3.2: Πρόγραμμα υπολογισμού τετραγωνικής ρίζας στην Karooma.

μοποιούνται μέσα στις εκφράσεις της γλώσσας. Δηλαδή ενώ, για παράδειγμα, μπορούμε να χρησιμοποιήσουμε μία παράμετρο μετά τη δήλωσή της ως μέρος μίας λογικής πρότασης, δεν μπορούμε να καλέσουμε μία συνάρτηση που έχουμε δηλώσει κανονικά, μέσα στις λογικές προτάσεις. Οι συναρτήσεις των λογικών προτάσεων ορίζονται με διαφορετικό τρόπο από τις κανονικές συναρτήσεις και έχουν διαφορετικό σκοπό ύπαρξης, τον οποίο θα δούμε παρακάτω.

Στο σχήμα 3.2 παραθέτουμε ένα πρώτο παράδειγμα προγράμματος στη γλώσσα Karooma. Στο παράδειγμα αυτό υλοποιείται η συνάρτηση τετραγωνικής ρίζας φυσικού αριθμού ($f(n) = \lfloor\sqrt{n}\rfloor$). Θα εξετάσουμε τη λειτουργία του προγράμματος πιο αναλυτικά στο υποκεφάλαιο 3.4.1.

Σε αυτό το σημείο τα μόνα στοιχεία της γλώσσας που δεν έχουμε περιγράψει είναι οι δηλώσεις representation και theorems. Στα επόμενα υποκεφάλαια θα δούμε τη σημασία αυτών των δηλώσεων.

3.2 Η δήλωση theorems

Κατά τη μεταγλώττιση ενός προγράμματος Karooma, προκύπτουν λόγω των προ-συνθηκών και μετα-συνθηκών διάφορες λογικές προτάσεις που πρέπει να αποδειχθούν. Για παράδειγμα κατά την κλήση της συνάρτησης sqrt_aux που είδαμε στο σχήμα 3.2 πρέπει να αποδειχθεί ότι το τετράγωνο της παραμέτρου aux είναι μικρότερο ίσο της παραμέτρου n. Οι περιπτώσεις όπου πρέπει να αποδειχθεί μια λογική πρόταση είναι, περιληπτικά, οι παρακάτω. Πιο αναλυτικά θα δούμε τις απαιτήσεις αυτές στο κεφάλαιο 4.

- όταν καλείται μία συνάρτηση, πρέπει να μπορούν να αποδειχθούν οι λογικές προτάσεις που αντιστοιχούν στις προσυνθήκες της, αφού αντικατασταθούν καταλλήλως σε αυτές οι παράμετροι που περνούμε

- όταν προκύπτει μία τιμή, αποτέλεσμα συνάρτησης, πρέπει να μπορούν να αποδειχθούν οι λογικές προτάσεις που αντιστοιχούν στις μετασυνθήκες της, αφού αντικατασταθεί η τιμή επιστροφής σε αυτές

- όταν έχουμε μία έκφραση if-then-else, πρέπει να μπορεί να αποδειχθεί η λογική πρόταση που αντιστοιχεί στη συνθήκη του if στο πρώτο παρακλάδι της δομής, εάν η συνθήκη ισχύει, και η άρνησή τής, εάν η συνθήκη δεν ισχύει. Έτσι εξασφαλίζουμε ότι και στα δύο παρακλάδια του if έχουμε απόδειξη των προτάσεων που γνωρίζουμε ότι ισχύουν. Γι'αυτό το λόγο, απαιτούμε την ύπαρξη απόδειξης για τη λογική πρόταση Cond(συνθήκη του if, συνθήκη του if ως λογική πρόταση, άρνηση της συνθήκης του if ως λογική πρόταση). Η συνάρτηση λογικών προτάσεων Cond(b, p_1, p_2) είναι μια προκαθορισμένη ειδική συνάρτηση, η οποία είναι ισοδύναμη με τη λογική πρόταση p_1 όταν η συνθήκη b είναι αληθής, και ισοδύναμη με τη λογική πρόταση p_2 όταν η συνθήκη είναι ψευδής. Έτσι, έχοντας απόδειξη για αυτή, έχουμε απόδειξη για την κατάλληλη πρόταση εκ των δύο πιθανών.

Οι προτάσεις αυτές αποδεικνύονται από έναν μηχανισμό αυτόματης απόδειξης που διαθέτει ο μεταγλωττιστής. Αυτός χρησιμοποιεί δύο σύνολα για την κατασκευή των αποδείξεων: το σύνολο των διαθέσιμων αποδείξεων, και το σύνολο των θεωρημάτων. Συνθέτει καταλλήλως τα στοιχεία αυτών των συνόλων προς κατασκευή των αποδείξεων των προτάσεων. Αν οι προτάσεις δεν μπορούν να αποδειχθούν, το πρόγραμμα θεωρείται εσφαλμένο και η μεταγλώττισή του τερματίζεται.

Το σύνολο των διαθέσιμων αποδείξεων εξαρτάται από το σημείο του προγράμματος όπου βρισκόμαστε. Μέσα στο σώμα μιας συνάρτησης με ορισμένες προσυνθήκες, διαθέτουμε αποδείξεις των λογικών προτάσεων που αντιστοιχούν σε αυτές. Στο αληθές μέρος μίας έκφρασης if-then-else, έχουμε απόδειξη για την λογική πρόταση που αντιστοιχεί στη συνθήκη του if, ενώ στο ψευδές, έχουμε απόδειξη για την άρνηση αυτής της πρότασης. Τέλος μετά την κλήση μίας συνάρτησης με ορισμένες μετασυνθήκες, έχουμε απόδειξη των προτάσεων που αντιστοιχούν σε αυτές, αν αντικαταστήσουμε καταλλήλως τις τιμές των παραμέτρων της συνάρτησης.

Το σύνολο των θεωρημάτων είναι σταθερό σε ένα πρόγραμμα, και περιγράφεται σε ξεχωριστό αρχείο με συγκεκριμένη σύνταξη, το οποίο ονομάζουμε αρχείο των θεωρημάτων. Η δήλωση *theorems* ενός προγράμματος Karooma υπάρχει ακριβώς ώστε να προσδιορίζεται το αρχείο των θεωρημάτων που θα χρησιμοποιηθεί από το μηχανισμό αυτόματης απόδειξης του μεταγλωττιστή. Τα θεωρήματα αποτελούνται από μία λογική πρόταση-στόχο, καθώς και από λογικές προτάσεις-υποθέσεις. Επίσης επιτρέπεται να έχουν κάποιες ελεύθερες μεταβλητές. Για παράδειγμα, το θεώρημα "εάν ο φυσικός αριθμός ν είναι διάφορος του 0 τότε είναι μεγαλύτερος ίσος του 1", θα είχε ως ελεύθερη μεταβλητή το ν, ως πρόταση-στόχο την "ν μεγαλύτερος ίσος του 1", και ως πρόταση-υπόθεση την "ν διάφορο του 0". Επιτρέπονται επίσης θεωρήματα χωρίς καμία λογική πρόταση-υπόθεση, τα οποία αποτελούν αξιώματα.

Το αρχείο των θεωρημάτων περιέχει εκτός από τα θεωρήματα, και τις δηλώσεις χρήστη των συναρτήσεων που μπορούν να χρησιμοποιηθούν στους όρους των λογικών προτάσεων. Η ύπαρξή τους κρίνεται αναγκαία ώστε να μπορούν να εκφραστούν λογικές προτάσεις που δεν μπορούν να εκφραστούν αλλιώς, με αποτέλεσμα να έχουμε δυνατότητα για πιο ακριβείς προδιαγραφές ορθότητας των συναρτήσεων. Για παράδειγμα, εάν μία συνάρτηση επιστρέφει πρώτους αριθμούς, με δεδομένη τη μορφή των λογικών προτάσεων, δεν μπορεί να εκφραστεί κάτι τέτοιο στη μετασυνθήκη της συνάρτησης, εκτός και εάν ορίσουμε ένα σχετικό κατηγόρημα prime με όρισμα ακεραίου μέσα στο αρχείο των θεωρημάτων. Παρομοίως, θα μπορούσαμε να ορίσουμε μία συνάρτηση λογικών προτάσεων power με δύο ορίσματα ακεραίων, η οποία θα αντιστοιχεί στην ύψωση αριθμού σε εκθέτη. Έτσι θα μπορούσαμε να δώσουμε μία ακριβή μετασυνθήκη για την αντίστοιχη συνάρτηση μέσα σε ένα πρόγραμμα Karooma.

Να σημειώσουμε ότι η ενδεικνυόμενη τακτική για τον προγραμματισμό στην Karooma επιβάλλει το αρχείο θεωρημάτων να είναι σταθερό μεταξύ των διαφόρων προγραμμάτων. Έτσι, τα θεωρήματα που έχουν εισαχθεί για ένα πρόγραμμα μπορούν να επαναχρησιμοποιηθούν για κάποιο άλλο, και μόνον εάν τα ήδη υπάρχοντα θεωρήματα δεν αρκούν για την απόδειξη όλων των προτάσεων ενός προγράμματος πρέπει να προστίθενται νέα.

Στο σχήμα 3.3 φαίνεται η σύνταξη του αρχείου θεωρημάτων. Ο αρχικός κανόνας της γραμματικής είναι ο *thrmfile*. Όπως φαίνεται στο σχήμα, η δήλωση ενός θεωρήματος περιλαμβάνει

$$\begin{array}{lcl}
\langle\langle \text{ptype} \rangle\rangle & ::= & \text{``}\textbf{nat}\text{''} \mid \text{``}\textbf{bool}\text{''} \mid \text{``}\textbf{prop}\text{''} \\
\langle\langle \text{thrmfile} \rangle\rangle & ::= & \langle \text{predfunc} \rangle^* \ \langle \text{theorem} \rangle^* \\
\langle\langle \text{predfunc} \rangle\rangle & ::= & \text{``}\textbf{predicate}\text{''} \ \langle \text{id} \rangle \ \text{``:''(} \ \langle \text{ptype} \rangle \ \text{``,'')}^* \ \langle \text{ptype} \rangle \ \text{``}\rightarrow\text{''} \ \langle \text{ptype} \rangle \\
\langle\langle \text{theorem} \rangle\rangle & ::= & \langle \text{prooftrm} \rangle \ \langle \text{tvars} \rangle \ \langle \text{rule} \rangle \\
\langle\langle \text{rule} \rangle\rangle & ::= & \langle \text{pred} \rangle \ \text{``}\leftarrow\text{''(} \ \langle \text{pred} \rangle \ \text{``,'')}^* \ \langle \text{pred} \rangle \ \text{``.''} \\
 & \mid & \langle \text{pred} \rangle \ \text{``.''} \\
\langle\langle \text{prooftrm} \rangle\rangle & ::= & \text{``(''} \ \langle \text{id} \rangle \ \text{``)''} \\
\langle\langle \text{tvars} \rangle\rangle & ::= & \epsilon \mid \text{``[''(} \ \langle \text{tvar} \rangle \ \text{``,'')}^* \ \langle \text{tvar} \rangle \ \text{``]''} \\
\langle\langle \text{tvar} \rangle\rangle & ::= & \langle \text{id} \rangle \ \text{``:''} \ \langle \text{ptype} \rangle
\end{array}$$

Σχήμα 3.3: Σύνταξη του αρχείου θεωρημάτων της Karooma.

predicate prime : nat → prop.

(notEQ0isLE1) $[n : nat]$ n $\geq$ 1 $\leftarrow$ n != 0.

Σχήμα 3.4: Παράδειγμα δηλώσεων του αρχείου θεωρημάτων της Karooma.

τέσσερα στοιχεία: ένα αναγνωριστικό, τη δήλωση των ελεύθερων μεταβλητών του θεωρήματος, την λογική πρόταση-στόχο του θεωρήματος, και τις λογικές προτάσεις-υποθέσεις. Το αναγνωριστικό πρέπει να αντιστοιχεί σε έναν όρο NFLINT που αποτελεί την απόδειξη του θεωρήματος. Ο όρος αυτός πρέπει να περιλαμβάνεται, μαζί με άλλα στοιχεία που θα δούμε παρακάτω, στο αρχείο της αναπαράστασης. Για τη δήλωση μίας συνάρτησης λογικών προτάσεων αρκεί ο προσδιορισμός των τύπων των παραμέτρων της και του τύπου αποτελέσματός της. Να σημειώσουμε την προσθήκη του νέου τύπου `prop`, ο οποίος είναι ο τύπος των λογικών προτάσεων – δηλαδή κάθε λογική πρόταση πρέπει να έχει αυτό τον τύπο. Επομένως, για να ορίσουμε ένα λογικό κατηγόρημα, πρέπει να δώσουμε αυτόν τον τύπο επιστροφής. Τα στοιχεία αυτά θα εξεταστούν με μεγαλύτερη λεπτομέρεια στο κεφάλαιο 4.

Στο σχήμα 3.4 φαίνεται ένα παράδειγμα δήλωσης συνάρτησης λογικών προτάσεων (του κατηγορήματος prime για πρώτους αριθμούς), και ένα παράδειγμα δήλωσης θεωρήματος (του θεωρήματος $\nu \neq 0 \Rightarrow \nu \geq 1$).

3.3 Η δήλωση representation

Ένα πρόγραμμα Karooma μεταγλωττίζεται, όπως έχουμε πει, σε ενδιάμεσο κώδικα για το σύστημα NFLINT. Ο σχεδιασμός του συστήματος αυτού μοιάζει στη γλώσσα Vero που είδαμε στο κεφάλαιο 2.5, με τη διαφορά ότι το NFLINT είναι χαμηλότερου επιπέδου. Έτσι το NFLINT περιλαμβάνει μία γλώσσα τύπων (αντίστοιχη της γλώσσα αποδείξεων της Vero), και μία γλώσσα υπολογισμών. Γίνεται χρήση singleton τύπων για τους ακεραίους, και έτσι ένας ακέραιος όρος στη γλώσσα υπολογισμών του NFLINT εξαρτάται από μία αναπαράσταση αυτού του όρου στη γλώσσα τύπων.

Η δήλωση representation γίνεται σε ένα πρόγραμμα Karooma ώστε να προσδιοριστεί ένα αρχείο στη γλώσσα του συστήματος NFLINT, το οποίο περιλαμβάνει τον ορισμό του είδους της αναπαράστασης των ακεραίων, καθώς και την αναπαράσταση των στοιχειωδών πράξεων μεταξύ ακεραίων, στο επίπεδο των τύπων. Επίσης, πρέπει να περιλαμβάνει τον ορισμό των όρων που αποδεικνύουν τα διάφορα θεωρήματα που περιγράφονται στο αρχείο των θεωρημάτων. Τέλος, πρέπει να ορίζονται και οι όροι στο επίπεδο των τύπων που αναπαριστούν τις συναρτήσεις λογικών προτάσεων που ορίζονται στο αρχείο των θεωρημάτων.

(GTorLEprf)	[*x:nat, y:nat*]	Cond(x>y, x>y, x≤y).
(LE_0_n)	[*n:nat*]	0*0 ≤ n.

Σχήμα 3.5: Θεωρήματα για το παράδειγμα τετραγωνικής ρίζας.

Ανάλογα με το είδος της αναπαράστασης των ακεραίων στο επίπεδο των τύπων που επιλέγουμε, και με την πληροφορία που περιέχεται στην αναπαράσταση αυτή, προσδιορίζεται και η δυσκολία απόδειξης των θεωρημάτων. Εάν, για παράδειγμα, επιλέξουμε η αναπαράσταση των ακεραίων να είναι ο μοναδιαίος τύπος, οπότε έχουμε μηδενική πληροφορία στο επίπεδο των τύπων αφού όλοι οι ακέραιοι αναπαρίστανται με την ίδια, μοναδική, σταθερά, τότε η απόδειξη όλων των θεωρημάτων είναι μία τετριμμένη διαδικασία. Για παράδειγμα, μπορούμε να "αποδείξουμε" για κάθε ν και μ ότι $\nu < \mu$, αφού η σχέση μικρότερο υποβιβάζεται στην πάντα αληθή πρόταση. Η επιλογή αυτής της αναπαράστασης, όπως είναι φυσικό, συνεπάγεται ότι δεν έχουμε ουσιαστική απόδειξη μερικής ορθότητας για τα προγράμματα, αφού όλες οι συνθήκες ισχύουν τετριμμένα. Αντίθετα, αν επιλέξουμε την πλήρη αναπαράσταση για τους ακεραίους –δηλαδή, ως στοιχεία ενός τύπου που έχει κατασκευαστές για το μηδέν και τη συνάρτηση επομένου– η απόδειξη των θεωρημάτων είναι συχνά πολύπλοκη διαδικασία, αλλά εξασφαλίζουμε τη μερική ορθότητα των προγραμμάτων.

Σε μελλοντική έκδοση της γλώσσας όπου θα υποστηρίζονται και αναδρομικοί τύποι ορισμένοι από το χρήστη, όπως ο τύπος λίστας, το αρχείο αναπαράστασης θα πρέπει να έχει παρόμοια δήλωση αναπαράστασης στο επίπεδο τύπων και για αυτούς τους τύπους. Έτσι, θα μπορούμε να επιλέγουμε για παράδειγμα, αν οι λίστες θα αναπαρίστανται με τον μοναδιαίο τύπο (οπότε ο τύπος της λίστας δεν είναι, ουσιαστικά, εξαρτώμενος), με ακέραιο που αντιστοιχεί στο μήκος της λίστας (οπότε ο τύπος της λίστας είναι εξαρτώμενος από το μήκος της λίστας, και μπορούμε για παράδειγμα να αποδεικνύουμε ότι η συνάρτηση συνένωσης λιστών επιστρέφει λίστα με μήκος ίσο με το άθροισμα των μηκών των λιστών-ορισμάτων), ή ακόμη και με λίστα του επιπέδου των τύπων (οπότε μπορούμε να προδιαγράφουμε επακριβώς και να αποδεικνύουμε πολύπλοκα κατηγορήματα όπως το εάν μία λίστα είναι ταξινομημένη).

Από τα παραπάνω είναι φανερή η χρησιμότητα της δυνατότητα παραμετροποίησης της αναπαράστασης που θα χρησιμοποιηθεί μέσω της σχετικής δήλωσης. Έτσι, εάν δεν μας ενδιαφέρει η μερική ορθότητα του προγράμματος, επιλέγουμε τη μοναδιαία αναπαράσταση και προγραμματίζουμε με τον ίδιο τρόπο όπως σε μία κοινή συναρτησιακή γλώσσα. Εάν όμως μας ενδιαφέρει η ορθότητα, επιλέγουμε την πλήρη αναπαράσταση, όπου θα πρέπει να εισάγουμε κατάλληλα θεωρήματα και αποδείξουμε για όσες προτάσεις προκύψουν κατά την υλοποίηση του προγράμματος.

Όπως και το αρχείο των θεωρημάτων, θεωρούμε ότι και το αρχείο της αναπαράστασης είναι σταθερό (δεν γράφεται ξεχωριστά για κάθε πρόγραμμα), ώστε οι επεκτάσεις αυτού για κάποιο πρόγραμμα να είναι διαθέσιμες και για άλλα. Επίσης, είναι φανερή η σύνδεση των δύο δηλώσεων: ένα αρχείο θεωρημάτων που έχει γραφτεί προς χρήση με την κανονική αναπαράσταση δεν μπορεί να χρησιμοποιηθεί με την μοναδιαία ή κάποια άλλη.

Κλείνοντας να αναφέρουμε ότι η τρέχουσα έκδοση της Karooma έχει τις δύο πιθανές αναπαραστάσεις, και τα αντίστοιχα αρχεία θεωρημάτων που αναφέραμε: την μοναδιαία αναπαράσταση η οποία αναφέρεται ως **unit** και την πλήρη αναπαράσταση που αναφέρεται ως **default**.

3.4 Παραδείγματα προγραμμάτων Karooma

3.4.1 Τετραγωνική ρίζα φυσικού αριθμού

Επιστρέφουμε στο παράδειγμα του σχήματος 3.2, για να εξετάσουμε πιο αναλυτικά τη λειτουργία του. Όπως είχαμε αναφέρει παραπάνω, στο πρόγραμμα αυτό υλοποιείται η συνάρτηση τετραγωνικής ρίζας φυσικού αριθμού. Για το αποτέλεσμα της συνάρτησης αυτής πρέπει να ισχύει: $(\lfloor\sqrt{n}\rfloor)^2 \leq n \wedge (\lfloor\sqrt{n}\rfloor + 1)^2 > n)$. Αυτή είναι και η μετα-συνθήκη της συνάρτησης sqrt του προγράμματος. Η sqrt καλεί την βοηθητική συνάρτηση sqrt_aux, η οποία δοκιμάζει σειριακά τους φυσικούς αριθμούς ξεκινώντας από το 0, ώστε να βρει το αποτέλεσμα της συνάρτησης. Η sqrt_aux δέχεται δύο μεταβλητές – το αρχικό όρισμα **n** και την τρέχουσα τιμή που δοκιμάζεται, **aux**. Η προσυνθήκη της συνάρτησης επιβάλλει το τετράγωνο της **aux** να είναι μικρότερο ίσο του **n**. Επομένως για να ικανοποιείται η μετασυνθήκη αρκεί το τετράγωνο του επομένου της **aux** να είναι μεγαλύτερο του **n**, και αυτός είναι ο έλεγχος που γίνεται στην υλοποίηση της συνάρτησης. Στην περίπτωση που αυτό δεν ισχύει, δοκιμάζουμε με τον επόμενο αριθμό, για τον οποίο μπορεί να αποδειχθεί ότι το τετράγωνό του είναι μικρότερο ίσο του **n** (είναι η άρνηση της συνθήκης του if), δηλαδή η προσυνθήκη της sqrt_aux ισχύει.

Όπως παρατηρούμε έχει χρησιμοποιηθεί η πλήρης αναπαράσταση, οπότε έχει ενδιαφέρον να δούμε ποια θεωρήματα είναι απαραίτητα για την επιτυχή απόδειξη της ορθότητας του συγκεκριμένου προγράμματος. Στο σχήμα 3.5 φαίνονται όλα τα θεωρήματα που είναι απαραίτητα, και παρακάτω θα δούμε πώς αποδεικνύονται όλες οι λογικές προτάσεις που προκύπτει ότι πρέπει να αποδειχθούν από το πρόγραμμα.

Ξεκινούμε από τη συνάρτηση sqrt_aux. Κατ'αρχάς, η χρήση της δομής if-then-else με τη συνθήκη `(aux + 1) * (aux + 1) > n`, προϋποθέτει απόδειξη για την λογική πρόταση:

$$\begin{aligned}&\mathrm{Cond}((\mathrm{aux}{+}1) * (\mathrm{aux}{+}1) > \mathrm{n},\\ &\qquad (\mathrm{aux}{+}1) * (\mathrm{aux}{+}1) > \mathrm{n},\\ &\qquad (\mathrm{aux}{+}1) * (\mathrm{aux}{+}1) \leq \mathrm{n})\end{aligned}$$

Η πρόταση αυτή αποδεικνύεται με χρήση του θεωρήματος *GTorLEprf*, αντικαθιστώντας τη μεταβλητή `x` με `(aux+1)*(aux+1)`, και τη μεταβλητή `y` με `n`. Η επιστροφή της τιμής `aux` ως αποτέλεσμα της συνάρτησης, στο αληθές παρακλάδι του if, προϋποθέτει απόδειξη για την πρόταση `aux * aux <= n`, η οποία παρέχεται απευθείας από την προσυνθήκη της συνάρτησης, και για την πρόταση `(aux+1) * (aux+1) > n` υπάρχει απόδειξη λόγω του if. Η κλήση της συνάρτησης sqrt_aux στο ψευδές παρακλάδι του if, προϋποθέτει απόδειξη για την πρόταση `(aux+1) * (aux+1) <= n`, η οποία παρέχεται απευθείας λόγω του if. Το αποτέλεσμα αυτής της κλήσης είναι και το αποτέλεσμα της συνάρτησης. Έτσι έχοντας την απόδειξη των μετασυνθηκών για το αποτέλεσμα αυτό από την κλήση της συνάρτησης, έχουμε προφανώς και την απόδειξη των μετασυνθηκών της συνάρτησης.

Στη συνάρτηση sqrt, λόγω της κλήσης της sqrt_aux, πρέπει να αποδειχθεί η πρόταση `0*0<=n`. Αυτή αποδεικνύεται με χρήση του θεωρήματος *LE_0_n*. Η απόδειξη των μετασυνθηκών είναι προφανής.

Τέλος, επειδή η συνάρτηση sqrt δεν έχει προσυνθήκες, η κλήση της στο αποτέλεσμα δεν προϋποθέτει την ύπαρξη απόδειξης για κάποια πρόταση.

3.4.2 Ύψωση αριθμού σε δύναμη

Στο σχήμα 3.6 δίνεται ο κώδικας για ένα πρόγραμμα ύψωσης αριθμού σε δύναμη. Η υλοποίηση της κύριας συνάρτησης του προγράμματος γίνεται με τον προφανή τρόπο, δηλαδή επιστρέφοντας 1 εάν ο εκθέτης είναι 0, ή επιστρέφοντας το γινόμενο της βάσης επί το αποτέλεσμα της αναδρομικής κλήσης της συνάρτησης με τον προηγούμενο εκθέτη.

Για να ορίσουμε επακριβώς τη μετασυνθήκη της συνάρτησης, έχουμε ορίσει στο αρχείο θεωρημάτων μία νέα συνάρτηση λογικών προτάσεων, την power, η οποία λαμβάνει δύο φυσικούς αριθμούς ως ορίσματα και επιστρέφει επίσης φυσικό αριθμό. Επίσης η υλοποίηση αυτής στο επίπεδο των τύπων υπάρχει στο αρχείο αναπαράστασης. Έτσι η μετασυνθήκη της συνάρτησης είναι απλά, το αποτέλεσμα της συνάρτησης να ισούται με την ύψωση της βάσης στον εκθέτη.

Οι δηλώσεις στο αρχείο των θεωρημάτων που είναι απαραίτητες για την επιτυχή μεταγλώττιση του συγκεκριμένου αρχείου φαίνονται στο σχήμα 3.7. Η χρήση του if προϋποθέτει την ύπαρξη απόδειξης για την πρόταση `Cond(y==0, y==0, y!=0)`, η οποία κατασκευάζεται μέσω του θεωρήματος *EQorNEQprf*. Στο αληθές παρακλάδι του if, η επιστροφή του 1 ως αποτέ-

representation default.
theorems default.

function power :
    x : nat, y : nat → r : nat [*power(x,y) == r*]
is
    if y == 0 **then**
        1
    else
        x * power(x,y-1)

result is power(5,4).

Σχήμα 3.6: Πρόγραμμα ύψωσης αριθμού σε δύναμη στην Karooma.

predicate power : nat, nat → nat.

(EQorNEQprf)	[*x:nat, y:nat*]	Cond(x==y, x==y, x!=y).
(natPowerZero)	[*x:nat, y:nat*]	power(x,y) == 1 ← y == 0.
(natPowerNext)	[*x:nat, y:nat, r:nat*]	power(x,y) == x * r ← power(x,y-1) == r, y > 0.
(notEq0isGT0)	[*n:nat*]	n > 0 ← n != 0.

Σχήμα 3.7: Θεωρήματα για το παράδειγμα ύψωσης αριθμού σε δύναμη.

λεσμα της συνάρτησης προϋποθέτει την ύπαρξη απόδειξης για την πρόταση `power(x,y)==1`, η οποία αποδεικνύεται από το θεώρημα *natPowerZero*, αν λάβουμε υπόψη και την απόδειξη για τη συνθήκη του if που διαθέτουμε στο παρακλάδι αυτό. Στο ψευδές παρακλάδι, η επιστροφή του αποτελέσματος προϋποθέτει την ύπαρξη απόδειξης για την πρόταση `power(x,y) == x*r`, εάν γνωρίζουμε ότι για το `r` ισχύει `power(x,y-1) == r`. Η πρόταση αποδεικνύεται με τη χρήση του θεωρήματος *natPowerNext*, το οποίο απαιτεί και μία απόδειξη της πρότασης `y > 0`. Αυτή αποδεικνύεται με τη χρήση του θεωρήματος *notEq0isGT0*.

3.4.3 Ύψωση αριθμού σε δύναμη, με χρήση αναδρομής ουράς

Στο σχήμα 3.8 παρουσιάζεται ένα πρόγραμμα που υλοποιεί την ύψωση αριθμού σε δύναμη με τρόπο πιο αποδοτικό από το παράδειγμα του σχήματος 3.6. Αυτό επιτυγχάνεται με τη χρήση αναδρομής ουράς (tail recursion), μία γνωστή τεχνική βελτιστοποίησης προγραμμάτων συναρτησιακών γλωσσών. Βάσει της τεχνικής αυτής, όταν γίνεται μία αναδρομική κλήση συνάρτησης, πρέπει να μην χρησιμοποιείται το αποτέλεσμα αυτής ως μέρος μιας πιο πολύπλοκης έκφρασης, αλλά να επιστρέφεται ως έχει σαν αποτέλεσμα της συνάρτησης. Ένας διερμηνέας ή μεταγλωττιστής της εκάστοτε γλώσσας, εάν εντοπίσει αναδρομή ουράς, μπορεί να εκτελέσει ή να μεταγλωττίσει την αναδρομική κλήση της συνάρτησης με τρόπο πιο αποδοτικό από ό,τι μία αναδρομική κλήση που χρησιμοποιείται ως μέρος έκφρασης.

Αυτό δικαιολογείται ως εξής. Ως γνωστόν η κλήση συναρτήσεων χρησιμοποιεί μία στοίβα όπου εισάγονται οι παράμετροι καθώς και η τιμή αποτελέσματος της εκάστοτε κλήσης. Εάν η τιμή αποτελέσματος μίας αναδρομικής κλήσης χρησιμοποιείται ως μέρος έκφρασης, τότε μετά το πέρας της κλήσης πρέπει να γίνουν υπολογισμοί μέσα στην αρχική κλήση της συνάρτησης, οπότε οι παράμετροι αυτής θα πρέπει να έχουν διατηρηθεί στη στοίβα. Έτσι, στο αρχικό παράδειγμα ύψωσης σε δύναμη ν, κατανοούμε ότι επειδή γίνονται ν "φωλιασμένες" αναδρομικές κλήσεις, όπου το αποτέλεσμα της εσωτερικότερης χρησιμοποιείται στην αμέσως εξωτερικότερη,

```
representation default.
theorems default.

function power_tlaux :
        x : nat, y : nat, orig : nat, r : nat [ power(x, orig) == power(x,y) * r ]
            → res : nat [ res == power(x,orig) ]
is
        if y == 0 then
            r
        else
            power_tlaux(x,y-1,orig,x*r)

function power_tlrec :
        x : nat, y : nat → r : nat [ r == power(x, y) ]
is
        power_tlaux(x,y,y,1)

result is power_tlrec(5,4).
```

Σχήμα 3.8: Πρόγραμμα ύψωσης αριθμού σε δύναμη με χρήση αναδρομής ουράς.

```
predicate power : nat, nat → nat.

(EQorNEQprf)      [x:nat, y:nat]                  Cond(x==y, x==y, x!=y).
(notEq0isGT0)     [n:nat]                         n > 0 ← n != 0.
(natPowerZero)    [x:nat, y:nat]                  power(x,y) == 1 ← y == 0.

(natEqSym)        [x:nat, y:nat]                  x == y ← y == x.
(natEqRefl)       [x:nat]                         x == x.
(mult_one)        [a:nat, b:nat, c:nat]           a == b ← a == c * b, c == 1.
(mult_one2)       [a:nat]                         a == a*1.
(natPowerNext2)   [x:nat, y:nat, r:nat, a:nat]    a == power(x,y-1)*(x*r)
                                                      ← y > 0, a == power(x,y)*r.
```

Σχήμα 3.9: Θεωρήματα για το πρόγραμμα ύψωσης αριθμού σε δύναμη με χρήση αναδρομής ουράς.

κ.ο.κ., το μέγεθος της στοίβας αυξάνει γραμμικά ως προς τον εκθέτη. Εάν όμως η τιμή αποτελέσματος μίας αναδρομικής κλήσης χρησιμοποιείται ως έχει ως αποτέλεσμα και της αρχικής κλήσης (όπως στην περίπτωση της αναδρομής ουράς), τότε η εγγραφή της αρχικής κλήσης στη στοίβα μπορεί να αντικατασταθεί από την εγγραφή της αναδρομικής κλήσης. Κατανοούμε ότι σε μία υλοποίηση της ύψωσης σε δύναμη με χρήση αναδρομής ουράς η στοίβα του προγράμματος έχει σταθερό μέγεθος ως προς τον εκθέτη. Έτσι γίνεται εξοικονόμηση μνήμης.

Μετατρέπουμε συνήθως μία συνάρτηση χωρίς χρήση αναδρομής ουρά σε ένα με χρήση, εισάγοντας μία μεταβλητή όπου συσσωρεύεται το αποτέλεσμα της συνάρτησης. Στο παράδειγμα αυτό φαίνεται στη συνάρτηση *power_tlaux* η οποία κάνει τον ουσιαστικό υπολογισμό. Αυτή δέχεται τέσσερις παραμέτρους: τη βάση `x`, την παράμετρο συσσώρευσης `r`, την παράμετρο `y` που εκφράζει πόσες φορές πρέπει να πολλαπλασιαστεί ακόμη η βάση `x` με την παράμετρο συσσώρευσης ώστε να έχουμε το ορθό αποτέλεσμα, καθώς και τον αρχικό εκθέτη `orig`, ο οποίος χρησιμοποιείται μόνο για την ακριβή προδιαγραφή της συνάρτησης, δηλαδή μόνο μέσα στις προ- και μετα-συνθήκες. Όταν η παράμετρος y γίνει μηδενική, τότε στην παράμετρο συσσώρευσης έχουμε το αποτέλεσμα της συνάρτησης. Σε διαφορετική περίπτωση, η παράμετρος συσσώρευσης πολλαπλασιάζεται ακόμη μία φορά και η παράμετρος `y` μειώνεται κατά 1. Στις προσυνθήκες της συνάρτησης προδιαγράφεται η έννοια των παραμέτρων `y` και `r`, ενώ η μετασυνθήκη ορίζει ότι το αποτέλεσμα της συνάρτησης ισούται με τη βάση υψωμένη στον αρχικό εκθέτη.

Η κύρια συνάρτηση του προγράμματος, *power_tlrec*, καλεί τη συνάρτηση *power_tlaux* με αρχική τιμή 1 για τον συσσωρευτή. Έχει την ίδια προδιαγραφή με τη συνάρτηση *power* του παραδείγματος του σχήματος 3.6, και έτσι βλέπουμε ότι μία συνάρτηση με συγκεκριμένη προδιαγραφή μπορεί να υλοποιηθεί με πολλούς τρόπους.

Η απόδειξη της μερικής ορθότητας του προγράμματος είναι πιο δύσκολη από το προηγούμενο παράδειγμα, λόγω των πιο πολύπλοκων προδιαγραφών. Στο σχήμα 3.9 φαίνονται τα θεωρήματα που είναι απαραίτητα. Από αυτά, τα τρία υπήρχαν ήδη από το προηγούμενο παράδειγμα, αλλά έχουμε και πέντε καινούρια. Ας δούμε πώς αυτά χρησιμοποιούνται. Στη συνάρτηση *power_tlaux*, στο αληθές παρακλάδι του if πρέπει να αποδείξουμε ότι `r == power(x,orig)`. Από το θεώρημα *natPowerZero* έχουμε ότι `power(x,y) == 1`, αφού `y==0` λόγω της συνθήκης του if. Από το θεώρημα *mult_one* έχουμε ότι `power(x,orig) == r`, αφού `power(x,y) == 1` και `power(x,orig) == power(x,y) * r` από τη προσυνθήκη της συνάρτησης. Τέλος από το θεώρημα *natEqSym* που εκφράζει τη συμμετρία της σχέσης "ίσο", έχουμε ότι `r == power(x, orig)` αφού `power(x,orig) == r`. Στο ψευδές παρακλάδι του if η απόδειξη της μετασυνθήκης της συνάρτησης για την τιμή επιστροφής είναι προφανής, αρκεί να αποδείξουμε για την αναδρομική κλήση την ισχύ της προσυνθήκης. Ουσιαστικά πρέπει να αποδείξουμε την πρόταση `power(x,orig) == power(x,y-1)*(x*r)`, κάτι που επιτυγχάνουμε μέσω του θεωρήματος *natPowerNext2*. Οι υποθέσεις αυτού του θεωρήματος αποδεικνύονται με τη χρήση της απόδειξης που έχουμε από το ψευδές παρακλάδι του if (`y!=0`), το θεώρημα *notEq0isGT0*, και της προσυνθήκης της συνάρτησης. Στη συνάρτηση *power_tlrec*, αρκεί να αποδείξουμε την ισχύ της προσυνθήκης της *power_tlaux* για την αρχική της κλήση. Αποδεικνύουμε έτσι ότι `power(x,y) == power(x,y) * 1` απευθείας με χρήση του θεωρήματος *mult_one2*. Η απόδειξη της ισχύος της μετασυνθήκης της *power_tlrec* παρέχεται από την λογική πρόταση που ισχύει λόγω της κλήσης της *power_tlaux*.

Κεφάλαιο 4

Κανόνες τύπων και σημασιολογία της Karooma

Στο προηγούμενο κεφάλαιο κάναμε μία άτυπη παρουσίαση του συνόλου της γλώσσας Karooma, κατάλληλη για προγραμματιστές. Στο κεφάλαιο αυτό θα παρουσιάσουμε τους κανόνες τύπων και τη λειτουργική σημασιολογία της Karooma, ώστε να έχουμε μία αυστηρή φορμαλιστική περιγραφή αυτής, κάτι που διευκολύνει τη δημιουργία ενός ελεγκτή τύπων και ενός μεταγλωττιστή ή διερμηνέα για τη γλώσσα. Στο τέλος του κεφαλαίου θα δούμε κάποια στοιχεία για το πώς υλοποιήσαμε τον ελεγκτή τύπων της γλώσσας, ενώ σε επόμενο κεφάλαιο θα περιγράψουμε αναλυτικά τον τρόπο που γίνεται ο μετασχηματισμός από Karooma σε NFLINT. Ο συνδυασμός αυτού του μετασχηματισμού με έναν διερμηνέα για το σύστημα NFLINT αποτελεί έναν διερμηνέα για την Karooma, ο οποίος υλοποιεί τη λειτουργική σημασιολογία που θα δούμε στο παρόν κεφάλαιο.

4.1 Κανόνες τύπων της Karooma

Το αφηρημένο συντακτικό της Karooma που θα χρησιμοποιήσουμε παρακάτω είναι το εξής:

$$
\begin{array}{lcl}
e & ::= & id \mid n \mid b \mid \textbf{binop}(op, e_1, e_2) \mid \textbf{unop}(op, e) \mid \textbf{if}(e_1, e_2, e_3) \\
 & \mid & id(e_1,\ e_2,\ \cdots,\ e_n) \\
p & ::= & id \mid n \mid b \mid \textbf{binop}(op, p_1, p_2) \mid \textbf{unop}(op, p) \mid id(p_1,\ p_2,\ \cdots,\ p_n) \\
\tau & ::= & \textbf{nat} \mid \textbf{bool} \\
\tau p & ::= & \tau \mid \textbf{prop} \\
f & ::= & id : ((id : \tau)^+\ [p^*] \to id : \tau\ [p^*]) := e \\
r & ::= & ((id : \tau p)^*,\ p,\ p^*) \\
pf & ::= & id : \tau p^+ \to \tau p \\
prg & ::= & (pf^*, r^*, f^*, e)
\end{array}
$$

Στο συντακτικό αυτό, e είναι το σύμβολο για τις εκφράσεις, id το σύμβολο των αναγνωριστικών, n το σύμβολο των ακέραιων σταθερών, b των λογικών σταθερών, ενώ p είναι το σύμβολο για τις λογικές προτάσεις. Οι τύποι της γλώσσας για τις εκφράσεις συμβολίζονται με τ, ενώ για τις λογικές προτάσεις συμβολίζονται με τp. Με f συμβολίζουμε τον ορισμό συνάρτησης, με pf τη δήλωση συνάρτησης λογικών προτάσεων, με r τη δήλωση θεωρήματος, και με prg ένα πλήρες πρόγραμμα Karooma. Αυτό ουσιαστικά αποτελείται από τις δήλωση συναρτήσεων λογικών προτάσεων, τις δηλώσεις των θεωρημάτων, τους ορισμούς των συναρτήσεων, και την έκφραση-αποτέλεσμα του προγράμματος.

Τα περιβάλλοντα που θα χρησιμοποιήσουμε στους κανόνες τύπων είναι τα εξής:

$$
\begin{array}{lll}
ctx & \Gamma & ::= \cdot \mid \Gamma, id : \tau \\
functx & \Phi & ::= \cdot \mid \Phi, id : \tau^{+} \rightarrow \tau \\
prctx & P\Gamma & ::= \cdot \mid P\Gamma, id : \tau p \\
prfunctx & P\Phi & ::= \cdot \mid P\Phi, pf \\
proofctx & \Pi & ::= \cdot \mid \Pi, p \\
rulectx & \Sigma & ::= \cdot \mid \Sigma, r \\
fullfnctx & \Psi & ::= \cdot \mid \Psi, f
\end{array}
$$

Το περιβάλλον Γ διατηρεί τις δεσμεύσεις ονομάτων, περιλαμβάνει για παράδειγμα τα ονόματα και τους τύπους των παραμέτρων μίας συνάρτησης για εκφράσεις που βρίσκονται μέσα σε αυτή. Το περιβάλλον Φ διατηρεί τους τύπους των ορισμένων συναρτήσεων. Τα περιβάλλοντα $P\Gamma$ και $P\Phi$ έχουν την ίδια έννοια αλλά αφορούν τις εκφράσεις των λογικών προτάσεων. Έτσι το $P\Phi$ περιέχει τις συναρτήσεις των λογικών προτάσεων που έχουν δηλωθεί από το χρήστη στο αρχείο των θεωρημάτων, καθώς και τις προκαθορισμένες συναρτήσεις:

$\mathbf{Succ} : \mathbf{nat} \rightarrow \mathbf{nat}$	συνάρτηση επόμενου
$\mathbf{Cond} : \mathbf{bool}, \mathbf{prop}, \mathbf{prop} \rightarrow \mathbf{prop}$	συνάρτηση επιλογής λογικής πρότασης υπό συνθήκη

Το περιβάλλον Π περιλαμβάνει τις λογικές προτάσεις για τις οποίες έχουμε αποδείξεις σε κάποιο σημείο του προγράμματος, ενώ το Σ τα διαθέσιμα θεωρήματα. Τέλος το περιβάλλον Ψ διατηρεί την πλήρη πληροφορία για τις ορισμένες συναρτήσεις (ονόματα και τύπους παραμέτρων και τιμής επιστροφής, προ- και μετα-συνθήκες). Θεωρούμε ότι όλα τα περιβάλλοντα είναι σύνολα, και έτσι δεν επιτρέπεται διπλή δέσμευση του ίδιου ονόματος σε κάποιο από αυτά.

Οι τυπικές αποφάσεις (typing judgements) που αποτελούν τη στατική σημασιολογία της Karooma φαίνονται στον πίνακα 4.1.

Τυπική απόφαση	Εξήγηση
$\Phi; \Gamma \vdash e : \tau$	τύπος έκφρασης
$P\Phi; P\Gamma \vdash p$ ok	έγκυρη λογική πρόταση
$P\Phi; \Phi \vdash f$ ok	έγκυρη συνάρτηση
$P\Phi \vdash r$ ok	έγκυρο θεώρημα
$\cdot \vdash prg$ ok	έγκυρο πρόγραμμα
$P\Phi; \Sigma; \Psi \vdash^{*} f$ ok	έγκυρη και λογικά έγκυρη συνάρτηση
$\cdot \vdash^{*} prg$ ok	έγκυρο και λογικά έγκυρο πρόγραμμα

Πίνακας 4.1: Τυπικές αποφάσεις της Karooma.

Ένα πρόγραμμα Karooma θεωρείται καλώς-τυποποιημένο (well-typed), εάν μπορεί να παραχθεί η τυπική απόφαση $\cdot \vdash^{*} prg$ ok βάσει των κανόνων τύπων. Θα εξετάσουμε πρώτα τις τυπικές αποφάσεις που δεν ελέγχουν τη λογική εγκυρότητα, και μετά αυτές που την ελέγχουν, διότι είναι πιο πολύπλοκες.

4.1.1 Τυπικές αποφάσεις χωρίς έλεγχο λογικής εγκυρότητας

Παραθέτουμε τους κανόνες τύπων για τις τυπικές αποφάσεις χωρίς έλεγχο λογικής εγκυρότητας που περιγράψαμε παραπάνω στα σχήματα 4.1 - 4.5.

Ένα σημείο που πρέπει να προσέξουμε είναι το εξής: στους κανόνες τύπων για τις λογικές προτάσεις, ο τύπος **bool** είναι υποτύπος (subtype) του τύπου **prop**, δηλαδή, όπου πρέπει να χρησιμοποιηθεί έκφραση τύπου **prop**, είναι επιτρεπτό να τοποθετηθεί έκφραση τύπου **bool**. Επίσης οι λογικοί τελεστές μπορούν να εφαρμοστούν για τελούμενα και των δύο τύπων. Να σημειώσουμε εδώ ότι δεν μπορούμε να αποκλείσουμε τις εκφράσεις τύπου **bool** από τις λογικές προτάσεις, ταυτίζοντας τον τύπο **bool** με τον τύπο **prop**, λόγω των συναρτήσεων που δέχονται

$$\textit{(e-var)}\ \frac{id : \tau \text{ in } \Gamma}{\Phi;\Gamma \ \vdash \ id : \tau} \qquad \textit{(e-nat)}\ \frac{}{\Phi;\Gamma \ \vdash \ n : \mathbf{nat}} \qquad \textit{(e-bool)}\ \frac{}{\Phi;\Gamma \ \vdash \ b : \mathbf{bool}}$$

$$\textit{(e-uop-bool)}\ \frac{\Phi;\Gamma \ \vdash \ e : \mathbf{bool} \quad \textit{op} \text{ in } \{\ !\ \}}{\Phi;\Gamma \ \vdash \ \mathbf{unop}(\textit{op},\ e) : \mathbf{bool}}$$

$$\textit{(e-binop-arith)}\ \frac{\Phi;\Gamma \ \vdash \ e_1 : \mathbf{nat} \quad \Phi;\Gamma \ \vdash \ e_2 : \mathbf{nat} \quad \textit{op} \text{ in } \{\ +, -, *, /\ \}}{\Phi;\Gamma \ \vdash \ \mathbf{binop}(\textit{op},\ e_1,\ e_2) : \mathbf{nat}}$$

$$\textit{(e-binop-rel)}\ \frac{\Phi;\Gamma \ \vdash \ e_1 : \mathbf{nat} \quad \Phi;\Gamma \ \vdash \ e_2 : \mathbf{nat} \quad \textit{op} \text{ in } \{\leq, \geq, <, >, ==, !=\}}{\Phi;\Gamma \ \vdash \ \mathbf{binop}(\textit{op},\ e_1,\ e_2) : \mathbf{bool}}$$

$$\textit{(e-binop-bool)}\ \frac{\Phi;\Gamma \ \vdash \ e_1 : \mathbf{bool} \quad \Phi;\Gamma \ \vdash \ e_2 : \mathbf{bool} \quad \textit{op} \text{ in } \{\&\&, ||\}}{\Phi;\Gamma \ \vdash \ \mathbf{binop}(\textit{op},\ e_1,\ e_2) : \mathbf{bool}}$$

$$\textit{(e-if)}\ \frac{\Phi;\Gamma \ \vdash \ e_1 : \mathbf{bool} \quad \Phi;\Gamma \ \vdash \ e_2 : \tau \quad \Phi;\Gamma \ \vdash \ e_3 : \tau}{\Phi;\Gamma \ \vdash \ \mathbf{if}(e_1,\ e_2,\ e_3) : \tau}$$

$$\textit{(e-func)}\ \frac{\begin{array}{c} id : \tau_1,\ \tau_2,\ \cdots,\ \tau_n \rightarrow \tau_{ret} \text{ in } \Phi \\ \Phi;\Gamma \ \vdash \ e_1 : \tau_1 \quad \Phi;\Gamma \ \vdash \ e_2 : \tau_2 \quad \cdots \quad \Phi;\Gamma \ \vdash \ e_n : \tau_n \end{array}}{\Phi;\Gamma \ \vdash \ id(e_1,\ e_2,\ \cdots,\ e_n) : \tau_{ret}}$$

Σχήμα 4.1: Τυπική απόφαση $\Phi;\Gamma \ \vdash \ e : \tau$.

$$\textit{(p-ok)}\ \frac{P\Phi;P\Gamma \ \vdash \ p : \mathbf{prop}}{P\Phi;P\Gamma \ \vdash \ p \text{ ok}} \qquad \textit{(p-var)}\ \frac{id : \tau p \text{ in } P\Gamma}{P\Phi;P\Gamma \ \vdash \ id : \tau p}$$

$$\textit{(p-nat)}\ \frac{}{P\Phi;P\Gamma \ \vdash \ n : \mathbf{nat}} \qquad \textit{(p-bool)}\ \frac{}{P\Phi;P\Gamma \ \vdash \ b : \mathbf{bool}}$$

$$\textit{(puBorP)}\ \frac{P\Phi;P\Gamma \ \vdash \ p : \tau p \quad \tau p \text{ in } \{\mathbf{bool}, \mathbf{prop}\} \quad \textit{op} \text{ in } \{\ !\ \}}{P\Phi;P\Gamma \ \vdash \ \mathbf{unop}(\textit{op},\ e) : \tau p}$$

$$\textit{(pbAr)}\ \frac{P\Phi;P\Gamma \ \vdash \ p_1 : \mathbf{nat} \quad P\Phi;P\Gamma \ \vdash \ p_2 : \mathbf{nat} \quad \textit{op} \text{ in } \{\ +, -, *, /\ \}}{P\Phi;P\Gamma \ \vdash \ \mathbf{binop}(\textit{op},\ p_1,\ p_2) : \mathbf{nat}}$$

$$\textit{(pbRel)}\ \frac{P\Phi;P\Gamma \ \vdash \ p_1 : \mathbf{nat} \quad P\Phi;P\Gamma \ \vdash \ p_2 : \mathbf{nat} \quad \textit{op} \text{ in } \{\leq, \geq, <, >, ==, !=\}}{P\Phi;P\Gamma \ \vdash \ \mathbf{binop}(\textit{op},\ p_1,\ p_2) : \mathbf{bool}}$$

$$\textit{(pbBorP)}\ \frac{P\Phi;P\Gamma \ \vdash \ p_1 : \tau p \quad P\Phi;P\Gamma \ \vdash \ p_2 : \tau p \quad \tau p \text{ in } \{\mathbf{bool}, \mathbf{prop}\} \quad \textit{op} \text{ in } \{\&\&, ||\}}{P\Phi;P\Gamma \ \vdash \ \mathbf{binop}(\textit{op},\ p_1,\ p_2) : \tau p}$$

$$\textit{(p-func)}\ \frac{\begin{array}{c} id : \tau p_1,\ \tau p_2,\ \cdots,\ \tau p_n \rightarrow \tau p_{ret} \text{ in } P\Phi \\ P\Phi;P\Gamma \ \vdash \ p_1 : \tau p_1 \quad P\Phi;P\Gamma \ \vdash \ p_2 : \tau p_2 \quad \cdots \quad P\Phi;P\Gamma \ \vdash \ p_n : \tau p_n \end{array}}{P\Phi;P\Gamma \ \vdash \ id(p_1,\ p_2,\ \cdots,\ p_n) : \tau p_{ret}}$$

$$\textit{(p-subtyping)}\ \frac{P\Phi;P\Gamma \ \vdash \ p : \tau p \quad \tau p <: \tau p'}{P\Phi;P\Gamma \ \vdash \ p : \tau p'} \qquad \textit{(p-bsubp)}\ \frac{}{\mathbf{bool} <: \mathbf{prop}}$$

Σχήμα 4.2: Τυπική απόφαση $P\Phi;P\Gamma \ \vdash \ p$ ok.

$$\text{(f-ok)}\frac{\Phi;\overrightarrow{params} \vdash e:\tau_{ret} \qquad \forall p \text{ in } \overrightarrow{p_{pre}}:\ P\Phi;\overrightarrow{params} \vdash p \text{ ok} \qquad \forall p \text{ in } \overrightarrow{p_{post}}:\ P\Phi;\overrightarrow{params}, id_{ret}:\tau_{ret} \vdash p \text{ ok}}{P\Phi;\Phi \vdash id_{func}:(\overrightarrow{params}[\overrightarrow{p_{pre}}] \to id_{ret}:\tau_{ret}[\overrightarrow{p_{post}}]) := e \text{ ok}}$$

Σχήμα 4.3: Τυπική απόφαση $P\Phi;\Phi \vdash f$ ok.

$$\text{(r-ok)}\frac{P\Phi;\overrightarrow{vars} \vdash p_{goal} \text{ ok} \qquad \forall p \text{ in } \overrightarrow{p_{hyp}}:\ P\Phi;\overrightarrow{vars} \vdash p \text{ ok}}{P\Phi \vdash (\overrightarrow{vars}, p_{goal}, \overrightarrow{p_{hyp}}) \text{ ok}}$$

Σχήμα 4.4: Τυπική απόφαση $P\Phi \vdash r$ ok.

$$\text{(prg-ok)}\frac{\forall r \text{ in } \overrightarrow{r}:\ P\Phi \vdash r \text{ ok} \qquad \forall f \text{ in } \overrightarrow{f}:\ P\Phi;\overrightarrow{f} \vdash f \text{ ok} \qquad \Phi;\emptyset \vdash e:\tau \qquad (\text{όπου } P\Phi = \overrightarrow{prfuns},\ Cond:(\mathbf{bool},\mathbf{prop},\mathbf{prop}) \to \mathbf{prop},\ Succ:(\mathbf{nat}) \to \mathbf{nat})}{\cdot \vdash (\overrightarrow{prfuns}, \overrightarrow{r}, \overrightarrow{f}, e) \text{ ok}}$$

Σχήμα 4.5: Τυπική απόφαση $\cdot \vdash prg$ ok.

ορίσματα τύπου **bool**. Όμως, θέλουμε να χρησιμοποιούμε μία έκφραση τύπου **bool** στη θέση μίας έκφρασης τύπου **prop**, και γι'αυτό εισάγουμε τη σχέση υποτύπων. Έτσι, για παράδειγμα, μία λογική πρόταση μπορεί να είναι μία απλή ανισότητα μεταξύ ακεραίων.

4.1.2 Τυπικές αποφάσεις με έλεγχο λογικής εγκυρότητας

Οι τυπικές αποφάσεις με έλεγχο λογικής εγκυρότητας ελέγχουν το κατά πόσο μπορούν να βρεθούν αποδείξεις για τις διάφορες λογικές προτάσεις που πρέπει να ικανοποιούνται, μέσω των θεωρημάτων και των διαθέσιμων αποδείξεων. Στηρίζονται στην απόφαση $\Sigma;\Pi \Vdash p$ η οποία καθορίζει το κατά πόσον υπάρχει απόδειξη για τη λογική πρόταση p δεδομένου του περιβάλλοντος θεωρημάτων Σ και του περιβάλλοντος διαθέσιμων αποδείξεων Π. Η απόφαση αυτή ορίζεται στο σχήμα 4.8. Επειδή η συγκεκριμένη απόφαση δεν είναι απαραίτητα αποκρίσιμη, ο έλεγχος τύπων με έλεγχο λογικής εγκυρότητας ενός προγράμματος Karooma δεν τερματίζει απαραίτητα. Ωστόσο, επειδή το σύστημα NFLINT έχει αποκρίσιμο έλεγχο τύπων, αν ο έλεγχος λογικής εγκυρότητας του προγράμματος Karooma τερματίσει και ο μετασχηματισμός σε κώδικα NFLINT γίνει επιτυχώς, το τελικό εκτελέσιμο θα μπορεί να ελεγχθεί με μία αποκρίσιμη διαδικασία για μερική ορθότητα, πριν την εκτέλεσή του.

Στο σημείο αυτό εισάγουμε έναν μετασχηματισμό από την αρχική μορφή των εκφράσεων σε μία νέα μορφή, η οποία απλοποιεί αρκετά τους κανόνες τύπων των τυπικών αποφάσεων αυτών. Επίσης είναι χρήσιμη κατά τη μετάφραση στο σύστημα NFLINT. Στη μετασχηματισμένη μορφή των εκφράσεων, διακρίνουμε τις απλές εκφράσεις (σταθερές, δεσμευμένες μεταβλητές, και πράξεις μεταξύ αυτών) από τις κλήσεις συναρτήσεων και τη διακλάδωση if-then-else. Μία μετασχηματισμένη έκφραση αποτελείται είτε από μία απλή έκφραση είτε από μία διακλάδωση if-then-else, με τη συνθήκη να είναι απλή έκφραση. Έτσι οι διακλαδώσεις είναι φωλιασμένες, και δεν εμφανίζονται ως μέρος των απλών εκφράσεων. Επίσης, η κλήση συναρτήσεων γίνεται δεσμεύοντας το αποτέλεσμά τους σε μία νέα μεταβλητή. Τυπικά, η σύνταξη των μετασχηματισμένων εκφράσεων είναι η εξής:

$$\begin{array}{lcl} te & ::= & \mathbf{bind}\ be\ \mathbf{in}\ te \mid ise \\ ise & ::= & se \mid \mathbf{if}(se,\ te_1,\ te_2) \\ se & ::= & id \mid n \mid b \mid \mathbf{binop}(op,\ se_1,\ se_2) \mid \mathbf{unop}(op,\ se) \\ be & ::= & (id, id(se_1,\ se_2,\ \cdots,\ se_n)) \end{array}$$

$$[\![id]\!]_{if} = id \quad [\![n]\!]_{if} = n \quad [\![b]\!]_{if} = b$$

$$[\![\mathbf{binop}(op,\, e_1,\, e_2)]\!]_{if} \;=\; \begin{cases} [\![\mathbf{if}(e_1',\, \mathbf{binop}(op,\, e_2',\, e_2),\, \mathbf{binop}(op,\, e_3',\, e_2))]\!]_{if} \\ \qquad\qquad ,\ \textit{εάν}\ [\![e_1]\!]_{if} = \mathbf{if}(e_1',\, e_2',\, e_3') \\ [\![\mathbf{if}(e_1',\, \mathbf{binop}(op,\, e_1,\, e_2'),\, \mathbf{binop}(op,\, e_1,\, e_3'))]\!]_{if} \\ \qquad\qquad ,\ \textit{εάν}\ [\![e_2]\!]_{if} = \mathbf{if}(e_1',\, e_2',\, e_3') \\ \mathbf{binop}(op,\, e_1,\, e_2),\ \textit{διαφορετικά} \end{cases}$$

$$[\![\mathbf{unop}(op,\, e)]\!]_{if} \;=\; \begin{cases} [\![\mathbf{if}(e_1',\, \mathbf{unop}(op,\, e_2'),\, \mathbf{unop}(op,\, e_3'))]\!]_{if} \\ \qquad\qquad ,\ \textit{εάν}\ [\![e]\!]_{if} = \mathbf{if}(e_1',\, e_2',\, e_3') \\ \mathbf{unop}(op,\, e),\ \textit{διαφορετικά} \end{cases}$$

$$[\![\mathbf{if}(e_1,\, e_2,\, e_3)]\!]_{if} \;=\; \begin{cases} \mathbf{if}(e_1',\, [\![\mathbf{if}(e_2',\, e_2,\, e_3)]\!]_{if},\, [\![\mathbf{if}(e_3',\, e_2,\, e_3)]\!]_{if}) \\ \qquad\qquad ,\ \textit{εάν}\ [\![e_1]\!]_{if} = \mathbf{if}(e_1',\, e_2',\, e_3') \\ \mathbf{if}(e_1,\, [\![e_2]\!]_{if},\, [\![e_3]\!]_{if}),\ \textit{διαφορετικά} \end{cases}$$

$$[\![id(e_1,\ e_2,\ \cdots,\ e_n)]\!]_{if} \;=\; \begin{cases} [\![\mathbf{if}(e_{k1},\, id(e_1,\ e_2,\ \cdots,\ e_{k2},\ \cdots,\ e_n), \\ \quad id(e_1,\ e_2,\ \cdots,\ e_{k3},\ \cdots,\ e_n))]\!]_{if} \\ \qquad\qquad ,\ \textit{εάν}\ [\![e_k]\!]_{if} = \mathbf{if}(e_{k1},\, e_{k2},\, e_{k3}) \\ id(e_1,\ e_2,\ \cdots,\ e_n),\ \textit{διαφορετικά} \end{cases}$$

Σχήμα 4.6: Μετασχηματισμός $[\![\cdot]\!]_{if} \ :\ e \ \rightarrow\ e.$

$$[\![id]\!]_{fun} = (\emptyset,\ id) \quad [\![n]\!]_{fun} = (\emptyset,\ n) \quad [\![b]\!]_{fun} = (\emptyset,\ b)$$

$$[\![\mathbf{binop}(op,\, e_1,\, e_2)]\!]_{fun} = (be_1 \oplus be_2,\ \mathbf{binop}(op,\, ise_1,\, ise_2)),\ \textit{εάν}\ [\![e_1]\!]_{fun} = (be_1,\ ise_1) \\ \textit{και}\ [\![e_2]\!]_{fun} = (be_2,\ ise_2)$$

$$[\![\mathbf{unop}(op,\, e)]\!]_{fun} = \ (be,\ \mathbf{unop}(op,\, ise)),\ \textit{εάν}\ [\![e]\!]_{fun} = (be,\ ise)$$

$$[\![\mathbf{if}(e_1,\, e_2,\, e_3)]\!]_{fun} = \ (be_1,\ \mathbf{if}(ise_1,\, [\![e_2]\!]_{fun2},\, [\![e_3]\!]_{fun2})),\ \textit{εάν}\ [\![e_1]\!]_{fun2} = (be_1,\ ise_1)$$

$$[\![id(e_1,\ e_2,\ \cdots,\ e_n)]\!]_{fun} = \ (be_1 \oplus \cdots \oplus be_n \oplus \{(\mathbf{new}, id(ise_1,\ ise_2,\ \cdots,\ ise_n))\},\ \mathbf{new}) \\ ,\ \textit{εάν}\ [\![e_1]\!]_{fun} = (be_1,\ ise_1),\ \cdots,\ [\![e_n]\!]_{fun} = (be_n,\ ise_n)$$

$$[\![e]\!]_{fun_2} = \ \mathbf{bind}\ b_1\ \mathbf{in}\ (\mathbf{bind}\ b_2\ \mathbf{in}\ (\cdots\ (\mathbf{bind}\ b_n\ \mathbf{in}\ (ise)))) \\ ,\ \textit{εάν}\ [\![e]\!]_{fun} = (be, ise)\ \textit{και}\ be = \{b_1,\ b_2,\ \cdots,\ b_n\}$$

Σχήμα 4.7: Μετασχηματισμός $[\![\cdot]\!]_{fun} : e \rightarrow (be^*, ise)$ και $[\![\cdot]\!]_{fun2} : e \rightarrow te.$

Η αρχική μορφή μετασχηματίζεται στη μορφή αυτή σε δύο στάδια. Στο πρώτο ($[\![\cdot]\!]_{if}$), γίνεται η μετατροπή των if σε φωλιασμένα if, και στο δεύτερο ($[\![\cdot]\!]_{fun2}$) γίνονται δεσμεύσεις μεταβλητών για τις κλήσεις συναρτήσεων και έτσι εξάγονται από τις απλές εκφράσεις. Ο συνολικός μετασχηματισμός ορίζεται ως $[\![\cdot]\!]_{full} = [\![\ [\![\cdot]\!]_{if}\ ]\!]_{fun2}$. Ο ορισμός των μετασχηματισμών φαίνεται στα σχήματα 4.6 - 4.7. Σε αυτά συμβολίζουμε την κενή λίστα με $\emptyset$ και τη συνένωση λιστών με $L_1 \oplus L_2$.

Μπορούμε τώρα να δώσουμε τους κανόνες τύπων για τις τυπικές αποφάσεις ελέγχου λογικής εγκυρότητας. Σε αυτούς χρησιμοποιούμε κάποια στοιχεία χωρίς να τα ορίσουμε επακριβώς, όπως την ταυτόχρονη αντικατάσταση μεταβλητών με εκφράσεις (με συμβολισμό $e[x := e']$) και την δομική ισότητα μεταξύ εκφράσεων (με συμβολισμό $e_1 = e_2$). Επίσης, για απλοποίηση των κανόνων, χρησιμοποιούμε σε κάποια σημεία το περιβάλλον Ψ στη θέση του περιβάλλοντος Φ, όπως και μία απλή έκφραση (se) στη θέση λογικής πρότασης (p), αφού οι μετατροπές μεταξύ

$$\frac{p \text{ in } \Pi}{\Sigma; \Pi \Vdash p}$$

$$\frac{((\overrightarrow{x} : \overrightarrow{\tau p}), p_{goal}, \overrightarrow{p_{hyp}}) \text{ in } \Sigma \qquad \exists \overrightarrow{v} : \overrightarrow{\tau p} : (p = p_{goal}[\overrightarrow{x} := \overrightarrow{v}] \ \wedge \ \forall p_{hyp} \text{ in } \overrightarrow{p_{hyp}} : \Sigma; \Pi \Vdash p_{hyp}[\overrightarrow{x} := \overrightarrow{v}])}{\Sigma; \Pi \Vdash p}$$

Σχήμα 4.8: Τυπική απόφαση $\Sigma; \Pi \Vdash p$.

$$\frac{\Sigma; \Pi \Vdash p[id := se]}{\Psi; \Sigma; \Pi \Vdash p \ (id \rightsquigarrow se)}$$

$$\frac{\Sigma; \Pi \Vdash \mathbf{Cond}(se_1, se_1, negate(se_1)) \qquad \Psi; \Sigma; \Pi, se_1 \Vdash p \ (id \rightsquigarrow te_2) \qquad \Psi; \Sigma; \Pi, negate(se_1) \Vdash p \ (id \rightsquigarrow te_3)}{\Psi; \Sigma; \Pi \Vdash p \ (id \rightsquigarrow \mathbf{if}(se_1, \, te_2, \, te_3))}$$

$$\frac{id_{func} : (\overrightarrow{id_{params}} : \overrightarrow{\tau_{params}}[\overrightarrow{p_{pre}}] \rightarrow id_{ret} : \tau_{ret}[\overrightarrow{p_{post}}]) := e_{func} \text{ in } \Psi \qquad \forall p' \text{ in } \overrightarrow{p_{pre}} : \Sigma; \Pi \Vdash p'[\overrightarrow{id_{params}} := \overrightarrow{se}] \qquad \Psi; \Sigma; \Pi, \overrightarrow{p_{post}}[\overrightarrow{id_{params}} := \overrightarrow{se}, id_{ret} := id_{bind}] \Vdash p \ (id \rightsquigarrow te')}{\Psi; \Sigma; \Pi \Vdash p \ (id \rightsquigarrow \mathbf{bind} \ (id_{bind}, id_{func}(\overrightarrow{se})) \ \mathbf{in} \ te')}$$

Σχήμα 4.9: Τυπική απόφαση $\Psi; \Sigma; \Pi \Vdash p \ (id \rightsquigarrow te)$.

$$\begin{gathered}
negate(id) = \mathbf{unop}(!, \, id) \quad negate(b) = notb \\
negate(\mathbf{binop}(\leq , \, se_1, \, se_2)) = \mathbf{binop}(> , \, se_1, \, se_2) \\
negate(\mathbf{binop}(\geq , \, se_1, \, se_2)) = \mathbf{binop}(< , \, se_1, \, se_2) \\
negate(\mathbf{binop}(< , \, se_1, \, se_2)) = \mathbf{binop}(\geq , \, se_1, \, se_2) \\
negate(\mathbf{binop}(> , \, se_1, \, se_2)) = \mathbf{binop}(\leq , \, se_1, \, se_2) \\
negate(\mathbf{binop}(== , \, se_1, \, se_2)) = \mathbf{binop}(! = , \, se_1, \, se_2) \\
negate(\mathbf{binop}(! = , \, se_1, \, se_2)) = \mathbf{binop}(== , \, se_1, \, se_2) \\
negate(\mathbf{binop}(\&\&, \, se_1, \, se_2)) = \mathbf{binop}(||, \, negate(se_1), \, negate(se_2)) \\
negate(\mathbf{binop}(||, \, se_1, \, se_2)) = \mathbf{binop}(\&\&, \, negate(se_1), \, negate(se_2)) \\
negate(\mathbf{unop}(!, \, se)) = se
\end{gathered}$$

Σχήμα 4.10: Η συνάρτηση άρνησης λογικής πρότασης negate.

αυτών είναι τετριμμένες. Οι κανόνες δίνονται στα σχήματα 4.8 - 4.12, μαζί με κάποιες βοηθητικές τυπικές αποφάσεις. Να σημειώσουμε ότι με το συμβολισμό $p \ (id \rightsquigarrow te)$ εννοούμε την αντικατάσταση της μεταβλητής id με τη σύνθετη έκφραση te μέσα στη λογική πρόταση p, κάτι που γίνεται αποδομώντας τη σύνθετη έκφραση σε απλές, όπως φαίνεται στην αντίστοιχη τυπική απόφαση.

4.2 Δυναμική σημασιολογία της Karooma

Στο κεφάλαιο αυτό θα περιγράψουμε τη δυναμική σημασιολογία της Karooma ορίζοντας τη λειτουργική σημασιολογία της (operational semantics). Οι τιμές της γλώσσας είναι οι ακέραιες και δυαδικές σταθερές. Η κατάσταση που θα χρησιμοποιήσουμε αποτελείται από την τρέχουσα έκφραση υπό αποτίμηση και την αντιστοίχηση ονομάτων συναρτήσεων με τα ονόματα των παραμέτρων τους και το σώμα τους. Έτσι:

$$\frac{\begin{array}{c} P\Phi;\Psi \ \vdash \ id_{func} : (\overrightarrow{params}[\overrightarrow{p_{pre}}] \to id_{ret} : \tau_{ret}[\overrightarrow{p_{post}}]) := e_{func} \text{ ok} \\ \forall p \text{ in } \overrightarrow{p_{post}} : \Psi;\Sigma;\overrightarrow{p_{pre}} \ \Vdash \ p \ (id_{ret} \rightsquigarrow [\![e_{func}]\!]_{full}) \end{array}}{P\Phi;\Sigma;\Psi \ \vdash^* \ id_{func} : (\overrightarrow{params}[\overrightarrow{p_{pre}}] \to id_{ret} : \tau_{ret}[\overrightarrow{p_{post}}]) := e_{func} \text{ ok}}$$

Σχήμα 4.11: Τυπική απόφαση $P\Phi;\Sigma;\Psi \ \vdash^* \ f$ ok.

$$\frac{\begin{array}{c} \cdot \ \vdash \ (\overrightarrow{prfuns}, \overrightarrow{rules}, \overrightarrow{f}, e) \text{ ok} \\ \forall f in \overrightarrow{f} : P\Phi;\overrightarrow{rules};\overrightarrow{f} \ \vdash^* \ f \text{ ok} \\ \overrightarrow{f};\emptyset \ \vdash \ e : \tau \qquad P\Phi;\overrightarrow{rules};\overrightarrow{f} \ \vdash^* \ \text{“result”} : (\emptyset)[\emptyset] \to \text{“resid”} : \tau[\emptyset] := e \text{ ok} \\ (\text{όπου } P\Phi = \overrightarrow{prfuns},\ Cond : (\mathbf{bool}, \mathbf{prop}, \mathbf{prop}) \to \mathbf{prop},\ Succ : (\mathbf{nat}) \to \mathbf{nat}) \end{array}}{\cdot \ \vdash^* \ (\overrightarrow{prfuns}, \overrightarrow{rules}, \overrightarrow{f}, e) \text{ ok}}$$

Σχήμα 4.12: Τυπική απόφαση $\cdot \ \vdash^* \ prg$ ok.

$$\begin{array}{lcl} v & ::= & n \mid \mathbf{true} \mid \mathbf{false} \\ s & ::= & (e, F) \text{ όπου } F : (id \rightharpoonup (id^*, e))^* \end{array}$$

Με $[\![\diamond]\!] : v \to v$ συμβολίζουμε τη σημασία του τελεστή ενός τελούμενου $\diamond$. Αντίστοιχα με $[\![\circ]\!] : v \times v \to v$ συμβολίζουμε τη σημασία του τελεστή δύο τελούμενων $\circ$.

Βάσει αυτών, δίνουμε στο σχήμα 4.13 τη λειτουργική σημασιολογία της γλώσσας. Τα μόνα στοιχεία που χρήζουν σχολιασμού είναι το γεγονός ότι οι κλήσεις συναρτήσεων γίνονται με πέρασμα κατά τιμή (call-by-value), και έτσι η γλώσσα χαρακτηρίζεται από πρόθυμη αποτίμηση (eager evaluation), καθώς επίσης και το γεγονός ότι οι λογικές προτάσεις δεν επηρεάζουν την εκτέλεση του προγράμματος.

Η τιμή του προγράμματος $(\overrightarrow{pf}, \overrightarrow{r}, \overrightarrow{f}, e_{res})$ προκύπτει από την τιμή της τελικής κατάστασης που προκύπτει από την αρχική κατάσταση (e_{res}, F), όπου

$$\begin{array}{c} F(id_{func}) = (\overrightarrow{id_{params}}, e_{func}) \Leftrightarrow \\ id_{func} : (\overrightarrow{id_{params}} : \overrightarrow{\tau_{params}}[\overrightarrow{p}] \to id_{ret} : \tau_{ret}[\overrightarrow{p'}]) := e_{func} \text{ in } \overrightarrow{f} \end{array}$$

4.3 Υλοποίηση

Η υλοποίηση του ελεγκτή τύπων και του μεταφραστή από Karooma σε NFLINT έγινε σε γλώσσα Ocaml η οποία είναι μία ιδιαιτέρως αποδοτική συναρτησιακή γλώσσα στο στυλ της ML. Προτιμήθηκε διότι η υλοποίηση του συστήματος NFLINT έχει γίνει στη γλώσσα αυτή, κυρίως για λόγους απόδοσης, οπότε η σύνδεση του μεταφραστή με το υπάρχον σύστημα είναι απλή.

Ο ελεγκτής τύπων υλοποιήθηκε βάσει των παραπάνω κανόνων τύπων. Για αποφυγή της επανάληψης παρόμοιου κώδικα, στην υλοποίηση έχουμε συνθέσει τον έλεγχο τύπων για εκφράσεις και για λογικές προτάσεις λόγω της ομοιότητάς τους. Επίσης, ο έλεγχος λογικής εγκυρότητας γίνεται καθώς κατασκευάζονται οι αποδείξεις από το μηχανισμό αυτόματης εύρεσης αποδείξεων κατά τη μεταγλώττιση του προγράμματος, και όχι από πριν. Έτσι ο έλεγχος προκύπτει ελαφρώς πιο αυστηρός από τον έλεγχο που γίνεται βάσει των παραπάνω κανόνων: στην τυπική απόφαση $\Sigma;\Pi \ \Vdash \ p$ αρκεί να υπάρχει κατάλληλη απόδειξη για την λογική πρόταση p, αλλά στην υλοποίησή μας πρέπει μία τέτοια απόδειξη να μπορεί να βρεθεί από το μηχανισμό αυτόματης εύρεσης αποδείξεων. Επειδή αυτός ο μηχανισμός δεν είναι στην παρούσα φάση τόσο ισχυρός ώστε να βρίσκει αποδείξεις για όλες τις προτάσεις που μπορούν να αποδειχθούν βάσει της τυπικής απόφασης αυτής, ο έλεγχος λογικής εγκυρότητας είναι κάπως πιο αυστηρός.

$$(\diamond v, F) \;\longrightarrow\; (\llbracket \diamond \rrbracket(v), F) \qquad (v_1 \circ v_2, F) \;\longrightarrow\; (\llbracket \circ \rrbracket(v_1, v_2), F)$$

$$(\mathbf{if}(\mathbf{true},\, e_1,\, e_2), F) \;\longrightarrow\; (e_1, F) \qquad (\mathbf{if}(\mathbf{false},\, e_1,\, e_2), F) \;\longrightarrow\; (e_2, F)$$

$$\frac{(e, F) \;\longrightarrow\; (e', F)}{(\diamond e, F) \;\longrightarrow\; (\diamond e', F)}$$

$$\frac{(e_1, F) \;\longrightarrow\; (e_1', F)}{(e_1 \circ e_2, F) \;\longrightarrow\; (e_1' \circ e_2, F)} \qquad \frac{(e_2, F) \;\longrightarrow\; (e_2', F)}{(v_1 \circ e_2, F) \;\longrightarrow\; (v_1 \circ e_2', F)}$$

$$\frac{(e, F) \;\longrightarrow\; (e', F)}{(\mathbf{if}(e,\, e_1,\, e_2), F) \;\longrightarrow\; (\mathbf{if}(e',\, e_1,\, e_2), F)}$$

$$\frac{F(id) = (\overrightarrow{id_{params}}, e_{body})}{(id(\overrightarrow{v_{params}}), F) \;\longrightarrow\; (e_{body}[\overrightarrow{id_{params}} := \overrightarrow{v_{params}}], F)}$$

$$\frac{(e_k, F) \;\longrightarrow\; (e_k', F)}{(id(v_1, v_2, \cdots, e_k, e_{k+1}, \cdots, e_n), F) \;\longrightarrow\; (id(v_1, v_2, \cdots, e_k', e_{k+1}, \cdots, e_n), F)}$$

Σχήμα 4.13: Λειτουργική σημασιολογία της Karooma.

Κεφάλαιο 5

Το σύστημα πιστοποιημένων εκτελέσιμων NFLINT

Το σύστημα πιστοποιημένων εκτελέσιμων για το οποίο παράγει κώδικα ο μεταγλωττιστής της Karooma ονομάζεται NFLINT. Είναι ουσιαστικά μία ενδιάμεση συναρτησιακή γλώσσα με ισχυρό σύστημα τύπων, παραλλαγή της γλώσσας που περιγράφεται στο Shao *et al.* (2002). Το NFLINT χωρίζεται λογικά σε δύο επιμέρους τμήματα: τη γλώσσα τύπων και τη γλώσσα υπολογισμών. Η πρώτη περιλαμβάνει τους κανόνες των τύπων και σε αυτήν περιγράφονται οι τύποι των δομικών μονάδων της γλώσσας υπολογισμών. Στην δεύτερη αναπαριστώνται οι διάφορες υπολογιστικές εντολές (πράξεις μεταξύ ακεραίων, κλήση συναρτήσεων, κ.λπ.) και κατα συνέπεια όλη η λειτουργικότητα των παραγόμενων προγραμμάτων.

Η γλώσσα τύπων είναι στενά συνδεδεμένη με τη γλώσσα υπολογισμών. Κάθε πρόγραμμα εκφρασμένο στην δεύτερη έχει και έναν αντίστοιχο τύπο, εκφρασμένο στην πρώτη. Αν αλλάξει ένα μέρος του προγράμματος στην υπολογιστική γλώσσα, τότε θα υπάρξει αναντιστοιχία με τον τύπο του. Αυτό έχει σαν αποτέλεσμα να μπορεί να εξακριβωθεί ανα πάσα στιγμή η εγκυρότητα του προγράμματος.

Ένα ακόμη χαρακτηριστικό της ενδιάμεσης γλώσσας είναι η δυνατότητα περιγραφής στην γλώσσα τύπων αποδείξεων για ορισμένες ιδιότητες του προγράμματος. Οι ιδιότητες αυτές μπορούν να είναι αρκετά απλές, όπως η απόδειξη ότι κάποια συνάρτηση επιστέφει μία τιμή συγκεκριμένου τύπου, έως αρκετά πολύπλοκες, όπως η απόδειξη της ισχύος των προσυνθηκών μίας συνάρτησης κατά την κλήση της.

5.1 Η Γλώσσα των Τύπων

Η γλώσσα των τύπων αποτελεί έναν πολυμορφικό λ-λογισμό με υποστήριξη επαγωγικών δηλώσεων, και είναι αρκετά κοντά στον λογισμό των επαγωγικών κατασκευών.

Ο ορισμός της γλώσσας τύπων σε BNF μορφή, είναι:

$$\begin{array}{lcl} s & ::= & \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Ext} \\ X & ::= & z \mid k \mid t \\ A, B & ::= & s \mid X \mid \Pi X\!:\!A.\,B \mid \lambda X\!:\!A.\,B \mid A\,B \\ & \mid & \mathsf{Ind}(X\!:\!A)\{\underline{A}\} \mid \mathsf{Constr}(n, A) \mid \mathsf{Elim}[A'](A\!:\!B\,\vec{B})\{\underline{A}\} \end{array}$$

Η γλώσσα δεν κάνει συντακτική διάκριση μεταξύ των όρων και των τύπων, ενώ ορίζονται τριών ειδών Sorts, τα Set, Type και Ext. Τα δύο πρώτα αντιστοιχούν στα $*$ και $\square$. Όπως έχει αναφερθεί στη θεωρία του λ-calculus, Set είναι ο τύπος των τύπων, και Type ο τύπος των πολυμορφικών τύπων. Ext είναι ο τύπος του Type.

Τα υπόλοιπα στοιχεία της γλώσσας, είναι κατά τα γνωστά από τον λογισμό των επαγωγικών κατασκευών:

- $\Pi x\!:\!A.\,B$ που ορίζει τύπους, εξαρτώμενους από όρους [1]
- $\lambda x\!:\!A.\,B$ που ορίζει συναρτήσεις

[1] Ως όροι αναφέρονται όλα τα στοιχεία της γλώσσας, δηλαδή και οι τύποι.

- AB που ορίζει την εφαρμογή ενός όρου σε όρο
- $\mathsf{Ind}(X\!:\!A)\{\underline{A}\}$ που ορίζει έναν επαγωγικό τύπο
- $\mathsf{Constr}(n, A)$ που ορίζει έναν κατασκευαστή τύπου A
- $\mathsf{Elim}[A'](A\!:\!B\,\vec{B})\{\underline{A}\}$ που ορίζει την αναδρομική κλήση στον κατασκευαστή ενός στοιχείου.

Παρακάτω χρησιμοποιούνται οι εξής συμβολισμοί:

$$\begin{array}{lcll}
A \to B & \equiv & \Pi X\!:\!A.\,B & \text{αν } X \notin \mathsf{FV}(B) \\
B\,\vec{A} & \equiv & B\;\; A_1A_2\ldots A_n & \\
\Pi \vec{X}\!:\!\vec{A}.\,B & \equiv & \Pi X_1\!:\!A_1.\,\Pi X_2\!:\!A_2.\,\ldots\Pi X_n\!:\!A_n.\,B & \\
\lambda \vec{X}\!:\!\vec{A}.\,B & \equiv & \lambda X_1\!:\!A_1.\,\lambda X_2\!:\!A_2.\,\ldots\lambda X_n\!:\!A_n.\,B &
\end{array}$$

ενώ με $\underline{A}$ συμβολίζουμε μία ακολουθία από όρους.

Παραπάνω χρησιμοποιήθηκε το σύμβολο $\mathsf{FV}(A)$, το οποίο παριστάνει το σύνολο των ελεύθερων μεταβλητών του όρου A. Ελεύθερες χαρακτηρίζουμε τις μεταβλητές ενός όρου, οι οποίες δεν δεσμεύονται από κάποιο λ, Π ή Ind. Ο ορισμός του FV() είναι:

$$\begin{array}{lcl}
\mathsf{FV}(s) & = & \emptyset \\
\mathsf{FV}(X) & = & \{X\} \\
\mathsf{FV}(\Pi X\!:\!A.\,B) & = & \mathsf{FV}(A) \cup (\mathsf{FV}(B) - \{X\}) \\
\mathsf{FV}(\lambda X\!:\!A.\,B) & = & \mathsf{FV}(A) \cup (\mathsf{FV}(B) - \{X\}) \\
\mathsf{FV}(A\,B) & = & \mathsf{FV}(A) \cup \mathsf{FV}(B) \\
\mathsf{FV}(\mathsf{Ind}(X\!:\!A)\{\underline{A}\}) & = & \mathsf{FV}(A) \cup (\mathsf{FV}(\underline{A}) - \{X\}) \\
\mathsf{FV}(\mathsf{Constr}(n, A)) & = & \mathsf{FV}(\vec{A}) \\
\mathsf{FV}(\mathsf{Elim}[A'](A\!:\!B\,\vec{B})\{\underline{A}\}) & = & \mathsf{FV}(A') \cup \mathsf{FV}(A) \cup \mathsf{FV}(B) \cup \mathsf{FV}(\vec{B}) \cup \mathsf{FV}(\underline{A}) \\
\mathsf{FV}(\underline{A}) & = & \cup_i \mathsf{FV}(A_i)
\end{array}$$

Η αντικατάσταση ορίζεται ως:

$$\begin{array}{lcl}
s[\underline{Z} \mapsto \underline{R}] & \equiv & s \\
X[\underline{Z} \mapsto \underline{R}] & \equiv & \begin{cases} X & \text{αν } X \neq Z_i, \text{ για κάθε } i \\ R_i & \text{αν } X = Z_i, \text{ για κάποιο } i \end{cases} \\
(\Pi X\!:\!A.\,B)[\underline{Z} \mapsto \underline{R}] & \equiv & \begin{cases} \Pi X\!:\!A[\underline{Z} \mapsto \underline{R}].\,B[\underline{Z} \mapsto \underline{R}] & \text{, περ. (i)} \\ \Pi Y\!:\!A[\underline{Z} \mapsto \underline{R}].\,B[X \mapsto Y][\underline{Z} \mapsto \underline{R}] & \text{, περ. (ii)} \end{cases} \\
(\lambda X\!:\!A.\,B)[\underline{Z} \mapsto \underline{R}] & \equiv & \begin{cases} \lambda X\!:\!A[\underline{Z} \mapsto \underline{R}].\,B[\underline{Z} \mapsto \underline{R}] & \text{, περ. (i)} \\ \lambda Y\!:\!A[\underline{Z} \mapsto \underline{R}].\,B[X \mapsto Y][\underline{Z} \mapsto \underline{R}] & \text{, περ. (ii)} \end{cases} \\
(A\,B)[\underline{Z} \mapsto \underline{R}] & \equiv & A[\underline{Z} \mapsto \underline{R}]\,B[\underline{Z} \mapsto \underline{R}] \\
(\mathsf{Ind}(X\!:\!A)\{\underline{A}\})[\underline{Z} \mapsto \underline{R}] & \equiv & \begin{cases} \mathsf{Ind}(X\!:\!A[\underline{Z} \mapsto \underline{R}])\{\underline{A}[\underline{Z} \mapsto \underline{R}]\} & \text{, περ. (i)} \\ \mathsf{Ind}(Y\!:\!A[\underline{Z} \mapsto \underline{R}])\{\underline{A}[X \mapsto Y][\underline{Z} \mapsto \underline{R}]\} & \text{, περ. (ii)} \end{cases} \\
(\mathsf{Constr}(n, A))[\underline{Z} \mapsto \underline{R}] & \equiv & \mathsf{Constr}(n, A[\underline{Z} \mapsto \underline{R}]) \\
(\mathsf{Elim}[A'](A\!:\!B\,\underline{B})\{\underline{A}\})[\underline{Z} \mapsto \underline{R}] & \equiv & \mathsf{Elim}[A'[\underline{Z} \mapsto \underline{R}]](A[\underline{Z} \mapsto \underline{R}]\!:\!B[\underline{Z} \mapsto \underline{R}]\,\underline{B}[\underline{Z} \mapsto \underline{R}]) \\
& & \quad \{\underline{A}[\underline{Z} \mapsto \underline{R}]\}
\end{array}$$

Οι δύο περιπτώσεις που αναφέρονται αντιστοιχούν στις παρακάτω συνθήκες:

$$\begin{array}{ll}
\text{περίπτωση (i)} & X \neq Z_i, \text{ για κάθε } i, \text{ και } X \notin \mathsf{FV}(\underline{R}) \\
\text{περίπτωση (ii)} & X = Z_i, \text{ για κάποιο } i, \text{ ή } X \in \mathsf{FV}(\underline{R})
\end{array}$$

και Y είναι μια καινούργια μεταβλητή, που θεωρούμε οτι δεν συναντάται σε κανέναν όρο.

5.2 Λειτουργική σημασιολογία

Προκειμένου η γλώσσα τύπων που ορίσαμε να βρίσκει εφαρμογή, θα έπρεπε να ορίσουμε τόσο τον τρόπο που οι όροι της γλώσσας αποτιμώνται σε απλούστερες εκφράσεις. Παρακάτω ορίζουμε τις προϋποθέσεις κάτω από τις οποίες γίνεται αποδεκτή μια σχέση μεταξύ όρων, και ακόμα τις β, η και ι αναγωγές, αντίστοιχα με αυτές που ορίσαμε στον λ-λογισμό. Αναγκαίο ακόμα είναι να ορίσουμε και τη σχέση ισότητας μεταξύ των όρων της γλώσσας .

5.2.1 Συμβατές σχέσεις

Έστω μια σχέση $\sim$ μεταξύ όρων της γλώσσας. Ορίζουμε τη σχέση $\sim^*$ σε ένα σύνολο από όρους ως την ελάχιστη σχέση που ικανοποιεί τις παρακάτω προϋποθέσεις για όλους του όρους A , A' και σύνολα από όρους $\underline{B}$, $\underline{B}'$:

$$\begin{array}{lcl} A \sim A' & \Rightarrow & A; \underline{B} \sim^* A'; \underline{B} \\ \underline{B} \sim \underline{B}' & \Rightarrow & A; \underline{B} \sim^* A; \underline{B}' \end{array}$$

Μια σχέση $\sim$ θεωρείται αποδεκτή στη γλώσσα τύπων, αν ισχύουν τα παρακάτω:
Έστω X μεταβλητές, A, A', B, B', Q, Q' , όροι και $\underline{A}$, $\underline{B}$ σύνολα από όρους. Τότε πρέπει:

$$\begin{array}{lcl} A \sim A' & \Rightarrow & \Pi X{:}A.\,B \sim \Pi X{:}A'.\,B \\ B \sim B' & \Rightarrow & \Pi X{:}A.\,B \sim \Pi X{:}A.\,B' \\ A \sim A' & \Rightarrow & \lambda X{:}A.\,B \sim \Pi X{:}A'.\,B \\ B \sim B' & \Rightarrow & \lambda X{:}A.\,B \sim \Pi X{:}A.\,B' \\ A \sim A' & \Rightarrow & A\,B \sim A'\,B \\ B \sim B' & \Rightarrow & A\,B \sim A\,B' \\ A \sim A' & \Rightarrow & \mathsf{Ind}(X{:}A)\{\underline{A}\} \sim \mathsf{Ind}(X{:}A')\{\underline{A}\} \\ \vec{A} \sim^* \vec{A'} & \Rightarrow & \mathsf{Ind}(X{:}A)\{\underline{A}\} \sim \mathsf{Ind}(X{:}A)\{\underline{A'}\} \\ A \sim A' & \Rightarrow & \mathsf{Constr}(i, A) \sim \mathsf{Constr}(i, A') \\ Q \sim Q' & \Rightarrow & \mathsf{Elim}[Q](A{:}B\,\vec{B})\{\underline{A}\} \sim \mathsf{Elim}[Q'](A{:}B\,\vec{B})\{\underline{A}\} \\ A \sim A' & \Rightarrow & \mathsf{Elim}[Q](A{:}B\,\vec{B})\{\underline{A}\} \sim \mathsf{Elim}[Q](A'{:}B\,\vec{B})\{\underline{A}\} \\ B \sim B' & \Rightarrow & \mathsf{Elim}[Q](A{:}B\,\vec{B})\{\underline{A}\} \sim \mathsf{Elim}[Q](A{:}B'\,\vec{B})\{\underline{A}\} \end{array}$$

5.2.2 Αναγωγές

Ορίζουμε τις β, η και ι αναγωγές, ως τις μικρότερες συμβατές σχέσεις που ικανοποιούν:

$$(\lambda X{:}A.\,B)\,A' \quad \to_\beta \quad B[X \mapsto A']$$

$$\lambda X{:}A.\,B\,X \quad \to_\eta \quad B$$
(αν $X \notin \mathsf{FV}(B)$)

$$\mathsf{Elim}[A'](\mathsf{Constr}(I,B)\,\vec{A'}{:}B\,\vec{B})\{\underline{A}\} \quad \to_\iota$$
$$\Phi_{X,I}(C_i, A_i, \lambda\vec{X}{:}\vec{B'}.\,\lambda Y{:}B\,\vec{X}.\,\mathsf{Elim}[A'](Y{:}B\,\vec{X})\{\underline{A}\})\,\vec{A'}$$
(όπου $B = \mathsf{Ind}(X{:}B')\{\underline{C}\}$ και $B' = \Pi\vec{X}{:}\vec{B'}.\,\mathsf{Set}$)

Η συνάρτηση Φ ορίζεται ως:

$$\begin{array}{lcl} \Phi_{X,I}(X\,\vec{A'}, A, B) & \equiv & A \\ \Phi_{X,I}(\Pi X'{:}A'.\,B', A, B) & \equiv & \lambda X'{:}A'.\,\Phi_{X,I}(B', A\,X', B) \\ \Phi_{X,I}((\Pi\vec{X'}{:}\vec{A'}.\,X\,\vec{B'}) \to B', A, B) & \equiv & \lambda Y{:}(\Pi\vec{X'}{:}\vec{A'}.\,I\,\vec{B'}). \\ & & \quad \Phi_{X,I}(B', A\,Y\,(\lambda\vec{X'}{:}\vec{A'}.\,B\,\vec{B'}\,(Y\,\vec{X'})), B) \end{array}$$

Ορίζουμε τη σχέση $\twoheadrightarrow_\chi$ ως το ανακλαστικό και μεταβατικό κλείσιμο της σχέσης $\to_\chi$ για $\chi \in \{\beta, \eta, \iota\}$. Ακόμα $\twoheadrightarrow_{\beta\eta\iota}$ είναι η ένωση των $\to_\beta$, $\to_\eta$ και $\to_\iota$, ενώ $\twoheadrightarrow_{\beta\eta\iota}$ είναι το ανακλαστικό και μεταβατικό της κλείσιμο. Με τον συμβολισμό $\to$ και $\twoheadrightarrow$ εννοούμε τα $\to_{\beta\eta\iota}$ και $\twoheadrightarrow_{\beta\eta\iota}$ αντίστοιχα.

5.2.3 Ισότητα

Ορίζουμε την σχέση ισότητας $=_\chi$, η οποία προέρχεται από την σχέση $\rightarrow_\chi$, όπου $\chi \in \{\beta, \eta, \iota\}$, και τη σχέση $=$ ως την ένωση τριών προηγούμενων.

5.3 Κανόνες τύπων

Το πιο σημαντικό στοιχείο μιας γλώσσας τύπων (όπως άλλωστε υποδεικνύει και το όνομα της) είναι οι κανόνες τύπων. Αρχικά ορίζουμε ένα σύνολο από βοηθητικές συναρτήσεις, και στη συνέχεια αναφέρουμε τις προϋποθέσεις κάτω από τις οποίες κάθε όρος της γλώσσας είναι αποδεκτός.

5.3.1 Βοηθητικές συναρτήσεις

Πριν ορίσουμε τους κανόνες τύπων της γλώσσας, απαραίτητο είναι να οριστούν κάποιοι βοηθητικοί όροι.

- Θετικές εμφανίσεις (Positive Occurrences).

$$\frac{X \neq Y_i \quad X \notin \mathsf{FV}(A_i) \quad X \notin \mathsf{FV}(B_i) \quad \textit{(για κάθε i)}}{\mathsf{pos}_X(\Pi \vec{Y}\!:\!\vec{A}.\, X\, \vec{B})}$$

- Καλά ορισμένοι κατασκευαστές (Well Founded Constructors).

$$\frac{X \notin \mathsf{FV}(A_i) \quad \textit{(για κάθε i)}}{\mathsf{wfc}_X(X\, \vec{A})} \qquad \frac{X \neq X' \quad X \notin \mathsf{FV}(A') \quad \mathsf{wfc}_X(B)}{\mathsf{wfc}_X(\Pi X'\!:\!A'.\, B)}$$

$$\frac{\mathsf{pos}_X(A) \quad \mathsf{wfc}_X(B)}{\mathsf{wfc}_X(A \rightarrow B)}$$

- Μικρός τύπος κατασκευαστή (Small constructor type)

  Η συνάρτηση small επιστρέφει αληθή τιμή, αν ο όρος είναι ακολουθία από γινόμενα, κάθε μεταβλητή των οποίων έχει τύπο Set και το σώμα του γινομένου είναι μια ακολουθία από εφαρμογές, η οποία ξεκινάει από την ελεύθερη μεταβλητή που δώσαμε ως παράμετρο.

$$\frac{\Delta \vdash A'_i : \mathsf{Set} \quad \textit{(για κάθε i)}}{\Delta \vdash \mathsf{small}_X(\Pi \vec{X'}\!:\!\vec{A'}.\, X\, \vec{A})}$$

- Συνάρτηση ψ.

  Χρησιμοποιείται γαι τον έλεγχο της αποσύνθεσης επαγωγικά ορισμένων τύπων.

$$\begin{array}{lcl}
\Psi_{X,I}(X\, \vec{A'}, A, B) & \equiv & A\, \vec{A'}\, B \\
\Psi_{X,I}(\Pi X'\!:\!A'.\, B', A, B) & \equiv & \Pi X'\!:\!A'.\, \Psi_{X,I}(B', A, B\, X') \\
\Psi_{X,I}((\Pi \vec{X'}\!:\!\vec{A'}.\, X\, \vec{B'}) \rightarrow B', A, B) & \equiv & \Pi Y\!:\!(\Pi \vec{X'}\!:\!\vec{A'}.\, I\, \vec{B'}).\, \Pi \vec{X'}\!:\!\vec{A'}. \\
 & & \quad A\, \vec{B'}\, (Y\, \vec{X'}) \rightarrow \Psi_{X,I}(B', A, B\, Y)
\end{array}$$

- Πολλαπλές εκφράσεις

  Οι πολλαπλές εκφράσεις δεν αποτελούν κάποιο στοιχείο της γλώσσας, αλλά είναι μια χρήσιμη συντόμευση.

$$\frac{}{\Delta \vdash \epsilon : (\epsilon : \epsilon)} \qquad \frac{\Delta \vdash A : B \quad \Delta, X : B \vdash \underline{A} : (\underline{X} : \underline{B})}{\Delta \vdash A; \underline{A} : (X; \underline{X} : B; \underline{B})}$$

5.3.2 Μεταβλητές και σταθερές

$$\frac{(X : A) \in \Delta}{\Delta \vdash X : A} \qquad \frac{}{\Delta \vdash \mathsf{Set} : \mathsf{Type}} \qquad \frac{}{\Delta \vdash \mathsf{Type} : \mathsf{Ext}}$$

5.3.3 Ισοδυναμία τύπων

$$\frac{\Delta \vdash X : A \quad A = A'}{\Delta \vdash X : A'}$$

5.3.4 Γινόμενο, αφαίρεση και εφαρμογή

$$\frac{\Delta \vdash A : s_1 \quad \Delta, X : A \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Delta \vdash \Pi X{:}A.\, B : s_2}$$

$$\frac{\Delta, X : A \vdash B : B' \quad \Delta \vdash \Pi X{:}A.\, B' : s}{\Delta \vdash \lambda X{:}A.\, B : \Pi X{:}A.\, B'} \qquad \frac{\Delta \vdash A : \Pi X{:}B'.\, A' \quad \Delta \vdash B : B'}{\Delta \vdash A\, B : A'[X \mapsto B]}$$

$$\text{όπου} \quad \mathcal{R} = \left\{ \begin{array}{lll} (\mathsf{Set}, \mathsf{Set}), & (\mathsf{Type}, \mathsf{Set}), & (\mathsf{Ext}, \mathsf{Set}), \\ (\mathsf{Set}, \mathsf{Type}), & (\mathsf{Type}, \mathsf{Type}), & (\mathsf{Ext}, \mathsf{Type}), \\ (\mathsf{Set}, \mathsf{Ext}), & (\mathsf{Type}, \mathsf{Ext}) & \end{array} \right\}$$

5.3.5 Επαγωγικοί ορισμοί

$$\frac{\begin{array}{c} \Delta \vdash A : \mathsf{Type} \qquad A = \Pi \vec{X}{:}\vec{B}.\, \mathsf{Set} \\ \Delta, X : A \vdash A_i : \mathsf{Set} \quad \mathsf{wfc}_X(A_i) \quad (\textit{για κάθε } i) \end{array}}{\Delta \vdash \mathsf{Ind}(X{:}A)\{\underline{A}\} : A} \qquad \frac{A = \mathsf{Ind}(X{:}A')\{\underline{A}\} \quad \Delta \vdash A : A'}{\Delta \vdash \mathsf{Constr}(n, A) : A_n[X \mapsto A]}$$

$$\frac{\begin{array}{c} B = \mathsf{Ind}(X{:}B')\{\underline{A'}\} \qquad\qquad B' = \Pi \vec{X}{:}\vec{B'}.\, \mathsf{Set} \\ \Delta \vdash \vec{B} : (\vec{X} : \vec{B'}) \quad \Delta \vdash A : B\, \vec{B} \quad \Delta \vdash A' : \Pi \vec{X}{:}\vec{B'}.\, B\, \vec{X} \to \mathsf{Set} \\ \Delta \vdash A_i : \Psi_{X,B}(A'_i, A', \mathsf{Constr}(i, B)) \qquad (\textit{για κάθε } i) \end{array}}{\Delta \vdash \mathsf{Elim}[A'](A{:}B\, \vec{B})\{\underline{A}\} : A'\, \vec{B}\, A}$$

$$\frac{\begin{array}{c} B = \mathsf{Ind}(X{:}B')\{\underline{A'}\} \qquad\qquad B' = \Pi \vec{X}{:}\vec{B'}.\, \mathsf{Set} \\ \Delta \vdash \vec{B} : (\vec{X} : \vec{B'}) \quad \Delta \vdash A : B\, \vec{B} \quad \Delta \vdash A' : \Pi \vec{X}{:}\vec{B'}.\, B\, \vec{X} \to \mathsf{Type} \\ \Delta \vdash A_i : \Psi_{X,B}(A'_i, A', \mathsf{Constr}(i, B)) \quad \Delta, X : B' \vdash \mathsf{small}_X(A'_i) \quad (\textit{για κάθε } i) \end{array}}{\Delta \vdash \mathsf{Elim}[A'](A{:}B\, \vec{B})\{\underline{A}\} : A'\, \vec{B}\, A}$$

5.3.6 Παραγόμενες Θεωρίες

Με βάση την θεωρία και τους κανόνες που αναφέρθηκαν μπορούν να ορισθούν τα παρακάτω στοιχεία της γλώσσας τύπων, τα οποία αποτελούν μια βάση για την υλοποίηση των απαραίτητων τύπων που χρησιμοποιούνται στην γλώσσα υπολογισμών. Να σημειώσουμε ότι η γλώσσα τύπων υποστηρίζει τις δηλώσεις συνθετημάτων (modules), που ομαδοποιούν πολλές δηλώσεις σε ένα κοινό χώρο ονομάτων. Η υποστήριξη αυτή είναι καθαρά συντακτική, αφού δεν επηρεάζει τους κανόνες τύπων και τη σημασιολογία της γλώσσας. Εάν μία δήλωση για τη σταθερά B έχει γίνει μέσα στο συνθέτημα A τότε αναφερόμαστε σε αυτήν ως A%B. Συχνά παραλείπουμε το όνομα του συνθετήματος, όταν αυτό είναι προφανές.

Τιμές Αληθείας: Συνθέτημα Bool

$$\begin{array}{lcl} \mathsf{Bool} & : & \mathsf{Set} \\ & \equiv & \mathsf{Ind}(X:\mathsf{Set})\{X; X\} \\ \mathsf{true} & : & \mathsf{Bool} \\ & \equiv & \mathsf{Constr}(0, \mathsf{Bool}) \\ \mathsf{false} & : & \mathsf{Bool} \\ & \equiv & \mathsf{Constr}(1, \mathsf{Bool}) \end{array}$$

$$\begin{array}{lcl} \mathsf{bnot} & : & \mathsf{Bool} \to \mathsf{Bool} \\ \mathsf{band} & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\ \mathsf{bor} & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\ \mathsf{Cond} & : & \mathsf{Bool} \to \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \end{array}$$

Φυσικοί Αριθμοί: Συνθέτημα Nat

$$\begin{array}{lcl} \mathsf{Nat} & : & \mathsf{Set} \\ & \equiv & \mathsf{Ind}(X:\mathsf{Set})\{X, X \to X, X \to X\} \\ \mathsf{natZero} & : & \mathsf{Nat} \\ & \equiv & \mathsf{Constr}(0, \mathsf{Nat}) \\ \mathsf{natSucc} & : & \mathsf{Nat} \to \mathsf{Nat} \\ & \equiv & \mathsf{Constr}(1, \mathsf{Nat}) \end{array}$$

$$\begin{array}{lcl} \mathsf{natPlus}, \mathsf{natMinus}, \mathsf{natMult}, \mathsf{natDiv} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{bool\_le}, \mathsf{bool\_lt}, \mathsf{bool\_ge}, \mathsf{bool\_gt}, \mathsf{bool\_eq}, \mathsf{bool\_neq} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \\ \mathsf{LE}, \mathsf{LT}, \mathsf{GE}, \mathsf{GT} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set} \end{array}$$

Προτασιακός Λογισμός: Συνθέτημα Prop

$$\begin{array}{lcl} \mathsf{propTrue} & : & \mathsf{Set} \\ \mathsf{propFalse} & : & \mathsf{Set} \\ \mathsf{propTruePrf} & : & \mathsf{True} \\ \mathsf{propNot} & : & \mathsf{Set} \to \mathsf{Set} \\ \mathsf{propAnd} & : & \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \\ \mathsf{propOr} & : & \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \\ \mathsf{propEq} & : & \Pi \mathsf{a}:Set.\, \mathsf{a} \to \mathsf{a} \to \mathsf{Set} \end{array}$$

Λοιπά

$$\begin{array}{lcl} \mathsf{Unit} & : & \mathsf{Set} \\ & \equiv & \mathsf{Ind}(X:\mathsf{Set})\{X\} \\ \mathsf{unit} & : & \mathsf{Unit} \\ & \equiv & \mathsf{Constr}(0, \mathsf{Unit}) \end{array}$$

5.4 Η Γλώσσα Υπολογισμών

Η γλώσσα υπολογισμών είναι μια γλώσσα σχετικά υψηλού επιπέδου η οποία χρησιμοποιείται από το μεταγλωττιστή της Karooma ως ενδιάμεση. Σε αυτήν αναπαρίστανται όλες οι δομικές μονάδες και οι υπολογιστικές λειτουργίες της αρχικής γλώσσας. Τα προγράμματα που είναι εκφρασμένα στο NFLINT μπορούν να εκτελεστούν σε έναν μεταγλωττιστή (compiler) ή διερμηνέα (interpreter), αφού αφαιρεθεί η γλώσσα τύπων και παραμείνει μόνο η γλώσσα υπολογισμών.

Εδώ θα εξετάσουμε το υποσύνολο της γλώσσας υπολογισμών του NFLINT που είναι χρήσιμο για τη μετάφραση της Karooma. Το πλήρες σύστημα έχει επίσης υποστήριξη για τύπο πλειάδας, υπαρξιακούς και καθολικούς τύπους ως προς στοιχείο με τύπο Type (αντί μόνο για Set), και ακέραιους.

5.4.1 Τύποι της Γλώσσας Υπολογισμών

Κατ'αρχάς πρέπει να προσδιοριστούν στο συνθέτημα Repr οι αναπαραστάσεις στο επίπεδο των τύπων των στοιχείων του επιπέδου της γλώσσας υπολογισμού. Επειδή μόνον οι φυσικοί αριθμοί έχουν παραμετρική αναπαράσταση στο επίπεδο των τύπων, το συνθέτημα αυτό πρέπει να έχει την παρακάτω μορφή. Επίσης πρέπει να προστεθούν και οι αναπαραστάσεις των συναρτήσεων λογικών προτάσεων που έχουν οριστεί στο αρχείο των θεωρημάτων.

$$
\begin{array}{lcl}
\mathsf{Repr\%Nat} & : & \mathsf{Set} \\
\mathsf{Repr\%natZero} & : & \mathsf{Repr\%Nat} \\
\mathsf{Repr\%natSucc} & : & \mathsf{Repr\%Nat} \to \mathsf{Repr\%Nat} \\
\mathsf{Repr\%natPlus} & : & \mathsf{Repr\%Nat} \to \mathsf{Repr\%Nat} \to \mathsf{Repr\%Nat} \\
\vdots & & \\
\mathsf{Repr\%natLTbool} & : & \mathsf{Repr\%Nat} \to \mathsf{Repr\%Nat} \to \mathsf{Bool\%Bool} \\
\vdots & & \\
\mathsf{Repr\%natLTprop} & : & \mathsf{Repr\%Nat} \to \mathsf{Repr\%Nat} \to \mathsf{Set} \\
\vdots & &
\end{array}
$$

Το σύνολο Ω όλων των τύπων της γλώσσας υπολογισμών ορίζεται επαγωγικά στην γλώσσα τύπων ως εξής:

$$
\begin{array}{lcl}
\Omega & : & \mathsf{Set} \\
\mathsf{snat} & : & \mathsf{Repr\%Nat} \to \Omega \\
\mathsf{sbool} & : & \mathsf{Bool} \to \Omega \\
\mathsf{arrow} & : & \Omega \to \Omega \to \Omega \\
\mathsf{forall}_{\mathsf{Set}} & : & \Pi k\!:\!\mathsf{Set}.\,(k \to \Omega) \to \Omega \\
\mathsf{exists}_{\mathsf{Set}} & : & \Pi k\!:\!\mathsf{Set}.\,(k \to \Omega) \to \Omega
\end{array}
$$

Παρατηρούμε ότι για τους φυσικούς αριθμούς και τις δυαδικές τιμές χρησιμοποιούνται singleton τύποι. Ορίζουμε επίσης τις παρακάτω συντμήσεις για τους υπόλοιπους τύπους:

$$
\begin{array}{lcll}
\tau_1 \twoheadrightarrow \tau_2 & \equiv & \mathsf{arrow}\,\tau_1\,\tau_2 & \tau_1, \tau_2 : \Omega \\
\forall_{\mathsf{Set}}\, t\!:\!\kappa.\,\tau & \equiv & \mathsf{forall}_{\mathsf{Set}}\,\kappa\,(\lambda t\!:\!\kappa.\,\tau) & \kappa : \mathsf{Set}, \tau : \Omega \\
\exists_{\mathsf{Set}}\, t\!:\!\kappa.\,\tau & \equiv & \mathsf{exists}_{\mathsf{Set}}\,\kappa\,(\lambda t\!:\!\kappa.\,\tau) & \kappa : \mathsf{Set}, \tau : \Omega
\end{array}
$$

5.4.2 Όροι της Γλώσσας Υπολογισμών

Η αφηρημένη σύνταξη της γλώσσας υπολογισμών είναι η ακόλουθη:

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid b \mid op\, e \mid e_1\, op\, e_2 \\
 & \mid & \texttt{lambda}\; x\!:\!\tau.\ e \mid e_1\ e_2 \mid \texttt{poly}\; X\!:\!A.\ e \mid e\ [A] \mid \texttt{letrec}\; \vec{x}\!:\!\vec{\tau} = \vec{e_1}\; \texttt{in}\; e_2 \\
 & \mid & \texttt{pack}\,(X\!:\!B = A, e\!:\!\tau) \mid \texttt{unpack}\; e_1\; \texttt{as}\; (X\!:\!B, x\!:\!\tau)\; \texttt{in}\; e_2 \\
 & \mid & \texttt{if}\,[B, A]\,(e, X_1.\, e_1, X_2.\, e_2) \\
op & ::= & ! \mid + \mid - \mid * \mid / \mid == \mid != \mid < \mid > \mid <= \mid >= \mid \&\& \mid ||
\end{array}
$$

όπου x	:	μεταβλητή της γλώσσας υπολογισμών
X	:	μεταβλητή της γλώσσας τύπων
A, B	:	όροι της γλώσσας τύπων
n, i, c, b	:	κλειστοί όροι της γλώσσας τύπων
p	$\in$	$\mathbb{N}$

Πιο αναλυτικά οι εκφράσεις της γλώσσας είναι οι εξής

n : κατασκευάζει μια σταθερά φυσικού αριθμού, τύπου $\mathsf{snat}\hat{n}$, όπου $\hat{n}$ η αναπαράσταση στο επίπεδο των τύπων της σταθεράς

b : κατασκευάζει μια σταθερά bool, τύπου $\mathsf{sbool}\hat{b}$, όπου $\hat{b}$ η αντίστοιχη τιμή στο επίπεδο των τύπων (δηλαδή είτε Bool%true είτε Bool%false)

$op\ e$: εφαρμογή του τελεστή με ένα τελούμενο op στον όρο e.

$e_1\ op\ e_2$: εφαρμογή του δυαδικού τελεστή op στους όρους της υπολογιστικής γλώσσας e_1 και e_2.

`lambda` $x\!:\!\tau.\ e$: ανώνυμη συνάρτηση με μία παράμετρο x τύπου τ και σώμα e.

$e_1\ \ e_2$: εφαρμογή της συνάρτησης e_1 στην έκφραση e_2.

`poly` $X\!:\!A.\ e$: ανώνυμη πολυμορφική συνάρτηση με παράμετρο τύπου X η οποία έχει τύπο υψηλότερης τάξης A και με σώμα τον όρο e.

$e\ [A]$: εφαρμογή της πολυμορφικής συνάρτησης e πάνω στον τύπο της υπολογιστικής γλώσσας A.

`letrec` $\vec{x}\!:\!\vec{\tau} = \vec{e}$ `in` e' : επιτρέπει την ανάθεση συμβολικών ονομάτων x_i για τους όρους e_i, δηλώνοντας ότι αυτοί έχουν τύπο τ_i αντίστοιχα. Με τον τρόπο αυτό μπορεί μέσα στον όρο e_2 να γίνονται αναφορές στα x_i οι οποίες αντικαθίστανται με τον αντίστοιχο όρο e_i. Με τον τρόπο αυτό γίνεται δυνατή η κλήση συναρτήσεων βάσει συμβολικών ονομάτων, όπως και η υλοποίηση της αναδρομής.

`pack` $(X\!:\!B = A, e\!:\!\tau)$: δημιουργεί ένα υπαρξιακό "πακέτο" με ένα ζευγάρι μεταβλητής τύπου X και έκφρασης e. Οι τύποι τους είναι αντίστοιχα οι B και τ. Στην μεταβλητή X μπορεί να ανατεθεί η τιμή του όρου A, ίδιου τύπου. Έτσι οι αναφορές της X μέσα στο e δεσμεύονται στην τιμή αυτή.

`unpack` e_1 `as` $(X\!:\!B, x\!:\!\tau)$ `in` e_2 : ανοίγει ένα υπαρξιακό πακέτο e_1 που έχει δημιουργηθεί με την εντολή `pack` και δεσμεύει μέσα στον όρο e_2 την μεταβλητή X στην τιμή που έχει ο όρος τύπου στο πακέτο και την μεταβλητή x στον όρο που περιείχε το πακέτο. Με τον τρόπο αυτό, τόσο η μεταβλητή τύπου, όσο και ο όρος του πακέτου μπορούν να χρησιμοποιηθούν από τον όρο e_2.

`if` $[B, A]\,(e, X_1.\,e_1, X_2.\,e_2)$: δομή διακλάδωσης στην οποία e είναι η συνθήκη, e_1 ο όρος που επιλέγεται στην περίπτωση που η συνθήκη είναι αληθής και e_2 ο όρος που επιλέγεται στην περίπτωση που η συνθήκη είναι ψευδής. X_1 είναι η απόδειξη ότι το e είναι αληθές μέσα στην e_1 και X_2 είναι η απόδειξη ότι το e είναι ψευδές μέσα στο e_2. B είναι η συνάρτηση που επιστρέφει τον τύπο όλης της δομής συναρτήσει της τιμής της συνθήκης και A είναι μια απόδειξη για τον τύπο που επιστρέφει το B.

Οι τελεστές της γλώσσας υπολογισμού βρίσκονται σε αντιστοιχία με αυτούς της Karooma και περιγράφονται από τον κανόνα op.

5.4.3 Κανόνες Τύπων

Οι κανόνες τύπων στην υπολογιστική γλώσσα αναγράφονται με τον συμβολισμό $\Delta;\Gamma \vdash e \triangleright \tau$. Αυτό σημαίνει ότι υπό τις προϋποθέσεις που ορίζονται στα περιβάλλοντα Δ (της γλώσσας τύπων) και Γ (της γλώσσας υπολογισμών), ο όρος e της γλώσσας υπολογισμών έχει τύπο τ, ο οποίος ανήκει στο σύνολο Ω.

Μεταβλητές και Σταθερές

$$\frac{(x \triangleright \tau) \in \Gamma}{\Delta;\Gamma \vdash x \triangleright \tau} \qquad \frac{}{\Delta;\Gamma \vdash \texttt{true} \triangleright \mathsf{sbool}\,\mathsf{Bool\%true}}$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{false} \triangleright \mathsf{sbool}\,\mathsf{Bool\%false}}$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{n} \triangleright \mathsf{snat}\,(\underbrace{\mathsf{Repr\%natSucc}\cdots(\mathsf{Repr\%natSucc}}_{n \text{ φορές}}\ \mathsf{Repr\%natZero}))}$$

Μετατροπή

$$\frac{\Delta;\Gamma \vdash e \triangleright \tau \qquad \tau = \tau'}{\Delta;\Gamma \vdash e \triangleright \tau'}$$

Τελεστές

Αριθμητικοί Τελεστές

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 + e_2 \triangleright \mathsf{snat}\,(\mathsf{Repr\%natPlus}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 - e_2 \triangleright \mathsf{snat}\,(\mathsf{Repr\%natMinus}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 * e_2 \triangleright \mathsf{snat}\,(\mathsf{Repr\%natMult}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 \,/\, e_2 \triangleright \mathsf{snat}\,(\mathsf{Repr\%natDiv}\,n_1\,n_2)}$$

Σχεσιακοί Τελεστές

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 == e_2 \triangleright \mathsf{sbool}\,(\mathsf{Repr\%natEQbool}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 \,!= e_2 \triangleright \mathsf{sbool}\,(\mathsf{Repr\%natNEbool}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 < e_2 \triangleright \mathsf{sbool}\,(\mathsf{Repr\%natLTbool}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 > e_2 \triangleright \mathsf{sbool}\,(\mathsf{Repr\%natGTbool}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 <= e_2 \triangleright \mathsf{sbool}\,(\mathsf{Repr\%natLEbool}\,n_1\,n_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \triangleright \mathsf{snat}\,n_1 \qquad \Delta;\Gamma \vdash e_2 \triangleright \mathsf{snat}\,n_2}{\Delta;\Gamma \vdash e_1 >= e_2 \triangleright \mathsf{sbool}\,(\mathsf{Repr\%natGEbool}\,n_1\,n_2)}$$

Λογικοί Τελεστές

$$\frac{\Delta;\Gamma \vdash e_1 \rhd \mathsf{sbool}\, b_1 \qquad \Delta;\Gamma \vdash e_2 \rhd \mathsf{sbool}\, b_2}{\Delta;\Gamma \vdash e_1\, \mathsf{and}\, e_2 \rhd \mathsf{sbool}\,(\mathsf{Bool\%band}\, b_1\, b_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 \rhd \mathsf{sbool}\, b_1 \qquad \Delta;\Gamma \vdash e_2 \rhd \mathsf{sbool}\, b_2}{\Delta;\Gamma \vdash e_1\, \mathsf{and}\, e_2 \rhd \mathsf{sbool}\,(\mathsf{Bool\%bor}\, b_1\, b_2)}$$

$$\frac{\Delta;\Gamma \vdash e \rhd \mathsf{sbool}\, b}{\Delta;\Gamma \vdash \mathtt{not}\, e \rhd \mathsf{sbool}\,(\mathsf{Bool\%bnot}\, b)}$$

Συναρτήσεις

$$\frac{\Delta \vdash \tau : \Omega \qquad \Delta;\Gamma, x \rhd \tau \vdash e \rhd \tau'}{\Delta;\Gamma \vdash \mathtt{lambda}\, x\!:\!\tau.\ e \rhd \tau \twoheadrightarrow \tau'} \qquad \frac{\Delta;\Gamma \vdash e_1 \rhd \tau \twoheadrightarrow \tau' \qquad \Delta;\Gamma \vdash e_2 \rhd \tau}{\Delta;\Gamma \vdash e_1\ e_2 \rhd \tau'}$$

$$\frac{\Delta \vdash A : s \qquad \Delta, X : A;\Gamma \vdash e \rhd \tau \qquad s \in \{\mathsf{Set}, \mathsf{Type}\}}{\Delta;\Gamma \vdash \mathtt{poly}\ X\!:\!A.\ e \rhd \forall_s X\!:\!A.\, \tau}$$

$$\frac{\Delta;\Gamma \vdash e \rhd \mathsf{forall}_s\, A'\, \tau \qquad \Delta \vdash A : A'}{\Delta;\Gamma \vdash e\ [A] \rhd \tau\, A} \qquad \frac{\Delta;\Gamma, \vec{x} \rhd \vec{\tau} \vdash \vec{e_1} \rhd \vec{\tau} \qquad \Delta;\Gamma, \vec{x} \rhd \vec{\tau} \vdash e_2 \rhd \tau}{\Delta;\Gamma \vdash \mathtt{letrec}\ \vec{x}\!:\!\vec{\tau} = \vec{e_1}\ \mathtt{in}\ e_2 \rhd \tau}$$

Πακέτα

$$\frac{\Delta \vdash A : B \qquad \Delta \vdash B : s \qquad \Delta;\Gamma \vdash e \rhd \tau\{X \mapsto A\} \qquad s \in \{\mathsf{Set}, \mathsf{Type}\}}{\Delta;\Gamma \vdash \mathtt{pack}\,(X\!:\!B = A, e\!:\!\tau) \rhd \exists_s X\!:\!B.\, \tau}$$

$$\frac{\Delta;\Gamma \vdash e_1 \rhd \mathsf{exists}_s\, B\, \tau'' \qquad \Delta, X : B \vdash \tau : \Omega \qquad \Delta, X : B;\Gamma, x \rhd \tau''\, X \vdash e_2 \rhd \tau'}{\Delta;\Gamma \vdash \mathtt{unpack}\ e_1\ \mathtt{as}\ (X\!:\!B, x\!:\!\tau)\ \mathtt{in}\ e_2 \rhd \tau'}$$

Δομές Διακλάδωσης

$$\frac{\begin{array}{c}\Delta \vdash B : \mathsf{Bool} \to \mathsf{Set} \qquad \Delta;\Gamma \vdash e \rhd \mathsf{sbool}\, b \qquad \Delta \vdash A : B\, b \\ \Delta, X_1 : B\, \mathsf{true};\Gamma \vdash e_1 \rhd \tau \qquad \Delta, X_2 : B\, \mathsf{false};\Gamma \vdash e_2 \rhd \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{if}\,[B, A]\,(e, X_1.\, e_1, X_2.\, e_2) \rhd \tau}$$

5.4.4 Λειτουργική Σημασιολογία

Τιμές

Στις τιμές ανήκουν οι σταθερές φυσικών αριθμών και λογικών τιμών της γλώσσας υπολογισμού, οι συναρτήσεις ως προς όρο της γλώσσας υπολογισμού ή της γλώσσας τύπων, καθώς επίσης και οι τιμές που αποθηκεύονται σε πακέτα.

$$\begin{array}{rcl} v & ::= & n \mid b \mid \mathtt{lambda}\ x\!:\!\tau.\ e \mid \mathtt{poly}\ X\!:\!A.\ v \\ & \mid & \mathtt{pack}\,(X\!:\!B = A, v\!:\!\tau) \end{array}$$

Η Σχέση Υπολογισμού

Η κατάσταση που θα χρησιμοποιηθεί στη σχέση υπολογισμού αποτελείται μόνον από την έκφραση e υπό αποτίμηση. Ορίζουμε την σχέση υπολογισμού $\hookrightarrow$ σαν την μικρότερη υπολογιστικά συμβατή σχέση η οποία ικανοποιεί τις αναγωγές που δίνονται στις παρακάτω ενότητες. Σημειώνεται ότι η σχέση υπολογισμού αγνοεί εντελώς όλα τα στοιχεία της γλώσσας τύπων.

Συμβολίζουμε ως $[\![\diamond]\!] : v \to v$ τη σημασία του τελεστή ενός τελούμενου $\diamond$, και ως $[\![\circ]\!] : v \to v \to v$ τη σημασία του τελεστή δύο τελούμενων $\circ$.

$$
\begin{array}{lcl}
v_1 \circ v_2 & \hookrightarrow & [\![\circ]\!](v_1,\ v_2) \\
\diamond\, v & \hookrightarrow & [\![\diamond]\!](v) \\
(\texttt{lambda}\ x\!:\!\tau.\ e)\ \ v & \hookrightarrow & e\{x \mapsto v\} \\
(\texttt{poly}\ X\!:\!A.\ e)\ \ [B] & \hookrightarrow & e \\
\texttt{letrec}\ \vec{x}\!:\!\vec{\tau} = \vec{v}\ \texttt{in}\ e & \hookrightarrow & e\{\vec{x} \mapsto \vec{v}\{\vec{x} \mapsto \vec{w}\}\} \\
 & & \text{όπου}\ \ w_i = \texttt{letrec}\ \vec{x}\!:\!\vec{\tau} = \vec{v}\ \texttt{in}\ x_i \\
\texttt{unpack}\ \texttt{pack}\,(X\!:\!B = A, v\!:\!\tau)\ \texttt{as}\ (X'\!:\!B', y\!:\!\tau')\ \texttt{in}\ e & \hookrightarrow & e\{y \mapsto v\} \\
\texttt{if}\,[B,A]\,(\texttt{cbool}\,[b], X_1.\,e_1, X_2.\,e_2) & \hookrightarrow & e_1 \qquad \text{αν}\ \ b = \mathsf{true} \\
\texttt{if}\,[B,A]\,(\texttt{cbool}\,[b], X_1.\,e_1, X_2.\,e_2) & \hookrightarrow & e_2 \qquad \text{αν}\ \ b = \mathsf{false}
\end{array}
$$

Κεφάλαιο 6

Μετασχηματισμός της Karooma σε NFLINT

Η διαδικασία μετάφρασης ενός προγράμματος Karooma σε κώδικα για το NFLINT είναι η εξής. Αρχικά, συμπεριλαμβάνουμε τον κώδικα του αρχείου που ορίζεται από την δήλωση της αναπαράστασης (representation), στον οποίο πρέπει να περιλαμβάνονται δηλώσεις στο επίπεδο τύπων, όπως ο επαγωγικός ορισμός των ακεραίων, το συνθέτημα της αναπαράστασης που αντιστοιχεί τους βασικούς τύπους με τον τύπο της αναπαράστασής τους στο επίπεδο των τύπων, καθώς και την αναπαράσταση των στοιχειωδών τους πράξεων, καθώς και τον επαγωγικό ορισμό του συνόλου Ω, δηλαδή των τύπων της γλώσσας υπολογισμού. Έπειτα μεταφράζονται οι ορισμοί συναρτήσεων σε αντίστοιχους όρους της γλώσσας υπολογισμού του NFLINT, καθώς και το αποτέλεσμα, ως μία συνάρτηση που δεν δέχεται κανένα όρισμα. Το συνολικό πρόγραμμα είναι μία δομή letrec, όπου γίνεται αμοιβαία αναδρομική δήλωση των συναρτήσεων, με σώμα την έκφραση που έχει προκύψει από τη μετάφραση του αποτελέσματος.

Στα παρακάτω θα δώσουμε μία αυστηρή περιγραφή του πώς γίνεται η μετάφραση μίας συνάρτησης, ορίζοντας ένα σύνολο ξεχωριστών μετασχηματισμών που θα χρησιμοποιηθούν. Ένα σημαντικό μέρος της διαδικασίας μετάφρασης, ο μηχανισμός κατασκευής αποδείξεων, θα παρουσιαστεί σε ξεχωριστό υποκεφάλαιο, λόγω της αυξημένης πολυπλοκότητάς του.

6.1 Μετάφραση συνάρτησης

Κατ'αρχάς να σημειώσουμε ότι στη διαδικασία μετάφρασης χρησιμοποιούνται οι μετασχηματισμένες εκφράσεις που ορίσαμε στο κεφάλαιο 4.1.2 και όχι οι αρχικές εκφράσεις.

Θα χρειαστούμε τις εξής επιμέρους διαδικασίες:

- μετάφραση τύπου της Karooma σε κατασκευαστή τύπου από το Ω (συμβολισμός $ctor(\tau)$)
- μετάφραση τύπου της Karooma στον τύπο της αναπαράστασης στο επίπεδο τύπων των στοιχείων του (συμβολισμός $rset(\tau)$)
- μετασχηματισμός απλής έκφρασης στην αναπαράστασή της στο επίπεδο των τύπων του NFLINT (συμβολισμός $repr(se)$)
- μετασχηματισμός λογικής πρότασης σε όρο του επιπέδου τύπων του NFLINT (συμβολισμός $prop(p)$)
- διαδικασία κατασκευής απόδειξης στο επίπεδο τύπων του NFLINT για συγκεκριμένη λογική πρόταση (συμβολισμός $proof(p)$), η οποία θα περιγραφεί στο επόμενο υποκεφάλαιο
- μετασχηματισμός έκφρασης σε όρο της γλώσσας υπολογισμού του NFLINT (συμβολισμός $expr(te)$)
- μετατροπή όρου της γλώσσας υπολογισμού σε όρο κατάλληλο για επιστροφή από συνάρτηση (συμβολισμός $rtrn(te)$)
- τελική μετάφραση συνάρτησης με χρήση των διαδικασιών αυτών (συμβολισμός $func(f)$)

$$\begin{array}{rcl} ctor(\mathbf{nat}) & = & \mathsf{snat} \\ ctor(\mathbf{bool}) & = & \mathsf{sbool} \\ rset(\mathbf{nat}) & = & \mathsf{Repr.Nat} \\ rset(\mathbf{bool}) & = & \mathsf{Repr.Bool} \end{array}$$

Σχήμα 6.1: Οι διαδικασίες ctor και rset.

Οι διαδικασίες αυτές χρειάζονται, όπως είναι φυσικό, και πληροφορίες για τις διαθέσιμες συναρτήσεις (το περιβάλλον Ψ που είδαμε στο κεφάλαιο 4.1), τις συναρτήσεις των λογικών προτάσεων (το περιβάλλον $P\Phi$) καθώς και τις διαθέσιμες αποδείξεις σε κάθε σημείο. Προς απλοποίηση της μορφής των ορισμών, τα περιβάλλοντα αυτά θεωρούμε ότι είναι διαθέσιμα στις διαδικασίες χωρίς να αποτελούν παράμετρό τους, ενώ σημειώνουμε πού γίνεται χρήση ή ενημέρωση αυτών.

Στα σχήματα 6.1 - 6.6 γίνεται ο ορισμός των διαδικασιών αυτών. Από αυτές χρήζουν σχολιασμού οι prop, expr και rtrn. Στην prop, γίνεται χρήση μίας βοηθητικής διαδικασίας, της rprp, η οποία επιστρέφει την αναπαράσταση στο επίπεδο τύπων των υποεκφράσεων μίας λογικής πρότασης που είναι τύπου διαφορετικού του **prop** (άρα τύπου **nat** ή **bool**). Οι υποεκφράσεις αυτές, είτε είναι κλήσεις συναρτήσεων των λογικών προτάσεων, οπότε η αναπαράστασή τους στο επίπεδο των τύπων είναι η εφαρμογή της αναπαράστασης των παραμέτρων στον όρο που αναπαριστά τη συνάρτηση, είτε έχουν μορφή παρόμοια με τις απλές εκφράσεις. Σε αυτή την περίπτωση η διαδικασία rprp έχει παρόμοιο αποτέλεσμα με την repr, και γι'αυτό δεν επαναλαμβάνουμε τον πλήρη ορισμό.

Στην expr έχει ενδιαφέρον η μετάφραση του if και η μετάφραση της δέσμευσης μεταβλητής για κλήση συνάρτησης. Όπως έχουμε αναφέρει στο προηγούμενο κεφάλαιο, η δομή if της γλώσσας υπολογισμού του FLINT δέχεται ως παραμέτρους μία συνάρτηση στο επίπεδο τύπων που δεδομένης μίας δυαδικής τιμής επιστρέφει τον τύπο της απόδειξης που θα είναι διαθέσιμη στο αντίστοιχο παρακλάδι, καθώς και έναν όρο με τύπο τη συνάρτηση, εφαρμοσμένη πάνω στην αναπαράσταση της συνθήκης της δομής. Ο δεύτερος αυτός όρος κατασκευάζεται από το μηχανισμό αυτόματης εύρεσης απόδειξης, ζητώντας απόδειξη για λογική πρόταση που χρησιμοποιεί την ειδική συνάρτηση Cond. Σε κάθε παρακλάδι, γίνεται διαθέσιμη η απόδειξη της κατάλληλης λογικής πρότασης, προς χρήση από τον μηχανισμό αυτόματης εύρεσης απόδειξης.

Η κλήση συνάρτησης μεταφράζεται ως εξής. Κατ'αρχάς εφαρμόζονται πάνω στον όρο της συνάρτησης οι αναπαραστάσεις στο επίπεδο τύπων των παραμέτρων, οι παράμετροι, καθώς και αποδείξεις για την ισχύ των προ-συνθηκών της συνάρτησης. Το αποτέλεσμα της συνάρτησης είναι ένα υπαρξιακό, που περιέχει την αναπαράσταση του αποτελέσματος, τις αποδείξεις των μετασυνθηκών, και το αποτέλεσμα αυτό καθεαυτό. Το υπαρξιακό "ανοίγεται" πλήρως, ώστε τα επιμέρους στοιχεία να είναι διαθέσιμα για την μετάφραση της έκφρασης στην οποία δεσμεύεται η κλήση συνάρτησης.

Η διαδικασία rtrn χρησιμοποιείται ώστε στα σημεία όπου γίνεται επιστροφή αποτελέσματος από τη συνάρτηση, να δημιουργηθούν τα κατάλληλα υπαρξιακά. Έτσι το αποτέλεσμα συνδυάζεται με την αναπαράστασή του, καθώς και με τις αποδείξεις των μετασυνθηκών της συνάρτησης.

6.2 Μηχανισμός αυτόματης κατασκευής αποδείξεων

Στο σημείο εδώ θα περιγράψουμε το πώς λειτουργεί ο μηχανισμός αυτόματης κατασκευής αποδείξεων που αποτελεί μέρος του μετασχηματισμού ενός προγράμματος Karooma σε κώδικα NFLINT. Επειδή η παρούσα έκδοση αυτού του μηχανισμού επιδέχεται βελτιώσεις και δεν θεωρείται τελική, δεν θα δώσουμε αυστηρή περιγραφή αυτού, όπως έγινε για τις υπόλοιπες διαδικασίες που χρησιμοποιούνται κατά τον μετασχηματισμό.

$$
\begin{array}{rcl}
repr(id) & = & \textsf{id} \\
repr(n) & = & \underbrace{\textsf{Repr.natSucc}(\cdots(\textsf{Repr.natSucc}}_{\text{n φορές}}\ \textsf{Repr.natZero})) \\
repr(\mathbf{true}) & = & \textsf{Bool.true} \\
repr(\mathbf{false}) & = & \textsf{Bool.false} \\
repr(\mathbf{binop}(+, se_1, se_2)) & = & \textsf{Repr.natPlus}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(-, se_1, se_2)) & = & \textsf{Repr.natMinus}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(*, se_1, se_2)) & = & \textsf{Repr.natTimes}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(/, se_1, se_2)) & = & \textsf{Repr.natDiv}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(\leq, se_1, se_2)) & = & \textsf{Repr.natLEbool}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(\geq, se_1, se_2)) & = & \textsf{Repr.natGEbool}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(<, se_1, se_2)) & = & \textsf{Repr.natLTbool}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(>, se_1, se_2)) & = & \textsf{Repr.natGTbool}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(==, se_1, se_2)) & = & \textsf{Repr.natEQbool}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(!=, se_1, se_2)) & = & \textsf{Repr.natNEbool}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(\&\&, se_1, se_2)) & = & \textsf{Bool.band}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{binop}(||, se_1, se_2)) & = & \textsf{Bool.bor}\ repr(se_1)\ repr(se_2) \\
repr(\mathbf{unop}(!, se)) & = & \textsf{Bool.bnot}\ repr(se)
\end{array}
$$

Σχήμα 6.2: Η διαδικασία repr.

$$
\begin{array}{rcl}
prpr(id(p_1,\ p_2,\ \cdots,\ p_n)) & = & \textsf{Repr.(id)}\ prpr(p_1)\ \cdots\ prpr(p_n) \\
prpr(\cdots) & \cong & repr(\cdots) \\
prop(id) & = & \textsf{Prop.propEq Bool.Bool id Bool.true} \\
prop(\mathbf{true}) & = & \textsf{Prop.propTrue} \\
prop(\mathbf{false}) & = & \textsf{Prop.propFalse} \\
prop(\mathbf{binop}(\leq, p_1, p_2)) & = & \textsf{Repr.natLEprop}\ prpr(p_1)\ prpr(p_2) \\
prop(\mathbf{binop}(\geq, p_1, p_2)) & = & \textsf{Repr.natGEprop}\ prpr(p_1)\ prpr(p_2) \\
prop(\mathbf{binop}(<, p_1, p_2)) & = & \textsf{Repr.natLTprop}\ prpr(p_1)\ prpr(p_2) \\
prop(\mathbf{binop}(>, p_1, p_2)) & = & \textsf{Repr.natGTprop}\ prpr(p_1)\ prpr(p_2) \\
prop(\mathbf{binop}(==, p_1, p_2)) & = & \textsf{Repr.natEQprop}\ prpr(p_1)\ prpr(p_2) \\
prop(\mathbf{binop}(!=, p_1, p_2)) & = & \textsf{Repr.natNEprop}\ prpr(p_1)\ prpr(p_2) \\
prop(\mathbf{binop}(\&\&, p_1, p_2)) & = & \textsf{Prop.propAnd}\ prop(p_1)\ prop(p_2) \\
prop(\mathbf{binop}(||, p_1, p_2)) & = & \textsf{Prop.propOr}\ prop(p_1)\ prop(p_2) \\
prop(\mathbf{unop}(!, p)) & = & \textsf{Prop.propNot}\ prop(p)
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{εάν } id : \tau p_1,\ \tau p_2,\ \cdots,\ \tau p_n \to \mathbf{prop} :} \\
prop(id(p_1,\ p_2,\ \cdots,\ p_n)) & = & \textsf{Repr.(id)}\ t_1\ \cdots\ t_n \\
\multicolumn{3}{l}{\text{εάν } id : \tau p_1,\ \tau p_2,\ \cdots,\ \tau p_n \to \mathbf{bool} :} \\
prop(id(p_1,\ p_2,\ \cdots,\ p_n)) & = & \textsf{Prop.propEq Bool.Bool (Repr.(id)}\ t_1\ \cdots\ t_n\textsf{) Bool.true}
\end{array}
$$

$$\textit{όπου } t_i = prop(p_i) \textit{ εάν } \tau p_i = \mathbf{prop} \textit{ ή } t_i = prpr(p_i) \textit{ διαφορετικά}$$

Σχήμα 6.3: Η διαδικασία prop.

$$
\begin{aligned}
expr(id) &= \texttt{id}\\
expr(n) &= \texttt{n}\\
expr(b) &= \texttt{b}\\
expr(\textbf{binop}(op,\, se_1,\, se_2)) &= expr(se_1)\ (\mathsf{op})\ expr(se_2)\\
\\
expr(\textbf{unop}(op,\, se)) &= (\mathsf{op})\ expr(se)\\
expr(\textbf{if}(se_1,\, te_2,\, te_3)) &= \texttt{if}\,[\lambda \mathsf{b}{:}\mathsf{Bool}.\ \mathsf{Cond\ b}\ prop(se_1)\ prop(negate(se_1)),\\
&\qquad proof(Cond(se_1, se_1, negate(se_1)))]\,(expr(se_1),\\
&\qquad X_{true}.\,rtrn(te_2), X_{false}.\,rtrn(te_3))
\end{aligned}
$$

(η απόδειξη X_{true} του se_1 είναι διαθέσιμη στη μετάφραση του te_2,
και η απόδειξη X_{false} του $negate(se_1)$ είναι διαθέσιμη στη μετάφραση του te_3)

εάν $id_{func} : (\overrightarrow{id_{params}} : \overrightarrow{\tau_{params}})[\overrightarrow{p^{pre}}] \rightarrow id_{ret} : \tau_{ret}[\overrightarrow{p^{post}}]$ τότε:

$expr(\textbf{bind}\ (id, id_{func}(\overrightarrow{se}))\ \textbf{in}\ te') =$

$\texttt{unpack}\ (\texttt{t} : \exists \texttt{id}_{\texttt{ret}} : \mathrm{rset}(\tau_{\texttt{ret}}).\exists \texttt{prf}_1 : \texttt{p}_1^*. \cdots \exists \texttt{prf}_\texttt{m} : \texttt{p}_\texttt{m}^*.\mathrm{ctor}(\tau_{\texttt{ret}})\ \texttt{id}_{\texttt{ret}})\ \texttt{as}$

$(\texttt{id} : \mathrm{rset}(\tau_{\texttt{ret}}), \texttt{imm}_0 : \exists \texttt{prf}_1 : \texttt{p}_1^*. \cdots \exists \texttt{prf}_\texttt{m} : \texttt{p}_\texttt{m}^*.\mathrm{ctor}(\tau_{\texttt{ret}})\ \texttt{id})\ in$

$\texttt{unpack imm}_0\ \texttt{as}\ (\texttt{prf}_1 : \texttt{p}_1^*, \texttt{imm}_1 : \exists \texttt{prf}_2 : \texttt{p}_2^*. \cdots \exists \texttt{prf}_\texttt{m} : \texttt{p}_\texttt{m}^*.\mathrm{ctor}(\tau_{\texttt{ret}})\ \texttt{id})\ \texttt{in}$

$\vdots$

$\texttt{unpack imm}_{\texttt{m}-1}\ \texttt{as}\ (\texttt{prf}_\texttt{m} : \texttt{p}_\texttt{m}^*, \texttt{id} : \mathrm{ctor}(\tau_{\texttt{ret}})\ \texttt{id})\ \texttt{in}\ rtrn(te')$

$\texttt{t}\ =\ \texttt{id}_{\texttt{func}}\ repr(se_1) \cdots repr(se_n)\ proof(p'_1) \cdots proof(p'_r)\ expr(se_1) \cdots expr(se_n)$

$p'_i\ =\ p_i^{pre}[\overrightarrow{id_{params}} := \overrightarrow{se}]$

$p^*_i\ =\ prop(p_i^{post}[\overrightarrow{id_{params}} := \overrightarrow{se}, id_{ret} := id])$

(οι αποδείξεις prf_1 έως prf_m των λογικών προτάσεων $\overrightarrow{p^{post}}[\overrightarrow{id_{params}} := \overrightarrow{se}, id_{ret} := id]$
είναι διαθέσιμες στην μετάφραση του te')

Σχήμα 6.4: Η διαδικασία expr.

$rtrn(\textbf{if}(se_1,\, te_2,\, te_3)) = expr(\textbf{if}(se_1,\, te_2,\, te_3))$

$rtrn(\textbf{bind}\ be\ \textbf{in}\ te') = expr(\textbf{bind}\ be\ \textbf{in}\ te')$

εάν βρισκόμαστε μέσα στη συνάρτηση
$id_{func} : (\overrightarrow{id_{params}} : \overrightarrow{\tau_{params}})[\overrightarrow{p^{pre}}] \rightarrow id_{ret} : \tau_{ret}[p_1,\ p_2,\ \cdots,\ p_n]$ και
$p'_i = p_i[id_{ret} := repr(se)]$ τότε:

$$
\begin{aligned}
rtrn(se) = \texttt{pack}\ (&\texttt{id}_{\texttt{ret}} : rset(\tau_{ret})\ =\ repr(se),\\
&\texttt{pack}(\texttt{prf}_1 : prop(p'_1)\ =\ proof(p'_1),\\
&\vdots\\
&\texttt{pack}(\texttt{prf}_\texttt{n} : prop(p'_n)\ =\ proof(p'_n),\\
&\quad expr(se) : ctor(\tau_{ret})\ repr(se))\ :\ \exists \mathsf{prf_n} :\ prop(p'_1).ctor(\tau_{ret})\ repr(se))\\
&\quad : \exists \mathsf{prf_{n-1}} :\ prop(p'_{n-1}).\exists \mathsf{prf_n} :\ prop(p'_1).ctor(\tau_{ret})\ repr(se))\\
&\quad \vdots\\
&: \exists \mathsf{prf_1} : prop(p_1). \cdots \exists \mathsf{prf_n} : prop(p_n).ctor(\tau_{ret})\ \mathsf{id_{ret}})
\end{aligned}
$$

Σχήμα 6.5: Η διαδικασία rtrn.

$func(id_{func} : (\overrightarrow{id_{params}} : \overrightarrow{\tau_{params}})[\overrightarrow{p^{pre}}] \rightarrow id_{ret} : \tau_{ret}[\overrightarrow{p^{post}}] := e_{func}) =$
$\texttt{poly}\ \mathsf{id_{params,1}} :\ rset(\tau_{params,1}). \cdots \texttt{poly}\ \mathsf{id_{params,n}} :\ rset(\tau_{params,n}).$
$\texttt{poly}\ \mathsf{pprf_1} :\ prop(p_1^{pre}). \cdots \texttt{poly}\ \mathsf{pprf_m} :\ prop(p_m^{pre}).$
$\texttt{lambda}\ \mathsf{id_{params,1}} :\ ctor(\tau_{params,1})\ \mathsf{id_{params,1}}. \cdots \texttt{lambda}\ \mathsf{id_{params,n}} :\ ctor(\tau_{params,n})\ \mathsf{id_{params,n}}.$
$rtrn(e_{func})$

(στη μετάφραση της e_{func} είναι διαθέσιμες οι αποδείξεις $pprf_1,\ pprf_2,\ \cdots,\ pprf_m$ των προτάσεων $\overrightarrow{p^{pre}}$)

Σχήμα 6.6: Η διαδικασία func.

Ο μηχανισμός αυτός χρησιμοποιεί τις διαθέσιμες αποδείξεις και τα θεωρήματα που περιγράφονται στο αντίστοιχο αρχείο προς κατασκευή των αποδείξεων. Οι διαθέσιμες αποδείξεις αποτελούν δυάδες ενός όρου NFLINT και της λογικής πρότασης για την οποία ο όρος αυτός είναι απόδειξη. Τα θεωρήματα αποτελούνται από έναν όρο σε NFLINT, ένα σύνολο από ελεύθερες μεταβλητές, μία λογική πρόταση-στόχο και πιθανώς και λογικές προτάσεις-υποθέσεις. Ο όρος NFLINT που αντιστοιχεί σε κάθε θεώρημα πρέπει να έχει τον κατάλληλο τύπο, να είναι δηλαδή συνάρτηση ως προς τις αναπαραστάσεις των τύπων των μεταβλητών και τις λογικές προτάσεις-υποθέσεις, προς τη λογική πρόταση-στόχο, ώστε να αποτελεί απόδειξη του θεωρήματος. Μπορούμε να κατασκευάσουμε μία απόδειξη της λογικής πρότασης-στόχου εάν βρεθεί κατάλληλη αντικατάσταση για τις μεταβλητές του θεωρήματος, και αποδείξεις για τις υποθέσεις, εφαρμόζοντας στον όρο NFLINT του θεωρήματος τις αναπαραστάσεις των μεταβλητών και τους όρους που αποδεικνύουν τις υποθέσεις.

Για να βρεθεί απόδειξη για μία λογική πρόταση, αρχικά αναζητείται η ίδια λογική πρόταση μέσα στο σύνολο των διαθέσιμων αποδείξεων. Εάν δεν βρεθεί, τότε ξεκινά μία διαδικασία αναζήτησης απόδειξης με χρήση των θεωρημάτων. Ελέγχεται, για κάθε θεώρημα, κατά πόσον η προς απόδειξη πρόταση μπορεί να ενοποιηθεί με τη πρόταση-στόχο του εκάστοτε θεωρήματος. Δύο προτάσεις είναι ενοποιήσιμες, όταν έχουν την ίδια δομή μετά από αντικατάσταση των αντίστοιχων υποεκφράσεων της μίας στη θέση των ελεύθερων μεταβλητών της άλλης. Συγκεκριμένα, η πρόταση-στόχος έχει (πιθανώς) ελεύθερες μεταβλητές συγκεκριμένου τύπου. Αν κατά τη δομική σύγκρισή της με την πρόταση προς απόδειξη προκύψει η περίπτωση στη θέση μίας υποέκφρασης κάποιου τύπου της πρότασης προς απόδειξη να βρίσκεται μία ελεύθερη μεταβλητή του ίδιου τύπου της πρότασης-στόχο, η τιμή αυτής προσδιορίζεται ως ίση με την υποέκφραση. Η διαδικασία αυτή επαναλαμβάνεται, και στην περίπτωση που δεν προκύψει κάποια σύγκρουση στη δομή των δύο προτάσεων, οι δύο προτάσεις είναι ενοποιήσιμες.

Σε περίπτωση που η πρόταση προς απόδειξη ενοποιείται με την πρόταση-στόχο ενός θεωρήματος, γίνονται οι αντικαταστάσεις μεταβλητών που έχουν προκύψει κατά την ενοποίηση και στις προτάσεις-υποθέσεις. Στο σημείο αυτό, οι υποθέσεις μπορεί να έχουν και ορισμένες ελεύθερες μεταβλητές οι τιμές των οποίων δεν έχουν προσδιοριστεί· δεν είναι απαραίτητο ότι όλες οι μεταβλητές ενός θεωρήματος εμφανίζονται στην πρόταση-στόχο, κάποιες μπορεί να εμφανίζονται μόνον στις υποθέσεις. Ιδανικά θα θέλαμε να εκτελέσουμε αναδρομικά τη διαδικασία εύρεσης απόδειξης για τις προτάσεις-υποθέσεις, αυτό όμως δεν μπορεί να γίνει στη γενική περίπτωση, αφού η διαδικασία αυτή στην παρούσα έκδοση απαιτεί η λογική πρόταση προς απόδειξη να μην έχει ελεύθερες μεταβλητές. Έτσι έχουμε μία διαδικασία που προσπαθεί να ανιχνεύσει μία κατάλληλη αντικατάσταση για τις μεταβλητές των προτάσεων-υποθέσεων που δεν έχουν προσδιοριστεί, δοκιμάζοντας ενοποίηση μόνον με τις διαθέσιμες αποδείξεις και όχι με πλήρη αναζήτηση απόδειξης και μέσω των θεωρημάτων. Η υλοποίηση της διαδικασίας εύρεσης αποδείξεων έτσι απλοποιείται αρκετά, σε σχέση με την περίπτωση που επιτρέπονταν αδέσμευτες μεταβλητές για τις προς απόδειξη προτάσεις. Η αρνητική συνέπεια της επιλογής αυτής είναι ότι ενώ κάποιες λογικές προτάσεις μπορούν θεωρητικά να αποδειχθούν από τις διαθέσιμες αποδείξεις και τα θεωρήματα, πρακτικά δεν μπορεί να βρεθεί μία τέτοια απόδειξη, και έτσι χρειάζονται περαιτέρω θεωρήματα για την απόδειξή τους.

Ο μηχανισμός εύρεσης αποδείξεων μπορεί να χαρακτηριστεί ως back-chaining και αναζήτησης πρώτα σε βάθος. Χαρακτηρίζεται ως back-chaining διότι ξεκινάμε από τον στόχο που θέλουμε να αποδείξουμε και αναζητούμε πώς μπορεί να προκύψει αυτός "προς τα πάνω", σε αντίθεση με την περίπτωση που θα ξεκινούσαμε από τα διαθέσιμα θεωρήματα και τις διαθέσιμες αποδείξεις, και θα κατασκευάζαμε αποδείξεις που προκύπτουν από αυτά, μέχρι να βρεθεί μία απόδειξη της πρότασης-στόχου. Επίσης αναζητεί πρώτα σε βάθος, διότι αν βρεθεί ένα θεώρημα, ο στόχος του οποίου ενοποιείται με την εκάστοτε πρόταση προς απόδειξη, γίνεται πλήρης αναζήτηση του κατά πόσον μπορούν να αποδειχθούν οι προτάσεις-υποθέσεις, πριν δοκιμαστεί άλλο θεώρημα.

Μπορούμε να πούμε ότι ο μηχανισμός εύρεσης απόδειξης είναι ανάλογος του μηχανισμού που χρησιμοποιείται προς επίλυση ερωτημάτων ενός προγράμματος στη λογική γλώσσα Prolog. Οι προτάσεις για τις οποίες έχουμε διαθέσιμες αποδείξεις είναι κανόνες Prolog χωρίς μεταβλητές και χωρίς δεξί μέρος, ενώ τα θεωρήματα αντιστοιχούν σε κανόνες Prolog με στόχο την πρόταση-στόχο, και προαπαιτούμενα τις προτάσεις-υποθέσεις. Η διαδικασία εύρεσης απόδειξης χρησιμοποιεί μάλιστα την έννοια της ενοποίησης, όπως χρησιμοποιείται και στην Prolog. Ιδανικά, ο μηχανισμός εύρεσης απόδειξης θα λειτουργούσε όπως μία πλήρης μηχανή Prolog, ώστε να μπορούν να βρεθούν αποδείξεις για όλες τις προτάσεις που είναι θεωρητικά αποδείξιμες.

Μία εναλλακτική δυνατότητα που θα είχε ενδιαφέρον, όσον αφορά την απόδειξη των λογικών προτάσεων, θα ήταν να καταργηθεί το αρχείο των θεωρημάτων, και σε συνδυασμό με έναν βοηθό αποδείξεων για το σύστημα NFLINT, για παράδειγμα παρόμοιο με το πρόγραμμα Coq, οι αποδείξεις να προκύπτουν αυτόματα, ή με αλληλεπίδραση με το χρήστη, από μία δεδομένη βιβλιοθήκη αποδείξεων σε NFLINT. Έτσι κατά τη μετάφραση ενός προγράμματος, θα προέκυπταν προτάσεις-θεωρήματα που πρέπει να αποδειχτούν, είτε μέσω τακτικών αυτόματης εύρεσης αποδείξεων, είτε μέσω απ'ευθείας απόδειξης από τον χρήστη. Η προσέγγιση αυτή θα είχε το πλεονέκτημα ότι ο μηχανισμός αυτόματης εύρεσης αποδείξεων δεν θα αποτελούσε μέρος του μεταγλωττιστή της Karooma, αλλά μέρος του NFLINT και θα ήταν περισσότερο επεκτάσιμος. Επίσης δεν θα χρειαζόταν, όταν προσθέτουμε ένα νέο θεώρημα στη βιβλιοθήκη αποδείξεων του NFLINT, να ενημερώνουμε και το αρχείο θεωρημάτων με την περιγραφή του θεωρήματος αυτού.

Κεφάλαιο 7

Συμπεράσματα

7.1 Συνεισφορά

Η συνεισφορά της εργασίας μπορεί να συνοψιστεί ως εξής.

- Προτάθηκε μία νέα γλώσσα προγραμματισμού με αποδείξεις, η Karooma, που χρησιμοποιεί την ιδέα των προ- και μετα-συνθηκών για την προδιαγραφή της ορθότητας ενός προγράμματος. Η σχεδίασή της επικεντρώθηκε στην ευκολία χρήσης από προγραμματιστές και αυτό είναι το ισχυρό της σημείο σε σχέση με υπάρχουσες παρόμοιες γλώσσες. Ωστόσο, υστερεί ως προς την εκφραστικότητα του συστήματος τύπων.

- Επιτεύχθηκε πλήρης διαχωρισμός της φάσης προγραμματισμού ενός προγράμματος στη γλώσσα αυτή και της φάσης απόδειξης της μερικής ορθότητάς του. Σε συνδυασμό με την υποστήριξη παραμετρικών αναπαραστάσεων, ένας προγραμματιστής μπορεί να προγραμματίζει στην Karooma χωρίς να ασχολείται με την απόδειξη της μερικής ορθότητας, όπως θα έκανε σε μία οποιαδήποτε γλώσσα συναρτησιακού προγραμματισμού, και ένας γνώστης μαθηματικής λογικής μπορεί έπειτα να παρέχει την αυστηρή απόδειξη.

- Υλοποιήθηκε ένας μεταγλωττιστής αυτής της γλώσσας που παράγει κώδικα για το σύστημα NFLINT. Πριν την εκτέλεση του προκύπτοντα κώδικα, μπορεί να επιβεβαιωθεί μέσω αυτού του συστήματος η μερική ορθότητα του κώδικα, ενώ κατά το χρόνο εκτέλεσης δεν γίνεται κάποιος σχετικός έλεγχος, με αποτέλεσμα να μην έχουμε κόστος στην απόδοση του τελικού εκτελέσιμου.

- Υλοποιήθηκαν στη γλώσσα Karooma ένα σύνολο από προγράμματα για συναρτήσεις φυσικών αριθμών, με πλήρη απόδειξη μερικής ορθότητας. Για το σκοπό αυτό, εμπλουτίσαμε τη βιβλιοθήκη του NFLINT με τα θεωρήματα που φάνηκαν απαραίτητα.

7.2 Μελλοντική έρευνα

Μία σημαντική κατεύθυνση για μελλοντική έρευνα πάνω στις γλώσσες προγραμματισμού με αποδείξεις είναι η διερεύνηση του κατά πόσο μία σχεδίαση όπως αυτή της γλώσσας που περιγράψαμε σε αυτή την εργασία, με διαχωρισμό της φάσης προγραμματισμού από τη φάση απόδειξης, και χωρίς εμφανείς εξαρτώμενους τύπους, είναι πρακτική και μπορεί να επεκταθεί σε μεγαλύτερα προγράμματα. Έχει ενδιαφέρον να δούμε εάν, δεδομένης μιας αρκετά μεγάλης βιβλιοθήκης θεωρημάτων, μπορεί να επιτευχθεί ο στόχος των αποδείξιμα ορθών προγραμμάτων, ακόμη και για προγράμματα με πρακτική χρησιμότητα – χωρίς η διαδικασία της απόδειξης να δυσχεραίνει υπερβολικά το έργο του προγραμματιστή.

Προς αυτή την κατεύθυνση θα βοηθούσε μία επέκταση της υπάρχουσας γλώσσας ώστε να υποστηρίζει ένα πιο πλούσιο σύστημα τύπων. Θα μας ενδιέφερε ιδιαίτερα η υποστήριξη αλγεβρικών τύπων δεδομένων ορισμένων από το χρήστη, παρόμοιων με αυτούς που συναντάμε στις τρέχουσες διαδεδομένες γλώσσες συναρτησιακού προγραμματισμού. Η υποστήριξη τέτοιων τύπων συνεπάγεται τη δυνατότητα ορισμού δομών δεδομένων όπως λίστες και δέντρα, που θα επέτρεπαν την υλοποίηση ενδιαφερόντων αλγορίθμων όπως η ταξινόμηση και η αναζήτηση. Στην περίπτωση αυτή, πρέπει να εξεταστεί πώς πρέπει να εμπλουτιστεί και η γλώσσα των λογικών προτάσεων, ώστε να μπορούμε να εκφράζουμε και να αποδεικνύουμε τη μερική ορθότητα της υλοποίησης τέτοιων αλγορίθμων.

Η ιδέα των προ- και μετα-συνθηκών θα μπορούσε τότε να επεκταθεί και στους ορισμούς τύπων από το χρήστη, ώστε να υποστηριχθούν κάποιοι τύποι δεδομένων που έχουν ιδιαίτερο ενδιαφέρον στην Επιστήμη της Πληροφορικής. Τέτοιος τύπος είναι για παράδειγμα ο τύπος των ταξινομημένων λιστών, που εν αντιθέσει με τις κανονικές λίστες, ο κατασκευαστής Cons της δημιουργίας νέας λίστας από στοιχείο-κεφαλή και υπάρχουσα λίστα-ουρά, μπορεί να εφαρμοστεί μόνον εάν η προσθήκη του νέου στοιχείου διατηρεί την ταξινομημένη σειρά της λίστας. Άλλα τέτοια παραδείγματα τύπων είναι τα κόκκινα-μαύρα δέντρα (red-black trees) και οι σωροί (heaps).

Η γλώσσα θα γινόταν ακόμη πιο εκφραστική εάν επεκτείναμε το σύστημα τύπων με τύπους συναρτήσεων. Έτσι θα υπήρχε δυνατότητα για συναρτήσεις υψηλότερου βαθμού (που δέχονται ως παράμετρο συνάρτηση, ή επιστρέφουν συνάρτηση ως αποτέλεσμα). Στην περίπτωση αυτή έχει ενδιαφέρον το πώς χειριζόμαστε συναρτήσεις με διαφορετικές προ- και μετα-συνθήκες.

Ένα ακόμη επιθυμητό χαρακτηριστικό της γλώσσας είναι η υποστήριξη για παρενέργειες (side-effects), που θα επέτρεπε την ύπαρξη αναφορών, συναρτήσεων εισόδου-εξόδου, και γενικά υποστήριξης χαρακτηριστικών των προστακτικών γλωσσών προγραμματισμού.

Τέλος, ενδιαφέρον έχει η περίπτωση να αντικατασταθεί ο μηχανισμός κατασκευής αποδείξεων της γλώσσας, καθώς και το αρχείο θεωρημάτων, από ένα, πιθανώς διαδραστικό, σύστημα αποδείξεων όπως το Coq για το σύστημα NFLINT. Στην περίπτωση αυτή, εάν η υπάρχουσα βιβλιοθήκη θεωρημάτων του συστήματος NFLINT δεν επαρκεί για την απόδειξη της μερικής ορθότητας ενός προγράμματος, εμπλουτίζεται με νέα θεωρήματα, τα οποία αποδεικνύονται με τρόπο παρόμοιο με το σύστημα Coq – δίνοντας μία σειρά από τακτικές απόδειξης και όχι έναν πλήρη λ-όρο που αναπαριστά την απόδειξη, ο οποίος δύσκολα κατασκευάζεται.

Βιβλιογραφία

[Augu98] Lennart Augustsson, “Cayenne – a language with dependent types”, in *ICFP ’98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pp. 239–250, New York, NY, USA, 1998, ACM Press.

[CDev03] The Coq Development Team, “The Coq Proof Assistant Reference Manual”, Version 7.4, URL: `http://coq.inria.fr/`, February 2003.

[Chen05] Chiyan Chen and Hongwei Xi, “Combining Programming with Theorem Proving”, in *ACM International Conference on Functional Programming (ICFP)*, September 2005.

[Feng04] Xinyu Feng, “Design of the Certifying Programming Language – Vero”, Independent project report, Yale University, May 2004. URL: http://flint.cs.yale.edu/feng/research/publications/vero690.pdf.

[Shao02] Z. Shao, B. Saha, V. Trifonov and N. Papaspyrou, “A Type System for Certified Binaries”, in *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL 2002)*, pp. 217–232, Portland, OR, USA, January 2002.

[Shea04] Tim Sheard, “Languages of the future”, *SIGPLAN Not.*, vol. 39, no. 12, pp. 119–132, 2004.

[Shea05a] Tim Sheard, *Ωmega Users Guide*, November 2005. URL: http://www.cs.pdx.edu/˜sheard/Omega/.

[Shea05b] Tim Sheard, “Putting Curry-Howard to work”, in *Haskell ’05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pp. 74–85, New York, NY, USA, 2005, ACM Press.

[Xi98] Hongwei Xi, *Dependent Types in Practical Programming*, Ph.D. thesis, Carnegie Mellon University, December 1998.

[Xi04] Hongwei Xi, “Applied Type System”, in *TYPES 2003*, vol. 3085 of *Lecture Notes in Computer Science*, pp. 394–408, Springer Verlag, February 2004.