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

Υλοποίηση μιας Γλώσσας Τύπων με Εφαρμογές στην Πιστοποίηση Εκτελέσιμου Κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΜΙΧΑΛΗΣ Α. ΠΑΠΑΚΥΡΙΑΚΟΥ

Επιβλέπων : Νίκος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Αθήνα, Οκτώβριος 2004

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

Υλοποίηση μιας Γλώσσας Τύπων με Εφαρμογές στην Πιστοποίηση Εκτελέσιμου Κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΜΙΧΑΛΗΣ Α. ΠΑΠΑΚΥΡΙΑΚΟΥ

Επιβλέπων : Νίκος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20 Οκτωβρίου 2004.

.....	.....	.....
Στάθης Ζάχος Καθηγητής Ε.Μ.Π.	Γιώργος Κολέτσος Αναπληρωτής Καθηγητής Ε.Μ.Π.	Νίκος Παπασπύρου Λέκτορας Ε.Μ.Π.

Αθήνα, Οκτώβριος 2004

.....

Μιχάλης Α. Παπακυριάκου

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

Copyright © Μιχάλης Α. Παπακυριάκου, 2004.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σκοπός της εργασίας είναι η υλοποίηση ενός ελεγκτή τύπων (typechecker) και ενός διερμηνέα (interpreter) για μια γλώσσα τύπων, βασισμένη στο λογισμό των επαγωγικών κατασκευών (Calculus of Inductive Constructions — CIC). Η γλώσσα αυτή είναι αρκετά εκφραστική ώστε να μπορεί να χρησιμοποιηθεί για την κωδικοποίηση της κατηγορηματικής λογικής υψηλής τάξης. Η υλοποίησή μας μπορεί αφενός να χρησιμεύσει στο πλαίσιο ενός αυτόματου ή ημιαυτόματου συστήματος αποδείξεων θεωρημάτων. Αφετέρου, μπορεί να ενσωματωθεί στο σύστημα τύπων μιας γλώσσας προγραμματισμού ώστε η τελευταία να υποστηρίζει την παραγωγή πιστοποιημένου εκτελέσιμου κώδικα.

Η ανάγκη για αξιόπιστο και πιστοποιημένα ασφαλή εκτελέσιμο κώδικα είναι ακόμα μεγαλύτερη σήμερα, από ό,τι στο παρελθόν. Σε πολυάριθμες περιπτώσεις, προβλήματα ασφάλειας και συμβατότητας προγραμμάτων που έγιναν γνωστά, είχαν ως αποτέλεσμα να θέσουν σε κίνδυνο στην λειτουργία μεγάλων συστημάτων και συνεπώς οικονομικές επιπτώσεις στους οργανισμούς που τα χρησιμοποιούσαν. Για την παραγωγή πιστοποιημένου κώδικα μία τεχνική που κερδίζει έδαφος τα τελευταία χρόνια είναι η πιστοποίηση μέσω ισχυρών συστημάτων τύπων που ενσωματώνουν κατάλληλα συστήματα λογικής, μέσω των οποίων διατυπώνονται και αποδεινύονται προτάσεις ασφαλείας.

Σε ένα τέτοιο σύστημα πιστοποίησης της ασφάλειας εκτελέσιμου κώδικα, η γλώσσα των τύπων που είναι το θέμα αυτής της εργασίας αποτελεί το τμήμα της γλώσσας προγραμματισμού στο οποίο γίνονται όλοι οι τυπικοί συλλογισμοί. Μια παραλλαγή αυτής της γλώσσας έχει υλοποιηθεί στο εργαλείο υποστήριξης αποδείξεων Coq. Σε ένα πολύγλωσσο προγραμματιστικό περιβάλλον, η γλώσσα των τύπων θα αποτελεί τον κοινό παρονομαστή όλων των γλωσσών προγραμματισμού που μπορούν να χρησιμοποιηθούν για την υλοποίηση συστημάτων. Όμως, η γλώσσα τύπων που υλοποιείται στην παρούσα εργασία είναι χρήσιμη και αυτοτελώς, στο πλαίσιο ενός συστήματος κωδικοποίησης θεωριών και αποδείξεως θεωρημάτων βασισμένου στην κατηγορηματική λογική υψηλής τάξης.

Η υλοποίηση της γλώσσας τύπων βασίστηκε σε τεχνικές που βελτιστοποιούν την απόδοση, όπως ο κατακερματισμός (hash-consing), η κωδικοποίηση με ανώνυμες μεταβλητές μέσω των δεικτών de-Bruijn και η χρήση της ασθενούς κανονικής μορφής κεφαλής (weak head normal form) για την βελτίωση της απόδοσης του ελέγχου τύπων. Με κριτήριο την απόδοση και την λειτουργικότητα που προσφέρει επιλέχθηκε επίσης η γλώσσα υλοποίησης. Χρησιμοποιήθηκε η OCaml, μία γλώσσα που αναπτύχθηκε με τέτοιου είδους χαρακτηριστικά ώστε να προσφέρεται για την ανάπτυξη τέτοιων εφαρμογών. Από τα αποτελέσματα που παρατηρήθηκαν, ελέγχθηκε η ορθότητα και μετρήθηκε η απόδοση της παρούσας υλοποίησης της γλώσσας τύπων.

Λέξεις κλειδιά

Συστήματα τύπων, λογισμός των επαγωγικών κατασκευών, προγραμματισμός με αποδείξεις, ασφάλεια εκτελέσιμου κώδικα, OCaml.

Abstract

The purpose of this diploma dissertation is the implementation of a typechecker and an interpreter for a type language, based on the Calculus of Inductive Constructions (CIC). This type language is expresive enough to allow the encoding of higher-order predicate logic. Our implementation can be useful as part of an automatic or semi- automatic proof-assistant system. It can also be merged with the type- system of a programming language, in such a way that the latter produces certified binaries.

The need for reliable and certifiably secure code is even greater today than it whas in the past. In many cases, security and software compatibility issues put in danger the operation of large systems, with the subsequent financial consequences. In the recent past, a method for producing certified code that is gaining ground is through strong type systems incorporating various systems of logic, through which one formulates and proves security properties.

In such a certification system, the type language which is the subject of this dissertation is the part of the language that is responsible for all types of formal reasoning. A variaton of this language has been implemented in the Coq proof-assistant system. In a multilingual programming environment, the type language will be the common denominator of all programming languages that are used in software systems development. Nevertheless, the type language implemented in this dissertation is also useful per se, as part of a system for encoding theories and proving theorems, based on higher-order predicate logic.

The implementation of the type language was based on techniques which aim to optimize performance, such as hash-consing, representation of variables with de Bruijn indices and the use of weak head normal form to improve the performance of type checking. The programming language that was used for the implementation is OCaml, which was chosen because it combines improved performance and fuctionality for this type of applications. The obtained results were useful in assessing the correctness and performance of our implementation of the type language.

Key words

Type systems, calculus of inductive constructions, programming with proofs, security of executable code, OCaml.

Ευχαριστίες

Κατ' αρχήν θέλω να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου, κ. Νίκο Παπασπύρου για το ενδιαφέρον που έδειξε και για τον ατέλειωτο χρόνο που αφιέρωσε σε αυτή την εργασία. Θέλω επίσης να ευχαριστήσω τους γονείς που μου εξασφάλισαν τη δυνατότητα να διεκπαιρεώσω απερίσπαστος τις σπουδές μου. Τέλος θέλω να ευχαριστήσω τα μέλη του Εργαστηρίου Τεχνολογίας Λογισμικού για τη συνεργασία που είχαμε κατα διάρκεια αυτού του χρόνου.

Μιχάλης Α. Παπακυριάκου,

Αθήνα, 18η Οκτωβρίου 2004.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-3-04, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2004.

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

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Περιεχόμενα 11

Πίνακες 13

Σχήματα 15

1. Εισαγωγή 17
1.1 Αποδείξεις θεωρημάτων 17
1.2 Ασφάλεια εκτελέσιμου κώδικα 18
1.3 Βιβλιογραφική ανασκόπηση 20
1.4 Σύνοψη της εργασίας 22

2. Λάμβδα λογισμός, επαγωγικοί τύποι και λογική 25
2.1 Λάμβδα λογισμός, χωρίς τύπους 25
2.1.1 Ορισμός 25
2.1.2 Αντικατάσταση στον λ-λογισμό 26
2.1.3 Μετατροπές 26
2.1.4 Κανονικές μορφές 27
2.2 Δείκτες de Bruijn: ανώνυμες μεταβλητές 28
2.3 Λάμβδα λογισμός, με απλούς τύπους 29
2.3.1 Ορισμός 29
2.3.2 Παραδείγματα τύπων 29
2.4 Λάμβδα λογισμός δευτέρας τάξης 30
2.4.1 Ορισμός 30
2.4.2 Παράσταση εκφράσεων 31
2.4.3 Παραδείγματα τύπων 31
2.5 Λάμβδα λογισμός ω-τάξης 31
2.5.1 Ορισμός 31
2.5.2 Παράσταση εκφράσεων 32
2.6 Λογισμός κατασκευών 32
2.6.1 Ορισμός 32
2.7 Αμιγή συστήματα τύπων 32
2.7.1 Ορισμός 32
2.7.2 Παράσταση συστημάτων λ-λογισμού με PTS 33
2.8 Λογισμός επαγωγικών κατασκευών 34
2.8.1 Ορισμός 34
2.8.2 Παραδείγματα 34

2.9 Προτάσεις ως τύποι . . . 35

3. Περιγραφή της γλώσσας τύπων nflint . . . 37
3.1 Ορισμός . . . 37
3.2 Λειτουργική σημασιολογία . . . 38
3.2.1 Συμβατές σχέσεις . . . 38
3.2.2 Αναγωγές . . . 39
3.2.3 Ισότητα . . . 39
3.3 Κανόνες τύπων . . . 39
3.3.1 Βοηθητικές συναρτήσεις . . . 40
3.3.2 Μεταβλητές και σταθερές . . . 40
3.3.3 Ισοδυναμία τύπων . . . 40
3.3.4 Γινόμενο, αφαίρεση και εφαρμογή . . . 41
3.3.5 Επαγωγικοί ορισμοί. . . . 41
3.4 Μετα-θεωρία της γλώσσας τύπων . . . 41

4. Θέματα υλοποίησης . . . 43
4.1 Σύνοψη . . . 43
4.2 Γλώσσα υλοποίησης . . . 44
4.3 Hash-consing . . . 44
4.4 Ονόματα και σταθερές . . . 45
4.5 Πρωτογενείς όροι . . . 47
4.6 Όροι . . . 48
4.7 Κανονικές μορφές . . . 50
4.8 Έλεγχος τύπων . . . 50
4.9 Αποτίμηση όρων . . . 52

5. Αξιολόγηση . . . 53
5.1 Θεωρίες . . . 53
5.1.1 Λογικές τιμές . . . 53
5.1.2 Ζεύγη . . . 54
5.1.3 Προτάσεις . . . 54
5.1.4 Συγκρίσεις . . . 55
5.1.5 Φυσικοί αριθμοί . . . 55
5.1.6 Ακέραιοι αριθμοί . . . 61
5.1.7 Λίστες . . . 61
5.2 Έλεγχος της υλοποίησης . . . 62
5.2.1 Παραδείγματα υπολογισμών . . . 62
5.2.2 Παραδείγματα θεωρήματων . . . 63
5.2.3 Μετρήσεις απόδοσης . . . 64

6. Συμπεράσματα . . . 71
6.1 Συνεισφορά . . . 71
6.2 Μελλοντική έρευνα . . . 71

Βιβλιογραφία . . . 73

Πίνακες

2.1 Κανόνες τύπων του $\lambda \rightarrow$. 29
2.2 Κανόνες τύπων του $\lambda 2$. 30
2.3 Κανόνες τύπων του $\lambda(S, A, R)$. 33

5.1 Μετρήσεις χρόνου εύρεσης τύπου. 68
5.2 Μετρήσεις χρόνου αποτίμησης της συνάρτησης fibonacci. 68
5.3 Μετρήσεις χρόνου αποτίμησης της συνάρτησης factorial. 68

Σχήματα

1.1 Ο διαχωρισμός της γλώσσας προγραμματισμού σε γλώσσα τύπων και γλώσσα υπολογισμών. 21

5.1 Χρόνος υπολογισμού της ακολουθίας Fibonacci. 69

Κεφάλαιο 1

Εισαγωγή

Σκοπός της εργασίας είναι η υλοποίηση ενός ελεγκτή τύπων (typechecker) και ενός διερμηνέα (interpreter) για μια γλώσσα τύπων, βασισμένη στο λογισμό των επαγωγικών κατασκευών. Η γλώσσα αυτή είναι αρκετά εκφραστική ώστε να μπορεί να χρησιμοποιηθεί για την κωδικοποίηση της κατηγορηματικής λογικής υψηλής τάξης. Η υλοποίησή μας μπορεί αφενός να χρησιμεύσει στο πλαίσιο ενός αυτόματου ή ημιαυτόματου συστήματος αποδείξεων θεωρημάτων. Αφετέρου, μπορεί να ενσωματωθεί στο σύστημα τύπων μιας γλώσσας προγραμματισμού ώστε η τελευταία να υποστηρίζει την παραγωγή πιστοποιημένου εκτελέσιμου κώδικα.

1.1 Αποδείξεις θεωρημάτων

Κατά το παρελθόν έχει γίνει αρκετή έρευνα για το πώς ένα σύστημα αποδείξεων μπορεί να προκύψει από ένα σύστημα τύπων, καθώς και πώς μπορεί κανείς να συσχετίσει τα συστήματα τύπων με συστήματα λογικής. Στην εργασία αυτή, ως συστήματα τύπων εννοούμε το λ-λογισμό με τύπους και τις επεκτάσεις αυτού, ιδιαίτερα μάλιστα το *σύστημα κατασκευής επαγωγικών τύπων* (Calculus of Inductive Constructions).

Η πρώτη περιγραφή της αναπαράστασης αποδείξεων με συστήματα τύπων διατυπώθηκε από τους Curry και Howard και είναι γνωστή ως *ισομορφισμός Curry-Howard* [Curr68, Howa80]. Σε αυτή δόθηκε μια τυπική μετάφραση των αποδείξεων σε λ-όρους με τύπους. Για τη συνεπαγωγή $A \supset B$ και τον καθολικό ποσοδείκτη $\forall x{\in}A.\, P(x)$, ο κανόνας εισαγωγής αντιστοιχεί στη αφαίρεση του λ-λογισμού και ο κανόνας απαλοιφής στην εφαρμογή. Αργότερα έγιναν και άλλες προσπάθειες τυποποίησης της αντιστοίχησης της λογικής σε συστήματα τύπων, με σημαντικότερη τη προσέγγιση του de Bruijn τη ίδια χρονιά, με το σύστημα *Automath*.

Σε όλες τις προτάσεις που έχουν γίνει κατά το παρελθόν και για οποιοδήποτε σύστημα λογικής, οι αποδείξεις παριστάνονται ως τύποι και οι αποδείξεις τους είναι οι όροι της γλώσσας τύπων. Η αναπαράσταση αυτή δίνει μάλιστα πολλές δυνατότητες όπως την τυποποίηση της λογικής των αποδείξεων ή την εξάρτηση όρων από αποδείξεις.

Στην πιο συνηθισμένη αναπαράσταση αποδείξεων με λ-όρους, οι τύποι-προτάσεις θεωρούνται ως σύνολα και οι όροι-αποδείξεις ως στοιχεία των συνόλων. Σε αυτή την αναπαράσταση, για να αποδειχτεί ένας τύπος-πρόταση, αρκεί το σύνολο να είναι κατοικημένο, δηλαδή, στο πεδίο του λ-λογισμού να υπάρχει κάποιος όρος που να έχει αυτόν τον τύπο. Ένα απλό παράδειγμα είναι ο τύπος-πρόταση $\Pi x{:}A.\, P(x)$, όπου A ένα σύνολο. Αυτός ερμηνεύεται ως: "Για κάθε στοιχείο x του συνόλου A, ισχύει $P(x)$". Η απόδειξη της πρότασης αυτής είναι ένας όρος της μορφής $\lambda x{:}A.\, p(x)$ που έχει τον παραπάνω όρο ως τύπο. Αυτός ο όρος ερμηνεύεται ως: "Έστω ένα τυχαίο στοιχείο x του συνόλου A. Αποδεικνύεται ότι ισχύει $P(x)$", υπό την προϋπόθεση $p(x)$ να είναι μια απόδειξη της πρότασης $P(x)$.

Στον τομέα των αλληλεπιδραστικών συστημάτων αποδείξεων (interactive proof systems) έχει γίνει πολλή έρευνα [Bare01]. Τα γνωστότερα σήμερα τέτοια συστήματα είναι τα ακόλουθα:

- Coq `http://coq.inria.fr/`
  Το *Coq* είναι ένα σύστημα διαχείρισης τυπικών αποδείξεων που ενσωματώνει κάποιες θεωρίες, ενώ επιτρέπει τον ορισμό νέων θεωριών και την απόδειξη τους. Περιλαμβάνει ένα

μηχανισμό για αυτόματη παραγωγή των αποδείξεων, γραφικό περιβάλλον και διατίθεται από το *Γαλλικό Ινστιτούτο Έρευνας για την Επιστήμη των Υπολογιστών* (INRIA) ως λογισμικό ανοιχτού κώδικα (open source).

- HOL `http://hol.sourceforge.net/`
  Πρόκειται για ένα ακόμα αυτοματοποιημένο σύστημα αποδείξεων για υψηλής τάξης λογική. Ενσωματώνει και αυτό αποδείξεις για απλά θεωρήματα παρέχοντας ένα προγραμματιστικό περιβάλλον για την διατύπωση και απόδειξη θεωρημάτων και την ανάπτυξη αποδεικτικών εργαλείων. Παλαιότερα αναπτύχθηκε από το τα πανεπιστήμια του Cambridge, της Γλασκώβης και της Utah, καθώς και από τα Bell Labs. Σήμερα διατίθεται και αναπτύσσεται ως λογισμικό ανοιχτού κώδικα (open source) από το *Sourceforge* με το όνομα *HOL* 4.

- Isabelle `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`
  Ένα ακόμα σύστημα αποδείξεων, το οποίο όμως επικεντρώνεται στην τυπική επιβεβαίωση ότι τόσο λογισμικό όσο και υλικό υπολογιστών χαρακτηρίζονται από συγκεκριμένες ιδιότητες. Η πρωτοτυπία του έγκειται στο γεγονός ότι αποδέχεται ένα σύνολο από είδη λογισμών. Η έκδοση του συστήματος που είναι διαδεδομένη υποστηρίζει λογισμό υψηλής τάξης αλλά και θεωρία αξιωμάτων, καθώς και άλλους φορμαλισμούς. Συντηρείται από το πανεπιστήμιο του Cambridge και από το Τεχνολογικό Πανεπιστήμιο του Μονάχου.

- Lego `http://www.dcs.ed.ac.uk/home/lego/`
  Το σύστημα αποδείξεων αυτό, σχεδιασμένο και υλοποιημένο από τον Randy Pollack υλοποιεί αρκετά σχετικά συστήματα τύπων, όπως το *Edinburgh Logical Framework* (LF), τον λογισμό επαγωγικών κατασκευών, τον γενικευμένο λογισμό κατασκευών και την ενοποιημένη θεωρία των εξαρτώμενων τύπων. Το σύστημα αυτό αναπτύχθηκε αρχικά στο Πανεπιστήμιο του Εδιμβούργου ενώ τώρα συντηρείται ανεξάρτητα από προγραμματιστες.

- Nuprl `http://www.cs.cornell.edu/Info/Projects/NuPrl/`
  Το αντίστοιχο πρόγραμμα του Πανεπιστημίου του Cornell ξεκίνησε το 1979 με την ονομασία "Λογική για την Εκλέπτυνση Προγραμμάτων" (Program Refinement Logic), αλλά σήμερα στην ερμηνεία του ακρωνύμιου "PRL" τείνουν να αντικαταστήσουν τον όρο πρόγραμμα με τον όρο "αποδείξεις". Βασικός σκοπός του προγράμματος αυτού είναι τα προγράμματα για την απόδειξη θεωρημάτων να γίνουν απαραίτητα στους προγραμματιστές και τους μαθηματικούς, καθώς και η κατασκευή ενός πραγματικά "έξυπνου" proof assistant.

- PVS `http://pvs.csl.sri.com/`
  Το σύστημα PVS είναι ένα εργαλείο που παρέχει μια γλώσσα προδιαγραφών και τα κατάλληλα εργαλεία, μαζί με μία μέθοδο απόδειξης θεωρημάτων. Έχει εφαρμοστεί πάνω σε αλγόριθμους και αρχιτεκτονικές για ανοχή σφαλμάτων σε συστήματα ελέγχου πτήσεων, και στο σχεδιασμό συστημάτων συστημάτων πραγματικού χρόνου.

- Twelf `http//www.cs.cmu.edu/~twelf/`
  Αυτό το σύστημα επικεντρώνεται στο σχεδιασμό, την υλοποίηση και την εφαρμογή περιβαλλόντων της λογικής. Εμπεριέχει το λογικό πλαίσιο LF, τη γλώσσα λογικού προγραμματισμού με περιορισμούς (constraint logic programming language) *Elf* και ένα αυτόματο σύστημα απόδειξης μετα-θεωρημάτων. Αναπτύχθηκε στο πανεπιστήμιο Carnegie Mellon.

1.2 Ασφάλεια εκτελέσιμου κώδικα

Στις μέρες μας γίνεται όλο και πιο φανερή η ανάγκη αξιόπιστου και πιστοποιημένα ασφαλούς κώδικα. 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.

Στη σημερινή εποχή τα συστήματα λογισμικού αποτελούνται, ως επί το πλείστον, από μικρά ανεξάρτητα τμήματα, τα οποία συχνά έχουν αναπτυχθεί από διαφορετικές ομάδες ανθρώπων, με διαφορετική φιλοσοφία και πιθανότατα σε διαφορετικές πλατφόρμες. Τα ανεξάρτητα αυτά τμήματα συνεργάζονται μεταξύ τους μέσω διαπροσωπειών που ορίζουν το πρωτόκολλο διαλειτουργίας. Η ανάπτυξη της τεχνολογίας, τόσο με τον καθολικό χαρακτήρα του διαδικτύου όσο και με την ενσωμάτωση όλο και περισσότερων συσκευών με χαρακτηριστικά υπολογιστή στην καθημερινότητα του ανθρώπου, επιβάλλει τα τμήματα αυτά κώδικα να παρέχουν ασφάλεια και διαλειτουργικότητα ανεξάρτητα από την πλατφόρμα που θα εκτελεστούν. Ιδιαίτερα όσον αφορά στο διαδίκτυο, τμήματα κώδικα και εκτελέσιμα γίνονται διαθέσιμα θεωρώντας ότι είναι κατάλληλα προς χρήση, χωρίς να υπάρχουν πειστικά στοιχεία που να αποδεικνύουν κάτι τέτοιο.

Σήμερα η τάση είναι να δημιουργούνται συστήματα όπου η εκτέλεση κώδικα γίνεται κατανεμημένα σε πολλές υπολογιστικές συσκευές. Ο εκτελέσιμος κώδικας που μεταφέρεται μέσα σε ένα τέτοιο σύστημα υπολογιστικών συσκευών μπορεί να δημιουργήσει πολλά προβλήματα, είτε επίτηδες, αν περιλαμβάνει τμήματα κώδικα που δίνουν δυνατότητα σε ξένους προς το σύστημα να εκμεταλλευτούν πηγές του (π.χ. ιοί, επιθέσεις), είτε χωρίς πρόθεση, αν περιέχει σφάλματα που θέτουν σε κίνδυνο τη σταθερότητα και την ασφάλεια της συσκευής στην οποία εκτελείται.

Οι υπάρχοντες μεταγλωττιστές που παράγουν πιστοποιημένο κώδικα εφαρμόζουν ελέγχους κατά τη διάρκεια εκτέλεσης του προγράμματος, χωρίς να μπαίνουν αρκετά στην ουσία της πιστοποίησης. Συνήθως κάτι τέτοιο υλοποιείται με μία νοητή μηχανή και έναν μεταγλωττιστή της τελευταίας στιγμής (Just-In-Time Compiler — JIT). Οι πιο γνωστές τέτοιες υλοποιήσεις είναι της γλώσσας *Java* και της πλατφόρμας *.NET* της Microsoft. Η υλοποίηση μέσω νοητής μηχανής που διενεργεί ελέγχους σε χρόνο εκτέλεσης έχει ως αποτέλεσμα να μην είναι ικανοποιητική η λειτουργία αυτή, αλλά και να περιορίζει την απόδοση του συστήματος. Μια άλλη τεχνική που κερδίζει διαρκώς έδαφος είναι η πιστοποίηση του κώδικα, μέσω της ταυτοποίησης του παραγωγού του. Αυτό πραγματοποιείται με τη βοήθεια τεχνικών κρυπτογραφίας, ώστε να αντιστοιχηθεί ο κώδικας με τον παραγωγό πέραν αμφιβολίας. Η επιτυχία της τεχνικής είναι ανάλογη της εμπιστοσύνης που υπάρχει απέναντι στις πηγές που υποστηρίζουν την ταυτοποίηση.

Παρ 'ολ αυτά καμία από αυτές τις τεχνικές δεν είναι ικανοποιητική. Η πρώτη, με τον έλεγχο κατά τη διάρκεια της εκτέλεσης συνεπάγεται σημαντικές απώλειες στην απόδοση του προγράμματος, ενώ η δεύτερη είναι αρκετά περιοριστική από τη φύση της, αφού τελικά δεν πιστοποιεί κώδικα αλλά ομάδες προγραμματιστών.

Πρόσφατες έρευνες ενισχύουν την άποψη ότι, παρόλο που είναι αρκετά φιλόδοξο ως σχέδιο, είναι δυνατόν να οριστεί και να υλοποιηθεί ένα γενικό πλαίσιο, με τη χρήση του οποίου να 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.

Ο κύριος στόχος μίας τέτοιας ερευνητικής προσπάθειας είναι η δημιουργία ενός πλαισίου, για την εξασφάλιση της αξιοπιστίας του εκτελέσιμου κώδικα, παρέχοντας ασφάλεια και απόρρητο τόσο στο επίπεδο της εφαρμογής όσο και στο επίπεδο της υποδομής. Τα παραπάνω μπορούν να ενταχθούν σε μια γενικευμένη έννοια "αξιοπιστίας", η οποία συμπεριλαμβάνει τις στενότερες προαναφερθείσες έννοιες όπως αυτές συνήθως χρησιμοποιούνται σε σχέση με συστήματα λογισμικού, ώστε να οριστεί ένα πλαίσιο ικανό να επαληθεύει αυθαίρετες ιδιότητες, εκφρασμένες ως προδιαγραφές υψηλού επιπέδου σε μια κατάλληλη τυπική λογική.

Ένα τέτοιο σύστημα θα πρέπει να είναι τυπικά ορισμένο. Είναι γνωστό άλλωστε ότι μετά από περίπου μισό αιώνα εμπειρίας με τους υπολογιστές και το λογισμικό, η ανάπτυξη λογισμικού είναι ένας κλάδος της επιστήμης του μηχανικού. Αν και όλοι οι άλλοι κλάδοι αυτής της επιστήμης βασίζονται σε στερεές θεωρητικές, μαθηματικές βάσεις, στην περίπτωση της τεχνολογίας λογισμικού (software engineering) υπάρχει ακόμη ένα σημαντικό κενό μεταξύ των τυπικών τεχνικών για την ανάπτυξη προγραμμάτων και των εμπειρικών τεχνικών και μεθόδων που χρησιμοποιούνται στην πράξη από τους προγραμματιστές. Ένας σημαντικός στόχος της επιστήμης υπολογιστών είναι να γεφυρώσει το κενό αυτό και να δημιουργήσει νέα υψηλά πρότυπα για την ποιότητα του λογισμικού. Ένα εργαλείο που πιστοποιεί την αξιοπιστία του κώδικα είναι χρήσιμο τόσο για μία γλώσσα υψηλού επιπέδου, όσο και για μια ενδιάμεση ή συμβολική γλώσσα χαμηλού επιπέδου. Η ενσωμάτωση της τυπικής λογικής στο σύστημα τύπων της γλώσσας προγραμματισμού κάνει τη χρήση της περισσότερο δομημένη και οικεία προς στους προγραμματιστές.

Η τυπική λογική που θα χρησιμοποιηθεί θα πρέπει να είναι αρκετά ισχυρή ώστε να μπορεί να εκφράσει και να επαληθεύει αυθαίρετες ιδιότητες συστημάτων λογισμικού. Γνωστά ερευνητικά αποτελέσματα έχουν δείξει πώς η κατηγορηματική λογική υψηλής τάξης μπορεί να ενσωματωθεί σε συστήματα τύπων με βάση την αρχή των "Προτάσεων ως Τύπων" (propositions-as-types).

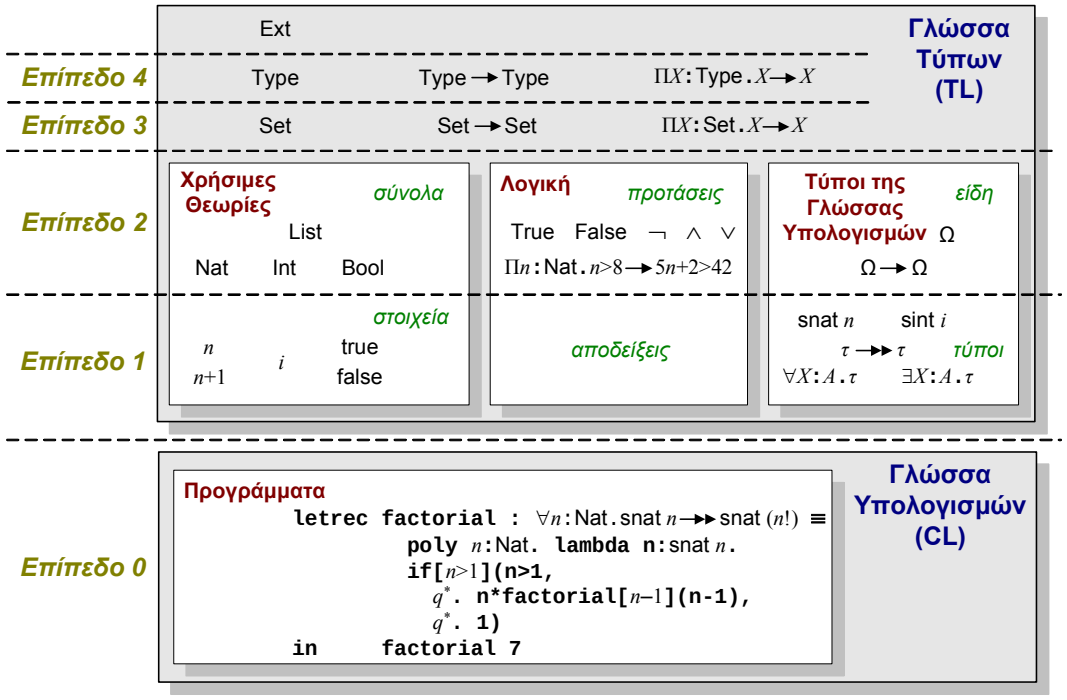
Κατά γενική ομολογία το πλαίσιο στο οποίο θα βασίζεται η αξιοπιστία στο λογισμικό πρέπει να είναι ευέλικτο με πολλούς τρόπους. Η ανεξαρτησία από τη γλώσσα είναι μια πολύ σημαντική προϋπόθεση για την ευελιξία. Ένα τέτοιο πλαίσιο θα πρέπει να επιτρέπει στους προγραμματιστές να γράφουν προγράμματα σε ένα πλήθος γλωσσών υψηλού επιπέδου, πιθανότατα ακολουθώντας τελείως διαφορετικά παραδείγματα προγραμματισμού, και ακολούθως να διενεργούν συλλογισμούς για τις ιδιότητες των προγραμμάτων τους. Θα πρέπει επίσης να υποστηρίζει πλήθος γλωσσών χαμηλού επιπέδου που να αντιστοιχούν στις πιο διαδεδομένες υπολογιστικές συσκευές, και να είναι αρκετά γενικό για να εφαρμοστεί σε νέες υπολογιστικές συσκευές με αποδοτικό τρόπο. Η ευελιξία είναι επίσης αρκετά σημαντική σε σχέση με την πολιτική ασφάλειας: σε διαφορετικές περιοχές εφαρμογών, διαφορετικοί αποδέκτες κώδικα ορίζουν την ασφάλεια με διαφορετικό τρόπο ο καθένας. Θα ήταν κατά συνέπεια χρήσιμο να μπορεί να πιστοποιηθεί όχι μόνο ότι ο εκτελέσιμος κώδικας καλύπτει τις ακριβείς προδιαγραφές ασφαλείας που περιμένει ένας παραλήπτης, αλλά και ένα σύνολο προδιαγραφών πιο ισχυρό από αυτό. Δεν είναι ξεκάθαρο σε ποιο βαθμό αυτό είναι δυνατό χωρίς ανθρώπινη παρέμβαση στη γενική περίπτωση. Θα ήταν επίσης χρήσιμο, αν οι παραλήπτες κώδικα μπορούσαν δυναμικά να αλλάζουν την πολιτική ασφάλειάς τους, ή ακόμη να απαιτούν τα διάφορα τμήματα προγραμμάτων που έρχονται από διαφορετικούς παραγωγούς κώδικα να ικανοποιούν διαφορετικές προδιαγραφές.

Προκειμένου να χρησιμεύσει σε ένα τέτοιο πλαίσιο πιστοποίησης εκτελέσιμου κώδικα, η γλώσσα τύπων που περιγράφεται και υλοποιείται σε αυτή την εργασία πρέπει να συνοδεύεται και από μια κατάλληλη, καλά ορισμένη γλώσσα υπολογισμών. Στο σχήμα 1.1 φαίνεται η σχέση μεταξύ της γλώσσας τύπων και της γλώσσας υπολογισμών, καθώς και η διαστρωμάτωση του όλου πλαισίου πιστοποίησης.

1.3 Βιβλιογραφική ανασκόπηση

Εξυπακούεται ότι η ανάπτυξη ενός συστήματος πιστοποίησης εκτελέσιμου κώδικα προϋποθέτει ένα συστηματικό τρόπο να μπορεί κανείς να αποδεικνύει προτάσεις στη γλώσσα τύπων. Σε αυτή την παράγραφο γίνεται μια σύντομη βιβλιογραφική ανασκόπηση σε συστήματα ανάπτυξης πιστοποιημένου κώδικα, με έμφαση στα συστήματα λογικής που υποστηρίζουν.

Η *Ενδιάμεση γλώσσα προγραμματισμού με Τύπους, TIL* [Harp95] και η *Ενδιάμεση γλώσσα μηχανής με Τύπους, TAL* [Morr98] είναι δύο από τις πρώτες προσπάθειες να επεκταθούν ενδιάμεσες γλώσσες και γλώσσες μηχανής με ισχυρά συστήματα τύπων που μπορούν να χρησιμοποιηθούν για πιστοποίηση κώδικα. Παρ όλο που τα πλεονεκτήματα από τη χρήση ενδιάμεσης



Σχήμα 1.1: Ο διαχωρισμός της γλώσσας προγραμματισμού σε γλώσσα τύπων και γλώσσα υπολογισμών.

ή τελικής γλώσσας με τύπους δεν περιορίζονται σε θέματα ασφαλείας, οι ιδιότητες που αποδίδονται στα προγράμματα που γράφονται σε μία τέτοια γλώσσα μπορούν να χρησιμοποιηθούν για να εγγυηθούν μερικές, απλές συνήθως, συνθήκες ασφαλείας κατά την εκτέλεση του προγράμματος. Για να επιτευχθεί αυτό είναι απαραίτητο ένας κατάλληλος ελεγκτής τύπων να πιστοποιήσει την ορθότητα του τελικού κώδικα πριν την εκτέλεση.

Το σύστημα *Proof-Carrying Code* (PCC) που προτάθηκε από τον Necula είναι ένα γενικό πλαίσιο που καταπιάνεται με την ακεραιότητα και την ασφάλεια ενός συστήματος, χρησιμοποιώντας τεχνικές από την μαθηματική λογική και την σημασιολογία γλωσσών προγραμματισμού [Necu96, Necu97, Necu98]. Η λογική του συστήματος αυτού είναι η εξής: Κατ' αρχήν οι δύο εμπλεκόμενες πλευρές, ο συγγραφέας του κώδικα και χρήστης, πρέπει να συμφωνούν στη πολιτική ασφαλείας γύρω από την οποία κυμαίνεται ολόκληρο το σύστημα. Η πολιτική αυτή, εκφραζόμενη σε κατάλληλη τυπική λογική αποτελείται από ένα σύνολο κανόνων ασφαλείας και απαιτήσεων ορθότητας που εξασφαλίζουν την ασφάλεια του προς εκτέλεση κώδικα. Από τη πλευρά του προγραμματιστή ο κώδικας περνάει από έναν μεταγλωττιστή που παράγει κατάλληλο πιστοποιημένο κώδικα. Ο κώδικας αυτός στο PCC είναι ένα πακέτο από τελικό κώδικα και την αναπαράσταση της απόδειξης ότι αυτός πληροί τις προϋποθέσεις ασφαλείας που τέθηκαν. Από τη πλευρά του χρήστη η ασφάλεια του εκτελέσιμου εξασφαλίζεται από την απόδειξη που έρχεται μαζί με τον κώδικα. Το σύστημα αυτό, συνοψίζοντας, απαιτεί ένα απλό μοντέλο εξασφάλισης εμπιστοσύνης, ενώ δεν συνεπάγεται καθυστέρηση σε χρόνο εκτέλεσης.

Σε σύγκριση με το TAL, το PCC χρησιμοποιεί μία γενικής-εφαρμογής κατηγοριματική λογική πρώτης τάξης και μπορεί να εκφράσει πιο πολύπλοκες ιδιότητες ασφαλείας. Από την άλλη μεριά το PCC χρησιμοποιεί ρητές αποδείξεις ασφάλειάς που, στη γενική περίπτωση, δεν μπορούν να παραχθούν αυτόματα από κάποιο εργαλείο. Ακόμα η γλώσσα στην οποία βασίζεται είναι λιγότερο εκφραστική από αυτή του TAL, η οποία υποστηρίζει ένα πολύ πιο περίτεχνο σύστημα τύπων. Παρ' ολ' αυτά και τα δύο συστήματα χαρακτηρίζονται από μια βασική ανάγκη η πολιτική ασφαλείας που εκφράζεται, πρέπει να είναι εκφρασμένη σε συνάρτηση με τη γλώσσα

που βρίσκεται πίσω από το σύστημα και κυρίως με το σύστημα τύπων. Αυτό σημαίνει ότι τόσο οι κανόνες τους οποίους περικλείει η πολιτική ασφαλείας, όσο και ο ελεγκτής τύπων και ο μεταφραστής στην ενδιάμεση γλώσσα πρέπει να μην έχουν κανένα ελάττωμα. Ευτυχώς κάτι τέτοιο είναι σημαντικά ευκολότερο από ότι σε εμπορικά συστήματα οπών η JVM (Java Virtual Machine) και η CLR (Common Language Runtime) της πλατφόρμας .NET της Microsoft, ενώ η ορθότητα τους μπορεί να αποδειχθεί με τη βοήθεια μιας μετα-θεωρίας.

Το σύστημα *Foundational Proof-Carrying Code* (FPCC), που παρουσίασαν οι Appel και Felty, είτε ως σκοπό του να παρακάμψει κάποια από τα προαναφερθέντα ελαττώματα ελαχιστοποιώντας το μέγεθος των τμημάτων που πρέπει να θεωρούνται ως πιστοποιημένα χωρίς απόδειξη σε ένα PCC system [Appe00, Appe01]. Το FPCC έναν γενικής χρήσης κατηγορηματικό λογισμό υψηλής τάξης και μερικά αξιώματα από την αριθμητική που μπορούν να χρησιμοποιηθούν ως βάση για τα σύγχρονα μαθηματικά. Μα μία τέτοια γλώσσα είναι δυνατόν να οριστεί τόσο μια πολιτική ασφαλείας όσο και ένα σύστημα τύπων και σημασιολογία για την γλώσσα προγραμματισμού. Στο σύστημα αυτό, ο πιστοποιημένος κώδικας είναι πάλι ένα πακέτο από τελικό κώδικα μαζί με μια αναπαράσταση της απόδειξης ασφαλείας. Οι αποδείξεις αυτές πρέπει να είναι καλά ορισμένες, δηλαδή όλες οι ζητούμενες ιδιότητες πρέπει να αποδεικνύονται με βάση μαθηματικούς ορισμούς.

Το FPCC είναι πιο ευέλικτο από τα TAL και PCC διότι δεν περιορίζεται από κάποια συγκεκριμένη γλώσσα προγραμματισμού ή από κάποιο υποσύνολο λογικής ή μαθηματικών. Οι αποδείξεις που ξεκινούν από θεμελιώδεις σχέσεις μπορούν να περιλαμβάνουν τον ορισμό νέων συστημάτων τύπων για την τελική γλώσσα και νέων ιδιοτήτων για την απόδειξη της ασφάλειας του συστήματος. Επιπλέον το FPCC είναι πιο ασφαλές αφού δέχεται ακόμα μικρότερα τμήματα ως ασφαλή. Φυσικά, και σε αυτό το σύστημα, για να μπορεί να εξασφαλιστεί η ασφάλεια πρέπει να μην υπάρχουν σφάλματα στον ορισμό και την υλοποίηση του συστήματος τύπων και του πιστοποιητή αποδείξεων. Όπως είναι αναμενόμενο, βέβαια, τα πλεονεκτήματα αυτά έχουν ως αποτέλεσμα η κατασκευή των αποδείξεων να είναι ακόμα πιο δύσκολη.

Η παρούσα εργασία βασίζεται στην προσέγγιση των Shao *et al.* [Shao02], η οποία προτείνει ένα τυπο-θεωρητικό πλαίσιο για την ανάπτυξη πιστοποιημένου κώδικα. Παρόμοια προσέγγιση προτάθηκε ανεξάρτητα από τους Crary και Vanderwaart [Crar02]. Οι τυπο-θεωρητικές αυτές προσεγγίσεις χτίζουν πάνω σε ένα μεγάλο οικοδόμημα έρευνας στην περιοχή της λογικής και της απόδειξης θεωρημάτων και προσπαθούν να καταστήσουν αυτή την έρευνα χρήσιμη στην κοινότητα των κατασκευαστών μεταγλωττιστών. Παρότι πολλά προβλήματα παραμένουν άλυτα ως σήμερα, η προσέγγιση αυτή θεωρείται ελπιδοφόρα.

1.4 Σύνοψη της εργασίας

Η υλοποίηση της γλώσσας τύπων έγινε στη γλώσσα συναρτησιακού προγραμματισμού OCaml. Η επιλογή της OCaml έγινε με κριτήρια την απόδοσή της και το γεγονός ότι προσφέρεται ιδιαίτερα για τον προγραμματισμό τέτοιου είδους εφαρμογών. Ο πηγαίος κώδικας της υλοποίησης είναι περίπου 2,500 γραμμές και ο κώδικας των θεωριών που χρησιμοποιήθηκαν για τον έλεγχό της άλλες 3,500 γραμμές. Στο πλαίσιο της εργασίας δεν υλοποιήθηκε λεκτικός και συντακτικός αναλυτής για τη γλώσσα τύπων.

Για την υλοποίηση της γλώσσας τύπων ακολουθήθηκε η εξής διαδικασία. Αρχικά κατασκευάστηκαν οι συναρτήσεις για τον αποδοτικό χειρισμό της μνήμης μέσω πίνακα κατακερματισμού. Στη συνέχεια σχεδιάστηκε ο χειρισμός των σταθερών όρων με αρθρώματα και υλοποιήθηκε κατάλληλα ώστε να μπορούν να οριστούν όροι βιβλιοθήκης χωρίς να υπάρχουν διενέξεις μεταξύ των ονομάτων. Ακολούθησε ο ορισμός της γλώσσας τύπων, η οποία περιγράφεται στο κεφάλαιο 3. Η σύνταξη και ο ορισμός της στη συνέχεια τροποποιήθηκαν κατάλληλα, ώστε η αναπαράσταση των όρων να γίνεται με ανώνυμες μεταβλητές. Για το σκοπό αυτό κατασκευάστηκαν δύο είδη όρων, ένα με ονομασμένες και ένα με ανώνυμες μεταβλητές, καθώς και διαδικασία μετατροπής μεταξύ τους. Το σημαντικότερο κομμάτι της υλοποίησης είναι η κα-

τασκευή του ελεγκτή τύπων και του διερμηνέα. Η υλοποίηση ολοκληρώθηκε με τη διαδικασία εκτύπωσης όρων της γλώσσας σε κατάλληλη μορφή, ώστε να είναι εύληπτη από τον χρήστη. Για τον έλεγχο της υλοποίησης διατυπώθηκαν μία σειρά από θεωρίες μαθηματικών και λογικής, καθώς και κάποιοι όροι προς αποτίμηση.

Η συνέχεια της διπλωματικής εργασίας έχει την ακόλουθη δομή:

Κεφάλαιο 2. Περιγραφή του λ-λογισμού και των παραλλαγών του με τύπους.

Κεφάλαιο 3. Ορισμός και περιγραφή της γλώσσας τύπων που υλοποιήθηκε.

Κεφάλαιο 4. Αναλυτική περιγραφή της υλοποίησης καθώς και αρχεία επικεφαλιδων για κάθε τμήμα της.

Κεφάλαιο 5. Περιγραφή θεωριών στη γλώσσα τύπων και έλεγχος καλής λειτουργίας της υλοποίησης μέσω των θεωριών αυτών.

Κεφάλαιο 6. Συμπεράσματα και κατευθύνσεις μελλοντικής έρευνας.

Κεφάλαιο 2

Λάμβδα λογισμός, επαγωγικοί τύποι και λογική

Ο λάμβδα λογισμός, στην αρχική του μορφή, είναι δημιούργημα του Alonso Church. Αναπτύχθηκε αρχικά στην δεκαετία του 1930 ως μέρος μιας θεωρίας που αποσκοπούσε στην θεμελίωση των μαθηματικών και της λογικής. Ενώ στη γενικότερη μορφή της η θεωρία ήταν ασυνεπής, παρουσιάζοντας κάτι ανάλογο με το παράδοξο του Russel, το κομμάτι που αφορούσε σε συναρτήσεις βρήκε σημαντικότατες εφαρμογές στην πληροφορική. Έχοντας αποδείξει ότι ο λ-λογισμός, όπως ονομάστηκε, είναι ισοδύναμος με την μηχανή Turing και ότι όλες οι αναδρομικές συναρτήσεις μπορούν να παρασταθούν σε αυτόν, αποτέλεσε έμπνευση για σχεδιασμό νέων αρχιτεκτονικών μοντέλων (μηχανές αναγωγής, υπολογιστές ροής δεδομένων).

Μια πιο σημαντική επιρροή του λ-λογισμού, δεδομένου ότι τα παραπάνω υπολογιστικά μοντέλα δεν έτυχαν ευρείας εφαρμογής, είναι η συνεισφορά του στη δημιουργία του συναρτησιακού προγραμματισμού. Ο John McCarthy σχεδίασε την γλώσσα Lisp εμπνευσμένος από τον λ-λογισμό.

Η σημαντικότερη συμβολή του όμως είναι στη μελέτη της σημασιολογίας των γλωσσών προγραμματισμού και των προβλημάτων σχεδίασης και υλοποίησής τους. Από τη δεκαετία του 1960 οι Christofer Strachey και Peter J. Landin παρατήρησαν ότι ο λ-λογισμός είναι πρόσφορος για την περιγραφή των σημασιολογικών ιδιοτήτων, ενώ αργότερα διατυπώθηκε η θεωρία πεδίων και θεμελιώθηκε το ερευνητικό πεδίο της σημασιολογίας γλωσσών προγραμματισμού.

Αρχικά η μορφή του ήταν χωρίς τύπους, όπου κάθε έκφραση μπορεί να εφαρμοστεί σε κάθε έκφραση. Αργότερα προτάθηκαν διάφορες παραλλαγές του με τύπους.

2.1 Λάμβδα λογισμός, χωρίς τύπους

2.1.1 Ορισμός

Ο λάμβδα λογισμός (λ-λογισμός) είναι μια θεωρία συναρτήσεων, όπου κάθε έκφραση θεωρείται συνάρτηση και μπορεί να εφαρμοστεί σε οποιαδήποτε έκφραση, συμπεριλαμβανομένου και του εαυτού της. Στην αρχική του μορφή, όπως προτάθηκε από τον Church, ο λ-λογισμός ήταν μια θεωρία χωρίς τύπους [Chur32, Chur33]. Η γραμματική του, σε μορφή BNF είναι:

$$e \ ::= \ V \mid \lambda V.\ e \mid e\ e$$

Συμβολίζουμε με Λ το σύνολο των όρων του λ-λογισμού, που ορίζεται από την παραπάνω γραμματική. Με V εννοούμε το σύνολο των μεταβλητών (variables), ενώ ο δεύτερος κανόνας ορίζει τις συναρτήσεις στον λ-λογισμό (abstraction). Οι συναρτήσεις δεν έχουν κάποιο συμβολικό όνομα, αλλά ορίζονται μόνο από το συμβολικό γράμμα λ και το όνομα της μεταβλητής. Εδώ αξίζει να αναφερθεί ότι οι μεταβλητές δεν έχουν κάποια φυσική σημασία, παρά μόνο χρησιμοποιούνται στον κανόνα αφαίρεσης συναρτήσεων και αντικαθίστανται κατά την εφαρμογή τους. Ο τρίτος κανόνας (application) αναφέρεται στην εφαρμογή μιας συνάρτησης σε κάποιον όρο. Όσον αφορά στην προτεραιότητα των τελεστών, όταν αυτή δεν καθορίζεται ρητά με παρενθέσεις θεωρούμε ότι η αφαίρεση εκτείνεται όσο το δυνατόν, ενώ η εφαρμογή είναι αριστερά προσεταιριστική. Για παράδειγμα

$$
\begin{array}{lcl}
FM_1M_2\ldots M_n & \equiv & (\ldots((FM_1)M_2)\ldots)M_n \\
\lambda x.\ M_1M_2\ldots M_n & \equiv & \lambda x.\ (M_1M_2\ldots M_n)
\end{array}
$$

2.1.2 Αντικατάσταση στον λ-λογισμό

Η αντικατάσταση στον λ-λογισμό είναι η βασική πράξη επεξεργασίας των λ-όρων και επηρεάζει τις ελεύθερες μεταβλητές. Με $M[x \mapsto N]$ συμβολίζουμε το αποτέλεσμα της αντικατάστασης στον όρο M όλων των ελεύθερων εμφανίσεων της μεταβλητής x με τον όρο N. Η αντικατάσταση ορίζεται επαγωγικά ως εξής:

$$
\begin{array}{lcll}
x[x \mapsto N] & \equiv & N & \\
y[x \mapsto N] & \equiv & y & \text{αν } y \neq x \\
(P\ Q)[x \mapsto N] & \equiv & P[x \mapsto N]\ Q[x \mapsto N] & \\
(\lambda x.\, P)[x \mapsto N] & \equiv & \lambda x.\, P & \\
(\lambda y.\, P)[x \mapsto N] & \equiv & \lambda y.\, P[x \mapsto N] & \text{αν } y \neq x \text{ και } (y \notin FV(N) \text{ ή } x \notin FV(P)) \\
(\lambda y.\, P)[x \mapsto N] & \equiv & \lambda z.\, P[y \mapsto z][x \mapsto N] & \text{αν } y \neq x \text{ και } y \in FV(N) \text{ και } x \in FV(P), \\
 & & & \text{όπου } z \notin (FV(P) \cup FV(N)
\end{array}
$$

2.1.3 Μετατροπές

Μια σχέση R μεταξύ όρων του λ-λογισμού λέγεται *συμβατή* όταν πληροί τις ακόλουθες προϋποθέσεις:

$$
M\ R\ N \quad \Rightarrow \quad \begin{cases} (\mathrm{Z\ M})\ \mathrm{R}\ (\mathrm{Z\ N}) \\ (\mathrm{M\ Z})\ \mathrm{R}\ (\mathrm{N\ Z}) \\ (\lambda x.\, M)\ \mathrm{R}\ (\lambda x.\, N) \end{cases}
$$

Στον λ-λογισμό ορίζονται τριών ειδών μετατροπές, οι οποίες αποτελούν συμβατές σχέσεις. Η απλούστερη είναι η α-αναγωγή. Ουσιαστικά συνίσταται στην μετονομασία της δεσμευμένης μεταβλητής σε μια αφαίρεση, υπό την προϋπόθεση η νέα μεταβλητή να μην είναι δεσμευμένη στην έκφραση στην οποία εφαρμόζεται. Για παράδειγμα:

$$\lambda x.\, xy \quad \rightarrow_\alpha \quad \lambda z.\, zy$$

Η ισότητα ως- προς-α ορίζεται ως το συμμετρικό, μεταβατικό και ανακλαστικό κλείσιμο του $\rightarrow_\alpha$ πάνω στο Λ.

$$X =_\alpha Y \quad \text{ανν} \quad X \rightarrow_\alpha X_1 \rightarrow_\alpha \ldots \leftarrow_\alpha \ldots \rightarrow_\alpha Y$$

Η δεύτερη μετατροπή του λ-λογισμού είναι η β-αναγωγή, που ουσιαστικά είναι η εφαρμογή μιας συνάρτησης σε έναν όρο. Ορίζεται ως η μικρότερη συμβατή σχέση πάνω στο Λ για την οποία, για κάθε M, $N \in \Lambda$ και για κάθε $x \in V$ ισχύει :

$$(\lambda x.\, M)\, N \quad \rightarrow_\beta \quad M[x \mapsto N]$$

Παραδείγματα β-μετατροπών :

$$
\begin{array}{lcl}
(\lambda x.\, zx)\, w & \rightarrow_\beta & z\ w \\
(\lambda x.\, \lambda y.\, zxy)\, w & \rightarrow_\beta & \lambda y.\, zw\ y
\end{array}
$$

Ορίζουμε ακόμα την ισότητα ως-προς-β ως το συμμετρικό, μεταβατικό και ανακλαστικό κλείσιμο του $\rightarrow_\beta$ πάνω στο Λ, και τη συμβολίζουμε με $=_\beta$

Η τρίτη μετατροπή είναι η η-μετατροπή, η οποία ουσιαστικά απλοποιεί τις συναρτήσεις εξαφανίζοντας περιττές αφαιρέσεις. Ορίζεται ως η μικρότερη συμβατή σχέση πάνω στον λ-λογισμό για την οποία για κάθε όρο $M \in \Lambda$ και για κάθε $x \in V$, τέτοια ώστε $x \notin FV(M)$, ισχύει:

$$\lambda x.\, M\ x \quad \to_\eta \quad M$$

Παραδείγματα η-μετατροπών είναι :

$$\begin{array}{lll} \lambda x.\, z\ x & \to_\eta & z \\ \lambda y.\, z\ x\ y & \to_\eta & z\ x \end{array}$$

Γενικά συμβολίζουμε με $\to$ την ένωση των $\to_\alpha$, $\to_\beta$ και $\to_\eta$, και με $\twoheadrightarrow$ ως το ανακλαστικό και μεταβατικό κλείσιμο της. Δηλαδή:

$$\begin{array}{lll} M \to N & \Rightarrow & M \twoheadrightarrow N \\ & & M \twoheadrightarrow M \\ M \twoheadrightarrow N, N \twoheadrightarrow P & \Rightarrow & M \twoheadrightarrow P \end{array}$$

Κατ' επέκταση ορίζεται και η ισότητα ως το συμμετρικό κλείσιμο του $\twoheadrightarrow$ πάνω στο Λ.

2.1.4 Κανονικές μορφές

Υπολογιστικά, την μεγαλύτερη σημασία στον λ-λογισμό την έχει η β-μετατροπή, αφού παριστάνει την εφαρμογή μιας συνάρτησης σε έναν όρο. Δευτερεύουσας σημασίας είναι η η-μετατροπή που ουσιαστικά απλοποιεί λ-όρους που παριστάνουν συναρτήσεις. Να σημειωθεί ότι συχνά στη βιβλιογραφία η η-μετατροπή αγνοείται εντελώς.

Ορίζουμε έναν όρο να βρίσκεται σε *κανονική μορφή* όταν δεν μπορεί να μετασχηματιστεί με β ή με η-μετατροπή.

Ένας όρος σε κανονική μορφή είναι πλήρως αποτιμημένος. Είναι προφανές ότι δεν λαμβάνεται υπ' όψιν η α-μετατροπή, διότι σε αυτή την περίπτωση όροι όπως ο $\lambda x.\, x$ δεν θα ήταν σε κανονική μορφή.

Παραδείγματα όρων σε κανονική μορφή είναι:
$\lambda f.\, \lambda x.\, x\ f$ και $\lambda x.\, x$, ενώ ο όρος

$$(\lambda z.\, \lambda f.\, \lambda x.\, z\ x)\ (\lambda y.\, y)$$

δεν είναι σε κανονική μορφή, αφού μπορεί να αναχθεί στο

$$(\lambda z.\, \lambda f.\, \lambda x.\, z\ x)\ (\lambda y.\, y) \quad \to_\beta \quad \lambda f.\, \lambda x.\, (\lambda y.\, y)\ x$$

Ορίζουμε ως ισχυρά κανονικοποιήσιμο έναν όρο που όλες οι ακολουθίες μετατροπής του καταλήγουν στην κανονική του μορφή. Να σημειωθεί ότι η κανονική μορφή ενός όρου είναι μοναδική (ως προς α μετατροπή), ενώ δεν είναι απαραίτητο κάθε ακολουθία αποτίμησης ενός όρου να καταλήγει σε κανονική μορφή, ακόμα και αν αυτή υπάρχει.

Παράδειγμα όρου που, ενώ έχει κανονική μορφή, μια ακολουθία αποτίμησης δεν καταλήγει ποτέ σε αυτήν είναι ο: $(\lambda z.\, y)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x))$

Αν ακολουθήσουμε την αριστερότερη αποτίμηση καταλήγουμε σε κανονική μορφή

$$(\lambda z.\, y)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x)) \quad \to_\beta \quad y$$

ενώ αν ακολουθούμε την δεξιά αποτίμηση δεν θα καταλήξει ποτέ σε κανονική μορφή.

$$(\lambda z.\, y)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x)) \quad \to_\beta \quad (\lambda z.\, y)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x)) \quad \to_\beta \quad \ldots$$

2.2 Δείκτες de Bruijn: ανώνυμες μεταβλητές

Ένα από τα μειονεκτήματα του λ-λογισμού είναι ο έλεγχος των ονομάτων των μεταβλητών και η μετονομασία τους όταν αυτό είναι απαραίτητο, όπως εύκολα διαπιστώνεται από τον ορισμό της αντικατάστασης. Επίσης ο έλεγχος ισότητας δύο όρων, ακόμα και αν αυτοί βρίσκονται σε κανονική μορφή, δεν είναι τετριμμένος γιατί πρέπει να επιτρέπει μετονομασίες μεταβλητών.

Έχουν γίνει αρκετές προτάσεις για αναπαράσταση του λ-λογισμού ώστε κάθε μεταβλητή να είναι μοναδική, όπως για παράδειγμα οι δεσμευμένες μεταβλητές να μετονομάζονται σε μοναδικά ονόματα ή κάποιες σε παραλλαγές του λ-λογισμού με συνδυαστική λογική (combinatory logic) [Curr68, Bare84] που βασίζεται σε combinators αντί για συναρτησιακή εφαρμογή.

Μια άλλη ιδέα διατυπώθηκε από τον Nicolas de Bruijn το 1972 [dBru72]. Ο de Bruijn πρότεινε κάθε μεταβλητή να παριστάνεται από έναν αριθμητικό δείκτη, έστω k. Κάθε δείκτης αντιστοιχεί στη k-οστή εσωτερική δέσμευση όρου. Έτσι, για παράδειγμα, έχουμε:

$$\begin{array}{lcl} \lambda x.\, x & \equiv & \lambda.\, 0 \\ \lambda x.\, \lambda y.\, x\,(y\,x) & \equiv & \lambda.\, \lambda.\, 1\,(0\,1) \end{array}$$

Χρησιμοποιώντας την αναπαράσταση με de Bruijn indices, η σύνταξη του λ-λογισμού γίνεται, σε μορφή BNF:

$$e \;::=\; k \mid \lambda.\, e \mid e\;e$$

Η χρήση αυτής της αναπαράστασης προϋποθέτει την χρήση και ενός *περιβάλλοντος* για κάθε όρο, αφού διαφορετικά ερμηνεύεται ένας όρος και διαφορετικά ο ίδιος όρος ως τμήμα κάποιου άλλου. Αν θεωρήσουμε ότι σε ένα περιβάλλον ο δείκτης 1 αντιστοιχεί στην μεταβλητή που είχαμε ονομάσει x, τότε στον $(\lambda : 0.)\,1$ όρο πράγματι αντιστοιχεί στην x. Όμως στον όρο $\lambda.\,\lambda.\,1$ ο δείκτης 1 αντιστοιχεί στην πρώτη δεσμευμένη μεταβλητή.

Προκειμένου να μπορούμε να εισαγάγουμε έναν όρο με ελεύθερες μεταβλητές μέσα σε μία αφαίρεση, ορίζουμε τη συνάρτηση *shift* που αυξάνει τον δείκτη στις ελεύθερες μεταβλητές ενός όρου. Την εφαρμογή αυτής της συνάρτησης σε ένα όρο τη συμβολίζουμε με $\uparrow_c^d (k)$.

$$\begin{array}{lcl} \uparrow_c^d (k) & = & \begin{cases} k & \text{αν } k \leq c \\ k+d & \text{αν } k > c \end{cases} \\ \uparrow_c^d (\lambda.\, t_1) & = & \lambda.\, \uparrow_{c+1}^d (t_1) \\ \uparrow_c^d (t_1\, t_2) & = & \uparrow_{c+1}^d (t_1)\; \uparrow_{c+1}^d (t_1) \end{array}$$

Με d παριστάνουμε τον αριθμό κατά τον οποίο θέλουμε να αυξήσουμε τους δείκτες των ελεύθερων μεταβλητών ενός όρου, και c είναι ο δείκτης από τον οποίο και πάνω θεωρούμε ότι πρόκειται για ελεύθερες, και όχι δεσμευμένες μεταβλητές. Αρχικά η συνάρτηση *shift* καλείται με $c = 0$. Με $\uparrow^d$ συμβολίζουμε το $\uparrow_0^d$.

Η αντικατάσταση κατά de Bruijn ορίζεται πολύ απλούστερα από ότι με χρήση ονομασμένων μεταβλητών:

$$\begin{array}{lcl} k[j \mapsto s] & = & \begin{cases} s & \text{αν } k = j \\ k & \text{αλλιώς} \end{cases} \\ (\lambda.\, t_1)[j \mapsto s] & = & \lambda.\, t_1[j+1 \mapsto \uparrow^1 (s)] \\ (t_1\, t_2)[j \mapsto s] & = & t_1[j \mapsto s]\, t_2[j \mapsto s] \end{array}$$

Πλέον πρέπει να ορίσουμε και την β-αναγωγή με de Bruijn indices:

$$(\lambda.\, t_1)\, v_2 \;\; \rightarrow_\beta \;\; \uparrow^{-1} (t_1[0 \mapsto \uparrow^1 v_2])$$

Στον όρο v_2 πρέπει να αυξηθούν κατά ένα οι δείκτες των ελεύθερων μεταβλητών του, αφού μπαίνει σε ένα περιβάλλον με δεσμευμένη μία ακόμα μεταβλητή. Στη συνέχεια, αφού γίνει η αντικατάσταση, η μεταβλητή παύει να είναι δεσμευμένη, οπότε χρειάζεται μείωση των δεικτών των ελεύθερων μεταβλητών κατά ένα.

Η πρόταση του Nicolas de Bruijn βρίσκει εφαρμογή σε υλοποιήσεις συστημάτων που βασίζονται στον λ-λογισμό για τις οποίες η απόδοση έχει καθοριστική σημασία. Για το λόγο αυτό θα χρησιμοποιηθεί στην παρούσα εργασία.

2.3 Λάμβδα λογισμός, με απλούς τύπους

2.3.1 Ορισμός

Αρχικά η έννοια των τύπων διατυπώθηκε από τον Curry (1934) σχετικά με τη θεωρία των κλειστών λ-όρων (λ-όροι χωρίς ελεύθερες μεταβλητές). Στην πορεία υπήρξε διαφοροποίηση μεταξύ του συστήματος τύπων που πρότεινε ο Curry και αυτού που πρότεινε ο Church [Chur40, Chur41]. Η ουσιαστικότερη διαφορά είναι ότι στο σύστημα του Curry, ο τύπος ενός όρου δεν είναι αυστηρά καθορισμένος, ενώ στου Church είναι. Εδώ ακολουθούμε την πρόταση του Church. Η γραμματική του $\lambda\!\rightarrow$ σε BNF μορφή:

$$\begin{aligned} \tau &::= \alpha \mid \tau \rightarrow \tau \\ e &::= V \mid \lambda V{:}\tau.e \mid e\,e \end{aligned}$$

όπου με τ συμβολίζουμε τους *τύπους* του $\lambda\!\rightarrow$. Ένας τύπος είναι είτε μια μεταβλητή τύπου, που συμβολίζεται με α, είτε ένας τύπος συνάρτησης Μέχρι τώρα, οι μόνοι περιορισμοί που είχαμε αφορούσαν στις α-μετατροπές, και την ονοματολογία των μεταβλητών. Με την εισαγωγή της έννοιας των τύπων, έχουμε και κάποιους επιπλέον περιορισμούς, τους οποίους διατυπώνουμε με κανόνες στην ακόλουθη μορφή:

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : D \qquad \ldots}{\Gamma \vdash E : F}$$

όπου ό,τι βρίσκεται κάτω από την γραμμή ισχύει όταν πληρούνται οι προϋποθέσεις που διατυπώνονται πάνω από την γραμμή. Ο συμβολισμός $\Gamma \vdash E : F$, σημαίνει ότι ο όρος E έχει τύπο F στο περιβάλλον Γ. Ως *περιβάλλον* ορίζουμε ένα σύνολο ζευγών της μορφής $x : \tau$, καθένα από τα οποία απεικονίζει μια μεταβλητή σε ένα τύπο.

Πίνακας 2.1: Κανόνες τύπων του $\lambda \rightarrow$.

αρχικός κανόνας	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$
απαλοιφή (elimination)	$\dfrac{\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$
εισαγωγή (introduction)	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\,M : A \rightarrow B}$

2.3.2 Παραδείγματα τύπων

Παρακάτω δίνονται μερικά παραδείγματα όρων με τους τύπους τους.

$$
\begin{array}{rcll}
 & \vdash & \lambda\alpha{:}A.\,\alpha & : \; A \to A \\
b : B & \vdash & \lambda\alpha{:}A.\,b & : \; A \to B \\
b : A & \vdash & (\lambda\alpha{:}A.\,\alpha)\,b & : \; A \\
c : A, b : B & \vdash & \lambda\alpha{:}A.\,b\,c & : \; B \\
 & \vdash & \lambda\alpha{:}A.\,\lambda b{:}B.\,\alpha & : \; A \to B \to A
\end{array}
$$

2.4 Λάμβδα λογισμός δευτέρας τάξης

2.4.1 Ορισμός

Η επέκταση του λ-λογισμού με τύπους, εξασφαλίζει απλά ότι θα εφαρμόζονται σωστά ορισμένες εκφράσεις, στις αντίστοιχες συναρτήσεις. Ο λ-2 ορίστηκε το 1971 από τον Girard [Gira72] και το 1974 από τον Reynolds [Reyn74] σε μία προσπάθεια να επεκτείνει το πρόβλημα "Curry-Hoard isomorphism" και αποτελεί μια η επέκταση του $\lambda \to$ ώστε να μπορούν να περιγραφούν πολυμορφικές συναρτήσεις. Η γραμματική του είναι:

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau \to \tau \mid \Pi\alpha{:}*.\,\tau \\
e & ::= & V \mid \lambda V{:}\tau.\,e \mid e\,e \mid \Lambda\alpha{:}*.\,e \mid e[\tau]
\end{array}
$$

Όπως και πριν, τ είναι το σύνολο των τύπων και e το σύνολο των εκφράσεων. Συντακτικά προστέθηκε η μορφή $\Pi\alpha{:}*.\,\tau$, που είναι ο τύπος πολυμορφικών εκφράσεων εξαρτώμενων από τύπους, και $\Lambda\alpha{:}*.\,\tau$ που ορίζει πολυμορφικές εκφράσεις. Με $e[\tau]$ συμβολίζουμε την εφαρμογή της πολυμορφικής έκφρασης e πάνω στον τύπο τ.

Στη βιβλιογραφία συχνά χρησιμοποιούνται τα σύμβολα $*$ ή Set προκειμένου να παραστήσουν τον τύπο των απλών τύπων και το σύμβολο $\forall$ αντί του Π. Στη μορφή $\Pi\alpha{:}*.\,\tau$ το α είναι μια μεταβλητή τύπου. Περισσότερες λεπτομέρειες σχετικά με υψηλότερου επιπέδου τύπους θα αναφερθούν στην επόμενη παράγραφο.

Προκειμένου να γίνει πιο κατανοητή η χρήση του λ-2 ακολουθούν μερικά παραδείγματα.

Η πολυμορφική ταυτοτική συνάρτηση	$\Lambda\alpha{:}*.\,\lambda x{:}\alpha.\,x$
Η πολυμορφική συνάρτηση apply	$\Lambda\alpha{:}*.\,\Lambda b{:}*.\,\lambda f{:}\alpha \to b.\,\lambda x{:}\alpha.\,f\,\alpha$

Πίνακας 2.2: Κανόνες τύπων του λ2.

αρχικός κανόνας	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$	
απαλοιφή (elimination)	$\dfrac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$	
εισαγωγή (introduction)	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\,M : A \to B}$	
Π-απαλοιφή	$\dfrac{\Gamma \vdash M : \Pi\alpha{:}*.\,A}{\Gamma \vdash M\,B : A[\alpha \mapsto B]}$	$, B \in T$
Π-εισαγωγή	$\dfrac{\Gamma \vdash M : A}{\Gamma \vdash \Lambda\alpha{:}*.\,M : \Pi\alpha{:}*.\,A}$	$, \alpha \notin FV(\Gamma)$

2.4.2 Παράσταση εκφράσεων

Ο λ-2 λογισμός είναι αρκετά εκφραστικός, ώστε να μπορεί να οριστεί το σύνολο των φυσικών ή ακεραίων αριθμών, καθώς επίσης και ο τύπος λίστας από οποιονδήποτε άλλο ορισμένο τύπο.

Όσον αφορά στους πραγματικούς αριθμούς, ορίζουμε τον τύπο *Nat* και δυο όρους *zero* και *succ* οι οποίοι παριστάνουν, αντίστοιχα το μηδέν και τη συνάρτηση που επιστρέφει το αμέσως επόμενο στοιχείο από αυτό που παίρνει ως παράμετρο.

$$\begin{array}{lcl} Nat & \equiv & \Pi X\colon *\,.\, X \to (X \to X) \to X \\ zero & \equiv & \Lambda X\colon *\,.\, \lambda x\colon X.\, \lambda f\colon X \to X.\, x \\ succ & \equiv & \lambda n\colon Nat.\, \Lambda X\colon *\,.\, \lambda x\colon X.\, \lambda f\colon X \to X.\, f\,(n\,[X]\,f) \end{array}$$

Ο παραπάνω ορισμός των φυσικών αριθμών, εύκολα επεκτείνεται και στους ακεραίους.

Ο τύπος *Bool* ορίζεται από δύο συναρτήσεις, μια που παριστάνει την αληθή και μία την ψευδή τιμή.

$$Bool \equiv \Pi A\colon *\,.\, A \to A \to A$$

Ο τύπος της λίστας των φυσικών αριθμών κατασκευάζεται από δύο συναρτήσεις, την *nil* που υλοποιεί την κενή λίστα και την *cons* που υλοποιεί την προσθήκη ενός φυσικού αριθμού στη λίστα.

$$\begin{array}{lcl} List_{Nat} & \equiv & \Pi A\colon *\,.\, A \to (Nat \to A \to A) \to A \\ nil_{Nat} & \equiv & \Lambda A\colon *\,.\, \lambda l\colon A.\, \lambda n\colon Nat \to A \to A.\, l \\ cons_{Nat} & \equiv & \lambda x\colon Nat.\, \lambda l\colon List.\, \Lambda A\colon *\,.\, \lambda n\colon A.\, \lambda c\colon Nat \to A \to A.\, c\,x\,(l\,[A]\,n\,c) \end{array}$$

Παρ' όλα αυτά στον λ-2 δεν μπορούμε να ορίσουμε κατά τον ίδιο τρόπο πολυμορφικούς τύπους, όπως για παράδειγμα πολυμορφικές λίστες.

2.4.3 Παραδείγματα τύπων

$$\begin{array}{rcl} & \vdash & \lambda\alpha\colon A.\, \alpha : A \to A \\ & \vdash & \Lambda t\colon *\,.\, \lambda\alpha\colon t.\, \alpha : \Pi t\colon *\,.\, t \to t \\ & \vdash & (\Lambda t\colon *\,.\, \lambda\alpha\colon t.\, \alpha)\, A : A \to A \\ b : A & \vdash & ((\Lambda t\colon *\,.\, \lambda\alpha\colon t.\, b)\, A)\, b : A \end{array}$$

2.5 Λάμβδα λογισμός ω-τάξης

2.5.1 Ορισμός

Η επιπλέον εκφραστική δυνατότητα που μας δίνει ο λ-λογισμός ω-τάξης [Gira72] συνίσταται στη δυνατότητα να ορίζουμε τύπους με παραμέτρους τύπους. Έτσι, για παράδειγμα, αντί για τον ορισμό ενός διαφορετικού τύπου λίστας για κάθε διαφορετικό τύπο στοιχείου, θα έχουμε τον ορισμό μίας πολυμορφικής λίστας, που παίρνει σαν παράμετρο τον τύπο των στοιχείων της. Η BNF γραμματική του λ-ω είναι:

$$\begin{array}{lcl} s & ::= & * \mid s \to s \\ \tau & ::= & \alpha \mid \tau \to \tau \mid \Pi\alpha\colon s.\, \tau \mid \lambda\alpha\colon s.\, \tau \mid \tau\,\tau \\ e & ::= & V \mid \lambda V\colon \tau.\, e \mid e\,e \mid \Lambda\alpha\colon s.\, e \mid e[\tau] \end{array}$$

Οι δύο νέες μορφές κανόνων στις μεταβλητές τύπων δίνουν τη δυνατότητα, όπως αναφέρθηκε, να κατασκευάζονται τύποι εξαρτώμενοι από τύπους. Ακόμα προστέθηκε ένα επιπλέον επίπεδο το οποίο παριστάνει τα *είδη* (kinds), δηλαδή τους τύπους των τύπων.

2.5.2 Παράσταση εκφράσεων

Ο πολυμορφικός τύπος λίστας ακολουθεί ακριβώς την ίδια λογική με τον τύπο λίστας φυσικών αριθμών, με επιπλέον παράμετρο τον τύπο των στοιχείων.

$$
\begin{array}{lcl}
List & \equiv & \lambda\beta{:}*.\,\Pi A{:}*\to *. \\
 & & \qquad (\Pi\alpha{:}*.\,A\,\alpha) \\
 & & \qquad \to (\Pi\alpha{:}*.\,\alpha\to A\,\alpha\to A\,\alpha) \\
 & & \qquad \to A\,\beta \\
nil & \equiv & \Lambda\beta{:}*.\,\Lambda A{:}*\to *. \\
 & & \qquad \lambda n{:}(\Pi\alpha{:}*.\,A\,\alpha). \\
 & & \qquad \lambda c{:}(\Pi\alpha{:}*.\,\alpha\to A\,\alpha\to A\,\alpha). \\
 & & \qquad n\,[\beta] \\
cons & \equiv & \Lambda\beta{:}*.\,\lambda x{:}\beta.\,\lambda l{:}List\,\beta. \\
 & & \qquad \Lambda A{:}*\to *. \\
 & & \qquad \lambda n{:}(\Pi\alpha{:}*.\,A\,\alpha). \\
 & & \qquad \lambda c{:}(\Pi\alpha{:}*.\,\alpha\to A\,\alpha\to A\,\alpha). \\
 & & \qquad c\,[\beta]\,x\,(l\,[\beta]\,n\,c)
\end{array}
$$

2.6 Λογισμός κατασκευών

2.6.1 Ορισμός

Ο λογισμός κατασκευών (Calculus of Constructions—CC) [Coqu88, Coqu90] αποτελεί την επέκταση των παραπάνω συστημάτων, με την προσθήκη της εξάρτησης τύπων από όρους. Η γραμματική του σε μορφή BNF είναι:

$$
\begin{array}{lcl}
s & ::= & * \mid \Pi V{:}\tau.\,s \mid \Pi\alpha{:}s.\,s \\
\tau & ::= & \alpha \mid \Pi V{:}\tau.\,\tau \mid \Pi\alpha{:}s.\,\tau \mid \lambda\alpha{:}s.\,\tau \mid \tau\,\tau \mid \lambda V{:}\tau.\,\tau \mid \tau\,e \\
e & ::= & V \mid \lambda V{:}\tau.\,e \mid e\,e \mid \Lambda\alpha{:}s.\,e \mid e[\tau]
\end{array}
$$

Όπως αναφέρθηκε, οι προσθήκες σε σχέση με τον λ λογισμό ω-τάξης, είναι η ύπαρξη εξαρτώμενων συναρτήσεων μεταξύ ειδών ($\Pi\alpha{:}s.\,s$), η ύπαρξη τύπων εξαρτωμένων από όρους ($\Pi V{:}\tau.\,s$, $\lambda V{:}\tau.\,\tau$, $\tau\,e$) και η ύπαρξη εξαρτώμενων συναρτήσεων μεταξύ όρων ($\Pi V{:}\tau.\,\tau$).

2.7 Αμιγή συστήματα τύπων

2.7.1 Ορισμός

Τα αμιγή συστήματα τύπων (pure type systems) [Bare92, Bera88] μπορούν να χρησιμοποιηθούν για να ορίσουν ένα σύνολο από παραλλαγές του λ-λογισμού με τύπους, το οποίο ονομάζεται λ-κύβος (ή κύβος του Barendregt) και συμπεριλαμβάνει τα συστήματα λ→, λ-2, λ-ω και CC που ορίστηκαν στις προηγούμενες ενότητες. Τα συστήματα αυτά μπορούν να οριστούν με την παρακάτω κοινή σύνταξη, η οποία δεν κάνει διάκριση μεταξύ όρων, τύπων και ειδών.

$$T \;::=\; * \mid \Box \mid V \mid T\,T \mid \lambda V{:}T.\,T \mid \Pi V{:}T.\,T$$

Το σύμβολο $*$ παριστάνει το είδος, δηλαδή τον τύπο των τύπων, και ομοίως το σύμβολο $\Box$ παριστάνει τον τύπο των ειδών, που θα βρισκόταν ένα επίπεδο πάνω από τα είδη. Το σύμβολο V παριστάνει μεταβλητές όρων και τύπων, χωρίς διάκριση.

Ένα αμιγές σύστημα τύπων ορίζεται από μια τριάδα (S, A, R), όπου:

S είναι το σύνολο των σταθερών του συστήματος, όπως είναι το $*$.
A είναι το σύνολο αξιωμάτων της μορφής $s_1 : s_2$, όπου $s_1, s_2 \in S$
R είναι το σύνολο των τριάδων από στοιχεία του S, που ορίζει τα είδη συναρτήσεων που μπορούν να κατασκευαστούν και σε ποιο στοιχείο του S αντιστοιχεί ο τύπος κάθε μίας.

Ένα αμιγές σύστημα τύπων περιγράφεται από τους κανόνες που ακολουθούν, όπου το s παίρνει τιμές από το σύνολο S. Σε αυτούς τους κανόνες, κάθε περιβάλλον Γ περιέχει απεικονίσεις τόσο μεταβλητών όρων σε τύπους όσο και μεταβλητών τύπων σε είδη.

Πίνακας 2.3: Κανόνες τύπων του $\lambda(S, A, R)$.

αξίωμα	$\dfrac{}{\vdash c : s}$	αν $c : s \in A$
αρχικός κανόνας	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \Gamma$
κανόνας αποδυνάμωσης	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$	$x \notin \Gamma$
κανόνας σχηματισμού τύπων	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.\, B : s_3}$	αν $(s_1, s_2, s_3) \in R$
κανόνας εφαρμογής	$\dfrac{\Gamma \vdash F : \Pi x{:}A.\, B \quad \Gamma \vdash \alpha : A}{\Gamma \vdash F\,\alpha : B[x \mapsto \alpha]}$	
κανόνας συνάρτησης	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash \Pi x{:}A.\, B : s}{\Gamma \vdash \lambda x{:}A.\, b : \Pi x{:}A.\, B}$	
κανόνας μετατροπής	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$	

2.7.2 Παράσταση συστημάτων λ-λογισμού με PTS

Αν, για συντομία, θεωρήσουμε οτι $(s_1, s_2) \equiv (s_1, s_2, s_2)$, τότε τα προηγούμενα ορισμένα συστήματα, παριστάνονται ως εξής:

$\lambda \rightarrow$

$$\begin{aligned} S &= \{*, \Box\} \\ A &= \{* : \Box\} \\ R &= \{*, *\} \end{aligned}$$

λ-2

$$\begin{aligned} S &= \{*, \Box\} \\ A &= \{* : \Box\} \\ R &= \{(*, *), (\Box, *)\} \end{aligned}$$

λ-ω

$$\begin{aligned} S &= \{*, \Box\} \\ A &= \{* : \Box\} \\ R &= \{(*, *), (\Box, *), (\Box, \Box)\} \end{aligned}$$

$$\begin{array}{lcl} S & = & \{*, \Box\} \\ A & = & \{* : \Box\} \\ R & = & \{(*,*), (\Box,*), (*,\Box), (\Box,\Box)\} \end{array}$$

2.8 Λογισμός επαγωγικών κατασκευών

Στα παραπάνω συστήματα ορίσαμε μια σειρά από αναπαραστάσεις εκφράσεων, τόσο με πεπερασμένο αριθμό στοιχείων (*Bool*) όσο και και με άπειρο (φυσικοί αριθμοί, λίστες). Με ανάλογο τρόπο θα μπορούσαμε να αναπαραστήσουμε και συναρτήσεις εφαρμογής σε τέτοιους όρους, όπως για παράδειγμα συνάρτηση άθροισης ή πολλαπλασιασμού για φυσικούς αριθμούς, συναρτήσεις not, and και or για Booleans και συναρτήσεις εύρεσης μήκους και εύρεσης ν-οστού στοιχείου για λίστες.

Παρ' όλα αυτά, αυτή η υλοποίηση έχει σημαντικά μειονεκτήματα:

- Ενώ ορίζονται διακριτά στοιχεία, π.χ. *true* και *false* για *Boolean*, δεν υπάρχει καμία απόδειξη οτι αυτά δεν ταυτίζονται. Αντίστοιχα και στην περίπτωση των φυσικών αριθμών δεν μπορεί κανείς να αποδείξει οτι το μηδέν δεν ταυτίζεται με το ένα.

- Δεν υπάρχει απόδειξη της επαγωγικής ιδιότητας των στοιχείων. Για τους φυσικούς αριθμούς, αν και μπορούμε να ορίσουμε συνάρτηση *less* , δεν μπορούμε να αποδείξουμε την ιδιότητα $less\ n\ (succ\ n)$

- Κάθε συνάρτηση που βρίσκει το προηγούμενο στοιχείο, καταναλώνει τουλάχιστον γραμμικό χρόνο. Για παράδειγμα στους φυσικούς αριθμούς η συνάρτηση $prev(n+1) = n$ υπολογίζεται σε n βήματα.

2.8.1 Ορισμός

Προκειμένου να αντιμετωπιστούν αυτά τα προβλήματα ορίστηκε μια γλώσσα η οποία μαζί με το ισχυρό σύστημα τύπων, που παρείχαν όλα τα προηγούμενα ορισμένα συστήματα, έχει και έναν αποδοτικό μηχανισμό υποστήριξης επαγωγικών ορισμών. Ο *λογισμός των επαγωγικών κατασκευών (Calculus of Inductive Constructions—CIC)* [Pfen90, Paul93, Paul96, Wern94] αποτελεί επέκταση του CC και η γραμματική του σε BNF μορφή είναι:

$$\begin{array}{lcl} T & ::= & * \mid \Box \mid V \mid T\,T \mid \lambda V{:}T.T \mid \Pi V{:}T.T \\ & \mid & \mathsf{Ind}(V{:}T)\{\underline{T}\} \mid \mathsf{Constr}(n,T) \mid \mathit{Elim}(T:T)\{\underline{T}\} \end{array}$$

Όπως στα αμιγή συστήματα τύπων, στο λογισμό επαγωγικών κατασκευών δεν γίνεται διαχωρισμός μεταξύ τύπων και όρων, αλλά δομείται σε επίπεδα. Αυτό θα φανεί καλύτερα σε επόμενο κεφάλαιο, στον ορισμό της *nflint* , της γλώσσας τύπων που υλοποιήθηκε. Με C συμβολίζουμε τις σταθερές, δηλαδή τα στοιχεία $*$ και $\Box$. Ο κανόνας *Ind* ορίζει έναν νέο αναδρομικό τύπο παρέχοντας τους τύπους των κατασκευαστών του, ο *Constr* ορίζει τους κατασκευαστές ενός τύπου, ενώ ο *Elim* ορίζει την ι-αναγωγή, δηλαδή έναν τρόπο αποσύνθεσης των επαγωγικών τύπων, που συγχρόνως υλοποιεί την αναδρομή και την επαγωγή.

Συχνά, αντί του $\Pi x{:}A.B$, όταν η μεταβλητή x δεν εμφανίζεται στον όρο A, τότε παριστάνουμε τον παραπάνω όρο ως $A \to B$.

2.8.2 Παραδείγματα

Ο τύπος *Nat* ορίζεται στον CIC ως εξής:

$$
\begin{array}{lcll}
\mathsf{Nat} & \equiv & \mathsf{Ind}(X:*)\{X \mid X \to X\} & \\
\mathsf{zero} & \equiv & \mathsf{Constr}(0, \mathsf{Nat}) & : \mathsf{Nat} \\
\mathsf{succ} & \equiv & \mathsf{Constr}(1, \mathsf{Nat}) & : \mathsf{Nat} \to \mathsf{Nat}
\end{array}
$$

Η συνάρτηση *sum* ορίζεται ως εξής:

$$
\mathsf{sum} \equiv \lambda n{:}\mathsf{Nat}.\, \lambda \mathsf{m}{:}\mathsf{Nat}.\, \mathsf{Elim}(\mathsf{n} : Nat)\{\mathsf{m} \mid \lambda \mathsf{y}{:}\mathsf{Nat}.\, \mathsf{succ}\}
$$

Ο κανόνας *Elim* αποτιμάται ως εξής:

$$
\begin{array}{l}
\mathsf{Elim}(\mathsf{zero}, \mathsf{Nat})\{\mathsf{m} \mid \lambda \mathsf{y}{:}\mathsf{Nat}.\, \mathsf{succ}\} \quad \to_\iota \quad \mathsf{m} \\
\mathsf{Elim}(\mathsf{succ\,k}, \mathsf{Nat})\{\mathsf{m} \mid \lambda \mathsf{y}{:}\mathsf{Nat}.\, \mathsf{succ}\} \\
\quad \to_\iota \quad (\lambda \mathsf{y}{:}\mathsf{Nat}.\, \mathsf{succ})\, \mathsf{k}\, \mathsf{Elim}(\mathsf{k}, Nat)\{\mathsf{m} \mid \lambda \mathsf{y}{:}\mathsf{Nat}.\, \mathsf{succ}\} \\
\quad \to_\beta \quad \mathsf{succ}\, \mathsf{Elim}(\mathsf{k}, \mathsf{Nat})\{\mathsf{m} \mid \lambda \mathsf{y}{:}\mathsf{Nat}.\, \mathsf{succ}\}
\end{array}
$$

Περισσότερες λεπτομέρειες σχετικά με τον λογισμό των αναδρομικών κατασκευών αναφέρονται στο κεφάλαιο που περιγράφει την γλώσσα *nflint* , που αποτελεί μια παραλλαγή ενός τέτοιου συστήματος.

2.9 Προτάσεις ως τύποι

Ο λ-λογισμός και ιδιαίτερα οι επεκτάσεις του έχουν βρει εφαρμογή στη θεωρία αποδείξεων [Curr68, Howa80, Gira89, Geuv92].

Η θεωρία τύπων της οποίας βάση αποτελεί ο λ-λογισμός, αποτελεί ένα σύστημα που περικλείει τόσο την έννοια του *υπολογισμού*, μέσω των όρων συναρτήσεων σε λ-λογισμό με τύπους, όσο και την έννοια των *αποδείξεων* όπου οι τύποι μπορούν να θεωρηθούν ως προτάσεις και οι όροι αποδείξεις των προτάσεων.

Χρησιμοποιούμε τον συμβολισμό:

$$\Gamma \vdash_L A$$

$$Proof.p$$

για μια μαθηματική απόδειξη, όπου Γ είναι το σύνολο από υποθέσεις που κάνουμε και A είναι η προς απόδειξης πρόταση. Με p συμβολίζουμε την απόδειξη της πρότασης. Μια τέτοια απόδειξη μπορούμε να την αντιστοιχήσουμε σε λ θεωρώντας τις προτάσεις ως τύπους και την απόδειξη ενός τύπου ως έναν όρο που κατοικεί σε αυτόν τον τύπο (δηλαδή έχει αυτόν τον τύπο).

$$
\begin{array}{lcl}
\vdash \mathrm{A}\text{ , με απόδειξη p} & \Leftrightarrow & \vdash \quad \mathrm{p} \quad : \quad A \\
\text{το A είναι αποδείξιμο} & \Leftrightarrow & \text{το A είναι κατοικημένο}
\end{array}
$$

οπότε ο συμβολισμός στην γλώσσα τύπων είναι

$$\Gamma \vdash \quad p \quad : \quad A$$

Ένα σύστημα απόδειξης τύπων συνήθως περιλαμβάνει και κάποιο σύστημα ανάπτυξης των όρων—αποδείξεων, συγκρίνοντας τελικά τον τύπο του όρου που κατασκευάστηκε με την προς απόδειξη πρόταση—τύπο.

Σύμφωνα με τη θεωρία αποδείξεων, βασισμένη σε συστήματα τύπων, μπορούμε αντιστοιχίσουμε μερικές βασικές πράξεις της λογικής σε λ-όρους.

$$
\begin{array}{lcl}
A \supset B & = & A \to B \\
A \wedge B & = & A \times B \\
A \vee B & = & A \cup B \\
\bot & = & \varnothing \\
\forall x \in X. Ax & = & \Pi x{:}X.\, A\, x \\
\exists x \in X. Ax & = & \Sigma x{:}X.\, A\, x
\end{array}
$$

Ο όρος $\Sigma x{:}A.\,B$ παριστάνει το άθροισμα, το οποίο δεν έχει οριστεί στις προεκτάσεις του λ-λογισμού που παρουσιάστηκαν Σε ένα σύστημα τύπων στο οποίο δεν είναι ορισμένο μπορούμε να εκφράσουμε τον υπαρξιακό ποσοδείκτη σε σχέση με τον καθολικό ποσοδείκτη:

$$\exists x \in X.Ax \quad = \quad \neg(\forall x \in X.\neg Ax)$$

Μια απόδειξη της πρώτης σχέση ($\supset$) ερμηνέυεται ως μια μέθοδος που μετασχηματίζει μια απόδειξη του A σε μία απόδειξη για το B. Μια απόδειξη της δεύτερης είναι ένα ζεύγος $< p, q >$, τέτοιο ώστε το p να είναι απόδειξη του A και το q του B. Αντίστοιχα για τη σχέση $A \vee B$ μια απόδειξη είναι είτε απόδειξη του A είτε του B. Για τη πρόταση $\perp$, που στην βιβλιογραφία αναφέρεται με την ονομασία *bottom*, δεν υπάρχει απόδειξη, ενώ για τον καθολικό ποσοδείκτη, η απόδειξη είναι μια συνάρτηση που μετασχηματίζει κάθε όρο $a \in A$ σε έναν όρο $A\,\alpha$.

Ένα παράδειγμα πρότασης είναι το εξής:

$$\forall x, y \in X.Rxy \quad \supset \quad \neg Ryx$$

όπου R είναι μια δυαδική σχέση και X ένα σύνολο. Αν υποθέσουμε οτι ισχύει το παραπάνω, τότε αποδεικνύεται ότι $\forall x \in X.\neg Rxx$, δηλαδή οτι μία αντισυμμετρικη σχέση είναι μη αντανακλαστική.

Το σύστημα που υλοποιήθηκε, και αποτελεί το θέμα αυτής της εργασίας, ενσωματώνει κάποιες απλές αποδείξεις θεωρημάτων αλλά δεν έχει την δυνατότητα να βοηθήσει τον χρήστη στην κατασκευή μιας απόδειξης.

Κεφάλαιο 3

Περιγραφή της γλώσσας τύπων nflint

3.1 Ορισμός

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 $\Box$. 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
- AB που ορίζει την εφαρμογή ενός όρου σε όρο
- $\mathsf{Ind}(X\!:\!A)\{\vec{A}\}$ 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:

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

ενώ με $\underline{\mathrm{A}}$ 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:

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

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

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.

3.2 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 .

3.2.1 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:

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

3.2.2 Αναγωγές

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

$$
\begin{array}{ll}
(\lambda\, X\!:\!A.\, B)\, A' & \rightarrow_\beta \quad B[X \mapsto A'] \\
\lambda\, X\!:\!A.\, B\, X & \rightarrow_\eta \quad B \\
\quad (\textit{αν } X \notin \mathsf{FV}(B)) & \\
\mathsf{Elim}[A'](\mathsf{Constr}(I, B)\, \vec{A}'\!:\!B\, \vec{B})\{\underline{A}\} & \rightarrow_\iota \\
\qquad \Phi_{X,I}(C_i, A_i, \lambda\, \vec{X}\!:\!\vec{B}'.\, \lambda\, Y\!:\!B\, \vec{X}.\, \mathsf{Elim}[A'](Y\!:\!B\, \vec{X})\{\underline{A}\})\, \vec{A}' & \\
\quad (\textit{όπου } B = \mathsf{Ind}(X\!:\!B')\{\underline{C}\} \textit{ και } B' = \Pi\, \vec{X}\!:\!\vec{B}'.\, \mathsf{Set}) &
\end{array}
$$

Η συνάρτηση Φ ορίζεται ως:

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

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

3.2.3 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.

3.3 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.

3.3.1 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).

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

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

- 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.

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

- Πολλαπλές εκφράσεις

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

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

3.3.2 Μεταβλητές και σταθερές

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

3.3.3 Ισοδυναμία τύπων

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

3.3.4 Γινόμενο, αφαίρεση και εφαρμογή

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

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

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

3.3.5 Επαγωγικοί ορισμοί.

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

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

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

3.4 Μετα-θεωρία της γλώσσας τύπων

Τα παρακάτω θεωρήματα μπορούν να αποδειχθούν για τη γλώσσα των τύπων. Οι αποδείξεις τους για το λογισμό των επαγωγικών κατασκευών περιέχονται στη διδακτορική διατριβή του Benjamin Werner [Wern94]. Αυτές μπορούν να προσαρμοστούν για τη γλώσσα *nflint*, όπως φαίνεται στην εργασία των Shao *et al.* [Shao02]. Αξίζει να σημειωθεί ότι οι αποδείξεις των παρακάτω πρέπει να γίνουν σε μεγάλο βαθμό από κοινού, καθώς το σύστημα τύπων και οι αναγωγές είναι πολύ στενά συνδεδεμένες, π.χ. μέσω του κανόνα της μετατροπής.

Η απροκρισιμότητα του ελέγχου τύπων και η μοναδικότητα των τύπων είναι απαραίτητες προϋποθέσεις για τον καλό ορισμό του συστήματος τύπων της *nflint*. Προφανώς, οι τύποι είναι μοναδικοί ως προς $=_{\alpha\beta\eta}$.

Θεώρημα 1 (Αποκρισιμότητα του Ελέγχου Τύπων): *Υπάρχει αλγόριθμος ο οποίος, δεδομένου ενός όρου A και ενός περιβάλλοντος Δ:*

- *είτε αποφασίζει ότι δεν υπάρχει όρος B τέτοιος ώστε $\Delta \vdash A : B$,*
- *ή επιστρέφει έναν όρο B τέτοιον ώστε $\Delta \vdash A : B$.*

Θεώρημα 2 (Μοναδικότητα των Τύπων): *Αν $\Delta \vdash A : B$ και $\Delta \vdash A : B'$, τότε $B = B'$.*

Τα παρακάτω θεωρήματα αφορούν στις σχέσεις αναγωγής και τις συνδέουν με τον έλεγχο τύπων. Τα θεωρήματα συμβολής και Church-Rosser ισχύουν μόνο για καλά διατυπωμένους όρους.

Θεώρημα 3 (Συμβολή): *Έστω* $\Delta \vdash A : B$. *Αν* $A \twoheadrightarrow A_1$ *και* $A \twoheadrightarrow A_2$, *τότε υπάρχει ένας όρος* A' *τέτοιος ώστε* $A_1 \twoheadrightarrow A'$ *και* $A_2 \twoheadrightarrow A'$.

Θεώρημα 4 (Church-Rosser): *Έστω* $\Delta \vdash A_1 : B$ *και* $\Delta \vdash A_2 : B$. *Αν* $A_1 = A_2$, *τότε υπάρχει ένας όρος* A' *τέτοιος ώστε* $A_1 \twoheadrightarrow A'$ *και* $A_2 \twoheadrightarrow A'$.

Θεώρημα 5 (Ορθότητα των Αναγωγών): *Αν* $\Delta \vdash A : B$ *και* $A \twoheadrightarrow A'$, *τότε* $\Delta \vdash A' : B$.

Ένας όρος A λέγεται *κανονική μορφή* αν δεν υπάρχει όρος A' τέτοιος ώστε $A \to A'$. Ένας όρος A λέγεται *κανονικοποιήσιμος* αν υπάρχει κανονική μορφή B τέτοια ώστε $A \twoheadrightarrow B$. Ένας όρος A λέγεται *ισχυρά κανονικοποιήσιμος* αν είναι κανονικοποιήσιμος και δεν υπάρχει άπειρη ακολουθία $\to$-αναγωγών που να ξεκινάει από αυτόν τον όρο.

Το τελευταίο ενδιαφέρον θεώρημα αποδεικνύει ότι κάθε σειρά αναγωγών που ξεκινά από έναν καλά διατυπωμένο όρο τερματίζεται.

Θεώρημα 6 (Ισχυρή Κανονικοποίηση): *Αν* $\Delta \vdash A : B$, *τότε ο όρος* A *είναι ισχυρά κανονικοποιήσιμος.*

Κεφάλαιο 4

Θέματα υλοποίησης

4.1 Σύνοψη

Η υλοποίηση επικεντρώθηκε τόσο στη δυνατότητα να μπορεί το σύστημα να επεκταθεί αργότερα με ένα φιλικό προς το χρήστη περιβάλλον, όσο και στην ταχύτητα εκτέλεσης. Με γνώμονα τα παραπάνω, χρησιμοποιήθηκαν τεχνικές που στη βιβλιογραφία αναφέρονται ως ιδιαίτερα αποδοτικές [Shao98]. Κατ' αρχήν, παρατηρήθηκε ότι οι υπο-όροι συχνά επαναλαμβάνονταν παρά πολλές φορές σε έναν όρο, με αποτέλεσμα να καταναλώνεται άσκοπα χώρος στη μνήμη. Ακόμα, οι απλές αντικαταστάσεις μεταβλητών χρειάζονται υπερβολικά πολύ υπολογιστική δυνατότητα, αν λάβουμε υπ' όψιν τον αριθμό αυτών που απαιτούνται να γίνουν στη αποτίμηση μιας έκφρασης.

Με γνώμονα τα παραπάνω, στην υλοποίηση χρησιμοποιήθηκε η τεχνική του hash-consing, η οποία αποσκοπεί στην αποθήκευση κάθε όρου στη μνήμη μόνο μία φορά. Για κάθε όρο χρησιμοποιείται στην ουσία ένας δείκτης στον πραγματικό όρο, χωρίς να μεταβάλλεται ποτέ ο όρος, αλλά μόνο ο δείκτης. Έτσι για παράδειγμα, κατά την αντικατάσταση μιας μεταβλητής από μια άλλη, δεν μεταβάλλεται η ίδια η μεταβλητή, αλλά αντικαθίσταται ο δείκτης που έδειχνε σε αυτή με ένα δείκτη που δείχνει στην νέα μεταβλητή. Επιπλέον, θεωρήθηκε άσκοπο να γίνονται διαρκώς υπολογισμοί μεταβλητών που παριστάνονται από αλφαριθμητικά. Όπως αναφέρθηκε και στον κεφάλαιο του λ-λογισμού, ιδιαίτερα χρήσιμη είναι η τεχνική που προτείνει ο Nikolas de Bruijn, όπου κάθε μεταβλητή αναπαριστάται από έναν ακέραιο αριθμό.

Η υλοποίηση αποτελείται από τα εξής επιμέρους κομμάτια:

- Hash-consing
  Είναι μια παραμετρική υλοποίηση γενικής χρήσης, η οποία χρησιμοποιείται σχεδόν για όλα τα υπόλοιπα κομμάτια του κώδικα.

- Ονόματα και σταθερές
  Υλοποίηση για το hashconsing των ονομάτων των σταθερών, και αποθήκευση των όρων και των τύπων τους με το βέλτιστο δυνατό τρόπο.

- Πρωτογενείς όροι
  Όροι, στη κανονική μορφή της γλώσσας, δηλαδή με ονομασμένες μεταβλητές, έτσι ώστε να δίνονται από το χρήστη σε πιο αναγνώσιμη μορφή.

- Όροι
  Πρόκειται για τους όρους της γλώσσας, όπως χρησιμοποιούνται στην υλοποίηση μας, με de Bruijn indices.

- Κανονικές μορφές
  Ορισμός της κανονικής μορφής και της ασθενούς κανονικής μορφής κεφαλής για την *nflint* .

- Έλεγχος τύπων
  Πρόκειται για το ουσιαστικότερο κομμάτι της γλώσσας. Ελέγχει κάθε ένα τους όρους αν είναι σωστά διατυπωμένοι, και εμφανίζει κατάλληλα μηνύματα σφάλματος.

- Αποτίμηση
  Μετατροπή στη κανονική μορφή της έκφρασης.

4.2 Γλώσσα υλοποίησης

Η γλώσσα υλοποίησης είναι η OCaml, μια επέκταση της ML η οποία αναπτύσσεται από το *Γαλλικό Ινστιτούτο έρευνας της επιστήμης των υπολογιστών, INRIA, http://www.inria.fr.*

Πρόκειται για μια συναρτησιακή γλώσσα προγραμματισμού η οποία ενσωματώνει και στοιχεία προστακτικού προγραμματισμού, επιτρέποντας έτσι παρενέργειες (side-effects). Η διάκριση μεταξύ του συναρτησιακού και του προστακτικού μέρους γίνεται ρητά από την ίδια τη γλώσσα. Ένα από τα χαρακτηριστικά της γλώσσας που την καθιστούν καταλληλότερη από οποιαδήποτε άλλη συναρτησιακή γλώσσα είναι ότι αποτιμά τους όρους πρόθυμα (eager evaluation). Η γλώσσα ενσωματώνει ακόμα, στοιχεία αντικειμενοστραφούς προγραμματισμού, τα οποία όμως δεν χρησιμοποιήθηκαν από τη παρούσα υλοποίηση.

Μερικά από τα χαρακτηριστικά της γλώσσας είναι ότι κάνει πολλούς ελέγχους ακεραιότητας πριν παράγει το τελικό εκτελέσιμο, ενώ συγχρόνως παρέχει και ένα ακέραιο σύστημα τύπων, δίνοντας τη δυνατότητα στον προγραμματιστή, εκτός από τους βασικούς και τους πιο πολύπλοκους τύπους δεδομένων, να κατασκευάσει νέους. Το σύστημα τύπων που ενσωματώνει η γλώσσα δεν απαιτεί από τον προγραμματιστή να καθορίσει ρητά τύπους για κάθε έκφραση, αλλά καθορίζονται αυτόματα από τον ίδιο τον μεταγλωττιστή. Ακόμα, η δέσμευση και αποδέσμευση χώρου στη μνήμη γίνεται αυτόματα, χωρίς να χρειάζεται ο προγραμματιστής να χρησιμοποιήσει τις, επικίνδυνες για την ασφάλεια του προγράμματος, malloc και free, ενώ η συλλογή αποδεσμευμένης μνήμης (garbage collection) γίνεται αυτόματα. Αυτό είναι και ένα από τα χαρακτηριστικά των συναρτησιακών γλωσσών γενικότερα, και ένας από τους λόγους που η υλοποίηση έγινε σε μια τέτοια γλώσσα και όχι σε μία προστακτική (imperative).

Η επιλογή της γλώσσας έγινε λόγω της ασφάλειας που τη διακρίνει και της ταχύτητας των εκτελέσιμων που παράγει. Χαρακτηριστικά αξίζει να αναφερθεί ότι με μια οκνηρής- αποτίμησης (lazy evaluation) γλώσσα ο χρόνος εκτέλεσης θα πολλαπλασιαζόταν, καθώς επίσης και με μία πιο ελεύθερη γλώσσα, όπως η C, η διόρθωση λαθών σχετιζόμενα με διαχείριση μνήμης θα ήταν σχεδόν αδύνατη.

4.3 Hash-consing

Το hashconsing υλοποιήθηκε ως ένα αυτόνομο και πολυμορφικό κομμάτι της υλοποίησης με σκοπό να χρησιμοποιηθεί σε κάθε κομμάτι της.

Συνοπτικά θα μπορούσε να αναφερθεί ότι δίνοντας του ως δεδομένα τον τύπο ενός όρου, και μια συνάρτηση σύγκρισης, αναλαμβάνει την αποθήκευση των στοιχείων αυτού του όρου.

Η γλώσσα OCaml παρέχει μια υλοποίηση για πίνακες κατακερματισμού, που είναι αρκετά συναφής με το hash-consing, αλλά έχει αρκετά μειονεκτήματα. Η συνήθης υλοποίηση των πινάκων κατακερματισμού υπολογίζει για κάθε όρο την συνάρτηση κατακερματισμού δύο φορές. Μία για την αναζήτηση και μία για την αποθήκευση, ενώ δεν αποθηκεύει πουθενά κάποιο χαρακτηριστικό των όρων ώστε να γίνεται αποδοτικότερα ο χειρισμός σε λειτουργίες που θα προστεθούν αργότερα. Για παράδειγμα, στη περίπτωση που θα χρειαζόταν να μεγαλώσει ο πίνακας κατακερματισμού, θα έπρεπε να ξαναϋπολογιστούν οι τιμές της συνάρτησης κατακερματισμού για κάθε έναν όρο.

Η υλοποίηση, όπως την προτείνει ο J.C. Filliâtre [Fill00], χρησιμοποιεί μια δομή πίνακα με στοιχεία λίστες. Κάθε όρος της λίστας αποτελείται από το στοιχείο τον οποίο αποθηκεύουμε, μια μοναδική για κάθε στοιχείο ακέραια μεταβλητή και την τιμή που επέτρεψε η συνάρτηση κατακερματισμού για αυτό το στοιχείο.

Αυτή είναι και η υλοποίηση που ακολουθήθηκε σε αυτή την εργασία. Αναλυτικά, το module του hashconsing παρέχει τρεις λειτουργίες: μία συνάρτηση μετατροπής του όρου στον δείκτη

που του αντιστοιχεί, μία που από τον δείκτη επιστρέφει τον πραγματικό όρο, και μια που προσθέτει το στοιχείο στον αποθηκευτικό χώρο.

Η διαδικασία που ακολουθείται είναι η εξής: Αρχικά υπολογίζεται η τιμή κατακερματισμού, η τιμή της οποίας διαιρείται με τον μέγεθος του πίνακα. Στην υλοποίηση δεν έχει προστεθεί δυνατότητα επέκτασης του πίνακα κατακερματισμού, καθώς το μέγεθος της εργασίας δεν απαιτεί τόσο χώρο. Είναι παρ' όλα αυτά μία από τις λειτουργίες που θα προστεθούν καθώς θα επεκτείνεται το σύστημα. Στη συνέχεια διατρέχει τη λίστα που είναι αποθηκευμένη στη θέση του πίνακα που βρέθηκε, και αν δεν βρεθεί το στοιχείο, το προσθέτει και επιστρέφει ένα δείκτη σε αυτό, ή αν βρεθεί επιστρέφει απλώς τον δείκτη σε αυτό.

Ένα από τα μειονεκτήματα της δομής που επιλέξαμε είναι ότι οι όροι που μπαίνουν μία φορά στον πίνακα κατακερματισμού δεν αφαιρούνται ποτέ, με αποτέλεσμα να σπαταλάτε χρόνος στην εισαγωγή νέων στοιχείων και χώρος στη μνήμη. Από την άλλη, το κέρδος που έχουμε από τη αποθήκευση κάθε όρου μια μόνο φορά, είναι σαφώς μεγαλύτερο. Ακόμα μεγάλο κέρδος σε υπολογιστική ισχύ έχουμε και από τις συγκρίσεις μεταξύ όρων, αφού πλέον χρησιμοποιούμε φυσική- ισότητα, δηλαδή ισότητα μεταξύ ακεραίων (δεικτών στη μνήμη) και όχι ισότητα σε ολόκληρη τη δομή κάθε όρου.

Το αρχείο επικεφαλίδων του module του hash-consing είναι:

```
type α hash_consed

val hash_value : α hash_consed → α
val hash_tag : α hash_consed → int

module type Comp =
  sig
        type t
        val equal : t → t → bool
        val hash : t → int
  end

module type S =
  sig
        type t
        val f : unit → t → t hash_consed
  end

module Make (X : Comp) : (S with type t = X.t)

val init : unit → unit
val register_hcons : (unit → α → α hash_consed) →
                     (unit → α → α hash_consed)
```

4.4 Ονόματα και σταθερές

Η γλώσσα που κατασκευάσαμε υποστηρίζει τον ορισμό όρων βιβλιοθήκης, σαν πρόσθετα αρθρωμάτα (modules). Η ονοματολογία αυτών είναι στη μορφή:

άρθρωμα.υποάρθρωμα1.υποάρθρωμαN.όνομα_σταθεράς

Για λόγους ταχύτητας, μιας και οι όροι αυτοί αναμένεται να επαναλαμβάνονται αρκετά συχνά, κατασκευάσαμε το module με όνομα outer, του οποίου τα στοιχεία αποθηκεύονται σε ένα πίνακα κατακερματισμού, όπως περιγράφηκε παραπάνω. Η δομή του όρου αυτού είναι η εξής

```
type sort = Set | Type | Ext
```

type *identifier*
val *identifier* : *string* → *identifier*
val *kind_of_identifier* : *identifier* → *string*

type *name*
val *name* : *identifier* × *identifier list* → *name*
val *kind_of_name* : *name* → *identifier* × *identifier list*

val *string_of_name* : *name* → *string*

όπου η λίστα από identifiers παριστάνει την δομή του αρθρώματος κάτω από το οποίο ορίζεται ο όρος.

Ο τύπος *sort* που ορίζεται σε αυτό το αρχείο παριστάνει το *sort* της γλώσσας τύπων, και ορίζεται εδώ γιατί χρησιμοποιείται τόσο από τους πρωτογενείς όρους όσο και από τους όρους της γλώσσας.

Κατά τον ορισμό κάθε σταθεράς οι αναγκαίες πληροφορίες είναι το όνομα, ο όρος που παριστάνει καθώς και ο τύπος αυτού. Συχνά χρειάζεται και μία μερικώς αποτιμημένη μορφή του τύπου του όρου, οπότε αν αυτή δεν δίνεται από τον χρήστη, τότε αποθηκεύεται την πρώτη φορά που χρειάζεται να υπολογιστεί. Η μερικώς αποτιμημένη μορφή αυτή είναι σε ασθενή κανονική μορφή κεφαλής (weak-head-normal-form) και ορίζεται στην παράγραφο 4.7 . Κάθε σταθερά που ορίζεται περιλαμβάνει επιπλέον πληροφορίες, αν βέβαια αυτές δοθούν κατά τον ορισμό, σχετικά με την εμφάνιση της σε μορφή που γίνεται άμεσα αντιληπτή από τον χρήστη. Αυτό περιλαμβάνει τόσο πληροφορίες για τελεστές με πολλαπλά ορίσματα (όπως είναι ο τελεστής πρόσθεσης και το if-then-else) όσο και για όρους με άμεση συσχέτιση με τον πραγματικό κόσμο, όπως οι θετικοί φυσικοί αριθμοί.

Σε κάθε νέο "πρόγραμμα" δημιουργείται αυτόματα το γενικό άρθρωμα το *outer* κάτω από το οποίο προστίθενται όλα τα όσα τυχόν ορίζονται. Συχνά μπορεί να προκύψουν προβλήματα με την ονοματολογία των σταθερών, έχοντας περισσότερες από μια ορισμένες με το ίδιο όνομα. Η λύση σε αυτό το πρόβλημα έρχεται από τον λεκτικό και συντακτικό αναλυτή, που είναι η πρώτη επέκταση της υλοποίησης που θα πραγματοποιηθεί. Η λύση που θα δοθεί είναι να επιτρέπεται η χρήση μιας σταθεράς με το όνομα της (και όχι ολόκληρη την ακολουθία από τα ονόματα των αρθρωμάτων) μόνο όταν δεν υπάρχει ορισμένη σταθερά με το ίδιο όνομα σε κανένα άλλο άρθρωμα. Αντίθετα αν υπάρχει τέτοια, υποχρεωτικά θα πρέπει να η αναφορά στη σταθερά να γίνεται με ολόκληρο το όνομα.

Σκοπός ακόμα είναι να κατασκευαστεί μια βιβλιοθήκη με σταθερούς όρους, ανάλογη της pervasives της OCaml που θα είναι διαθέσιμη σε κάθε πρόγραμμα, χωρίς να απαιτείται κάποια ενέργεια από τον χρήστη. Η υλοποίηση των σταθερών όρων περιλαμβάνει τις ακόλουθες δηλώσεις:

exception *ModuleNotFound* of *string*
exception *ConstantNotFound* of *string*

type *pp_info* = { *arity* :*int* ; *prec* :*int* ; *s0* :*string* ; *ll* : (*int* × *string*) *list*;
prinT : (*term* × *term list* → *bool* × *term list*) *option* }
and *constant* = { *tid* : *identifier*; *t* : *term*; *tyorig* : *term*; *ty* : *term*;
pp : *pp_info option*}
and *modul* = { *id* : *identifier* ; mutable *l* : *constant list* ;
mutable *m* : *modul list* }

val *outer* : *modul*

val *name_of_string* : *string* → *name*
val *findConstant* : *modul* → *name* → *constant*

val *addConstant* : *modul* → *identifier* → *term* → *term* → *term* → *pp_info option* → *unit*
val *addConstant'* : *modul* → *identifier* → *term* → *term* → *term* → *unit*
val *addModule* : *modul* → *identifier* → *modul*

4.5 Πρωτογενείς όροι

Οι πρωτογενείς όροι είναι στη μορφή που έχει ήδη περιγραφεί η γλώσσα. Πρόκειται για μία δομή που περιλαμβάνει όλα τα στοιχεία της γλώσσας, σε μορφή δείκτη προς τον πίνακα κατακερματισμού. Όπως φαίνεται και από τον κώδικα, η δομή *rtetm_node* ορίζεται από τα στοιχεία της type language, όπου όμως ως παράμετροι δίνονται δείκτες και όχι οι πραγματικοί όροι, ενώ η δομή *rterm* αποτελεί την αποθηκευμένη μορφή της παραπάνω. Ακόμα ορίζονται και συναρτήσεις για κάθε όρο της γλώσσας που με τις κατάλληλες παραμέτρους κατασκευάζουν και αποθηκεύουν τον όρο, επιστρέφοντας τον δείκτη προς αυτόν. Και εδώ το μεγάλο πλεονέκτημα είναι ότι η συνάρτηση ισότητας μεταξύ δύο όρων, δεν χρειάζεται να τον διατρέξει αναδρομικά, αλλά αρκεί η φυσική- ισότητα μεταξύ των υπό- όρων του.

Η χρησιμότητα τον πρωτογενών όρων έγκειται αφενός στο ότι είναι απαραίτητο να υπάρχει ολόκληρος ο όρος προτού μετατραπεί στη μορφή με de Bruijn indices, και αφ' ετέρου να είναι δυνατόν για κάποιον να γράψει κάποιον όρο, χωρίς να χρειάζεται να κάνει μια σειρά υπολογισμών για την μετατροπή του, όπως θα φανεί και παρακάτω.

Οι υλοποιούμενες συναρτήσεις είναι:

Raw terms

type *rterm*
type *rterm_node* =
    Rsort of *sort*
  | *Rvar* of *name*
  | *Rprod* of *identifier* × *rterm* × *rterm*
  | *Rlambda* of *identifier* × *rterm* × *rterm*
  | *Rapp* of *rterm* × *rterm*
  | *Rind* of *identifier* × *rterm* × *rterm list*
  | *Rconstr* of *int* × *rterm*
  | *Relim* of *rterm* × *rterm* × *rterm* × *rterm list* × *rterm list*

val *rsort* : *sort* → *rterm*
val *rvar* : *name* → *rterm*
val *rprod* : *identifier* → *rterm* → *rterm* → *rterm*
val *rlambda* : *identifier* → *rterm* → *rterm* → *rterm*
val *rapp* : *rterm* → *rterm* → *rterm*
val *rind* : *identifier* → *rterm* → *rterm list* → *rterm*
val *rconstr* : *int* → *rterm* → *rterm*
val *relim* : *rterm* → *rterm* → *rterm* → *rterm list* → *rterm list* → *rterm*

val (@→) : *rterm* → *rterm* → *rterm*

val *kind_of_rterm* : *rterm* → *rterm_node*

val *walk_rterm* :
    (*int* → *sort* → α) →
    (*int* → *name* → α) →
    (*int* → *identifier* → α → α → α) →
    (*int* → *identifier* → α → α → α) →

$(int \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow$
$(int \rightarrow identifier \rightarrow \alpha \rightarrow \alpha\ list \rightarrow \alpha) \rightarrow$
$(int \rightarrow int \rightarrow \alpha \rightarrow \alpha) \rightarrow$
$(int \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha\ list \rightarrow \alpha\ list \rightarrow \alpha)$
$\rightarrow int \rightarrow rterm \rightarrow \alpha$

type $ctx = (identifier \times name\ option)\ list$
val $term_of_rterm : rterm \rightarrow ctx \rightarrow term$

exception $Unbound_Variable$ of $string$

4.6 Όροι

Πρόκειται για τη μορφή των όρων με de Bruijn indices [dBru72]. Ο ορισμός των όρων αυτών ακολουθεί την ίδια λογική με τους πρωτογενείς όρους, παρέχοντας τη δυνατότητα ελέγχου ισότητας μεταξύ των όρων με απλή σύγκριση των δεικτών. Παρέχονται συναρτήσεις δημιουργίας των όρων, μετατροπής από πρωτογενείς, αντικατάστασης μεταβλητής από άλλον όρο καθώς και μια γενική συνάρτηση διάσχισης που παίρνει ως παραμέτρους το πως θα αντιμετωπίσει κάθε όρο. Φυσικά υπάρχει και η συνάρτηση που επιστρέφει τον πραγματικό όρο από τον δείκτη στον πίνακα κατακερματισμού.

Επιλέγουμε αυτή την αναπαράσταση μεταβλήτών ώστε να μην χρειάζεται καμμία μετονομασία κατα τη διάρκεια επεξεργασίας του όρου. Ένας επιπλέον λόγος για αυτή την επιλογή είναι ότι τα λάθη σε ένα πρόγραμμα γραμμένο με αυτή τη λογική τείνουν να γίνονται φανερά ακόμα και στα απλούστερα παραδείγματα, αντίθετα με υλοποιήσεις ονομασμένων μεταβλητών, που μπορεί να γίνουν ορατά ακόμα και πολύ καιρό μετά.

Η σύνταξη της γλώσσας τύπων με de Bruijn indices σε BNF μορφή, είναι:

$$
\begin{array}{lcl}
s & ::= & \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Ext} \\
A, B & ::= & s \mid k \mid \Pi\ A.\, B \mid \lambda A.\, B \mid A\, B \\
 & \mid & \mathsf{Ind}(A)\{\underline{A}\} \mid \mathsf{Constr}(n, A) \mid \mathsf{Elim}[A'](A\colon B\, \vec{B})\{\underline{A}\}
\end{array}
$$

Η αντίστοιχη συνάρτηση αυξησης των δεικτών των ελεύθερων μεταβλητών, για την *nflint* είναι:

$$
\begin{array}{lcl}
\uparrow_c^d (s) & = & s \\
\uparrow_c^d (k) & = & \begin{cases} k & \text{if } k \leq c \\ k + d & \text{if } k \geq c \end{cases} \\
\uparrow_c^d (\Pi\ A.\, B) & = & \Pi\ \uparrow_c^d (A).\ \uparrow_{c+1}^d (B) \\
\uparrow_c^d (\lambda A.\, B) & = & \lambda \uparrow_c^d (A).\ \uparrow_{c+1}^d (B) \\
\uparrow_c^d (A\, B) & = & \uparrow_c^d (A)\ \uparrow_c^d (B) \\
\uparrow_c^d (\mathsf{Ind}(A)\{\underline{A}\}) & = & \mathsf{Ind}(A)\{\uparrow_{c+1}^d (\underline{A})\} \\
\uparrow_c^d (\mathsf{Constr}(n, A)) & = & \mathsf{Constr}(n, \uparrow_c^d (A)) \\
\uparrow_c^d (\mathsf{Elim}[A'](A\colon B\, \vec{B})\{\underline{A}\}) & = & \mathsf{Elim}[\uparrow_c^d (A')](A\colon \uparrow_c^d (B)\ \uparrow_c^d (\vec{B}))\{\uparrow_{c+1}^d (\underline{A})\}
\end{array}
$$

όπου το s παίρνει τιμές από το σύνολο $\{\mathsf{Set}, \mathsf{Type}, \mathsf{Ext}\}$ και k είναι ο de Bruijn δείκτης κάποιας μεταβλητής.

Μπορούμε τώρα να ορίσουμε την αντικατάσταση μιας μεταβλητής από κάποιο όρο:

$$
\begin{array}{lcl}
s[j \mapsto r] & = & s \\
k[j \mapsto r] & = & \begin{cases} r & \text{αν } k = j \\ k & \text{αλλιώς} \end{cases} \\
(\Pi\ A.\, B)[j \mapsto r] & = & \Pi\ A[j \mapsto r].\, B[j+1 \mapsto\uparrow^1 r] \\
(\lambda A.\, B)[j \mapsto r] & = & \lambda A[j \mapsto r].\, B[j+1 \mapsto\uparrow^1 r] \\
(A\, B)[j \mapsto r] & = & A[j \mapsto r]\, B[j \mapsto r] \\
(\mathsf{Ind}(A)\{\underline{A}\})[j \mapsto r] & = & \mathsf{Ind}(A[j \mapsto r])\{\underline{A}\}j+1\uparrow^1 r \\
(\mathsf{Constr}(n, A))[j \mapsto r] & = & \mathsf{Constr}(n, A[j \mapsto r]) \\
(\mathsf{Elim}[A'](A\!:\!B\,\vec{B})\{\underline{A}\})[j \mapsto r] & = & \mathsf{Elim}[A'[j \mapsto r]](A[j \mapsto r]\!:\!B[j \mapsto r]\,\vec{B}[j \mapsto r])\{\underline{A}[j \mapsto r]\}
\end{array}
$$

Στην υλοποίηση μας μαζί με τα γινόμενα τις εφαρμογές και τους ορισμούς επαγωγικών τύπων που δεσμεύουν μεταβλητές, κρατάμε στον όρο και το όνομα της δεσμευμένης μεταβλητής, όπως το όρισε ο χρήστης, ώστε να μπορούμε να το τυπώσουμε αργότερα. Αναλυτικά οι συναρτήσεις είναι:

```
type nameinfo
type term
and term_node =
                Tsort of sort
        | Tvar of int × int
        | Tprod of identifier × term × term
        | Tlambda of identifier × term × term
        | Tapp of term × term
        | Tind of identifier × term × term list
        | Tconstr of int × term
        | Telim of term × term × term × term list × term list
        | Tconst of name

val tsort : sort → term
val tvar : int → int → term
val tprod : identifier → term → term → term
val tlambda : identifier → term → term → term
val tapp : term → term → term
val tind : identifier → term → term list → term
val tconstr : int → term → term
val telim : term → term → term → term list → term list → term
val tconst : name → term

val (@→) : term → term → term

val kind_of_term : term → term_node
val tag_of_term : term → int

val walk_term :
    (int → sort → α) →
    (int → int → int → α) →
    (int → identifier → α → α → α) →
    (int → identifier → α → α → α) →
    (int → α → α → α) →
    (int → identifier → α → α list → α) →
    (int → int → α → α) →
    (int → α → α → α → α list → α list → α) →
        (int → name → α) →
    int → term → α
```

val *subst_term* : *int* → *term* → *term* → *term*
val *shift_term* : *int* → *term* → *term*
val *shift_vec_term* : *int* → *term list* → *term list*
val *shift_Svec_term* : *int* → (*identifier* × *term*) *list*
→ (*identifier* × *term*) *list*

val *fv* : *int* → *term* → *bool*
val *fv_list_prod* : *int* → (*identifier* × *term*) *list* → *bool*
val *fv_list* : *int* → *term list* → *bool*

4.7 Κανονικές μορφές

Σε κανονική μορφή ορίζουμε να βρίσκεται ένας όρος ο οποίος δεν έχει κανέναν αναγώγιμο υπο-όρο. Έτσι για παράδειγμα ο όρος $(\lambda A.\, B)\, C$ δεν είναι σε κανονική μορφή. Η κανονική μορφή του προκύπτει από την αποτίμηση του, δηλαδή $B[0 \mapsto C]$. Στη γλώσσα μας ορίζουμε τρεις εκφράσεις ως αναγώγιμες (redex), μια για κάθε δυνατή αναγωγή β , η και ι .

- β-αναγωγή

  Εφαρμογή ενός όρου σε μία λ-αφαίρεση. Πρόκειται για την β-αναγωγή όπως ακριβώς ορίζεται και στον λ-λογισμό.

- η-αναγωγή

  λ-αφαίρεση η οποία είναι στη μορφή $\lambda.\, AB\, 0$, και υπό την προϋπόθεση ότι η δεσμευμένη μεταβλητή δεν είναι ελεύθερη στον όρο B (ή με de Bruijn indices, δεν εμφανίζεται η μεταβλητή με δείκτη μηδέν στον όρο B) αποτιμάται σε B.

- ι-αναγωγή

  Ουσιαστικά πρόκειται για ένα pattern match ενός στοιχείου κάποιου επαγωγικού τύπου, ανάλογα με τον κατασκευαστή αυτού από τον οποίο προέρχεται, με ταυτόχρονη αναδρομική κλήση στην παράμετρο του. Είναι ο όρος Elim της γλώσσας τύπων, όπου το στοιχείο που αποτιμάται είναι κάποιος κατασκευαστής (Constr). Δηλαδή, ένας όρος της μορφής

  $$\mathsf{Elim}[A'](\mathsf{Constr}(n, B)\, \vec{A'} : B\, \vec{B})\{\underline{A}\}$$

  είναι ένα ι redex.

Σε ασθενή κανονική μορφή κεφαλής (weak head normal form, whnf) ορίζεται να είναι ένας όρος όταν δεν είναι αναγώγιμος. Για παράδειγμα ο όρος $\lambda A.\, (\lambda B.\, 0)\, 1$ ενώ δεν είναι σε κανονική μορφή είναι σε ασθενή κανονική μορφή κεφαλής.

Στην υλοποίηση μας, φέρουμε κάθε όρο σε ασθενή κανονική μορφή κεφαλής πριν κάνουμε ταυτοποίηση προτύπου (pattern matching) προκείμενου να κατευθύνουμε τον εκάστοτε υπολογισμό.

4.8 Έλεγχος τύπων

Ο έλεγχος τύπων είναι το σημαντικότερο τμήμα μιας γλώσσας τύπων. Ο έλεγχος τύπων στη γλώσσα *nflint* γίνεται πάντα σε ένα *περιβάλλον*. Η διαδικασία εύρεσης του τύπου ενός όρου είναι η εξής:

Η συνάρτηση παίρνει ως παράμετρο το περιβάλλον (context) και τον όρο του οποίου τον τύπο επιστρέφει. Αρχικά γίνεται συνταίριασμα πρωτοτύπων (pattern match) του όρου με τις περιπτώσεις όρων που έχουμε. Οι περιπτώσεις είναι οι εξής

- Μεταβλητή

  Ο τύπος της μεταβλητής αντλείται από το περιβάλλον. Να σημειωθεί εδώ ότι το περιβάλλον είναι μια λίστα από όρους, όπου το πρώτο στοιχείο της παριστάνει τον τύπο της μεταβλητής με δείκτη 0, το δεύτερο με δείκτη 1 κ.ο.κ.

  Ιδιαίτερη αναφορά χρειάζεται για την περίπτωση που ο τύπος κάποιας μεταβλητής είναι ή περιέχει μεταβλητή. Σε αυτή τη περίπτωση είναι απαραίτητο οι ελεύθερες μεταβλητές του όρου αυτόυ να αυξηθούν (shift) κατά $n+1$, όπου n είναι ο δείκτης της μεταβλητής της οποίας τον τύπο αναζητούμε. Ο λόγος που χρειάζεται η μετατροπή αυτή, είναι ότι ο k-οστός του περιβάλλον είναι εκφρασμένος στο περιβάλλον από εκεί και μετά.

  Για παράδειγμα : αν $tctx = [\Pi\ 0.\, A; B]$ τότε χωρίς την παραπάνω προσαρμογή έχουμε ότι ο τύπος της μεταβλητής 0 είναι $\Pi\ 0.\, A$, όπου το 0 μέσα στο Π θα έχει τύπο τον ίδιο όρο $\Pi\ 0.\, A$ κάτι που φυσικά είναι λάθος αφού όπως φαίνεται από το παραπάνω περιβάλλον ο τύπος του είναι B. Με την προσαρμογή ο τύπος του 0 γίνεται $\Pi\ 1.\, A$ που είναι και το σωστό αποτέλεσμα.

- Σταθερά

  Ο τύπος της σταθεράς Set είναι Type και της Type είναι Ext

- Γινόμενο

  Ο όρος $\Pi\ A.\, B$ έχει τύπο τον τύπο του B, αν το ζεύγος s_1, s_2 , όπου s_1 ο τύπος του A και s_2 του B, ανήκει στο σύνολο

$$R = \left\{ \begin{array}{ccc} (\mathsf{Set}, \mathsf{Set}) & (\mathsf{Type}, \mathsf{Set}) & (\mathsf{Ext}, \mathsf{Set}) \\ (\mathsf{Set}, \mathsf{Type}) & (\mathsf{Type}, \mathsf{Type}) & (\mathsf{Ext}, \mathsf{Type}) \\ (\mathsf{Set}, \mathsf{Ext}) & (\mathsf{Type}, \mathsf{Ext}) & \end{array} \right\}$$

- Αφαίρεση

  Ο τύπος ενός όρου της μορφής $\lambda A.\, B$ έχει τύπο $k = \Pi\ A.\, B$ όπου B' είναι ο τύπος του B, υπό την προϋπόθεση ο τύπος του k να είναι $\beta\eta\iota$-ισοδύναμος[1] με κάποια σταθερά (Set, Type, Ext).

- Εφαρμογή

  Για να είναι σωστός ο όρος $A\,B$, δηλαδή η εφαρμογή του B πάνω στο A, πρέπει ο τύπος του A να είναι $\beta\eta\iota$-ισοδύναμος με το $\Pi\ B'.\, A'$ και ο τύπος του B να είναι ισοδύναμος με τον B'. Αν πληρούνται οι προϋποθέσεις, τότε ο τύπος της εφαρμογής είναι $A'[0 \mapsto B]$.

- Επαγωγή

  Ο τύπος ενός επαγωγικού ορισμού $\mathsf{Ind}(A)\{\underline{A}\}$ είναι A υπό τις εξής προϋποθέσεις:

  1. Ο τύπος του A είναι ισοδύναμος με Type
  2. Ο όρος A είναι ισοδύναμος με μια σειρά από γινόμενα, όπου το σώμα του τελευταίου είναι Set. (Δηλαδή $\Pi\ \vec{A}.\, \mathsf{Set}$)
  3. Στο εκτεταμένο περιβάλλον, όπου στην αρχή του προστέθηκε ο όρος A, πρέπει κάθε μέλος του $\underline{A}$ να έχει τύπο ισοδύναμο με Set και να είναι καλώς ορισμένος κατασκευαστής (well-founded-constructor).

- Κατασκευαστές

  Για να είναι σωστά διατυπωμένος ο όρος $\mathsf{Constr}(n, A,)$ πρέπει το A να είναι ισοδύναμο με κάποιον επαγωγικό ορισμό, $\mathsf{Ind}(A')\{\vec{A}\}$, όπου φυσικά ο τύπος του A θα πρέπει να

[1] Από εδώ και στο εξής, όπου αναφέρεται ο όρος "*ισοδύναμος*" εννοούμε $\beta\eta\iota$-ισοδυναμία

είναι ισοδύναμος με τον τύπο του Ind, δηλαδή με A'. Σε αυτή την περίπτωση ο τύπος του όρου είναι ο n-οστός όρος της ακολουθίας $\underline{A}$ στον οποίο έχουμε εφαρμόσει τον όρο A. Δηλαδή $A_n[0 \mapsto A]$.

- Απαλοιφή

  Ο τύπος μιας απαλοιφής $\mathsf{Elim}[A'](A{:}B\,\vec{B})\{\vec{A}\}$ θα είναι $A'\,\vec{B}\,A$ υπό τις εξής προϋποθέσεις:

  1. Το B είναι ισοδύναμο με έναν επαγωγικό ορισμό $B = \mathsf{Ind}(B')\{\underline{A'}\}$
  2. Ο παραπάνω όρος, B' είναι ισοδύναμος με μια ακολουθία από γινόμενα, ο τελευταίος όρος της οποίας είναι Set. Δηλαδή: $B' = \Pi\ \vec{B'}.\mathsf{Set}$
  3. Θα πρέπει να ισχύει: $\vec{B}{:}(\vec{X}{:}\vec{B'})$, όπου ο παραπάνω συμβολισμός είναι αυτός που αναφέρθηκε στο κεφάλαιο της type language. Αυτός ερμηνεύεται ως ότι το n-οστό στοιχείο της ακολουθίας όρων B_n έχει τύπο τον n-οστό στοιχείο της ακολουθίας B', B'_n σε περιβάλλον στο οποίο έχουν οι ανατεθεί τύποι για τις μεταβλητές με δείκτη μέχρι $n-1$ οι $n-1$ πρώτοι όροι του $\vec{B}$ αντίστοιχα.
  4. Ακόμα πρέπει ο τύπος του A' να είναι ισοδύναμος με μια σειρά από γινόμενα, της οποίας ο τελευταίος όρος είναι $B\,\vec{X} \to s$.
  5. Η τιμή s που αναφέρθηκε παραπάνω μπορεί να είναι είτε Set είτε Type. Ανάλογα καθορίζεται και το τελευταίο κριτήριο.

     Αν είναι Set αρκεί ο τύπος κάθε όρου A_i της ακολουθίας $\vec{A}$ να έχει τύπο ισοδύναμο με αυτόν που επιστρέφει η συνάρτηση Ψ.

     Αν είναι Type τότε πρέπει επιπλέον να ισχύει $small(A'_i)$ σε περιβάλλον όπου η μεταβλητή με δείκτη 0 έχει τύπο B'

     Η Ψ καλείται ως εξής: $\Psi_B(A'_i, A', Constr(i, B))$

     Στον ορισμό της η Ψ, στην γλώσσα *nflint* , έπαιρνε ως παράμετρο και μια μεταβλητή, η οποία εδώ θεωρείται ότι είναι η μεταβλητή με δείκτη μηδέν

Ενδεικτικά, κάποιες συναρτήσεις που ορίζονται για τον έλεγχο τύπων είναι οι:

type *typectx* = *Term.term list*

exception *TypeError* of *string*

val *whnf* : *Term.term* → *Term.term*
val *equal* : *typectx* → *Term.term* → *Term.term* → *bool*
val *wfc* : *int* → *Term.term* → *bool* × *int list*
val *psi* : *Term.term* → *Term.term* → *Term.term* → *Term.term* → *int list* → *Term.term*
val *small* : *Term.term list* → *Term.term* → *bool*
val *phi* : *Term.term* → *Term.term* → *Term.term* → *Term.term* → *int list* → *Term.term*
val *typecheck* : *typectx* → *Term.term* → *Term.term*

4.9 Αποτίμηση όρων

Η αποτίμηση είναι ένα από τα τμήματα της υλοποίησης στα οποία δεν δόθηκε ιδιαίτερη βαρύτητα, αφού το σημαντικό μέρος της εργασίας είναι ο έλεγχος τύπων. Παρότι ο έλεγχος τύπων προϋποθέτει, μερικές φορές, την αποτίμηση όρων (βλ. κανόνα μετατροπής), υλοποιείται με τη βοήθεια των ασθενών κανονικών μορφών κεφαλής και η απόδοσή του δεν επηρεάζεται.

Η διαδικασία αποτίμησης φέρει κάθε όρο σε ασθενή κανονική μορφή κεφαλής, και στη συνέχεια, αφού εφαρμόσει συνταίριασμα παραμέτρων, αποτιμά τους υπο-όρους αυτού, αν υπάρχουν.

Κεφάλαιο 5

Αξιολόγηση

5.1 Θεωρίες

Για την αξιολόγηση της υλοποίησης, και την εύρεση τυχόν λαθών κρίθηκε απαραίτητο να διατύπωθούν στοιχειώδεις θεωρίες στη γλώσσα *nflint* και να ελεγχθούν με τον ελεγκτή τύπων αυτής.

5.1.1 Λογικές τιμές

Ο πρώτος τύπος που κατασκευάσαμε είναι αυτός των boolean τιμών. Αυτός ορίζεται ως ένας επαγωγικός τύπος, που έχει δυο κατασκευαστές. Ο πρώτος είναι η τιμή true και ο δεύτερος η τιμή false .

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

Ακόμα κατασκευάστηκαν και συναρτήσεις για τις στοιχειώδεις πράξεις μεταξύ boolean τιμών.

$$
\begin{array}{lcl}
\mathsf{not} & : & \mathsf{Bool} \to \mathsf{Bool} \\
 & \equiv & \lambda b{:}\mathsf{Bool}.\ \mathsf{Elim}[\lambda b{:}\mathsf{Bool}.\ \mathsf{Bool}](b{:}\mathsf{Bool})\{\mathsf{false}; \mathsf{true}\} \\
\mathsf{and} & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
 & \equiv & \lambda b_1{:}\mathsf{Bool}.\ \mathsf{Elim}[\lambda b_1{:}\mathsf{Bool} \to \mathsf{Bool}.\ \mathsf{Bool}](b_1{:}\mathsf{Bool})\{\lambda b_2{:}\mathsf{Bool}.\ b_2; \lambda b_2{:}\mathsf{Bool}.\ \mathsf{false}\} \\
\mathsf{or} & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
 & \equiv & \lambda b_1{:}\mathsf{Bool}.\ \mathsf{Elim}[\lambda b_1{:}\mathsf{Bool} \to \mathsf{Bool}.\ \mathsf{Bool}](b_1{:}\mathsf{Bool})\{\lambda b_2{:}\mathsf{Bool}.\ \mathsf{true}; \lambda b_2{:}\mathsf{Bool}.\ b_2\} \\
\mathsf{Eq} & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
 & \equiv & \lambda b1{:}\mathsf{Bool}.\ \mathsf{Elim}[\lambda b{:}\mathsf{Bool}.\ (\mathsf{Bool} \to \mathsf{Bool})](b1{:}\mathsf{Bool})\{\lambda b2{:}\mathsf{Bool}.\ b2, \mathsf{not}\} \\
\mathsf{Ne} & : & \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
 & \equiv & \lambda b1{:}\mathsf{Bool}.\ \mathsf{Elim}[\lambda b{:}\mathsf{Bool}.\ (\mathsf{Bool} \to \mathsf{Bool})](b1{:}\mathsf{Bool})\{\mathsf{not}, \lambda b2{:}\mathsf{Bool}.\ b2\} \\
\mathsf{Cond}_{\mathsf{Set}} & : & \Pi k{:}\mathsf{Set}.\ \mathsf{Bool} \to k \to k \to k \\
 & \equiv & \lambda k{:}\mathsf{Set}.\ \lambda b{:}\mathsf{Bool}.\ \lambda x{:}k.\ \lambda y{:}k.\ \mathsf{Elim}[\lambda b{:}\mathsf{Bool}.\ k](b{:}\mathsf{Bool})\{x; y\} \\
\mathsf{Cond}_{\mathsf{Type}} & : & \Pi z{:}\mathsf{Type}.\ \mathsf{Bool} \to z \to z \to z \\
 & \equiv & \lambda z{:}\mathsf{Type}.\ \lambda b{:}\mathsf{Bool}.\ \lambda x{:}z.\ \lambda y{:}z.\ \mathsf{Elim}[\lambda b{:}\mathsf{Bool}.\ z](b{:}\mathsf{Bool})\{x; y\}
\end{array}
$$

Η συνάρτηση not παίρνει μια παράμετρο τύπου Bool και αν αυτή προέρχεται από τον πρώτο κατασκευαστή του τύπου Bool επιστρέφει false, ενώ αν προέρχεται από τον δεύτερο true .

Οι συναρτήσεις or και and βασίζονται στη λογική οτι:

$$
\begin{array}{lcl}
a\,or\,b & = & \text{if } a \text{ then } true \text{ else } b \\
a\,and\,b & = & \text{if } a \text{ then } b \text{ else } false
\end{array}
$$

Οι συναρτήσεις αυτές στη γλώσσα τύπων ορίζονται βάσει της αποσύνθεσης της πρώτης παραμέτρου τους.

Οι δύο συναρτήσεις Cond παριστάνουν τη δομή *if then else* για προτάσεις και θεωρήματα (παράμετροι με τύπο Set και Type αντίστοιχα).

5.1.2 Ζεύγη

Ο τύπος που ορίζει τα ζεύγη έχει έναν κατασκευαστή που ορίζει ένα ζεύγος από στοιχεία. Ο τύπος των στοιχείων δίνεται ως παράμετρος, καθιστώντας έτσι τον τύπο ζεύγους πολυμορφικό.

$$
\begin{array}{lcl}
\mathsf{Pair} & : & \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \\
 & \equiv & \lambda A{:}\mathsf{Set}.\, \lambda B{:}\mathsf{Set}.\, \mathsf{Ind}(X{:}\mathsf{Set})\{A \to B \to X\} \\
\mathsf{pair} & : & \Pi A{:}\mathsf{Set}.\, \Pi B{:}\mathsf{Set}.\, A \to \mathsf{B} \to \mathsf{Pair\,A\,B} \\
 & \equiv & \lambda A{:}\mathsf{Set}.\, \lambda B{:}\mathsf{Set}.\, \mathsf{Constr}(0, \mathsf{Pair\,A}\,B) \\
\mathsf{fst} & : & \Pi A{:}\mathsf{Set}.\, \Pi B{:}\mathsf{Set}.\, \mathsf{Pair}\,A\,B \to A \\
 & \equiv & \lambda A{:}\mathsf{Set}.\, \lambda B{:}\mathsf{Set}.\, \lambda \mathsf{p}{:}\mathsf{Pair}\,A\,\mathsf{B}.\, \mathsf{Elim}[\lambda \mathsf{p}{:}\mathsf{Pair}\,A\,\mathsf{B}.\,\mathsf{A}](\mathsf{p}{:}\mathsf{Pair}\,A\,B)\{\lambda a{:}A.\, \lambda b{:}B.\, a\} \\
\mathsf{snd} & : & \Pi A{:}\mathsf{Set}.\, \Pi B{:}\mathsf{Set}.\, \mathsf{Pair}\,A\,B \to B \\
 & \equiv & \lambda A{:}\mathsf{Set}.\, \lambda B{:}\mathsf{Set}.\, \lambda \mathsf{p}{:}\mathsf{Pair}\,A\,B.\, \mathsf{Elim}[\lambda \mathsf{p}{:}\mathsf{Pair}\,A\,B.\,B](\mathsf{p}{:}\mathsf{Pair}\,A\,B)\{\lambda \alpha{:}A.\, \lambda b{:}B.\, b\}
\end{array}
$$

Οι δύο συναρτήσεις που ορίζουμε, fst και snd , παίρνουν ως παραμέτρους τους τύπους των στοιχείων του ζεύγος, το ίδιο ζεύγος και επιστρέφουν αντίστοιχα το πρώτο ή το δεύτερο στοιχείο.

5.1.3 Προτάσεις

Όπως έχει ήδη αναφερθεί, μία από τις λειτουργίες του συστήματος που αναπτύχθηκε είναι η απόδειξη θεωρημάτων. Για το σκοπό αυτό ορίζουμε μια κωδικοποίηση της προτασικής λογικής.

Η αληθής και η ψευδής πρόταση ορίζονται ως επαγωγικοί τύποι. Η πρώτη έχει έναν κατασκευαστή (την τετριμμένη απόδειξη της αληθούς πρότασης) ενώ η δεύτερη δεν έχει κανέναν κατασκευαστή. Ορίζουμε ακόμα τους όρους για τις στοιχειώδεις πράξεις: propNot , propOr και propAnd , ενώ επιπλέον ορίζουμε όρους για την ισότητα, την ανακλαστικότητα της ισότητας και την ανισότητα μεταξύ όρων κάποιου τύπου.

$$
\begin{array}{lcl}
\mathsf{propTrue} & : & \mathsf{Set} \\
 & \equiv & \mathsf{Ind}(X{:}\mathsf{Set})\{X\} \\
\mathsf{propFalse} & : & \mathsf{Set} \\
 & \equiv & \mathsf{Ind}(X{:}\mathsf{Set})\{\} \\
\mathsf{propTruePrf} & : & \mathsf{propTrue} \\
 & \equiv & \mathsf{Constr}(0, \mathsf{propTrue}) \\
\mathsf{propNot} & : & \mathsf{Set} \to \mathsf{Set} \\
 & \equiv & \lambda p{:}\mathsf{Set}.\, (p \to \mathsf{propFalse}) \\
\mathsf{propAnd} & : & \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \\
 & \equiv & \lambda p_1{:}\mathsf{Set}.\, \lambda p_2{:}\mathsf{Set}.\, \mathsf{Ind}(X{:}\mathsf{Set})\{p_1 \to p_2 \to X\} \\
\mathsf{propOr} & : & \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \\
 & \equiv & \lambda p_1{:}\mathsf{Set}.\, \lambda p_2{:}\mathsf{Set}.\, \mathsf{Ind}(X{:}\mathsf{Set})\{p_1 \to X, p_2 \to X\} \\
\mathsf{propEq} & : & \Pi k{:}\mathsf{Set}.\, k \to k \to \mathsf{Set} \\
 & \equiv & \lambda k{:}\mathsf{Set}.\, \mathsf{Ind}(X{:}k \to k \to \mathsf{Set})\{\Pi a{:}k.\, X\,a\,a\} \\
\mathsf{propEqRefl} & : & \Pi k{:}\mathsf{Set}.\, \Pi a{:}k.\, \mathsf{propEq}\,k\,a\,a \\
 & \equiv & \lambda k{:}\mathsf{Set}.\, \mathsf{Constr}(0, \mathsf{propEq}\,k) \\
\mathsf{propNe} & : & \Pi k{:}\mathsf{Set}.\, k \to k \to \mathsf{Set} \\
 & \equiv & \lambda k{:}\mathsf{Set}.\, \lambda x{:}k.\, \lambda y{:}k.\, \mathsf{propNot}\,(\mathsf{propEq}\,k\,x\,y)
\end{array}
$$

5.1.4 Συγκρίσεις

Ο τύπος σύγκρισης (Comparison) κατασκευαστηκέ ώστε να είναι δυνατή η σύγκριση μεταξύ όρων του ιδίου τύπου. Οι τρεις κατασκευαστές παριστάνουν τις τρεις περιπτώσεις αποτελέσματος μιας σύγκρισης (ίσο, μικρότερο, μεγαλύτερο).

$$
\begin{array}{lcl}
\mathsf{Comparison} & : & \mathsf{Set} \\
 & \equiv & \mathsf{Ind}(X:\mathsf{Set})\{X, X, X\} \\
\mathsf{compEq} & : & \mathsf{Comparison} \\
 & \equiv & \mathsf{Constr}(0, \mathsf{Comparison}) \\
\mathsf{compLt} & : & \mathsf{Comparison} \\
 & \equiv & \mathsf{Constr}(1, \mathsf{Comparison}) \\
\mathsf{compGt} & : & \mathsf{Comparison} \\
 & \equiv & \mathsf{Constr}(2, \mathsf{Comparison})
\end{array}
$$

5.1.5 Φυσικοί αριθμοί

Οι θετικοί φυσικοί αριθμοί κατασκευάστηκαν με δυαδική αναπαράσταση, ώστε να υπολογίζονται γρηγορότερα από οτι θα χρειαζόταν αν κατασκευάζονταν με ακολουθιακή αναπαράσταση.

$$
\begin{array}{lcl}
\mathsf{PNat} & : & \mathsf{Set} \\
 & \equiv & \mathsf{Ind}(X:\mathsf{Set})\{X, X \to X, X \to X\} \\
\mathsf{pnat1} & : & \mathsf{PNat} \\
 & \equiv & \mathsf{Constr}(0, \mathsf{PNat}) \\
\mathsf{pnat2x} & : & \mathsf{PNat} \to \mathsf{PNat} \\
 & \equiv & \mathsf{Constr}(1, \mathsf{PNat}) \\
\mathsf{pnat2x1} & : & \mathsf{PNat} \to \mathsf{PNat} \\
 & \equiv & \mathsf{Constr}(2, \mathsf{PNat})
\end{array}
$$

Έτσι ο τύπος PNat που παριστάνει τους θετικούς φυσικούς αριθμούς έχει τρεις κατασκευαστές, έναν για τον αριθμό ένα, και δύο παραμετρικούς. Ο πρώτος παραμετρικός κατασκευαστής παίρνει ως όρισμα έναν φυσικό αριθμό και τον μετατρέπει στον διπλάσιο του, ενώ ο δεύτερος παραμετρικός κατασκευαστής επιστρέφει το διπλάσιο της παραμέτρου συν ένα.

Η συνάρτηση pnatSucc επιστρέφει τον επόμενο ακέραιο από αυτόν που παίρνει ως παράμετρο. Αν η παράμετρος προέρχεται από τον κατασκευαστή pnat1 τότε το αποτέλεσμα είναι pnat2x pnat1, δηλαδή 2. Αν προέρχεται από κατασκευαστή pnat2x p, τότε το αποτέλεσμα είναι pnat2x1 p, ενώ αν προέρχεται από pnat2x1 p τότε έχουμε pnat2x (pnatSucc p), εφαρμόζοντάς αναδρομικά την κλήση του Elim.

$$
\begin{array}{lcl}
\mathsf{pnatSucc} & : & \mathsf{PNat} \to \mathsf{PNat} \\
 & \equiv & \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{Elim}[\lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{PNat}](\mathsf{p}:\mathsf{PNat})\{ \\
 & & \quad \mathsf{pnat2x\ pnat1}, \\
 & & \quad \lambda \mathsf{p}':\mathsf{PNat}.\ \lambda \mathsf{s}':\mathsf{PNat}.\ \mathsf{pnat2x1\ p}', \\
 & & \quad \lambda \mathsf{p}':\mathsf{PNat}.\ \lambda \mathsf{s}':\mathsf{PNat}.\ \mathsf{pnat2x\ s}'\}
\end{array}
$$

Μια άλλη, απλή, συνάρτηση είναι αυτή που υπολογίζει το μισό ενός ακεραίου. Στην περίπτωση που στη συνάρτηση εφαρμοστεί ο αριθμός 1, το αποτέλεσμα είναι πάλι 1, αφού δεν υπάρχει μικρότερος PNat, ενω διαφορετικά το αποτέλεσμα είναι ο ακέραιος που αντιστοιχεί στο μισό του, στρογγυλοποιημένο πορς τα κάτω.

$$
\begin{array}{lcl}
\mathsf{pnat_half} & : & \mathsf{PNat} \to \mathsf{PNat} \\
 & \equiv & \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{Elim}[\lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{PNat}](\mathsf{p}:\mathsf{PNat})\{ \\
 & & \quad \mathsf{pnat1}, \\
 & & \quad \lambda \mathsf{p}:\mathsf{PNat}.\ \lambda \mathsf{s}:\mathsf{PNat}.\ \mathsf{p}, \\
 & & \quad \lambda \mathsf{p}:\mathsf{PNat}.\ \lambda \mathsf{s}:\mathsf{PNat}.\ \mathsf{p}\}
\end{array}
$$

Η περίπτωση της πρόσθεσης είναι πιο πολύπλοκη, αφού παίρνει δύο παραμέτρους, και εφαρμόζει elimination σε αυτούς. Η λογική είναι:

- αν η πρώτη παράμετρος είναι pnat1

  Τότε το αποτέλεσμα είναι pnat2x pnat1, pnat2x1 p, pnat2x (pnatSucc p) αντίστοιχα αν η δεύτερη παράμετρος προέρχεται από κατασκευαστές pnat1 , pnat2x p, pnat2x1 p.

- αν η πρώτη παράμετρος είναι pnat2x p

  Τότε το αποτέλεσμα είναι pnat2x1 p, pnat2x (s p'), pnat2x1 (s p') αντίστοιχα για τους κατασκευαστές της δεύτερης παραμέτρου. Με p παριστάνουμε το αποτέλεσμα της απαλοιφής (elimination) της πρώτης παραμέτρου, με p' της δεύτερης, ενώ s είναι το αποτέλεσμα της αναδρομικής κλήσης της συνάρτησης πάνω στην πρώτη παράμετρο. Στον όρο s εφαρμόζεται ο p' ώστε να επιστρέψει το αποτέλεσμα της πρόσθεσης μεταξύ των όρων.

- αν η πρώτη παράμετρος είναι pnat2x1 p

  Με ίδιο συμβολισμό, και αντίστοιχη λογική με την παραπάνω περίπτωση το αποτέλεσμα είναι

  pnat2x1 p' , pnat2x1 (s p'), pnat2x (pnatsucc s p')

$$
\begin{array}{lcl}
\mathsf{pnatPlus} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{PNat} \\
 & \equiv & \lambda p_1 : \mathsf{PNat}.\ \mathsf{Elim}[\lambda p_1 : \mathsf{PNat}.\ (\mathsf{PNat} \to \mathsf{PNat})](p_1 : \mathsf{PNat})\{ \\
 & & \quad \lambda p_2 : \mathsf{PNat}.\ \mathsf{Elim}[\lambda p_2 : \mathsf{PNat}.\ \mathsf{PNat}](p_2 : \mathsf{PNat})\{ \\
 & & \quad\quad \mathsf{pnat2x\,pnat1}, \\
 & & \quad\quad \lambda p_2' : \mathsf{PNat}.\ \lambda s2 : \mathsf{PNat}.\ \mathsf{pnat2x1}\ p_2', \\
 & & \quad\quad \lambda p_2' : \mathsf{PNat}.\ \lambda s2 : \mathsf{PNat}.\ \mathsf{pnat2x}\ (\mathsf{pnatSucc}\ p_2')\}, \\
 & & \quad \lambda p_1' : \mathsf{PNat}.\ \lambda s_1 : \mathsf{PNat} \to \mathsf{PNat}.\ \lambda p_2 : \mathsf{PNat}. \\
 & & \quad\quad \mathsf{Elim}[\lambda p_2 : \mathsf{PNat}.\ \mathsf{PNat}](p_2 : \mathsf{PNat})\{ \\
 & & \quad\quad\quad \mathsf{pnat2x1}\ p_1', \\
 & & \quad\quad\quad \lambda p_2' : \mathsf{PNat}.\ \lambda s2 : \mathsf{PNat}.\ \mathsf{pnat2x}\ (s_1\ p_2'), \\
 & & \quad\quad\quad \lambda p_2' : \mathsf{PNat}.\ \lambda s2 : \mathsf{PNat}.\ \mathsf{pnat2x1}\ (s_1\ p_2')\}, \\
 & & \quad \lambda p_1' : \mathsf{PNat}.\ \lambda s_1 : \mathsf{PNat} \to \mathsf{PNat}.\ \lambda p_2 : \mathsf{PNat}. \\
 & & \quad\quad \mathsf{Elim}[\lambda p_2 : \mathsf{PNat}.\ \mathsf{PNat}](p_2 : \mathsf{PNat})\{ \\
 & & \quad\quad\quad \mathsf{pnat2x}\ (\mathsf{pnatSucc}\ p_1'), \\
 & & \quad\quad\quad \lambda p_2' : \mathsf{PNat}.\ \lambda s2 : \mathsf{PNat}.\ \mathsf{pnat2x1}\ (s_1\ p_2'), \\
 & & \quad\quad\quad \lambda p_2' : \mathsf{PNat}.\ \lambda s2 : \mathsf{PNat}.\ \mathsf{pnat2x}\ (\mathsf{pnatSucc}\ (s_1\ p_2'))\}\}
\end{array}
$$

Ανάλογη πολυπλοκότητα έχει και ο όρος του πολλαπλασιασμού, που επιτρέπει όμως και τη σύνθεση με κάποια άλλη συνάρτηση. Ο όρος που ορίζουμε παρακάτω παίρνει ως παραμέτρους έναν PNat, μία συνάρτηση f από PNat σε PNat και έναν ακόμα PNat. Ουσιαστικά πολλαπλασιάζει τον πρώτο όρο με το αποτέλεσμα της εφαρμογής του δεύτερου στην συνάρτηση f.

Οι περιπτώσεις του Elim είναι:

p_2 εφαρμοσμένο στην f	αν p_1 είναι pnat1
αναδρομική κλήση στο Elim με παραμέτρους 2 * f και p_2	αν p_1 είναι pnat2x
p_2 εφαρμοσμένο στην f συν την αναδρομική κλήση στο Elim με παραμέτρους 2* f και p_2	αν p_1 είναι pnat2x1

$$
\begin{array}{lcl}
\mathsf{pnatTimesAuxil} & : & \mathsf{PNat} \to (\mathsf{PNat} \to \mathsf{PNat}) \to \mathsf{PNat} \to \mathsf{PNat} \\
& \equiv & \lambda \mathsf{p}_1 : \mathsf{PNat}. \\
& & \mathsf{Elim}[\lambda \mathsf{p}_1 : \mathsf{PNat}. ((\mathsf{PNat} \to \mathsf{PNat}) \to \mathsf{PNat} \to \mathsf{PNat})](\mathsf{p}_1 : \mathsf{PNat}\,)\{ \\
& & \quad \lambda \mathsf{f} : \mathsf{PNat} \to \mathsf{PNat}. \lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{f}\,\mathsf{p}_2, \\
& & \quad \lambda \mathsf{p}'_1 : \mathsf{PNat}. \lambda \mathsf{g} : (\mathsf{PNat} \to \mathsf{PNat}) \to \mathsf{PNat} \to \mathsf{PNat}. \\
& & \quad\quad \lambda \mathsf{f} : \mathsf{PNat} \to \mathsf{PNat}. \lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{g}\,(\lambda \mathsf{z} : \mathsf{PNat}. \mathsf{pnat2x}\,(\mathsf{f}\,\mathsf{z}))\,\mathsf{p}_2, \\
& & \quad \lambda \mathsf{p}'_1 : \mathsf{PNat}. \lambda \mathsf{g} : (\mathsf{PNat} \to \mathsf{PNat}) \to \mathsf{PNat} \to \mathsf{PNat}. \\
& & \quad\quad \lambda \mathsf{f} : \mathsf{PNat} \to \mathsf{PNat}. \lambda \mathsf{p}_2 : \mathsf{PNat}. \\
& & \quad\quad\quad \mathsf{pnatPlus}\,(\mathsf{f}\,\mathsf{p}_2)\,(\mathsf{g}\,(\lambda \mathsf{z} : \mathsf{PNat}. \mathsf{pnat2x}\,(\mathsf{f}\,\mathsf{z}))\,\mathsf{p}_2)\}
\end{array}
$$

Η pnatTimes είναι η εφαρμογή της παραπάνω ώστε να κάνει τον πολλαπλασιασμό μεταξύ φυσικών. Αυτό γίνεται δίνοντας ως f την ταυτοτική συνάρτηση $(\lambda \mathsf{z} : \mathsf{PNat}. \mathsf{z})$.

$$
\begin{array}{lcl}
\mathsf{pnatTimes} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{PNat} \\
& \equiv & \lambda \mathsf{p}_1 : \mathsf{PNat}. \lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{pnatTimesAuxil}\,\mathsf{p}_1\,(\lambda \mathsf{z} : \mathsf{PNat}. \mathsf{z})\,\mathsf{p}_2
\end{array}
$$

Ο όρος p πολλαπλασιαζόμενος επί 2, και μειωμένος κατά ένα υπολογίζεται ως εξής:

αν ο όρος είναι ο pnat1	$2 * 1 - 1 = 1$, άρα pnat1
αν είναι $2 * \mathsf{p}'$	τότε είναι $1 + 2 * (\mathsf{pnat2xMinus1}\,\mathsf{p}')$
αν είναι $1 + 2 * \mathsf{p}'$	τότε είναι $1 + 2 * \mathsf{p}' = \mathsf{pnat2x1}\,\mathsf{p}'$

$$
\begin{array}{lcl}
\mathsf{pnat2xMinus1} & : & \mathsf{PNat} \to \mathsf{PNat} \\
& \equiv & \lambda \mathsf{p} : \mathsf{PNat}. \mathsf{Elim}[\lambda \mathsf{p} : \mathsf{PNat}. \mathsf{PNat}](\mathsf{p} : \mathsf{PNat}\,)\{ \\
& & \quad \mathsf{pnat1}, \\
& & \quad \lambda \mathsf{p}' : \mathsf{PNat}. \lambda \mathsf{s}' : \mathsf{PNat}. \mathsf{pnat2x1}\,\mathsf{s}', \\
& & \quad \lambda \mathsf{p}' : \mathsf{PNat}. \lambda \mathsf{s}' : \mathsf{PNat}. \mathsf{pnat2x1}\,(\mathsf{pnat2x}\,\mathsf{p}')\}
\end{array}
$$

Οι παρακάτω συναρτήσεις υλοποιούν την σύγκριση μεταξύ θετικών φυσικών.

$$
\begin{array}{lcl}
\mathsf{pnatCompareC} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Comparison} \to \mathsf{Comparison} \\
\mathsf{pnatCompare} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Comparison} \\
\mathsf{pnatEq} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Bool} \\
\mathsf{pnatNe} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Bool} \\
\mathsf{pnatLt} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Bool} \\
\mathsf{pnatGt} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Bool} \\
\mathsf{pnatLe} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Bool} \\
\mathsf{pnatGe} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Bool}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{pnatMinus} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Int} \\
& \equiv & \lambda \mathsf{p}_1 : \mathsf{PNat}. \mathsf{Elim}[\lambda \mathsf{p}_1 : \mathsf{PNat}. (\mathsf{PNat} \to \mathsf{Int})](\mathsf{p}_1 : \mathsf{PNat}\,)\{ \\
& & \quad \lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{Elim}[\lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{Int}](\mathsf{p}_2 : \mathsf{PNat}\,)\{ \\
& & \quad\quad \mathsf{int0}, \\
& & \quad\quad \lambda \mathsf{p}'_2 : \mathsf{PNat}. \lambda \mathsf{r} : \mathsf{Int}. \mathsf{intN}\,(\mathsf{pnat2xMinus1}\,\mathsf{p}'_2), \\
& & \quad\quad \lambda \mathsf{p}'_2 : \mathsf{PNat}. \lambda \mathsf{r} : \mathsf{Int}. \mathsf{intN}\,(\mathsf{pnat2x}\,\mathsf{p}'_2)\}, \\
& & \quad \lambda \mathsf{p}'_1 : \mathsf{PNat}. \lambda \mathsf{s} : \mathsf{PNat} \to \mathsf{Int}. \lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{Elim}[\lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{Int}](\mathsf{p}_2 : \mathsf{PNat}\,)\{ \\
& & \quad\quad \mathsf{intP}\,(\mathsf{pnat2xMinus1}\,\mathsf{p}'_1), \\
& & \quad\quad \lambda \mathsf{p}'_2 : \mathsf{PNat}. \lambda \mathsf{r} : \mathsf{Int}. \mathsf{int2x}\,(\mathsf{s}\,\mathsf{p}'_2), \\
& & \quad\quad \lambda \mathsf{p}'_2 : \mathsf{PNat}. \lambda \mathsf{r} : \mathsf{Int}. \mathsf{int2xMinus1}\,(\mathsf{s}\,\mathsf{p}'_2)\}, \\
& & \quad \lambda \mathsf{p}'_1 : \mathsf{PNat}. \lambda \mathsf{s} : \mathsf{PNat} \to \mathsf{Int}. \lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{Elim}[\lambda \mathsf{p}_2 : \mathsf{PNat}. \mathsf{Int}](\mathsf{p}_2 : \mathsf{PNat}\,)\{ \\
& & \quad\quad \mathsf{intP}\,(\mathsf{pnat2x}\,\mathsf{p}'_1), \\
& & \quad\quad \lambda \mathsf{p}'_2 : \mathsf{PNat}. \lambda \mathsf{r} : \mathsf{Int}. \mathsf{int2x1}\,(\mathsf{s}\,\mathsf{p}'_2), \\
& & \quad\quad \lambda \mathsf{p}'_2 : \mathsf{PNat}. \lambda \mathsf{r} : \mathsf{Int}. \mathsf{int2x}\,(\mathsf{s}\,\mathsf{p}'_2)\}\}
\end{array}
$$

Οι θετικοί αριθμοί ορίστηκαν ως ο αριθμός μηδέν και οι θετικοί φυσικοί αριθμοί

$$\begin{array}{lcl} \mathsf{Nat} & : & \mathsf{Set} \\ & \equiv & \mathsf{Ind}(X:\mathsf{Set})\{X, \mathsf{PNat} \to X\} \end{array}$$

$$\begin{array}{lcl} \mathsf{nat0} & : & \mathsf{Nat} \\ & \equiv & \mathsf{Constr}(0, \mathsf{Nat}) \\ \mathsf{natP} & : & \mathsf{PNat} \to \mathsf{Nat} \\ & \equiv & \mathsf{Constr}(1, \mathsf{Nat}) \end{array}$$

Έτσι ο πρώτος κατασκευαστής του Nat παριστάνει τον αριθμό μηδέν και ο δεύτερος, που παίρνει ως παράμετρο έναν PNat, επιστρέφει τον ίδιο αριθμό αλλά με τύπο Nat.

Η συνάρτηση που επιστρέφει τον προηγούμενο ενός θετικού φυσικού αριθμού έχει τύπο PNat → Nat αφού μπορεί να επιστρέψει και μηδέν.

$$\begin{array}{lcl} \mathsf{pnatPred} & : & \mathsf{PNat} \to \mathsf{Nat} \\ & \equiv & \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{Elim}[\lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{Nat}](\mathsf{p}:\mathsf{PNat})\{ \\ & & \quad \mathsf{nat0}, \\ & & \quad \lambda \mathsf{p}':\mathsf{PNat}.\ \lambda \mathsf{s}:\mathsf{Nat}.\ \mathsf{natP}\ (\mathsf{pnat2xMinus1}\ \mathsf{p}'), \\ & & \quad \lambda \mathsf{p}':\mathsf{PNat}.\ \lambda \mathsf{s}:\mathsf{Nat}.\ \mathsf{natP}\ (\mathsf{pnat2x}\ \mathsf{p}')\} \end{array}$$

Ορίζουμε ακόμα τις ακόλουθες συναρτήσεις:

- nat2x

  Διπλασιασμός ενός φυσικού αριθμού. Αν είναι ο όρος nat0, τότε επιστρέφεται πάλι nat0 , ενώ αν προέρχεται από τον θετικό φυσικό p τότε το αποτέλεσμα είναι natP (pnat2x p), μετατροπή σε φυσικού του διπλασισμένου θετικού φυσικού αριθμού.

- nat2x1 , nat2xMinus1X , natPred , natSucc , natPlus

  Ακολουθεί την ίδια ακριβώς λογική με την παραπάνω, δηλαδή εφαρμογή της αντίστοιχης συνάρτησης στον θετικό φυσικό από τον οποίο προέρχεται, και μετατροπή σε φυσικό.

$$\begin{array}{lcl} \mathsf{nat2x} & : & \mathsf{Nat} \to \mathsf{Nat} \\ & \equiv & \lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n}:\mathsf{Nat})\{\mathsf{nat0};\ \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{natP}\ (\mathsf{pnat2x}\ \mathsf{p})\} \end{array}$$

$$\begin{array}{lcl} \mathsf{nat2x1} & : & \mathsf{Nat} \to \mathsf{Nat} \\ & \equiv & \lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n}:\mathsf{Nat})\{ \\ & & \quad \mathsf{natP}\ \mathsf{pnat1}, \\ & & \quad \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{natP}\ (\mathsf{pnat2x1}\ \mathsf{p})\} \end{array}$$

$$\begin{array}{lcl} \mathsf{nat2xMinus1X} & : & \mathsf{Nat} \to \mathsf{Nat} \\ & \equiv & \lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n}:\mathsf{Nat})\{ \\ & & \quad \mathsf{nat0}, \\ & & \quad \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{natP}\ (\mathsf{pnat2xMinus1}\ \mathsf{p})\} \end{array}$$

$$\begin{array}{lcl} \mathsf{natPred} & : & \mathsf{Nat} \to \mathsf{Nat} \\ & \equiv & \lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n}:\mathsf{Nat})\{ \\ & & \quad \mathsf{nat0}, \\ & & \quad \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{pnatPred}\ \mathsf{p}\} \end{array}$$

$$\begin{array}{lcl} \mathsf{natSucc} & : & \mathsf{Nat} \to \mathsf{Nat} \\ & \equiv & \lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n}:\mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n}:\mathsf{Nat})\{ \\ & & \quad \mathsf{natP}\ \mathsf{pnat1}, \\ & & \quad \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{natP}\ (\mathsf{pnatSucc}\ \mathsf{p})\} \end{array}$$

$$
\begin{array}{lll}
\mathsf{natPlus} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
 & \equiv & \lambda \mathsf{n1}\colon \mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n1}\colon \mathsf{Nat}.\ (\mathsf{Nat} \to \mathsf{Nat})](\mathsf{n1}\colon \mathsf{Nat})\{ \\
 & & \quad \lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{n2}, \\
 & & \quad \lambda \mathsf{p}_1\colon \mathsf{PNat}.\ \lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n2}\colon \mathsf{Nat})\{ \\
 & & \quad\quad \mathsf{n1}, \\
 & & \quad\quad \lambda \mathsf{p}_2\colon \mathsf{PNat}.\ \mathsf{natP}\ (\mathsf{pnatPlus}\ \mathsf{p}_1\ \mathsf{p}_2)\}\}
\end{array}
$$

Η συνάρτηση αφαίρεσης των θετικών φυσικών αριθμών p_1 και p_2 επιστρέφει έναν φυσικό αριθμό ως εξής:

- αν $\mathsf{p}_1 = 1$

  τότε το αποτέλεσμα είναι 0, αφού θεωρούμαι οτι η συνάρτηση θα επιστρέψει μηδέν αν το αποτέλεσμα είναι αρνητικό.

- $\mathsf{p}_1 = 2 * \mathsf{p}'_1$

$\mathsf{p}_2 = 1$	$2 * \mathsf{p}_2 - 1$	$\mathsf{pnat2xMinus1}\ \mathsf{p}'_1$
$\mathsf{p}_2 = 2 * \mathsf{p}'_2$	$2 * (\mathsf{p}'_1 - \mathsf{p}'_2)$	$\mathsf{nat2x}\ (\mathsf{pnatMinusX}\ \mathsf{p}'_1\ \mathsf{p}'_2)$
$\mathsf{p}_2 = 1 + 2 * \mathsf{p}'_2$	$2 * (\mathsf{p}'_1 - \mathsf{p}'_2) - 1$	$\mathsf{nat2xMinus1X}\ (\mathsf{pnatMinusX}\ \mathsf{p}'_1\ \mathsf{p}'_2)$

- $\mathsf{p}_1 = 1 + 2 * \mathsf{p}'_1$

$\mathsf{p}_2 = 1$	$2 * \mathsf{p}_2$	$\mathsf{natP}\ (\mathsf{pnat2x}\ \mathsf{p}'_1)$
$\mathsf{p}_2 = 2 * \mathsf{p}'_2$	$1 + 2 * (\mathsf{p}'_1 - \mathsf{p}'_2)$	$\mathsf{nat2x1}\ (\mathsf{pnatMinusX}\ \mathsf{p}'_1\ \mathsf{p}'_2)$
$\mathsf{p}_2 = 1 + 2 * \mathsf{p}'_2$	$2 * (\mathsf{p}'_1 - \mathsf{p}'_2)$	$\mathsf{nat2x}\ (\mathsf{pnatMinusX}\ \mathsf{p}'_1\ \mathsf{p}'_2)$

$$
\begin{array}{lll}
\mathsf{pnatMinusX} & : & \mathsf{PNat} \to \mathsf{PNat} \to \mathsf{Nat} \\
 & \equiv & \lambda \mathsf{p}_1\colon \mathsf{PNat}.\ \mathsf{Elim}[\lambda \mathsf{p}_1\colon \mathsf{PNat}.\ (\mathsf{PNat} \to \mathsf{Nat})](\mathsf{p}_1\colon \mathsf{PNat})\{ \\
 & & \quad \lambda \mathsf{p}_2\colon \mathsf{PNat}.\ \mathsf{nat0}, \\
 & & \quad \lambda \mathsf{p}'_1\colon \mathsf{PNat}.\ \lambda \mathsf{s}_1\colon \mathsf{PNat} \to \mathsf{Nat}.\ \lambda \mathsf{p}_2\colon \mathsf{PNat}. \\
 & & \quad\quad \mathsf{Elim}[\lambda \mathsf{p}_2\colon \mathsf{PNat}.\ \mathsf{Nat}](\mathsf{p}_2\colon \mathsf{PNat})\{ \\
 & & \quad\quad\quad \mathsf{natP}\ (\mathsf{pnat2xMinus1}\ \mathsf{p}'_1), \\
 & & \quad\quad\quad \lambda \mathsf{p}'_2\colon \mathsf{PNat}.\ \lambda \mathsf{s2}\colon \mathsf{Nat}.\ \mathsf{nat2x}\ (\mathsf{s}_1\ \mathsf{p}'_2), \\
 & & \quad\quad\quad \lambda \mathsf{p}'_2\colon \mathsf{PNat}.\ \lambda \mathsf{s2}\colon \mathsf{Nat}.\ \mathsf{nat2xMinus1X}\ (\mathsf{s}_1\ \mathsf{p}'_2)\}, \\
 & & \quad \lambda \mathsf{p}'_1\colon \mathsf{PNat}.\ \lambda \mathsf{s}_1\colon \mathsf{PNat} \to \mathsf{Nat}.\ \lambda \mathsf{p}_2\colon \mathsf{PNat}. \\
 & & \quad\quad \mathsf{Elim}[\lambda \mathsf{p}_2\colon \mathsf{PNat}.\ \mathsf{Nat}](\mathsf{p}_2\colon \mathsf{PNat})\{ \\
 & & \quad\quad\quad \mathsf{natP}\ (\mathsf{pnat2x}\ \mathsf{p}'_1), \\
 & & \quad\quad\quad \lambda \mathsf{p}'_2\colon \mathsf{PNat}.\ \lambda \mathsf{s2}\colon \mathsf{Nat}.\ \mathsf{nat2x1}\ (\mathsf{s}_1\ \mathsf{p}'_2), \\
 & & \quad\quad\quad \lambda \mathsf{p}'_2\colon \mathsf{PNat}.\ \lambda \mathsf{s2}\colon \mathsf{Nat}.\ \mathsf{nat2x}\ (\mathsf{s}_1\ \mathsf{p}'_2)\}\}
\end{array}
$$

Όμοια με προηγούμενες περιπτώσεις, ορίζονται και οι συναρτήσεις αφαίρεσης και πολλαπλασιασμού για φυσικούς, χρησιμοποιώντας τις αντίστοιχες συναρτήσεις των θετικών ακεραίων.

$$
\begin{array}{lll}
\mathsf{natMinusX} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
 & \equiv & \lambda \mathsf{n1}\colon \mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n1}\colon \mathsf{Nat}.\ (\mathsf{Nat} \to \mathsf{Nat})](\mathsf{n1}\colon \mathsf{Nat})\{ \\
 & & \quad \lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{nat0}, \\
 & & \quad \lambda \mathsf{p}_1\colon \mathsf{PNat}.\ \lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n2}\colon \mathsf{Nat})\{ \\
 & & \quad\quad \mathsf{n1}, \\
 & & \quad\quad \lambda \mathsf{p}_2\colon \mathsf{PNat}.\ \mathsf{pnatMinusX}\ \mathsf{p}_1\ \mathsf{p}_2\}\}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{natTimes} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
 & \equiv & \lambda \mathsf{n1}\colon \mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n1}\colon \mathsf{Nat}.\ (\mathsf{Nat} \to \mathsf{Nat})](\mathsf{n1}\colon \mathsf{Nat})\{ \\
 & & \quad \lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{nat0}, \\
 & & \quad \lambda \mathsf{p}_1\colon \mathsf{PNat}.\ \lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{Elim}[\lambda \mathsf{n2}\colon \mathsf{Nat}.\ \mathsf{Nat}](\mathsf{n2}\colon \mathsf{Nat})\{ \\
 & & \quad\quad \mathsf{nat0}, \\
 & & \quad\quad \lambda \mathsf{p}_2\colon \mathsf{PNat}.\ \mathsf{natP}\ (\mathsf{pnatTimes}\ \mathsf{p}_1\ \mathsf{p}_2)\}\}
\end{array}
$$

```
natCompare        :  Nat → Nat → Comparison
natEq             :  Nat → Nat → Bool
natNe             :  Nat → Nat → Bool
natLt             :  Nat → Nat → Bool
natGt             :  Nat → Nat → Bool
natLe             :  Nat → Nat → Bool
natGe             :  Nat → Nat → Bool
natLT             :  Nat → Nat → Set
pnatDivModAuxil   :  PNat → PNat → Pair Nat Nat
natDiv            :  Nat → Nat → Nat
natMod            :  Nat → Nat → Nat
```

Στη υλοποίηση πολλών συναρτήσεων, όπως είναι ο υπολογισμός του παραγοντικού ή ενός όρου της ακολουθίας Fibonacci, χρειάζεται η αναδρομική κλήση σε έναν αριθμό βάση του προηγούμενου του. Ορίζουμε έτσι τις ακόλουθες συναρτήσεις:

- pnatPlusWithSucc

  υπολογίζει την πρόσθεση p + k, εφαρμόζοντας k φορές την pnatSucc στο p

- pnatIterateWithSucc

  εφαρμόζει αναδρομική κλήση με χρήση του του pnatPlusWithSucc

- pnatRec

  που είναι και το ζητούμενο, δηλαδή η αντίστοιχη Elim, αν οι κατασκευαστές του pnat ήταν οι pnat1 και pnatSucc . Ο όρος αυτός παίρνει ως παραμέτρους έναν τύπο A , το αποτέλεσμα αν η παράμετρος p είναι pnat1, μια συνάρτηση που υπολογίζει αναδρομικά την κλήση της πάνω στην παράμετρο p και επιστρέφει αποτέλεσμα τύπου A.

```
pnatPlusWithSucc     :  PNat → PNat → PNat
                     ≡  λp:PNat. Elim[λp:PNat. PNat → PNat](p:PNat){
                            λq:PNat. pnatSucc q,
                            λp′:PNat. λz:PNat → PNat. λq:PNat. z (z q),
                            λp′:PNat. λz:PNat → PNat. λq:PNat. z(z (pnatSucc q))}
pnatIterateWithSucc  :  Π P:PNat → Set.
                            Π p:PNat. P p → P (pnatSucc p) → Π p:PNat.
                                Π q:PNat. P q → P (pnatPlusWithSucc p q)
                     ≡  λP:PNat → Set. λH:Π p:. P p → P(pnatSucc p). λp:PNat.
                        Elim[λp:PNat. Π q:PNat. P q → P (pnatPlusWithSucc p q)](
                            p:PNat PNat){
                            λq:PNat. λz:P q. H q z;
                            λp′:PNat. λv:Π q:PNat. P q → P (pnatPlusWithSucc p′ q).
                                λq:PNat. λz:P q. v (pnatPlusWithSucc p′ q) (v q z);
                            λp′:PNat. λv:Π q:PNat. P q → P (pnatPlusWithSucc p′ q).
                                λq:PNat. λz:P q. v (pnatPlusWithSucc p′ (pnatSucc q))(
                                v (pnatSucc q) (H q z))}
pnatRec              :  Π A:Set. A → (PNat → A → A) → PNat → A
                     ≡  λA:Set. λa:A. λf:PNat → A → A. λp:PNat.
                            Elim[λp:PNat. A](p:PNat PNat){
                                a,
                                λp′:PNat. λq:A. pnatIterateWithSucc (λp:. A) f p′ p′ q,
                                λp′:PNat. λq:A. f (pnatPlusWithSucc p′ p′)
                                    (pnatIterateWithSucc (λp:PNat. A) f p′ p′ q)}
```

5.1.6 Ακέραιοι αριθμοί

Παρόμοια λογική με τους φυσικούς ακολουθήσαμε για να επεκταθούμε στους ακεραίους. Ο τύπος των ακέραιων Int έχει τρεις κατασκευαστές. Ο πρώτος παριστάνει τον αριθμό μηδέν, ο δεύτερος παίρνει έναν θετικό φυσικό αριθμό και επιστρέφει τον ίδιο αριθμό, με τύπο Int και ο τρίτος παίρνει πάλι έναν θετικό φυσικό αριθμό και επιστρέφει τον αντίθετο του.

$$
\begin{array}{lcl}
\mathsf{Int} & : & \mathsf{Set} \\
 & \equiv & \mathsf{Ind}(X:\mathsf{Set})\{X, \mathsf{PNat} \to X, \mathsf{PNat} \to X\} \\
\mathsf{int0} & : & \mathsf{Int} \\
 & \equiv & \mathsf{Constr}(0, \mathsf{Int}) \\
\mathsf{intP} & : & \mathsf{PNat} \to \mathsf{Int} \\
 & \equiv & \mathsf{Constr}(1, \mathsf{Int}) \\
\mathsf{intN} & : & \mathsf{PNat} \to \mathsf{Int} \\
 & \equiv & \mathsf{Constr}(2, \mathsf{Int}) \\
\mathsf{pnatSucc} & : & \mathsf{PNat} \to \mathsf{PNat} \\
 & \equiv & \lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{Elim}[\lambda \mathsf{p}:\mathsf{PNat}.\ \mathsf{PNat}](\mathsf{p}:\mathsf{PNat})\{ \\
 & & \quad \mathsf{pnat2x\,pnat1}, \\
 & & \quad \lambda \mathsf{p}':\mathsf{PNat}.\ \lambda \mathsf{s}':\mathsf{PNat}.\ \mathsf{pnat2x1\,p}', \\
 & & \quad \lambda \mathsf{p}':\mathsf{PNat}.\ \lambda \mathsf{s}':\mathsf{PNat}.\ \mathsf{pnat2x\,s}'\}
\end{array}
$$

Για τους ακεραίους ορίστηκαν ακόμα συναρτήσεις για όλες τις βασικές πράξεις. Οι τύποι αυτών είναι όπως φαίνεται παρακάτω. Το σώμα των συναρτήσεων αυτών παραλείπεται λόγω της πολυπλοκότητας του, και επειδή δεν έχει να προσφέρει καμία επιπλέον τεχνική σε σχέση με τους προηγούμενα ορισμένους όρους.

$$
\begin{array}{lcl}
\mathsf{int2xMinus1} & : & \mathsf{Int} \to \mathsf{Int} \\
\mathsf{intNeg} & : & \mathsf{Int} \to \mathsf{Int} \\
\mathsf{intPlus} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{intMinus} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{intTimes} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{intDiv} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{intMod} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
\mathsf{intCompare} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Comparison} \\
\mathsf{intEq} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
\mathsf{intNe} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
\mathsf{intLt} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
\mathsf{intGt} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
\mathsf{intLe} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
\mathsf{intGe} & : & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool}
\end{array}
$$

5.1.7 Λίστες

Ορίζουμε τον πολυμορφικό. τύπο λίστας, δηλαδή τη λίστα από στοιχεία ο τύπος των οποίων είναι και αυτός μια παράμετρος.

Για την κατασκευή μιας λίστας απαραίτητα είναι η τιμή nil, δηλαδή η άδεια λίστα, και η συνάρτηση cons που προσθέτει ένα στοιχεία στην κεφαλή μιας λίστας. Ο τύπος λίστας παίρνει ως παράμετρο τον τύπο A των στοιχείων της. Ο πρώτος κατασκευαστής, ο nil, έχει τύπο τον τύπο της λίστας και ο δεύτερος A → X → X → X όπου X είναι ο τύπος που ορίζουμε, δηλαδή ο τύπος λίστας. Ο παραπάνω τύπος σημαίνει οτι ο δεύτερος κατασκευαστής παίρνει ένα στοιχείο, παίρνει μια λίστα και επιστρέφει μια λίστα αφού έχει προσθέσει το στοιχείο στην αρχή της λίστας.

$$
\begin{array}{lcl}
\mathsf{List} & : & \mathsf{Set} \to \mathsf{Set} \\
 & \equiv & \lambda A{:}\mathsf{Set}.\ \mathsf{Ind}(X{:}\mathsf{Set})\{X, A \to X \to X\} \\
\mathsf{nil} & : & \Pi k{:}\mathsf{Set}.\ \mathsf{List}\, k \\
 & \equiv & \lambda A{:}\mathsf{Set}.\ \mathsf{Constr}(0, \mathsf{List}\, A) \\
\mathsf{cons} & : & \Pi k{:}\mathsf{Set}.\ k \to \mathsf{List}\, k \to \mathsf{List}\, k \\
 & \equiv & \lambda A{:}\mathsf{Set}.\ \mathsf{Constr}(1, \mathsf{List}\, A) \\
\mathsf{append} & : & \Pi A{:}\mathsf{Set}.\ \mathsf{List}\, A \to \mathsf{List}\, A \to \mathsf{List}\, A \\
\mathsf{nth} & : & \Pi A{:}\mathsf{Set}.\ A \to \mathsf{List}\, A \to \mathsf{Nat} \to A
\end{array}
$$

Οι συναρτήσεις που αναφέρονται, έχουν ως σκοπό τον αποδοτικότερο έλεγχο των λιστών, δίνοντας δυνατότητα για προσθήκη ενός στοιχείου στο τέλος μιας λίστας και την εύρεση του n-οστού στοιχείου. Η πρώτη υλοποιείται διασχίζοντας την λίστα αναδρομικά, και εφαρμόζοντας διαδοχικά cons για κάθε στοιχείο της αρχικής σε μία λίστα από ένα στοιχείο, αυτό που θέλουμε να προσθέσουμε Η εύρεση του n-οστού στοιχείου γίνεται εφαρμόζοντας επαναληπτικό βρόχο, με ένα elim , και περνώντας σαν παράμετρο κάθε φορά την ουρά (tail) της λίστας.

5.2 Έλεγχος της υλοποίησης

Για τον έλεγχο της υλοποίησής μας, χρησιμοποιήσαμε ένα σύνολο όρων της γλώσσας τύπων. Ορισμένοι από αυτούς πραγματοποιούν αριθμητικούς υπολογισμούς μεταξύ ακεραίων, ενώ άλλοι αφορούν στην απόδειξη θεωρημάτων.

5.2.1 Παραδείγματα υπολογισμών

Οι αριθμητικοί υπολογισμοί που επιλέξαμε για τον έλεγχο της απόδοσης του συστήματος είναι η συνάρτηση του παραγωντικού και η συνάρτηση που επιστρέφει κάποιο όρο της ακολουθίας Fibonacci. Οι όροι που πραγματοποιούν αυτούς τους υπολογισμούς είναι:

$$
\begin{array}{lcl}
\mathsf{fibonacci} & \equiv & \mathsf{pnatRec}\,(\mathsf{PNat} \to \mathsf{PNat} \to \mathsf{PNat}) \\
 & & \quad (\lambda \mathsf{x}{:}\mathsf{PNat}.\ \lambda \mathsf{y}{:}\mathsf{PNat}.\ \mathsf{y}) \\
 & & \quad (\lambda \mathsf{p}{:}\mathsf{PNat}.\ \lambda \mathsf{z}{:}\mathsf{PNat} \to \mathsf{PNat} \to \mathsf{PNat}.\ \lambda \mathsf{x}{:}\mathsf{PNat}.\ \lambda \mathsf{y}{:}\mathsf{PNat}.\ \mathsf{z}\,\mathsf{y} \\
 & & \qquad (\mathsf{pnatPlus}\,\mathsf{x}\,\mathsf{y})) \\
 & & \quad \mathsf{pnat1}\,\mathsf{pnat1} \\
\mathsf{factorial} & \equiv & \mathsf{pnatRec}\,\mathsf{PNat}\,\mathsf{pnat1}\,(\lambda \mathsf{x}{:}\mathsf{PNat}.\ \mathsf{pnatTimes}\,(\mathsf{pnatSucc}\,\mathsf{x}))\,\mathsf{x}
\end{array}
$$

Ο υπολογισμός του n-οστού αριθμού Fibonacci γίνεται με έναν επαναληπτικό με δύο μεταβλητες, όπου σε κάθε βήμα αντικαθιστούμε κυκλικά τη μια με το αθροισμα τους. Σε γλώσσα C το αντίστοιχο πρόγραμμα θα ήταν

```
int fibonacci (int x)
{
  int a = 1,b = 1;
  for(int i=0 ; i <= x ; i++)
  {
    b = a + b;
    a = b - a;
  }
  return b;
}
```

Η συνάρτηση fibonacci ερμηνεύεται ως εξής:

Γίνεται ανδρομική κλήση στην παράμετρο, με βάση τον επόμενο αριθμό. Αυτό πραγματοποιείται με την συνάρτηση pnatRec η οποία αναφέρθηκε παραπάνω. Το αποτέλεσμα της αναδρομικής κλήσης είναι μια συνάρτηση που παίρνει ως παραμέτρους δυο ακεραίους και επιστρέφει ακέραιο.

- Αν η παράμετρος της αναδρομικής κλήσης είναι pnat1, επιστρέφεται η συνάρτηση που επιστρέφεται έχει τιμή την τιμή της δεύτερης παραμέτρου της (κατά αντιστοιχεία με το *return b* του παραπάνω προγράμματος).

- Διαφορετικά επιστρέφεται η αναδρομική κλήση πάνω στην παράμετρο της pnatRec, με ορίσματα το αθροισμα των δύο προηγουμενων, και τον προηγούμενο αριθμό (κατά αντιστοιχεία με τα $a = b$ και $b = \alpha+\beta$ που θα μπορούσε να πει κανείς ότι είναι οι "παράμετροι" κάθε επανάληψης του βρόχου στο παραπάνω πρόγραμμα).

Η συνάρτηση factorial είναι αρκετά πιο απλή. Εκτελεί αναδρομική κλήση πάνω στην παράμετρο της, επιστρέφοντας έναν ακέραιο, που είναι και το αποτέλεσμα που θέλουμε.

- Αν η παράμετρος είναι pnat1, τότε το αποτέλεσμα είναι pnat1.

- Διαφορετικά το αποτέλεσμα είναι το γινόμενο του αριθμού με την αναδρομική κλήση της συνάρτησης στον προηγούμενο του. Επειδή η pnatRec, όπων και η Elim δεν παρέχει δυνατότητα να χρησιμοποιηθεί η ίδια η παράμετρος, αλλά μονο η πραμετρος του κατασκευαστή από το οποίο προέρχεται, ο πρώτος όρος του γίνομενου πρέπει να υπολογιστεί. Εδώ, ως κατασκευαστές θεωρούνται οι pnat1 και pnatSucc, οποότε αρκεί μια κλήση στην pnatSucc για να πάρουμε το ζητούμενο.

5.2.2 Παραδείγματα θεωρήματων

Εκτός από τις μαθηματικές συνρτήσεις, διατυπώσαμε και κάποια απλά θεωρήματα. Τα θεωρήματα που διατυπώσαμε είναι τα εξής:

- propContradiction

  Το τετριμένο, ότι η ψευδής απόδειξη συνεπάγεται τα πάντα.

- fEqual

  Απόδειξη ότι δεδομένων δύο στοιχείων ενός συνόλου και μιας συνάρτησης από αυτό το σύνολο σε κάποιο άλλο, αν τα δύο στοιχεία είναι ίσα τότε και η εφαρμογές τους πάνω στην συνάρτηση είναι ίσες.

- propLeibniz

  Απόδειξη οτι αν μία πρόταση ισχύει για ένα στοιχείο ενός συνόλου, τότε ισχύει για κάθε στοιχείο του συνόλου αυτού.

- pnat_inj_pnat2x

  Απόδειξη ότι αν το διπλάσιο δύο θετικών φυσικών αριθμών είναι ίσο, τότε οι αριθμοί αυτοί είναι ίσοι.

```
propContradiction  ≡  λA:Set. λH:propFalse.
                          Elim[λx:propFalse. A](H:propFalse){}
fEqual             ≡  λA:Set. λB:Set. λx:A. λy:A. λf:A → B. λH:propEq A x y.
                          Elim[λa:A. λb:A. λe:propEq A a b. propEq B (f a) (f b)](
                              H:propEq A x y){λa:A. propEqRefl B (f a)}
propLeibniz        ≡  λA:Set. λx:A. λP:A → Set. λf:P x. λy:A. λp:propEq A x y.
                          (Elim[λa:A. λb:A. λe:propEq A x a. (P a) → (P b)](
                              p:propEq A x y){λa:A. λf:P a. f}) f
pnat_inj_pnat2x    ≡  λp:PNat. λq:PNat. λH:propEq PNat (pnat2x p) (pnat2x q).
                          (fEqual PNat PNat) (pnat2x p) (pnat2x q) pnat_half H
```

5.2.3 Μετρήσεις απόδοσης

Για να αξιολογηθεί το σύστημα εκτελέσαμε μια σειρά από δοκιμές, τις οποίες και χρονομετρήσαμε. Αρχικά μετρήθηκε ο χρόνος για τον έλεγχο τύπων των σταθερών που ορίζονται σε κάθε άρθρωμα.

Όπως έχει αναφερθεί στη σχετική με τις σταθερές παράγραφο 4.4, κατα τον ορισμό μιας σταθερές παρατίθεται και ο τύπος της. Ο έλεγχος που πραγματοποιείται σε αυτή τη δοκιμή συνίσταται στην προσπάθεια εύρεση του τύπου κάθε σταθερού όρου και αν αυτό επιτύχει στη σύγκριση του με αυτον που έχει δωθεί κατά τον ορισμό του. Στην περίπτωση που δεν επιτύχει η εύρεση τύπου για κάποιον όρο, κατί που σημαίνει οτι δεν είναι σωστά διατυπωμένος, το πρόγραμμα τερματίζεται εμφανίζοντας κατάλληλο μήνυμα λάθος.

Παρακάτω φαίνεται η έξοδος του προγάμματος. Αρχικά εμφανίζεται το όνομα του αρθρώματος στο οποίο ανήκουν οι σταθεροί όροι. Για κάθε όρο του αρθρώματος εμφανίζεται ένα μηνύμα επιτυχίας, ότι δηλαδή ο τύπος που βρέθηκε είναι ισοδύναμος με αυτόν που είχε δοθεί αρχικά, και στη συνέχεια το όνομα του όρου με τον τύπο του. Στο τέλος κάθε αρθρώματος εμφανίζεται ο χρόνος που χρειάστικε για όλους τους όρους αυτού. Στους όρους που ακολοθούν, το γινόμενο εμφανίζεται ως

```
#x:A. B
```

Οι αφαιρέσεις $\lambda x{:}A.B$ εμφανίζονται ως

```
\x:A. B
```

ενώ οι μεταβλητές εμφανίζονται τόσο με τον de Bruijn δείκτη τους, όσο και με το όνομα με το οποίο είχαν ορισθεί, πριν τη μετατροπή τους από πρωτογενείς όρους, στη μορφη:

$$\textit{ονομα_μεταβλητής\$δείκτης}$$

Άλλης μορφής όροι, εκτός φυσικά από τη εφαρμογή που δεν χρειάζεται κάποιο συμβολισμό, δεν εμφανίζονται, αφου τόσο οι επαγωγικοί ορισμοί όσο και κατασκευαστές εμφανίζονται με το όνομα του όρου με το οποίο έχουν οριστεί, το οποίο εμφανίζεται ως το όνομα του αρθρώματος και των υπό-αρθρωμάτων στο οποίο ανοίκει και το όνομα του, χωρισμένα με "." . Το εξωτερικότερο αρθρωμα *outer* δεν εμφανίζεται πουθενά.

```
Typechecking terms of module: Bool
OK:     Bool.Bool : Set
OK:     Bool.true : Bool.Bool
OK:     Bool.false : Bool.Bool
OK:     Bool.not : Bool.Bool -> Bool.Bool
OK:     Bool.and : Bool.Bool -> Bool.Bool -> Bool.Bool
OK:     Bool.or : Bool.Bool -> Bool.Bool -> Bool.Bool
OK:     Bool.CondS : #k:Set. Bool.Bool -> k$1 -> k$2 -> k$3
OK:     Bool.CondT : #z:Type. Bool.Bool -> z$1 -> z$2 -> z$3
time :  0.001722

Typechecking terms of module: Pair
OK:     Prop.propAnd : Set -> Set -> Set
OK:     Pair.pair : #A:Set. #B:Set. A$1 -> B$1 -> Prop.propAnd A$3 B$2
OK:     Pair.fst : #A:Set. #B:Set. Prop.propAnd A$1 B$0 -> A$2
OK:     Pair.snd : #A:Set. #B:Set. Prop.propAnd A$1 B$0 -> B$1
time :  0.002117

Typechecking terms of module: Prop
OK:     Prop.propTrue : Set
```

```
OK:     Prop.propFalse : Set
OK:     Prop.propTruePrf : Prop.propTrue
OK:     Prop.propNot : Set -> Set
OK:     Prop.propAnd : Set -> Set -> Set
OK:     Prop.propOr : Set -> Set -> Set
OK:     Prop.propEq : #k:Set. k$0 -> k$1 -> Set
OK:     Prop.propEqRefl : #k:Set. #a:k$0. Prop.propEq k$1 a$0 a$0
OK:     Prop.propNe : #k:Set. k$0 -> k$1 -> Set
OK:     Prop.Comparison : Set
OK:     Prop.compEq : Prop.Comparison
OK:     Prop.compLt : Prop.Comparison
OK:     Prop.compGt : Prop.Comparison
OK:     Prop.compIsEq : Prop.Comparison -> Bool.Bool
OK:     Prop.compIsNe : Prop.Comparison -> Bool.Bool
OK:     Prop.compIsLt : Prop.Comparison -> Bool.Bool
OK:     Prop.compIsGt : Prop.Comparison -> Bool.Bool
OK:     Prop.compIsLe : Prop.Comparison -> Bool.Bool
OK:     Prop.compIsGe : Prop.Comparison -> Bool.Bool
time :  0.002678

Typechecking terms of module: PNat
OK:     PNat.PNat : Set
OK:     1  : PNat.PNat
OK:     PNat.pnat2x : PNat.PNat -> PNat.PNat
OK:     PNat.pnat2x1 : PNat.PNat -> PNat.PNat
OK:     PNat.pnat2xMinus1 : PNat.PNat -> PNat.PNat
OK:     PNat.pnatSucc : PNat.PNat -> PNat.PNat
OK:     PNat.pnatPlus : PNat.PNat -> PNat.PNat -> PNat.PNat
OK:     PNat.pnatTimesAuxil : PNat.PNat -> (PNat.PNat -> PNat.PNat) ->
           PNat.PNat -> PNat.PNat
OK:     PNat.pnatTimes : PNat.PNat -> PNat.PNat -> PNat.PNat
OK:     PNat.pnatCompareC : PNat.PNat -> PNat.PNat -> Prop.Comparison ->
           Prop.Comparison
OK:     PNat.pnatCompare : PNat.PNat -> PNat.PNat -> Prop.Comparison
OK:     PNat.pnatEq : PNat.PNat -> PNat.PNat -> Bool.Bool
OK:     PNat.pnatNe : PNat.PNat -> PNat.PNat -> Bool.Bool
OK:     PNat.pnatLt : PNat.PNat -> PNat.PNat -> Bool.Bool
OK:     PNat.pnatGt : PNat.PNat -> PNat.PNat -> Bool.Bool
OK:     PNat.pnatLe : PNat.PNat -> PNat.PNat -> Bool.Bool
OK:     PNat.pnatGe : PNat.PNat -> PNat.PNat -> Bool.Bool
time :  0.02388

Typechecking terms of module: Nat
OK:     Nat.nat0 : Nat.Nat
OK:     Nat.natP : PNat.PNat -> Nat.Nat
OK:     Nat.pnat2xMinus2X : PNat.PNat -> Nat.Nat
OK:     Nat.nat2x : Nat.Nat -> Nat.Nat
OK:     Nat.nat2x1 : Nat.Nat -> Nat.Nat
OK:     Nat.nat2xMinus1X : Nat.Nat -> Nat.Nat
OK:     Nat.pnatPred : PNat.PNat -> Nat.Nat
OK:     Nat.natPred : Nat.Nat -> Nat.Nat
```

```
OK:     Nat.natSucc : Nat.Nat -> Nat.Nat
OK:     Nat.natPlus : Nat.Nat -> Nat.Nat -> Nat.Nat
OK:     Nat.pnatMinusX : PNat.PNat -> PNat.PNat -> Nat.Nat
OK:     Nat.natMinusX : Nat.Nat -> Nat.Nat -> Nat.Nat
OK:     Nat.natTimes : Nat.Nat -> Nat.Nat -> Nat.Nat
OK:     Nat.natCompare : Nat.Nat -> Nat.Nat -> Prop.Comparison
OK:     Nat.natEq : Nat.Nat -> Nat.Nat -> Bool.Bool
OK:     Nat.natNe : Nat.Nat -> Nat.Nat -> Bool.Bool
OK:     Nat.natLt : Nat.Nat -> Nat.Nat -> Bool.Bool
OK:     Nat.natGt : Nat.Nat -> Nat.Nat -> Bool.Bool
OK:     Nat.natLe : Nat.Nat -> Nat.Nat -> Bool.Bool
OK:     Nat.natGe : Nat.Nat -> Nat.Nat -> Bool.Bool
OK:     Nat.natLT : Nat.Nat -> Nat.Nat -> Set
OK:     Nat.pnatDivModAuxil : PNat.PNat -> PNat.PNat ->
           Prop.propAnd Nat.Nat Nat.Nat
OK:     Nat.natDiv : Nat.Nat -> Nat.Nat -> Nat.Nat
OK:     Nat.natMod : Nat.Nat -> Nat.Nat -> Nat.Nat
time :  0.080693

Typechecking terms of module: Int
OK:     Int.Int : Set
OK:     Int.int0 : Int.Int
OK:     Int.intP : PNat.PNat -> Int.Int
OK:     Int.intN : PNat.PNat -> Int.Int
OK:     Int.int2x : Int.Int -> Int.Int
OK:     Int.int2x1 : Int.Int -> Int.Int
OK:     Int.int2xMinus1 : Int.Int -> Int.Int
OK:     Int.intNeg : Int.Int -> Int.Int
OK:     Int.pnatMinus : PNat.PNat -> PNat.PNat -> Int.Int
OK:     Int.intPlus : Int.Int -> Int.Int -> Int.Int
OK:     Int.intMinus : Int.Int -> Int.Int -> Int.Int
OK:     Int.intTimes : Int.Int -> Int.Int -> Int.Int
OK:     Int.intDiv : Int.Int -> Int.Int -> Int.Int
OK:     Int.intMod : Int.Int -> Int.Int -> Int.Int
OK:     Int.intCompare : Int.Int -> Int.Int -> Prop.Comparison
OK:     Int.intEq : Int.Int -> Int.Int -> Bool.Bool
OK:     Int.intNe : Int.Int -> Int.Int -> Bool.Bool
OK:     Int.intLt : Int.Int -> Int.Int -> Bool.Bool
OK:     Int.intGt : Int.Int -> Int.Int -> Bool.Bool
OK:     Int.intLe : Int.Int -> Int.Int -> Bool.Bool
OK:     Int.intGe : Int.Int -> Int.Int -> Bool.Bool
OK:     Int.intLT : Int.Int -> Int.Int -> Set
time :  0.175389

Typechecking terms of module: List
OK:     List.List : Set -> Set
OK:     List.nil : #A:Set. List.List A$0
OK:     List.cons : #A:Set. A$0 -> List.List A$1 -> List.List A$2
OK:     List.append : #A:Set. List.List A$0 -> List.List A$1 ->
           List.List A$2
OK:     List.nth : #A:Set. A$0 -> List.List A$1 -> Nat.Nat -> A$3
```

```
OK:     List.pnatRepeat : #A:Set. PNat.PNat -> A$1 -> List.List A$2
OK:     List.repeat : #A:Set. Nat.Nat -> A$1 -> List.List A$2
time :  0.00761
```

Στη συνέχεια μετρήθηκε ο χρόνος για τον έλεγχο τύπων των θεωρημάτων της παραγράφου 5.2.2.

```
OK:     Proof.propContradiction : #A:Set. Prop.propFalse -> A$1
time :  0.000019

OK:     Proof.fEqual : #A:Set. #B:Set. #x:A$1. #y:A$2.
           #f:A$3 -> B$3. Prop.propEq A$4 x$2 y$1 ->
              Prop.propEq B$4 (f$1 x$3) (f$1 y$2)
time :  0.000303

OK:     Proof.propEqLeibniz : #A:Set. #x:A$0. #P:A$1 -> Set.
           P$0 x$1 -> #y:A$3. Prop.propEq A$4 x$3 y$0 -> P$3 y$1
time :  0.001087

OK:     Proof.pnat_half : PNat.PNat -> PNat.PNat
time :  0.000165

OK:     Proof.pnat_inj_pnat2x : #p:PNat.PNat. #q:PNat.PNat.
           Prop.propEq PNat.PNat (PNat.pnat2x p$1) (PNat.pnat2x q$0) ->
              Prop.propEq PNat.PNat p$2 q$1
time :  0.001186
```

Τέλος, μετρήθηκε ο χρόνος για την αποτίμηση ορισμένων όρων, οι οποίοι χρησιμοποιούν τη συνάρτηση του παραγοντικού και τη συνάρτηση υπολογισμού των όρων της ακολουθίας Fibonnaci, που ορίστηκαν στην παράγραφο 5.2.1.

Προκειμένου να είναι ευανάγνωστοι οι όροι, οι ακολουθίες από τους κατασκευαστές ακεραίων μετατρέπονται από το προγραμμα στον αριθμό τον οποίο παριστάνουν.

```
Trying: factorial 4
typechecking:   0.000099
interpreting:   2.505112
Result is: 24

Trying: factorial 5
typechecking:   0.000109
interpreting:   20.016020
Result is: 120

Trying: fibonacci 5
typechecking:   0.000591
interpreting:   0.991527
Result is: 8

Trying: fibonacci 8
typechecking:   0.000326
interpreting:   19.892813
Result is: 34
```

Παρατηρήσαμε ότι η αποτίμηση ακόμα και απλών όρων, όπως ο υπολογισμός του 8ου αριθμού της ακολουθίας Fibonacci ή του 5!, χρειάζονται περισσότερο χρόνο από αυτον που θα θέλαμε. Το γεγονός αυτό μας οδηγεί στο συμπέρασμα ότι παρ' ότι το σύστημα είναι σταθερό, υπάρχουν σημαντικά περιθώρια βελτίωσης στην απόδοση του.

Πίνακας 5.1: Μετρήσεις χρόνου εύρεσης τύπου.

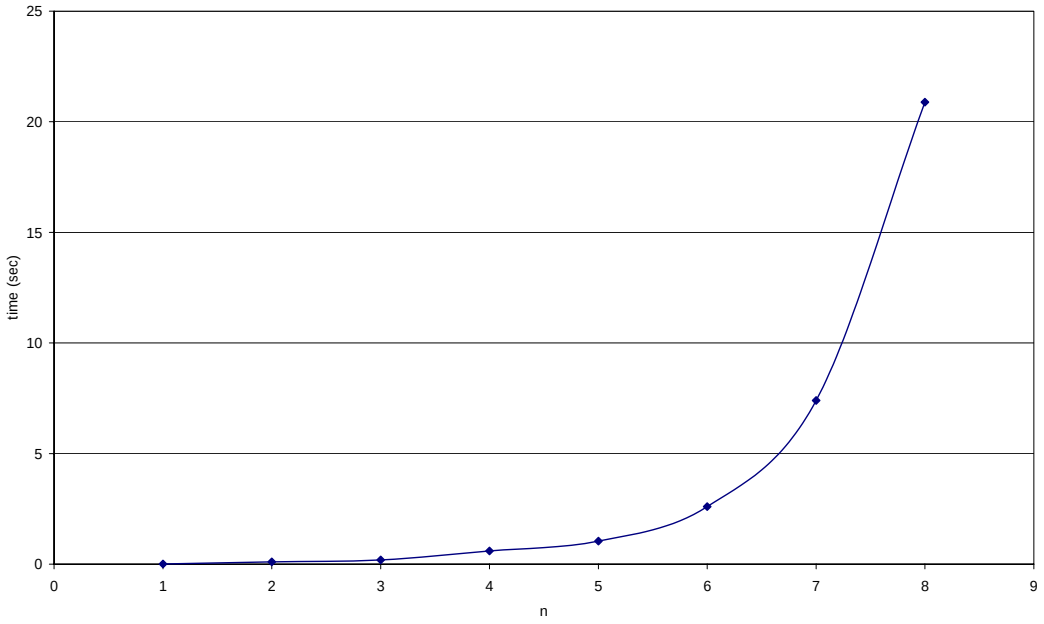
Bool	0.001722
Prop	0.002117
Prop	0.002678
Pnat	0.023880
Nat	0.080693
Int	0.175389
List	0.007610
propContradiction	0.000019
fEqual	0.000303
propEqLeibniz	0.001087
pnat_half	0.000165
pnat_inj_pnat2x	0.001186

Πίνακας 5.2: Μετρήσεις χρόνου αποτίμησης της συνάρτησης fibonacci.

fibonacci 1	0.005848
fibonacci 2	0.100498
fibonacci 3	0.190525
fibonacci 4	0.597450
fibonacci 5	1.044217
fibonacci 6	2.605271
fibonacci 7	7.398726
fibonacci 8	20.887324

Πίνακας 5.3: Μετρήσεις χρόνου αποτίμησης της συνάρτησης factorial.

factorial 4	2.505112
factorial 5	20.016020



Σχήμα 5.1: Χρόνος υπολογισμού της ακολουθίας Fibonacci.

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

Η συνεισφορά και τα συμπεράσματα της παρούσας εργασίας συνοψίζονται στα παρακάτω:

- Υλοποιήθηκε μια γλώσσα τύπων, βασισμένη στο λογισμό των επαγωγικών κατασκευών. Η γλώσσα αυτή είναι επέκταση εκείνης που ορίζεται στην εργασία των Shao *et al.* [Shao02]. Η υλοποίηση περιλαμβάνει ελεγκτή τύπων (type checker) και διερμηνέα (interpreter) για τη γλώσσα.

- Έμφαση δόθηκε σε θέματα απόδοσης της υλοποίησης. Για το λόγο αυτό, χρησιμοποιήθηκαν οι τεχνικές του hash-consing, των δεικτών de-Bruijn και της μετατροπής σε ασθενή κανονική μορφή κεφαλής (weak head normal form). Ως γλώσσα προγραμματισμού χρησιμοποιήθηκε η OCaml. Σε σχέση με μια παλιότερη υλοποίηση της γλώσσας τύπων, που είχε γίνει σε Haskell, η παρούσα υλοποίηση είναι ταχύτερη περίπου κατά μια τάξη μεγέθους.

- Από τα αποτελέσματα που παρατηρήθηκαν, διαπιστώθηκε η εφικτότητα της αποδοτικής υλοποίησης αυτής της γλώσσας. Η παρούσα υλοποίηση θα είναι χρήσιμη στο πλαίσιο της ευρύτερης ερευνητικής προσπάθειας που περιγράφεται στην επόμενη και τελευταία παράγραφο.

6.2 Μελλοντική έρευνα

Μακροπρόθεσμος στόχος αυτής της ερευνητικής προσπάθειας είναι η δημιουργία ενός πλαισίου για την εξασφάλιση της αξιοπιστίας του εκτελέσιμου κώδικα. Τα παραπάνω μπορούν να ενταχθούν σε μια γενικευμένη έννοια "αξιοπιστίας" ώστε να οριστεί ένα πλαίσιο ικανό να επαληθεύει αυθαίρετες ιδιότητες του κώδικα, εκφρασμένες ως προδιαγραφές υψηλού επιπέδου σε μια κατάλληλη τυπική λογική. Επιθυμητές ιδιότητες ενός τέτοιου πλαισίου είναι οι παρακάτω:

- *Τυπικότητα*: η τυπική λογική που θα χρησιμοποιηθεί για τον ορισμό προδιαγραφών των συστημάτων λογισμικού θα είναι ορατή και διαθέσιμη ως εργαλείο για τη διενέργεια συλλογισμών στο επίπεδο της γλώσσας προγραμματισμού.

- *Εκφραστική ικανότητα*: η τυπική λογική που θα χρησιμοποιηθεί πρέπει να είναι αρκετά ισχυρή ώστε να μπορεί να εκφράσει και να επαληθεύει αυθαίρετες ιδιότητες συστημάτων λογισμικού. Από την άλλη πλευρά, η αυτόματη εύρεση αποδείξεων για αυθαίρετα θεωρήματα σε μια τέτοια λογική δεν είναι γενικά δυνατή. Αυτό σημαίνει ότι, στην γενική περίπτωση, οι προγραμματιστές πρέπει να παρέχουν βοήθεια στην απόδειξη ιδιοτήτων των προγραμμάτων τους και υπάρχουν τουλάχιστον δύο ενδιαφέροντα ανοικτά ερευνητικά προβλήματα που σχετίζονται με αυτό:

  - Πρώτον, δεν είναι ξεκάθαρο αν αυτό είναι μια πρακτική λύση, με δεδομένη την απέχθεια και αναποτελεσματικότητα του μέσου προγραμματιστή όσον αφορά στις τυπικές αποδείξεις.

– Δεύτερον, είναι ανοικτό το ερώτημα αν υπάρχει ένα ενδιαφέρον σύνολο από σχετικά απλές ιδιότητες προγραμμάτων, για τις οποίες να είναι δυνατόν αυτόματα να βρεθούν αποδείξεις σε αυτό το γενικό πλαίσιο.

- *Ευελιξία*: ανεξαρτησία από την αρχική γλώσσα (πλατφόρμα ανάπτυξης), ανεξαρτησία από την τελική γλώσσα (πλατφόρμα εκτέλεσης), ευέλικτη πολιτική ασφαλείας.

- *Κλιμάκωση* (scalability): όταν το μέγεθος του κώδικα αυξάνει, η πολυπλοκότητα της ανάπτυξης και πιστοποίησής του δε θα πρέπει να αυξάνει δυσανάλογα.

Η γλώσσα των τύπων που υλοποιήθηκε στην παρούσα εργασία εξυπηρετεί εν μέρει το μακροπρόθεσμο στόχο. Στο πλαίσιο της μελλοντικής εργασίας για την ανάπτυξη ενός τέτοιου συστήματος πιστοποίησης κώδικα, είναι αναγκαίο:

- Να διατυπωθούν χρήσιμες θεωρίες στη γλώσσα τύπων. Η κωδικοποίηση των θεωριών από τα μαθηματικά είναι απαραίτητη για την αποδεικτική διαδικασία προτάσεων που χρησιμοποιούνται στις προδιαγραφές ασφαλείας. Ένα σχετικά πλήρες σύνολο από θεωρίες έχει διατυπωθεί σε σύστημα τύπων παρόμοιο με τον λογισμό των επαγωγικών κατασκευών, από την ομάδα του Coq, και έχει ενσωματωθεί στις βιβλιοθήκες του συστήματος αυτού. Η "μετάφραση" των θεωριών αυτών στη δική μας γλώσσα τύπων είναι σχετικά εύκολο να συστηματοποιηθεί.

- Να οριστεί και να υλοποιηθεί μια κατάλληλη γλώσσα υπολογισμών. Η γλώσσα αυτή θα αποτελεί την ενδιάμεση γλώσσα στην οποία θα μεταγλωττίζονται όλες οι δόμικές μονάδες της αρχικής γλώσσας προγραμματισμού.

- Να οριστεί και να υλοποιηθεί κατάλληλα μια γλώσσα προγραμματισμού υψηλού επιπέδου, η οποία θα εκμεταλλεύεται τη γλώσσα τύπων και θα μεταφράζεται στη γλώσσα υπολογισμών.

- Σημαντικό είναι επίσης να βελτιωθεί η υλοποίηση της γλώσσας τύπων που παρουσιάστηκε σε αυτή την εργασία. Όπως φάνηκε και από τις μετρήσεις απόδοσης, σημαντικά περιθώρια βελτίωσης υπάρχουν στον αλγόριθμο που εκτελεί τους υπολογισμούς. Τέλος απαραίτητη είναι η ανάπτυξη ενός αλγορίθμου χειρισμού των σφαλμάτων, έτσι ώστε να γίνονται αντιληπτά από τον χρήστη με ευκολία.

Βιβλιογραφία

[Appe00] A. W. Appel and A. P. Felty, "A Semantic Model of Types and Machine Instructions for Proof-Carrying Code", in *Proceedings of the 27th Annual Symposium on Principles of Programming Languages (POPL 2000)*, pp. 243–253, ACM Press, 2000.

[Appe01] A. W. Appel, "Foundational Proof-Carrying Code", in *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pp. 247–258, June 2001.

[Bare84] H. P. Barendregt, *The Lambda Calculus*, North Holland, revised edition, 1984.

[Bare92] H. P. Barendregt, "Lambda Calculi with Types", in S. Abramsky, D.M. Gabbay and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 2, Oxford Science Publications, 1992.

[Bare01] H. Barendregt and H. Geuvers, "Proof-Assistants Using Dependent Type Systems", in A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, vol. II, chapter 18, pp. 1149–1238, Elsevier Science Publishers, B.V., 2001.

[Bera88] S. Berardi, "Towards a Mathematical Analysis of the Coquand-Huet Calculus of Constructions and the other Systems in Barendregt's cube", Technical report, 1988.

[Chur32] A. Church, "A Set of Postulates for the Foundations of Logic", *Annals of Mathematics*, vol. 33, pp. 346–366, 1932. Part I.

[Chur33] A. Church, "A Set of Postulates for the Foundations of Logic", *Annals of Mathematics*, vol. 34, pp. 839–864, 1933. Part II.

[Chur40] A. Church, "A Formulation of a Simple Theory of Types", *Journal of Symbolic Logic*, vol. 5, pp. 56–58, 1940.

[Chur41] A. Church, *The Calculi of Lambda-Conversion*, vol. 6 of *Annals of Mathematical Studies*, Princeton University Press, Princeton, NJ, 1941.

[Coqu88] T. Coquand and G. Huet, "The Calculus of Constructions", *Information and Computation*, vol. 76, no. 2/3, pp. 95–120, 1988.

[Coqu90] T. Coquand, "Metamathematical investigations of a calculus of constructions", in P. Odifreddi, editor, *Logic and Computer Science*, vol. 31 of *APIC Studies in Data Processing*, pp. 91–122, Academic Press, London, UK, 1990.

[Crar02] K. Crary and J. C. Vanderwaart, "An Expressive, Scalable Type Theory for Certified Code", in *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pp. 191–205, Pittsburgh, PA, 2002.

[Curr68] H. B. Curry and R. Feys, *Combinatory Logic*, vol. 1, North Holland, 2nd edition, 1968.

[dBru72] N.G. de Bruijn, "Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation", *Indagationes Mathematicae*, vol. 34, pp. 381–392, 1972.

[Fill00] J.-C. Filliâtre, "Hash consing in an ML framework", Research Report 1368, LRI, Université Paris Sud, September 2000.

[Geuv92] H. Geuvers, "The calculus of constructions and higher order logic", in Ph. de Groote, editor, *The Curry-Howard Isomorphism*, vol. 8 of *Cahiers du Centre de logique*, pp. 139–191, Academia, Louvain-la-Neuve, Belgium, August 1992.

[Gira72] J.-Y. Girard, *Interprétation fonctionelle et élimination des coupures de l' arithmétique d' ordre supérieur*, Ph.D. thesis, Universit'e Paris VII, 1972.

[Gira89] J.-Y. Girard, Y. Lafont and P. Taylor, "Proofs and Types", *Tracks in Theoretical Computer Science*, 1989.

[Harp95] R. Harper and G. Morrisett, "Compiling Polymorphism Using Intensional Type Analysis", in *Proceedings of the 22nd Annual Symposium on Principles of Programming Languages (POPL 1995)*, pp. 130–141, ACM Press, 1995.

[Howa80] W. A. Howard, "The Formulae-as-Types Notion of Constructions", in *To H. B. Curry: Essays on Computation Logic, Lambda Calculus and Formalism*, Academic Press, 1980.

[Morr98] G. Morrisett, D. Walker, K. Crary and N. Glew, "From System F to Typed Assembly Language", in *Proceedings of the 25th Annual Symposium on Principles of Programming Languages (POPL 1998)*, pp. 85–97, ACM Press, January 1998.

[Necu96] G. Necula and P. Lee, "Safe Kernel Extensions without Run-Time Checking", in *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, pp. 229–243, USENIX Association, 1996.

[Necu97] G. Necula, "Proof-Carrying Code", in *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL 1997)*, pp. 106–119, New York, January 1997, ACM Press.

[Necu98] G. Necula, *Compiling with Proofs*, Ph.D. thesis, Carnegie Mellon University, September 1998.

[Paul93] Christine Paulin-Mohring, "Inductive Definitions in the System Coq: Rules and Properties", in M. Bezem and J. F. Groote, editors, *Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications (TLCA'93)*, vol. 664 of *Lecture Notes in Computer Science*, pp. 328–345, Springer-Verlag, Berlin, 1993.

[Paul96] C. Paulin-Mohring, *Définitions Inductives en Théorie des Types d' Ordre Supérieur*, Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[Pfen90] F. Pfenning and C. Paulin-Mohring, "Inductively defined types in the Calculus of Constructions", in *Proceedings of Mathematical Foundations of Programming Semantics*, vol. 442 of *Lecture Notes in Computer Science*, Berlin, 1990, Springer-Verlag. technical report CMU-CS-89-209.

[Reyn74] J. C. Reynolds, "Towards a Theory of Type Systems", in Ehring et al., editor, *Lecture Notes in Computer Science*, vol. 19, pp. 408–425, Springer-Verlag, 1974.

[Shao98] Zhong Shao, "Implementing Typed Intermediate Language", in *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pp. 313–323, Baltimore, Maryland, September 1998.

[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.

[Wern94] B. Werner, *Une Théorie des Constructions Inductives*, Ph.D. thesis, Université Paris VII, Paris, France, May 1994.