



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

**Αναφορές και Ανάθεση
στον Προγραμματισμό με Αποδείξεις**

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΗΣΤΟΣ Ι. ΔΗΜΟΥΛΑΣ

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

Αθήνα, Ιούλιος 2006



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Αναφορές και Ανάθεση
στον Προγραμματισμό με Αποδείξεις

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΗΣΤΟΣ Ι. ΔΗΜΟΥΛΑΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Ιουλίου 2006.

.....
Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

.....
Ευστάθιος Ζάχος
Καθηγητής Ε.Μ.Π.

.....
Κωνσταντίνος Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2006

Χρήστος Ι. Δημουλάς

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

Copyright © Χρήστος Ι. Δημουλάς, 2006.
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 μελέτη της δυνατότητας εισαγωγής αναφορών (*references*) και ανάθεσης (*destructive update*) στο μοντέλο του προγραμματισμού με αποδείξεις.

Η ανάγκη για την παραγωγή αξιόπιστου και πιστοποιημένου κώδικα είναι ακόμα μεγαλύτερη σήμερα από το παρελθόν. Σε πολλές περιπτώσεις, ζητήματα ασφάλειας και συμβατότητας λογισμικού έθεσαν σε κίνδυνο τη λειτουργία μεγάλων συστημάτων. Τα τελευταία χρόνια η επικρατέστερη μέθοδος παραγωγής πιστοποιημένου κώδικα είναι η χρήση γλωσσών με ισχυρά συστήματα τύπων που ενσωματώνουν διάφορα συστήματα λογικής, μέσω των οποίων εκφράζονται και αποδεικνύονται ιδιότητες που πρέπει να πληρεί το λογισμικό ώστε να είναι ασφαλές.

Οι προσπάθειες για τον ορισμό γλωσσών προγραμματισμού με ισχυρά συστήματα τύπων που υποστηρίζουν αποδείξεις επικεντρώθηκαν σε γλώσσες με αμιγώς συναρτησιακά χαρακτηριστικά. Οι συναρτησιακές, όμως, γλώσσες δεν υποστηρίζουν λειτουργίες όπως η ανάθεση ή οντότητες όπως οι αναφορές. Τα προστακτικά αυτά στοιχεία είναι πολύ χρήσιμα εργαλεία για τον προγραμματιστή γιατί του δίνουν τη δυνατότητα της άμεσης διαμόρφωσης της μνήμης και τη δυνατότητα της ροής των δεδομένων ανάμεσα στα διάφορα τμήματα ενός προγράμματος μέσω της μνήμης.

Η εισαγωγή τέτοιων χαρακτηριστικών σε συναρτησιακές γλώσσες δεν είναι υλοποιήσιμη με τετριμμένο τρόπο. Τα παραδοσιακά συστήματα δεν μπορούν να δώσουν πληροφορίες για τα περιεχόμενα της μνήμης και δεν μπορούν να προσφέρουν μία βάση πάνω στην οποία να διατυπωθούν και να αποδειχθούν λογικές προτάσεις που να περιγράφουν ιδιότητες της μνήμης.

Τα linear συστήματα τύπων προσφέρουν μια βάση που επιτρέπει την καταγραφή πληροφοριών για την κατάσταση της μνήμης και τις μεταβολές της. Έτιθενται, λοιπόν, τα θεμέλια για να περιγραφούν με λογικές προτάσεις οι ιδιότητες της μνήμης μέσα από το σύστημα τύπων μία γλώσσας προγραμματισμού.

Στο πλαίσιο της παρούσας εργασίας, κατ' αρχήν σχεδιάστηκε, ορίστηκε η σημασιολογία και διατυπώθηκε η μεταθεωρία για μία γλώσσα προγραμματισμού, το σύστημα τύπων της οποίας υποστηρίζει linear και unrestricted τύπους και ελεγχόμενη μετατροπή μεταξύ αυτών. Στη συνέχεια σχεδιάστηκε ένα σύστημα προγραμματισμού με αποδείξεις που υποστηρίζει αναφορές και ανάθεση, το σύστημα τύπων του οποίου βασίζεται σε μία παραλλαγή της προηγούμενης γλώσσας.

Λέξεις κλειδιά

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

Abstract

The purpose of this diploma dissertation is the introduction of references and destructive update in a system that produces certified binaries.

The need for reliable and certifiably secure code is even greater today than it was in the past. In many cases, security and software compatibility issues put in danger the operation of large systems. 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.

The attempts to design a strongly typed language that supports programming with proofs focus on the development of purely functional languages. These languages do not support references or destructive update. References and destructive update are very useful tools because they allow the programmer to directly manipulate the memory and control the data flow between different parts of a program.

The introduction of such characteristics in functional languages is not possible in a simple manner. Traditional type systems cannot provide enough information about the contents of the memory and so they cannot offer enough support to formulate and prove logical propositions that can describe the properties of memory.

Linear type systems offer a base that allow to track information about the state and changes in memory. This makes it possible to describe memory properties as logical formulae within a programming language's type system.

In this diploma thesis, we first design, define the operational semantics and formulate the metatheory of a programming language whose type system supports linear and unrestricted types and safe transformation from linear to unrestricted and vice versa. Then, we design a system that supports programming with proofs based on a variation of the previously defined programming language.

Key words

Type systems, references, destructive update, calculus of inductive constructions, programming with proofs, security of executable code.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου, κ. Νίκο Παπασπύρου. Πέρα από το ενδιαφέρον του και τη σημαντική βοήθεια κατά την πορεία της εργασίας, πίστεψε στις δυνατότητές μου, με ενθάρρυνσε και με ενέπνευσε να ασχοληθώ με τις γλώσσες προγραμματισμού. Επίσης, θέλω να ευχαριστήσω τον φίλο και πολύτιμο συνεργάτη Μιχάλη Παπακυριάκου που συνέδραμε στην ολοκλήρωση και συγγραφή της εργασίας. Ακόμα, θέλω να ευχαριστήσω τους γονείς και την αδερφή μου που εξασφάλισαν ιδανικές συνθήκες για να αφοσιωθώ απερίσπαστος στις σπουδές μου. Τέλος θέλω να ευχαριστήσω τα μέλη του Εργαστηρίου Τεχνολογίας Λογισμικού για τη συνεργασία που είχαμε κατά διάρκεια αυτού του χρόνου.

Χρήστος Ι. Δημουλάς,
Αθήνα, 17η Ιουλίου 2006.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-3-06, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2006.

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

Περιεχόμενα

Περίληψη	5
Abstract	7
Ευχαριστίες	9
Περιεχόμενα	11
1. Εισαγωγή	15
1.1 Σκοπός	15
1.2 Προγραμματισμός με Αποδείξεις	15
1.3 Σύνοψη της Εργασίας	16
2. Σχετική Ερευνητική Εργασία	17
2.1 Linear και Substructural Συστήματα Τύπων	17
2.1.1 Ορισμός και Ιδιότητες των Linear και Substructural Συστημάτων Τύπων	18
2.1.2 Εφαρμογές στη Διαχείριση Μνήμης	19
2.1.3 Σύνδεση Linear και Unrestricted Τύπων	24
2.2 Προγραμματισμός με Αποδείξεις	28
2.2.1 Η Βιομηχανική Προσέγγιση	28
2.2.2 Η Ερευνητική Προσέγγιση	28
3. Περιγραφή και Ορισμός της Γλώσσας let!	33
3.1 Οι Τύποι της Γλώσσας	33
3.1.1 Pretypes	33
3.1.2 Qualifiers	33
3.1.3 Scopes	33
3.1.4 Πλήρης Τύποι	34
3.2 Η Σύνταξη της Γλώσσας	34
3.3 Λειτουργική Σημασιολογία	34
3.3.1 Οι Τιμές της Γλώσσας	34
3.3.2 Εκτεταμένη Σύνταξη	34
3.3.3 Store και Scopes	35
3.3.4 Βοηθητικές Συναρτήσεις	35
3.3.5 Σχέση Υπολογισμού	35
3.4 Κανόνες του Συστήματος Τύπων	38
3.4.1 Περιβάλλοντα	38
3.4.2 Μεταβλητές και Σταθερές	39
3.4.3 Εναδικοί και Διαδικοί Τελεστές	39
3.4.4 Συναρτήσεις	40
3.4.5 Αναφορές	41
3.4.6 Εντολή Ελέγχου με Συνθήκη if	41

3.4.7	Ο 'Ορος let!	42
3.4.8	Ο 'Ορος auto!	42
3.5	Μεταθεωρία της Γλώσσας let!	42
3.5.1	Η Σχέση Συμφωνίας Σ και M	42
3.5.2	Safety	43
3.6	Παραδείγματα	43
3.6.1	Increase	43
3.6.2	Swap	44
3.6.3	Reverse	44
3.6.4	Map	44
4.	Περιγραφή της Γλώσσας Τύπων του Συστήματος NFlint	45
4.1	Αναλυτική Σύνταξη	45
4.2	Συνεπυγμένη Σύνταξη (PTS)	46
4.3	Απλοποιημένη Σύνταξη	46
4.4	Βοηθητικοί Κανόνες	46
4.4.1	Ελεύθερες Μεταβλητές	46
4.4.2	Αντικαταστάσεις	46
4.4.3	Θετικές Εμφανίσεις	47
4.4.4	Καλά Ορισμένοι Κατασκευαστές	47
4.4.5	Μικροί Τύποι Κατασκευαστών	47
4.4.6	Εφαρμογή της Επαγγωγής στους Επαγγωγικούς Ορισμούς	48
4.5	Λειτουργική Σημασιολογία	48
4.5.1	Αναγωγές	48
4.5.2	Ισότητες	49
4.6	Κανόνες Τύπων	49
4.6.1	Μεταβλητές και Σταθερές	49
4.6.2	Μετατροπή	49
4.6.3	Γινόμενα, Αφαίρεση και Εφαρμογή	49
4.6.4	Επαγγωγικοί Ορισμοί	50
4.6.5	Πολλαπλές Εκφράσεις, Εξάρτηση	50
4.7	Παραγόμενες Θεωρίες	50
4.7.1	Τιμές Αληθείας	50
4.7.2	Θετικοί Φυσικοί Αριθμοί	50
4.7.3	Φυσικοί Αριθμοί	51
4.7.4	Ακέραιοι Αριθμοί	51
4.7.5	Προτασιακός Λογισμός	51
4.7.6	Λίστες	52
4.7.7	Χαρακτήρες	52
4.8	Μεταθεωρία της Γλώσσας Τύπων	52
5.	Περιγραφή και Ορισμός του Συστήματος liNFlint	55
5.1	Η Γλώσσα Τύπων του Συστήματος liNFlint	55
5.2	Η Γλώσσα Υπολογισμών του Συστήματος liNFlint	55
5.2.1	Ορισμός Βοηθητικών Τύπων της Γλώσσας Υπολογισμών	56
5.2.2	Απλοποιημένη Σύνταξη για τους Βοηθητικούς Τύπους	57
5.2.3	Pretypes της Γλώσσας Υπολογισμών	57
5.2.4	Τύποι των Εκφράσεων της Γλώσσας Υπολογισμών	58
5.2.5	Απλοποιημένη Σύνταξη για τους Pretypes της Γλώσσας Υπολογισμών	58
5.2.6	Απλοποιημένη Σύνταξη για τους Τύπους της Γλώσσας Υπολογισμών	58
5.2.7	Η Σύνταξη της Γλώσσας	58
5.2.8	Περιβάλλοντα	59

5.2.9	Οι Κανόνες του Συστήματος Τύπων	60
5.3	Παραδείγματα	69
6.	Συμπεράσματα	71
6.1	Συνεισφορά	71
6.2	Μελλοντική Έρευνα	71
	Βιβλιογραφία	73

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σκοπός της εργασίας είναι η μελέτη της δυνατότητας εισαγωγής αναφορών (*references*) και ανάθεσης (*destructive update*) στο μοντέλο του προγραμματισμού με αποδείξεις. Κατ' αρχήν σχεδιάστηκε, ορίστηκε η σημασιολογία και διατυπώθηκε η μεταθεωρία για μία γλώσσα προγραμματισμού, το σύστημα τύπων της οποίας υποστηρίζει linear και unrestricted τύπους και ελεγχόμενη μετατροπή μεταξύ αυτών. Στη συνέχεια σχεδιάστηκε ένα σύστημα προγραμματισμού με αποδείξεις που υποστηρίζει αναφορές και ανάθεση, το σύστημα τύπων του οποίου βασίζεται σε μία παραλλαγή της προηγούμενης γλώσσας.

1.2 Προγραμματισμός με Αποδείξεις

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 ασφαλειας και συμβατότητας προγραμμάτων που είχαν ως αποτέλεσμα προβλήματα στην λειτουργία μεγάλων συστημάτων και συνεπώς οικονομικές επιπτώσεις στους οργανισμούς που τα χρησιμοποιούσαν.

Τα συστήματα λογισμικού, έτσι όπως αυτά αναπτύσσονται σήμερα, 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 (*API, application programming interface*) 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 (*interoperability*). 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. Επιπλέον, η τάση που κυριαρχεί είναι να δημιουργούνται συστήματα όπου η εκτέλεση κώδικα γίνεται κατανευμένα 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.

Ο προγραμματισμός με αποδείξεις είναι ένα ελπιδοφόρο μοντέλο που υπόσχεται να αλλάξει τον τρόπο με τον οποίο γίνεται η ανάπτυξη λογισμικού. Η δυνατότητα να περιγραφούν οι ιδιότητες των προγραμμάτων μέσα από τις γλώσσες προγραμματισμού και να ελεγχθεί η εγκυρότητά τους στατικά δίνει στην ασφάλεια του λογισμικού νέες διαστάσεις.

Η στενή σχέση ανάμεσα στη μαθηματική λογική και τη θεωρία τύπων καθιστά τα συστή-

ματα τύπων ιδανικούς φορείς των λογικών προτάσεων και των αποδείξεων τους στον κόσμο των γλωσσών προγραμματισμού. Η πρόοδος, λοιπόν, της έρευνας στο πεδίο των συστημάτων τύπων είναι αλληλένδετη με την ανάπτυξη αποδοτικών και εκφραστικών συστημάτων προγραμματισμού με αποδείξεις.

Οι προσπάθειες για τον ορισμό γλωσσών προγραμματισμού με ισχυρά συστήματα τύπων που υποστηρίζουν αποδείξεις επικεντρώθηκαν σε γλώσσες με αμιγώς συναρτησιακά χαρακτηριστικά. Οι συναρτησιακές, όμως, γλώσσες δεν υποστηρίζουν λειτουργίες όπως η ανάθεση ή οντότητες όπως οι αναφορές. Τα προστακτικά αυτά στοιχεία είναι πολύ χρήσιμα εργαλεία για τον προγραμματιστή γιατί του δίνουν τη δυνατότητα της άμεσης διαμόρφωσης της μνήμης και τη δυνατότητα της ροή των δεδομένων ανάμεσα στα διάφορα τμήματα ενός προγράμματος μέσω της μνήμης.

Η εισαγωγή τέτοιων χαρακτηριστικών σε συναρτησιακές γλώσσες δεν είναι υλοποιήσιμη με τετριμμένο τρόπο. Τα παραδοσιακά συστήματα δεν μπορούν να δώσουν πληροφορίες κατά το χρόνο μεταγλώττισης για την κατάσταση και τα περιεχόμενα της μνήμης. Ακόμα περισσότερο, δεν μπορούν να προσφέρουν μία βάση πάνω στην οποία να διατυπωθούν και να αποδειχθούν λογικές προτάσεις που να περιγράφουν ιδιότητες της μνήμης κατά τη διάρκεια εκτέλεσης του προγράμματος.

Τα linear συστήματα τύπων προσφέρουν ένα ισχυρό υπόβαθρο που μπορεί να καταγράψει αρκετές πληροφορίες για την κατάσταση της μνήμης και τις μεταβολές της. Τίθενται, λοιπόν, τα θεμέλια για να περιγραφούν με λογικές προτάσεις οι ιδιότητες της μνήμης μέσα από το σύστημα τύπων μίας γλώσσας προγραμματισμού.

Ο συνδυασμός των χαρακτηριστικών των linear συστημάτων τύπων και της ιδέας του προγραμματισμού με αποδείξεις μπορεί να οδηγήσει στην ανάπτυξη συστημάτων που παράγουν πιστοποιημένο κώδικα και υποστηρίζουν αναφορές και ανάθεση.

1.3 Σύνοψη της Εργασίας

Η συνέχεια της διπλωματικής εργασίας έχει την ακόλουθη δομή:

Κεφάλαιο 2. Περιγραφή της έρευνας στους τομείς των linear και substructural συστημάτων τύπων και του προγραμματισμού με αποδείξεις. Σύνοψη των κυριότερων συστημάτων.

Κεφάλαιο 3. Ορισμός και περιγραφή της γλώσσας let!. Η γλώσσα let! είναι μία απλή γλώσσα που υποστηρίζει πολυμορφισμό και διαθέτει linear και unrestricted τύπους. Μέσω της γλώσσας, προτείνεται ένας νέος αξιόπιστος τρόπος συνδυασμού των linear και των unrestricted τύπων.

Κεφάλαιο 4. Περιγραφή της γλώσσας τύπων του συστήματοςNFLint. Η γλώσσα τύπων του συστήματος είναι προϊόν εργασίας μελών του πανεπιστημίου του Yale και του Εργαστηρίου Τεχνολογίας Λογισμικού του Ε.Μ.Π. και αποτελεί τη βάση της γλώσσας τύπων της γλώσσας linNFLint που παρουσιάζεται στο επόμενο κεφάλαιο.

Κεφάλαιο 5. Ορισμός και περιγραφή του συστήματος linNFLint. Η γλώσσα linNFLint συνδυάζει τα αποτελέσματα της έρευνας στα πεδία των linear συστήματα τύπων και του προγραμματισμού με αποδείξεις και προτείνει ένα σύστημα προγραμματισμού με αποδείξεις που υποστηρίζει αναφορές και ανάθεση.

Κεφάλαιο 6. Συμπεράσματα και κατευθύνσεις μελλοντικής έρευνας.

Κεφάλαιο 2

Σχετική Ερευνητική Εργασία

2.1 Linear και Substructural Συστήματα Τύπων

Οι γλώσσες προγραμματισμού που διαθέτουν αναφορές και ανάθεση προσφέρουν σημαντικά οφέλη στον προγραμματιστή. Ο χρήστης έχει τη δυνατότητα να διαμορφώνει ρητά τη μνήμη, να κατασκευάζει δομές δεδομένων και να μεταφέρει δεδομένα μέσω της μνήμης ανάμεσα σε διαφορετικά σημεία του προγράμματός του. Επιπλέον, αν η αποδέσμευση της μνήμης γίνεται ρητά μέσω εκφράσεων της γλώσσας τότε αποφεύγονται πολύπλοκες διεργασίες ανάλυσης και αποδέσμευσης των ανενεργών τμημάτων της μνήμης κατά το χρόνο εκτέλεσης (*garbage collecting*) και αυξάνεται η αποδοτικότητα των προγραμμάτων. Η μνήμη που δεσμεύει το πρόγραμμα είναι το κοινό υπόστρωμα δεδομένων για όλο το πρόγραμμα και με τις αναφορές και την ανάθεση, ο χρήστης μπορεί να τη διαχειριστεί όπως επιθυμεί δεσμεύοντας, μεταβάλλοντας και αποδεσμεύοντας οποιοδήποτε τμήμα της.

Οι συνήθεις συναρτησιακές γλώσσες με ισχυρά συστήματα τύπων δεν υποστηρίζουν τα παραπάνω χαρακτηριστικά πλήρως ή και καθόλου. Το κύριο πλεονέκτημα των ισχυρά τυποποιημένων συναρτησιακών γλωσσών είναι ότι το σύστημα τύπων τους εγγυάται την μη εμφάνιση λαθών τύπων κατά το χρόνο εκτέλεσης που οδηγούν σε μη ομαλό τερματισμό του προγράμματος. Οι έλεγχοι εγκυρότητας του προγράμματος γίνονται στατικά μέσω των τύπων των δεδομένων (*typechecking*) και ο προγραμματιστής καλείται να διορθώσει πιθανά λάθη στο χρόνο μεταγλώττισης, πριν αποκτήσει τον εκτελέσιμο κώδικα που αντιστοιχεί στο πρόγραμμά του. Όταν εκτελέσει το διορθωμένο πρόγραμμα του, αυτό θα τερματίσει ομαλά για κάθε έγκυρη είσοδο. Αυτή η ιδιότητα των συστημάτων τύπων είναι γνωστή ως ασφάλεια (*safety*). Τα συστήματα τύπων που χρησιμοποιούν οι συνήθεις συναρτησιακές γλώσσες προκειμένου να εγγυηθούν την ασφάλεια αποκλείουν τους δείκτες και την ανάθεση ή όταν τους ενσωματώνουν δεν δίνουν τη δυνατότητα αποδέσμευσης της θέσης μνήμης που ελέγχει μία αναφορά. Η κυριότερη αιτία γι' αυτό το μειονέκτημα είναι η δυνατότητα των δεδομένων να χρησιμοποιούνται κατά βούλησιν, δηλαδή μπορούν να διπλασιάζονται ή και να μη χρησιμοποιούνται ποτέ. Με αυτό τον τρόπο θα μπορούσαν να προκύψουν δύο διαφορετικές αναφορές που να δείχνουν όμως στο ίδιο δεδομένο. Με χρήση της μίας από τις δύο αναφορές μπορεί να απαλειφθεί το κοινό δεδομένο (*free*) χωρίς αυτό να μπορεί κοινοποιηθεί στην άλλη αναφορά κατά το χρόνο μεταγλώττισης, δίνοντας την δυνατότητα για χρήση της δεύτερης που οδηγεί σε λάθος εκτέλεσης με αναφορά σε ξεκρέμαστο δείκτη (*dangling pointer*).

Η εισαγωγή της linear λογικής [Gira90] άνοιξε το δρόμο, μέσω του ισομορφισμού Curry-Howard [Howa69], για τη δημιουργία linear συστημάτων τύπων. Γρήγορα διαπιστώθηκε ότι τα linear συστήματα τύπων μπορούσαν να χρησιμοποιηθούν για τη σχεδίαση γλωσσών προγραμματισμού [Wadl90] με αναφορές και ανάθεσή Οι τιμές που αντιστοιχούν σε linear τύπους πρέπει να χρησιμοποιηθούν ακριβώς μία φορά, δηλαδή δεν μπορούν ούτε να διπλασιαστούν ούτε και να αγνοηθούν. Συνεπώς, είναι προφανές ότι μετά τη χρήση ενός δεδομένου αυτό μπορεί με ασφάλεια να αποδεσμευθεί ενώ συγχρόνως παραμένει δεσμευμένη κάθε στιγμή ακριβώς τόση μνήμη όση πρόκειται να χρησιμοποιηθεί.

Φυσικά, αν και τα interface συστήματα λύνουν το πρόβλημα της εισαγωγής αναφορών και ανάθεσης με ασφαλή τρόπο, η λύση που προτείνουν είναι πολύ περιοριστική. Ο διπλασιασμός

των αναφορών, που στα linear συστήματα τύπων δεν επιτρέπεται, είναι ιδιαίτερα χρήσιμος γιατί δίνει τη δυνατότητα πρόσβασης σε περιεχόμενα μνήμης από περισσότερα σημεία ενός προγράμματος παράλληλα. Τα δεδομένα μοιάζουν να εγκλωβίζονται μέσα στον κλάδο της ροής δεδομένων όπου γεννιούνται.

Κρίθηκε, λοιπόν, απαραίτητη η ανάμειξη δεδομένων με linear και μη linear τύπους ή ακόμα και με τύπους με ενδιάμεσα χαρακτηριστικά. Το αποτέλεσμα της συστηματικής μελέτης αυτών των συστημάτων αποχρυσταλλώθηκε στα substructural συστήματα τύπων [Walk05, Ahme05].

2.1.1 Ορισμός και Ιδιότητες των Linear και Substructural Συστημάτων Τύπων

Τα συστήματα τύπων εμφανίζουν τρεις βασικές δομικές ιδιότητες ανάλογα με τον τρόπο με τον οποίο χειρίζονται τις υποθέσεις:

- exchange: Αν μπορεί να αποδοθεί έγκυρος τύπος σε έναν όρο δεδομένου περιβάλλοντος Γ , τότε ο ίδιος τύπος αντιστοιχεί στον όρο για κάθε μετάθεση (permutation) του Γ .
- weakening: Αν μπορεί να αποδοθεί έγκυρος τύπος σε έναν όρο, δεδομένου περιβάλλοντος Γ , τότε ο ίδιος τύπος αντιστοιχεί στον όρο για κάθε περιβάλλον που προκύπτει από οποιαδήποτε επέκταση του Γ .
- contraction: Αν μπορεί να αποδοθεί έγκυρος τύπος σε έναν όρο, δεδομένου περιβάλλοντος Γ , που περιέχει δύο όμοιες υποθέσεις, τότε ο ίδιος τύπος αντιστοιχεί στον όρο για κάθε περιβάλλον που προκύπτει από την αντικατάσταση στο Γ και στον όρο των δύο όμοιων υποθέσεων με μία τρίτη όμοια με τις αρχικές υπόθεση

Με βάση τα παραπάνω, ένα linear σύστημα τύπων είναι ένα σύστημα στο οποίο ισχίο μόνο η ιδιότητα exchange.

Ανάλογα, ως substructural ορίζονται τα συστήματα όπου μία ή περισσότερες από τις δομικές ιδιότητες δεν ισχύουν για κάποια ή όλα τα δεδομένα. Τα substructural συστήματα τύπων πετυχαίνουν να αυξήσουν τον έλεγχο πάνω στη δομή και τις λειτουργίες των δεδομένων. Παρέχουν έναν ισχυρό στατικό μηχανισμό τόσο για την παρατήρηση των αλλαγών της κατάστασης των δεδομένων στην πορεία του προγράμματος, όσο και για τον περιορισμό των λειτουργιών που μπορούν να εκτελεστούν σε ένα δεδομένο κάθε χρονική στιγμή.

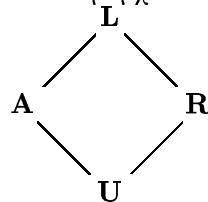
Ανάλογα με το είδος των υποθέσεων τα συστήματα διακρίνονται σε:

- unrestricted : Αντιστοιχεί σε υποθέσεις που μπορούν να χρησιμοποιηθούν 0, 1 ή και περισσότερες φορές. Διαφορετικά, πρόκειται για υποθέσεις για τις οποίες ισχύουν και οι τρεις δομικές ιδιότητες
- affine : Αντιστοιχεί σε υποθέσεις που μπορούν να χρησιμοποιηθούν 1 ή και περισσότερες φορές. Διαφορετικά, πρόκειται για υποθέσεις για τις οποίες ισχύουν οι δομικές ιδιότητες exchange και contraction.
- relevant : Αντιστοιχεί σε υποθέσεις που μπορούν να χρησιμοποιηθούν 0 ή 1 φορές. Διαφορετικά, πρόκειται για υποθέσεις για τις οποίες ισχύουν οι δομικές ιδιότητες exchange και weakening.
- linear : Αντιστοιχεί σε υποθέσεις που μπορούν να χρησιμοποιηθούν μόνο 1 φορά. Διαφορετικά, πρόκειται για υποθέσεις για τις οποίες ισχύει μόνο η δομική ιδιότητα exchange.
- ordered : Αντιστοιχεί σε υποθέσεις που μπορούν να χρησιμοποιηθούν μόνο 1 φορά και μάλιστα με τη σειρά που απαντώνται στο περιβάλλον όπου υπολογίζεται ο τύπος της έκφρασης. Διαφορετικά, πρόκειται για υποθέσεις για τις οποίες δεν ισχύει καμία δομική ιδιότητα.

Τις περισσότερες φορές, τα διάφορα συστήματα συνδυάζουν τα χαρακτηριστικά όλων των substructural συστημάτων. Ο κάθε τύπος συνοδεύεται και από έναν qualifier που είναι ενδεικτικός των ιδιοτήτων και της κατάστασης της μεταβλητής που φέρει τον τύπο. Συναντώνται χυρίως τέσσερις qualifiers U, A, R, L που αντιστοιχούν σε δεδομένα με unrestricted, affine, relevant και linear συμπεριφορά.

Οι qualifiers όπως και τα αντίστοιχα συστήματα σχετίζονται μέσω μιας ιεραρχικής δομής. Η έννοια της ιεραρχίας έγκειται στο ότι κάθε δεδομένο δομείται από δεδομένα που ο τύπος τους έχει qualifier που βρίσκεται στο ίδιο ή σε χαμηλότερα επίπεδα από τον δικό του. Για παράδειγμα, ένας δείκτης με linear τύπο μπορεί να δείχνει σε κάθε δεδομένα που ο τύπος τους έχει οποιοδήποτε qualifier, γιατί ο qualifier L βρίσκεται στην κορυφή της ιεραρχίας. Παρόμοια, ένας δείκτης με unrestricted τύπο μπορεί να δείχνει σε δεδομένα που ο τύπος τους έχει μόνο qualifier U, γιατί ο qualifier U βρίσκεται στο χαμηλότερο σημείο της ιεραρχίας. Αυτή ιδιότητα χάρισε στα παραπάνω συστήματα τον χαρακτηρισμό substructural.

Πίνακας 2.1: Η ιεραρχία των qualifiers.



2.1.2 Εφαρμογές στη Διαχείριση Μνήμης

Substructural Συστήματα Τύπων

Τα substructural συστήματα τύπων είναι ιδιαίτερα χρήσιμα για εφαρμογές που περιλαμβάνουν πρόσβαση σε αρχεία ή άλλα τμήματα μνήμης, δηλαδή σε εφαρμογές που απαιτούν γλώσσες με χαρακτηριστικά όπως references και destructive update. Σε αυτό το πλαίσιο οι qualifiers χρησιμοποιούνται τόσο για την ανάλυση της ροής των δεδομένων (*data flow analysis*) όσο και για τον έλεγχο των λειτουργιών που μπορούν να εκτελεστούν σε κάποιο δεδομένο. Ειδικότερα, όταν τα δεδομένα είναι προσβάσιμα μέσω κάποιου δείκτη τότε ο τρόπος με τον οποίο μπορεί να χρησιμοποιηθεί η θέση μνήμης στην οποία δείχνει ο δείκτης εξαρτάται άμεσα από τον qualifier του τύπου του δείκτη. Αναλυτικότερα, αν ορίσουμε τις λειτουργίες *read*, *write*, *swap* και *free* πάνω στο περιεχόμενο μίας θέσης μνήμης με προφανή σημασιολογία τότε για κάθε τύπο δεδομένων ισχύει [Morr05]:

- **linear:** επιτρέπονται οι λειτουργίες *swap* και *free*. Η λειτουργία *read* δεν επιτρέπεται γιατί ισοδυναμεί με διπλασιασμό των δεδομένων. Η λειτουργία *write* δεν επιτρέπεται επειδή αγνοεί τα linear δεδομένα που βρίσκονται ήδη στη θέση μνήμης και παραβιάζει τον περιορισμό της μη ισχύος της ιδιότητας *weakening*. Η εντολή *swap* εκτελεί και τις δύο προηγούμενες λειτουργίες διατηρώντας τους περιορισμούς του συστήματος. Η λειτουργεία *free* είναι ασφαλής καθώς δεν υπάρχουν αντίγραφα του δείκτη που πιθανώς να επιχειρήσουν να προσπελάσουν τα δεδομένα.
- **affine:** επιτρέπονται οι λειτουργίες *write*, *swap* και *free*. Η λειτουργία *write* επιτρέπεται επειδή ισχύει η ιδιότητα *weakening*. Η λειτουργία *read* δεν επιτρέπεται καθώς δεν ισχύει η ιδιότητα *contraction*. Η λειτουργία *free* είναι ασφαλής γιατί δεν υπάρχουν αντίγραφα του δείκτη που πιθανώς να επιχειρήσουν να προσπελάσουν τα δεδομένα.
- **relevant:** επιτρέπονται οι λειτουργίες *read*, *swap* και *free*. Η λειτουργία *write* δεν επιτρέπεται καθώς δεν ισχύει η ιδιότητα *weakening*. Η λειτουργία *read* επιτρέπεται γιατί

ισχύει η ιδιότητα contraction. Η λειτουργία *free* δεν επιτρέπεται επειδή δεν εξασφαλίζεται ότι δεν υπάρχουν άλλα αντίγραφα του δείκτη που πιθανώς να επιχειρήσουν να προσπελάσουν τα δεδομένα.

- unresticed: επιτρέπονται οι λειτουργίες *read*, *write* και *swap*. Η λειτουργία *free* δεν επιτρέπεται γιατί δεν εξασφαλίζεται ότι δεν υπάρχουν άλλα αντίγραφα του δείκτη που πιθανώς να επιχειρήσουν να προσπελάσουν τα δεδομένα.

Alias Types και Stateful Views

Παράλληλα με αυτές τις προσπάθειες, η διαισθητική αντίληψη πως η μνήμη είναι μοναδική και καθολική οντότητα για ένα πρόγραμμα οδήγησε σε μία διαφορετική προσέγγιση των linear τύπων. Αν είναι δυνατή η αναπαράσταση των περιεχομένων της μνήμης μέσω του συστήματος τύπων με τρόπο ώστε η εικόνα της μνήμης να είναι μοναδική κάθε χρονική στιγμή και ορατή από κάθε σημείο του προγράμματος, τότε είναι εφικτοί απλοί στατικοί έλεγχοι για το ποιες λειτουργίες είναι ασφαλείς. Αν για παράδειγμα η εικόνα της μνήμης εξασφαλίζει πως υπάρχει μόνο ένας δείκτης σε μία θέση μνήμης τότε είναι ασφαλές τόσο να αλλάξει το περιεχόμενό της όσο και να αποδεσμευθεί. Με βάση αυτούς τους συλλογισμούς κατασκευάστηκαν γλώσσες προγραμματισμού που ενσωματώνουν τα περιεχόμενα της μνήμης στο σύστημα τύπων τους. Η μνήμη αναπαρίσταται με ένα σύνολο προδιαγραφών που συνδέουν κάθε θέση μνήμης με τον τύπο των περιεχομένων της. Το σύστημα τύπων χειρίζεται το σύνολο των προδιαγραφών σαν μία linear οντότητα. Οι κανόνες του συστήματος τύπων αναλαμβάνουν να ελέγχουν αν πληρούνται οι απαιτούμενες προϋποθέσεις από τα περιεχόμενα της μνήμης και μεταβάλλουν τη μνήμη κατάλληλα, δηλαδή οι κανόνες τύπων διαδίδουν πληροφορία κατά τη ροή του προγράμματος.

Στο σύστημα των Applied Type Systems [Xi04, Zhu05] χρησιμοποιείται η ιδέα των stateful views. Τα stateful views είναι προδιαγραφές της μνήμης στη μορφή $l@t$ που δηλώνουν ότι στη θέση μνήμης l βρίσκονται αποθηκευμένα δεδομένα τύπου t .

Η χαρακτηριστική συνάρτηση *swap* φροντίζει να αλλάξει τα δεδομένα δύο θέσεων μνήμης καταναλλώνοντας τα stateful views που αντιστοιχούν στην κατάσταση της μνήμης πριν την κλήση της και παράγοντας στη θέση τους νέα που είναι συνεπή με τα νέα περιεχόμενα της μνήμης.

```
fun swap {t1 : type, t2 : type, l1 : addr, l2 : addr}
    (pf1 : t1@l1, pf2 : t2@l2 | p1 : ptr(l1), p2 : ptr(l2))
    : '(t1@l2, t2@l1 | unit) =
let val tmp := !p1 in p1 := !p2; p2 := tmp end
```

Τα stateful views μπορούν να χρησιμοποιηθούν για να περιγράψουν ολόκληρα μορφήματα της δομής των περιεχομένων της μνήμης που αντιστοιχούν σε δομές δεδομένων όπως απλά συνδεδεμένες λίστες.

```
datatype slseg (type, int, addr, addr) =
| (a : type, l : addr) SlsegNone (a, 0, 1, 1)
| (a : type, n : nat, first, next, last | first <> null)
  SlsegSome(a, n + 1, first, last) of
    ((a, ptr(next)@first, slseg (a, n, l, null)))
```

$$\text{viewdef } sllist (a, n, l) = slseg (a, n, l, null)$$

Το συγγενές σύστημα των alias τύπων [Smit00, Walk01a] χρησιμοποιεί περιορισμούς της μορφής $\{l \rightarrow t\}$ για να εκφράσει ότι η θέση μνήμης l περιέχει δεδομένα τύπου t . Διαθέτει αρκετά πλούσιο σύστημα τύπων με singleton τύπους και ορισμούς αναδρομικών τύπων όπως ο τύπος μίας λίστας ή ενός δέντρου.

List= $\mu\alpha.$ $< S(1) > \cup \exists[\rho | \{\rho \rightarrow \alpha\}]. < S(2), \text{int}, \text{ptr}(\rho) >$

Tree= $\mu\alpha.$ $< S(1) > \cup \exists[\rho_L, \rho_R | \{\rho_L \rightarrow \alpha, \rho_R \rightarrow \alpha\}]. < S(2), \text{ptr}(\rho_L), \text{ptr}(\rho_R) >$

Οι singleton τύποι είναι τύποι που έχουν μόνο ένα στοιχείο. Υπάρχει δηλαδή μία ένα προς ένα αντιστοιχία ανάμεσα στα αντικείμενα της γλώσσας με singleton τύπο και τον τύπο τους. Για παράδειγμα, στο σύστημα των alias τύπων, ο τύπος του φυσικού αριθμού 1 είναι ο τύπος $S(1)$ ενώ ο τύπος του φυσικού αριθμού 2 είναι ο τύπος $S(2)$.

Οι περιορισμοί των linear τύπων, και χωρίως το ότι δεν μπορούν τα δεδομένα τους να διπλασιαστούν, εξασφαλίζει τη συνέπεια της μνήμης. 'Όπως και στα συνήθη linear συστήματα, έτσι και εδώ έγινε προσπάθεια να συνδυαστούν linear και μη linear τύποι. Σε αυτό το πλαίσιο ένα unrestricted σύνολο προδιαγραφών της μνήμης αναπαριστά ένα τμήμα μνήμης που δεν μπορεί ούτε να αποδεσμευθεί ούτε και να μεταβληθούν οι τύποι των δεδομένων που περιέχει.

Ο Λογισμός των Regions και τα Linear Regions

Η αναζήτηση μεθόδων για την καλύτερη χρήση της μνήμης κατέληξε είτε σε χρήση garbage collectors είτε σε συστήματα όπου η δέσμευση και αποδέσμευση της μνήμης γίνεται από τον προγραμματιστή. Η διαχείριση μνήμης με garbage collectors είναι πολύπλοκη και δαπανηρή. Απαιτούνται σύνθετη έλεγχοι κατά το χρόνο εκτέλεσης που επιβαρύνουν σημαντικά την αποδοτικότητα των προγραμμάτων. Αντίθετα, η ρητή διαχείριση της μνήμης από το χρήστη δεν επηρεάζει το χρόνο εκτέλεσης του προγράμματος. Όλοι οι έλεγχοι μπορούν να γίνουν στατικά κατά το χρόνο μεταγλώττισης. Μία από τις κεντρικές ιδέες σε αυτή την κατεύθυνση είναι η διαχείριση μνήμης σε επίπεδο regions, δηλαδή συλλογές δεδομένων με παρόμοια διάρκεια ζωής. Στο λογισμό των regions όλα τα ενδιάμεσα αποτελέσματα των υπολογισμών αποθηκεύονται κατάλληλα σε regions. Η ομαδοποίηση των δεδομένων και η μαζική αποδέσμευσή τους με την καταστροφή του region συμβάλλει στην καλύτερη εκμετάλλευση του ελεύθερου χώρου της μνήμης. Η διαχείριση μνήμης σε επίπεδο regions αποδείχτηκε ιδιαίτερα αποδοτική και υιοθετήθηκε ακόμα και από γλώσσες με χαλαρά συστήματα τύπων όπως η C.

Η απόδειξη της συνέπειας και της ασφάλειας της χρήσης των regions, από τυποθεωρητική σκοπιά, σε συνδυασμό με την ανάπτυξη αλγορίθμων που επέτρεπαν την αυτόματη δημιουργία και καταστροφή τους, απαλλάσσοντας ουσιαστικά τον προγραμματιστή, [Toft94, Toft97, Toft98] κατέστησαν το λογισμό των regions ιδιαίτερα δημοφιλή.

Το μόνο σημαντικό μειονέκτημά είναι ότι η δέσμευση και αποδέσμευση των region γίνεται με την ίδια εντολή ορίζοντας ένα τμήμα του προγράμματος μόνο μέσα στο οποίο "ζει" το νέο region (*lexically scoped*):

```
letregion  $\rho$  in  $e$ 
```

όπου ρ το όνομα του νέου region.

Η σύνταξη αυτή επιλέχθηκε ώστε το σύστημα να είναι ασφαλές όμως δεν είναι αρκετά ευέλικτη ως προς τη διαχείριση μνήμης.

Σαν παράδειγμα, δίδεται μία αναδρομική συνάρτηση που μετράει αντίστροφα από κάποιον αριθμό μέχρι το μηδέν (συνάρτηση count) [Crary02]. Το σώμα της συνάρτησης αποθηκεύεται σε κάποιο region ρ_1 ενώ τα ενδιάμεσα αποτελέσματα του υπολογισμού μαζί με το τελικό αποτέλεσμα σε κάποιο άλλο region ρ_2 . Με το τέλος της κλήσης της συνάρτησης το region ρ_2 μπορεί να αποδεσμευθεί μαζί με τα περιεχόμενά του ενώ ο κώδικας μπορεί να παραμείνει στη μνήμη στο ενεργό region ρ_1 .

```
letregion  $\rho_1$  in  
  letregion  $\rho_2$  in  
    letrec count [ $\rho$ ] ( $x : < \text{int} >$  at  $\rho$ ) at  $\rho_1$  =  
      let  $n = \pi_1(x)$  in
```

```

if (n=0)
  then ()
else count [ $\rho$ ] (< n - 1 > at  $\rho$ )
in
count [ $\rho_2$ ] ( $\rho_2$ , < 10 > at  $\rho_2$ )

```

Η αποσύνδεση των εντολών δημιουργίας και καταστροφής των regions μπορεί να αποφέρει ακόμα πιο αποδοτική χρήση της μνήμης γιατί η διάρκεια ζωής των δεδομένων που περιέχονται σε κάποιο region δεν ταυτίζεται απαραίτητα με τη διάρκεια ζωής που ορίζει συντακτικά η εντολή *letregion*.

Όπως έχει ήδη σημειωθεί με χρήση των linear συστημάτων μπορεί να γίνει εξαιρετικά αποδοτική και ασφαλής χρήση της μνήμης. Προσφέρουν δηλαδή ένα πλαίσιο στο οποίο ο λογισμός των regions μπορεί να εφαρμοστεί με ασφάλεια [Walk01b, Flue06].

Η δομή ελέγχου *letregion* ρ x_ρ *in* e μπορεί να κωδικοποιηθεί με ασφάλεια με linear regions μέσω των εντολών *alloc()* και *free()* που δημιουργούν και καταστρέφουν ένα region αντίστοιχα [Walk01b].

```

letregion  $\rho$   $x$  in  $e$  ≡
  unpack  $\rho$ ,  $x = \text{alloc}()$  in
  let  $z = \epsilon$  in
  free  $x$ ;  $y$ 

```

όπου *unpack* η πράξη αποσύνθεσης υπαρξιακών τύπων.

Ένα linear region δεν μπορεί να διπλασιαστεί και πρέπει να αποδεσμευθεί πριν το τέλος του προγράμματος. Ο πρώτος περιορισμός εξασφαλίζει την ακεραιότητα της μνήμης καθώς τα regions ζουν και πεθαίνουν μόνο σε έναν κλάδο τη ροής δεδομένων του προγράμματος. Ο κλάδος αυτός είναι και ο μόνος ιδιοκτήτης του region με απόλυτη εξουσία πάνω στα δεδομένα που περιέχει. Αντίγραφα του region δεν υπάρχουν σε άλλους κλάδους του προγράμματος, καθιστώντας ασφαλή την αποδέσμευσή του. Ο δεύτερος περιορισμός εξασφαλίζει ότι όλα τα δεσμευμένα regions θα έχουν καταστραφεί με το τέλος του προγράμματος και το πρόγραμμα θα αποδεσμεύσει όλη τη μνήμη που χρησιμοποίησε κατά την εκτέλεσή του.

Ο Λογισμός των Capabilities

Ανάμεσα στις δύο παραπάνω προσεγγίσεις, βρίσκονται συστήματα που δανείζονται ιδέες από τον λογισμό των capabilities [Wulf81]. Τα capabilities είναι οντότητες του συστήματος τύπων που καθορίζουν τους περιορισμούς χρήσης ενός αντικειμένου της γλώσσας προγραμματισμού. Συνήθως χρησιμοποιούνται για τον έλεγχο πρόσβασης σε τμήματα μνήμης. Για να μπορέσει να χρησιμοποιηθεί π.χ. ένας δείκτης που δείχνει σε μία θέση μνήμης πρέπει, εκτός από τον ίδιο τον δείκτη, να δοθεί και το capability του.

Υπάρχουν δύο τρόποι να ενσωματωθούν τα capabilities σε ένα σύστημα τύπων με ασφάλεια είτε ως ένα καθολικό περιβάλλον είτε ως αντικείμενα με linear χαρακτηριστικά.

Στην περίπτωση που τα capabilities συνιστούν ένα καθολικό περιβάλλον [Crar99], διαδίδονται από τους κανόνες του συστήματος κατά μήκος της ροής των δεδομένων. Το περιβάλλον των capabilities είναι ένα linear αντικείμενο και ο τρόπος χειρισμού του μοιάζει με τον τρόπο που χρησιμοποιούνται οι προδιαγραφές της μνήμης στα συστήματα με stateful views ή alias τύπους. Επιπλέον, η ιδέα μπορεί να συνδυαστεί με regions δίνοντας μια τυποθεωρητική μέθοδο διαχείρισης μνήμης. Λόγω του linear χαρακτήρα του συστήματος η διαχείριση των regions μπορεί να γίνει χωρίς συντακτικούς περιορισμός (*non-lexically scoped*).

Η συνάρτηση *count*, που πριν δόθηκε σε περιγραφή που ακολουθούσε το λογισμό των regions, εδώ αποκτά διαφορετική μορφή [Crar99]. Κατά τη δημιουργία ενός νέου region, κατασκευάζεται το capability ρ που αντιστοιχεί στο region και ο handler x_ρ μέσω του οποίου

γίνεται η πρόσβαση στο region. Για να καταστραφεί ένα region πρέπει να δοθεί ο handler του ενώ συγχρόνως το capability του απομακρύνεται από το περιβάλλον των ενεργών capabilities.

```

let newrgn  $\rho_1$   $x_{\rho_1}$  in
let newrgn  $\rho_2$   $x_{\rho_2}$  in
let count =
  (fix count
    [ρ : Rgn, ρcont : Rgn, e ≤ {ρ1U, ρcontU}]
    e ⊕ ρL.xρ : ρ handle, x : < int > at ρ,
    k : (e) → 0 at ρ, let n = π1(x) in
    let freergn xρ, in
    if (n=0)
      then k()
      else
        let n' = n - 1 in
        let newrgn ρ' xρ' in
        let x' = < n' > at xρ' in
        count[ρ', ρcont, e](x'ρ', x', k)
    ), at xρ_1 in
  let ten = < 10 > at xρ_2 in
  let newrgn  $\rho_3$   $x_{\rho_3}$  in
  let cont =
    (λ({ρ1L, ρ3L}.
      let freergn xρ_3, in
      let freergn xρ_1, in
      halt0
    ) at xρ_3
  in
  count[ρ2, ρ3, {ρ1L, ρ3L}](xρ_2, ten, cont)

```

Στην περίπτωση που τα capabilities είναι linear οντότητες μπορούν να ζουν μόνο στους τύπους μίας γλώσσας [Flue06] ή να αντικατοπτρίζονται και στις εκφράσεις της [Morr05]. Η θεώρηση των capabilities ως linear αντικειμένων συνδυάζεται εύκολα και με τις γενικές ιδέες των substructural συστημάτων και των linear regions. Το πλεονέκτημα του συνδυασμού των τριών μοντέλων είναι πως πλέον ο έλεγχος των διαφόρων λειτουργιών μίας γλώσσας μπορεί να γίνει αποκλειστικά μέσω των capabilities και να αποσυνδεθεί από τους qualifiers των τύπων των εκφράσεων της γλώσσας. Μ' αυτό τον τρόπο η χρήση μίας αναφοράς εξαρτάται αποκλειστικά από την ύπαρξη του capability της. Το capability είναι ένα linear στοιχείο που εγγυάται από μόνο του την ασφαλή πρόσβαση στα δεδομένα που ελέγχει. Μπορούν, λοιπόν, να υπάρχουν ταυτόχρονα πολλές αναφορές για το ίδιο δεδομένο (π.χ. ένα region), δίνοντας μεγαλύτερη ευελιξία στον προγραμματιστή.

Οι τύποι των τελεστών που εφαρμόζονται πάνω σε regions έχουν ιδιαίτερο ενδιαφέρον [Flue06]:

newrgn	: $^U(L\text{unit} \multimap ^L\exists\rho. ^L(L\text{cap } \rho \otimes ^U\text{hnd } \rho))$
freergn	: $^U\forall\rho. ^U(L(L\text{cap } \rho \otimes ^U\text{hnd } \rho) \multimap ^L\text{unit})$
new	: $^U\forall\rho. ^U\forall\alpha. ^U(L(L\text{cap } \rho \otimes ^U\text{hnd } \rho \otimes ^{Uu}\alpha) \multimap ^L(L\text{cap } \rho \otimes (\text{ref } \rho ^U\alpha)))$
read	: $^U\forall\rho. ^U\forall\alpha. ^U(L(L\text{cap } \rho \otimes ^U(\text{ref } \rho ^U\alpha)) \multimap ^L(L\text{cap } \rho \otimes ^U\alpha))$
write	: $^U\forall\rho. ^U\forall\alpha. ^U(L(L\text{cap } \rho \otimes ^U(\text{ref } \rho ^U\alpha) \otimes ^U\alpha) \multimap ^L(L\text{cap } \rho \otimes ^U\text{unit}))$

όπου \multimap ο κατασκευαστής της linear συνάρτησης και \otimes ο κατασκευαστής του ζεύγους

'Οπως δηλώνει ο τύπος του τελεστή newrgn η δημιουργία ενός linear region ισοδυναμεί με την κατασκευή ενός δεδομένου υπαρξιακού τύπου που περιέχει ένα linear capability και

έναν unrestricticed handler για το νέο region. Ο υπαρξιακός τύπος είναι linear και μπορεί να χρησιμοποιηθεί μόνο μία φορά εξασφαλίζοντας ότι μία κλήση της newrgn μπορεί να δώσει μόνο ένα linear capability για το νέο region. To linear capability ελέγχει την πρόσβαση στο region που γίνεται μέσω του unrestricted handler. Αν και ο handler μπορεί να διπλασιαστεί το capability παραμένει πάντα μοναδικό και εγγυάται την ασφάλεια πρόσβαση στο region που ελέγχει. Σε κάθε προσπάθεια πρόσβασης στο region πρέπει απαραίτητα εκτός από τον handler να δοθεί και το μοναδικό capability. Οι τελεστές new, read και write καταναλώνουν το capability και το αναπαράγουν εξασφαλίζοντας ότι σε κάθε ενεργό region αντιστοιχεί ένα μοναδικό capability. Ο τελεστής free καταναλώνει το capability και δεν το αναπαράγει με αποτέλεσμα η πρόσβαση στο region να είναι πλέον ανέφικτη παρά την παρουσία του handler του region.

2.1.3 Σύνδεση Linear και Unrestricted Τύπων

Ένα από τα ζητήματα που απασχόλησε από νωρίς τους μελετητές των linear και substructural συστημάτων είναι το πώς ένα δεδομένο με τύπο που επιτρέπει κάποιες λειτουργίες μπορεί να αποκτήσει τύπο που επιτρέπει κάποιες άλλες λειτουργίες. Ή, με την ορολογία των substructural συστημάτων, πώς ένα δεδομένο που έχει τύπο με κάποιο qualifier μπορεί να αποκτήσει κάποιον άλλο qualifier. Η δυνατότητα αυτή είναι πολύ σημαντική γιατί ο qualifier δηλώνει την κατάσταση του δεδομένου και είναι σοβαρός περιορισμός να παραμένει η κατάσταση ενός δεδομένου σταθερή σε όλο το πρόγραμμα. Για παράδειγμα, αν μία αναφορά πρόκειται να χρησιμοποιηθεί τοπικά μόνο για ανάγνωση είναι χρήσιμο ο τύπος της να είναι unrestricted ώστε να μπορούν να δημιουργηθούν πολλά αντίγραφά (alias) και να μπορεί η ανάγνωση να γίνει από πολλά σημεία της ροής του προγράμματος παράλληλα. Στη συνέχεια, για να μπορεί να χρησιμοποιηθεί η αναφορά πάλι για εγγραφή πρέπει να μετατραπεί σε linear.

Ο 'Όρος let!

Η εισαγωγή μίας έκφρασης η οποία θα αναλαμβάνει να εκτελεί την μετατροπή δεν είναι τετριμένη γιατί μπορεί να δημιουργήσει κενά στην ασφάλεια του συστήματος. Για παράδειγμα αν ένας linear δείκτης μετατραπεί σε unrestricted τότε πριν επανέλθει στην αρχική του κατάσταση πρέπει να εξασφαλιστεί ότι δεν υπάρχουν άλλα ενεργά unrestricted αντίγραφά του. Διαφορετικά, εφόσον πλέον υπάρχουν ένας linear δείκτης και τουλάχιστον ένα unrestricted αντίγραφο, είναι δυνατή η αποδέσμευση της θέσης μνήμης στην οποία δείχνει ο δείκτης χωρίς να μπορεί να αποκλειστεί το ενδεχόμενο να γίνει προσπάθεια προσπέλασής της στο μέλλον μέσω του unrestricted αντιγράφου. Από τα παραπάνω, γίνεται, ακόμα, εμφανές ότι η διάρκεια ζώης του δεδομένου με το νέο qualifier πρέπει να καθορίζεται ρητά. Αυτό οδηγεί στην υιοθέτηση εκφράσεων που καθορίζουν συντακτικά τη διάρκεια ζώης του τροποποιημένου δεδομένου. Σε αντίθετη περίπτωση, δεν μπορεί να εξακριβωθεί αν ισχύουν οι περιορισμοί που απαιτούνται ώστε να επανέλθει ο τύπος του δεδομένου στην αρχική του κατάσταση.

Η πρώιμη μορφή της έκφρασης εμφανίστηκε σε γλώσσα με lazy αποτίμηση χωρίς πολυμορφισμό και με πολλές συντακτικές ομοιότητες με τη Haskell [Wadl90].

Ο όρος let! είχε πολλά κοινά με τον συμβατικό let όρο:

`let! (x) y = e1 in e2`

'Όμοια με έναν let όρο, πρώτα γίνεται η αποτίμηση της έκφρασης e₁ και το αποτέλεσμά της δεσμεύεται στη μεταβλητή y. Στη συνέχεια αποτιμάται η έκφραση e₂ σε διευρυμένο περιβάλλον που περιλαμβάνεις και τη μεταβλητή y. Αυτό που διαφοροποιεί το νέο όρο είναι ο τρόπος με τον οποίο χρησιμοποιείται η μεταβλητή x: Κατά την αποτίμηση της e₁ η μεταβλητή x έχει μη linear τύπο ενώ κατά την αποτίμηση της e₂ επανακτά τον αρχικό linear τύπο της. Στο όνομα της ασφάλειας, κρίθηκαν απαραίτητοι σημαντικοί περιορισμοί. Κατ' αρχήν η διαδικασία μετατροπής ενός linear τύπου σε μη linear διαδίδεται σε όλα τα συστατικά του τύπου και άρα

δεν είναι ισοδύναμη με απλή αλλαγή του *qualifier* του αρχικού τύπου. Επιπλέον, η αποτίμηση της e_1 πρέπει να ολοκληρωθεί πλήρως πριν την έναρξη του υπολογισμού της e_2 (*hyperstrict evaluation*). Αυτό εξασφαλίζει ότι δεν θα υπάρχουν ενεργές αναφορές στο x ή σε κάποιο από τα συστατικά του μέσα στο e_1 κατά την έναρξη της αποτίμησης του e_2 . Τέλος, από τον τύπο του e_1 πρέπει να εξασφαλίζεται ότι δεν μπορεί να αποδράσει το x ή κάποιο συστατικό του από την εμβέλεια του. Αυτό επιτυγχάνεται αν ο τύπος του e_1 δεν περιέχει οποιοδήποτε μη *linear* συστατικό που μπορεί να προκύψει από τη μετατροπή του x σε *linear* και αν επίσης δεν είναι συνάρτηση καθώς σε αυτή την περίπτωση το x μπορεί να "χρυφτεί" σαν ελεύθερη μεταβλητή. Η παραπάνω λύση, στην προσπάθεια της να εξασφαλίσει την ασφάλεια, αποδυναμώνει αρκετά την εκφραστικότητα της γλώσσας και δεν επεκτείνεται εύκολα σε γλώσσες που υποστηρίζουν πολυμορφισμό.

Με τη βοήθεια της έκφρασης *let!* μπορεί να γραφεί το πρόγραμμα *cat* που ενώνει τα περιεχόμενα των αρχείων ενός πίνακα αρχείων α που οι δείκτες τους περιέχονται στη μεταβλητή *files* και τα αποθηκεύει στο εικονικό αρχείο "*stdout*". Η ανάγνωση των αρχείων γίνεται παράλληλα ενώ η εγγραφή αρχίζει αφού έχουν ολοκληρωθεί όλες οι διαδικασίες ανάγνωσης. Στο παράδειγμα χρησιμοποιείται σύνταξη Haskell.

```
cat files a=
let! (a) y=concat[lookup i a|i ← files] in
    update "stdout"y
```

Observer Τύποι

Ο πολυμορφισμός είναι ένα πανίσχυρο εργαλείο προγραμματισμού και γρήγορα αναζητήθηκε λύση που να τον υποστηρίζει. Η παρουσία του πολυμορφισμού περιπλέκει την κατάσταση καθώς δίνει τη δυνατότητα να γραφούν συναρτήσεις παραμετρικές ως προς *linear* και *unrestricted* δεδομένα. Το σύστημα πρέπει, λοιπόν, να διαθέτει έναν ενιαίο τρόπο χρήσης όλων των δεδομένων. Σε αυτό το πλαίσιο η ασφάλεια του συστήματος γίνεται και πάλι διάτρητη.

Η χρήση των *observer* τύπων έδειξε μια άλλη προσέγγιση του προβλήματος [Oder92]. Οι *observers* είναι τύποι που στέκουν ανάμεσα στους *linear* και τους *unrestricted*. Ουσιαστικά είναι τύποι οι οποίοι αίρουν την απαγόρευση του μη διπλασιασμού των *linear* τύπων και δεν διαφοροποιούνται σημαντικά ως προς τις ιδιότητες τους από τους *unrestricted* τύπους. Η διαφορά τους έγκειται στο ότι η διάρκεια ζώης τους περιστέλλεται στην εμβέλεια ενός όρου παραπλήσιου με το αρχικό *let!*.

```
let! x=e1 in e2
```

Και πάλι όπως ένας *let* όρος, πρώτα αποτιμάται το e_1 , δεσμεύεται στη μεταβλητή x και στη συνέχεια αποτιμάται η έκφραση e_2 σε διευρυμένο περιβάλλον που περιλαμβάνει και τη μεταβλητή x . Όμως εδώ, το περιβάλλον στο οποίο αποτιμάται το e_1 δεν διαφέρει από το περιβάλλον στο οποίο αποτιμάται το e_2 μόνο ως προς το x . Κάθε *linear* τύπος μεταβλητής που υπάρχει στο περιβάλλον όπου ο όρος *let!* αποτιμάται, μετατρέπεται σε *observer* και στο νέο περιβάλλον αποτιμάται η e_1 . Η e_2 αποτιμάται στο αρχικό περιβάλλον με την προσθήκη της μεταβλητής x . Όπως και στον κλασσικό *let!* είναι υποχρεωτικοί σημαντικοί περιορισμοί. Διατηρείται η μετατροπή των τύπων από *linear* σε *observer* διαδίδεται σε όλο το βάθος του τύπου. Επιπλέον, ο τύπος του e_1 ή κάποιο από τα συστατικά του δεν πρέπει να είναι *observer*. Η λύση ξεπερνάει το πρόβλημα του πολυμορφισμού, ωστόσο είναι πολύ δύσχρηστη γιατί δεν επιτρέπει την επιλεκτική μετατροπή των τύπων κάποιων δεδομένων αλλά επεμβαίνει στο σύνολο των υποθέσεων εμποδίζοντας και την συνύπαρξη *linear* και *observer* τύπων. Το σύστημα τύπων όπου πρωτοεμφανίστηκαν οι *observer* τύποι [Oder92] δεν υποστηρίζει αποδέσμευση τμημάτων της μνήμης και συνεπώς οι συναρτήσεις δεν αποτελούν επικίνδυνο σημείο που απαιτεί ιδιαίτερη προσοχή.

Το σύστημα διαθέτει λίστες και μπορεί αποδοτικά να κωδικοποιήσει λειτουργίες πάνω στις λίστες όπως η αντικατάσταση του πρώτου στοιχείου($rplhd$), η αντικατάσταση n -ιοστού στοιχείου(upd) και η ανταλλαγή των περιεχομένων δύο στοιχείων μίας λίστας($swap$).

$$\begin{aligned}
 rplhd &: \forall t. t \rightarrow {}^L\text{list } t \rightarrow \text{list } t \\
 rplhd \; hd' \; hd :: tl = & \\
 & hd' :: tl \\
 \\
 upd &: \forall t. \text{int} \rightarrow t \rightarrow {}^L\text{list } t \rightarrow \text{list } t \\
 upd \; i \; o \; hd :: tl = & \\
 & \text{if } i = 0 \text{ then} \\
 & \quad rplhd \; o \; xs \\
 & \text{else } hd :: upd \; (i - 1) \; o \; tl \\
 \\
 swap &: \forall t. \text{int} \rightarrow \text{int} \rightarrow {}^L\text{list } t \rightarrow \text{list } t \\
 swap \; i \; j \; xs = & \\
 & \text{let! } x = xs!i \text{ in} \\
 & \quad \text{let! } y = xs!j \text{ in} \\
 & \quad upd \; i \; y \; (upd \; j \; x \; xs)
 \end{aligned}$$

όπου η έκφραση $x!i$ δηλώνει την ανάγνωση του i -οστού στοιχείου της λίστας x

Ο 'Όρος let! σε Συστήματα με Πολυμορφισμό

Η προσπάθεια για χρήση του κλασικού let! όρου σε συστήματα που υποστηρίζουν πολυμορφισμό και υπαρξιακής τύπους αποδείχτηκε εφικτή σε γλώσσες με διαχείριση της μνήμης με βάση τα regions [Walk01b]. Σε αυτή την περίπτωση δεν μελετάται η μετατροπή κάθε τύπου της γλώσσας αλλά μόνο του τύπου που αντιστοιχεί στο region ρ και του δείκτη r που εξασφαλίζει πρόσβαση σε ένα region:

$$\text{let! } (\rho) r = e_1 \text{ in } e_2$$

Ο τύπος είναι μοναδικός για κάθε region (*singleton*). Η ερμηνεία του όρου let! παραμένει ίδια με τον κλασικό ορισμό του όρου και συγχρόνως γίνεται προσπάθεια να επαναδιαπραγματευθούν οι συντακτικοί περιορισμοί διασφαλίζοντας την ασφάλεια του συστήματος. Για να αποφευχθεί η διαφυγή unrestricted regions έξω από την εμβέλεια που ορίζει το let! ο τύπος του e_1 δεν επιτρέπεται να περιέχει τον τύπο του region πάνω στο οποίο εκτελείται η μετατροπή. Λόγω της ύπαρξης των regions και της δέσμευσης και αποδέσμευσης μνήμης οι συναρτήσεις γίνονται εδώ μη ασφαλή σημεία. Για να ξεπεραστεί αυτό το πρόβλημα οι συναρτήσεις υποχρεώνονται να φέρουν στον τύπο τους τύπους των unrestricted regions που το σώμα τους χρησιμοποιεί. Ο περιορισμός υλοποιείται απαγορεύοντας το currying για συναρτήσεις με unrestricted ορίσματα.

Σαν παράδειγμα, δίδεται μία συνάρτηση που διπλασιάζει ένα δεδομένο x , αποθηκεύει το προκύπτων ζεύγος y σε ένα region ρ' και στη συνέχεια αποθηκεύει σε ένα άλλο region ρ ένα ζεύγος με τον δείκτη προς το region r' και το ζεύγος y .

$$\begin{aligned}
 \lambda[\rho](x : \text{int}, \text{gen}()) : \exists \rho'. {}^L\text{rgn}(\rho'), r : {}^U\text{rgn}(\rho)) \rightarrow & \\
 & \text{unpack } \rho, r' = \text{gen}() \text{ in} \\
 & \text{let! } (r') y = x \; {}^L \times \; x \; \text{at} \; r' \text{ in} \\
 & \quad \text{pack}[\rho', r' \; {}^L \times \; y \; \text{at} \; r] \text{ as } \tau_{res}
 \end{aligned}$$

όπου pack η πράξη σύνθεσης υπαρξιακών τύπων και unpack η πράξη αποσύνθεσης υπαρξιακών τύπων.

Ο Όρος *let!* σε Συστήματα με Capabilities

Σε γλώσσες με συστήματα τύπων που συνδυάζουν στοιχεία από τον λογισμό των capabilities και τα substactral συστήματα [Morr05], η μετάβαση ανάμεσα σε linear και μη linear τύπους αποκτά διαφορετικό νόημα. Πλέον ο έλεγχος των λειτουργιών που μπορούν να εφαρμοστούν πάνω σε ένα δεδομένο ελέγχεται από τα capabilities. Οπότε, για όλους τους τύπους πλην των capabilities η μετατροπή από linear σε μη linear δεν θέτει σε κίνδυνο την ασφάλεια του συστήματος αρκεί να διατηρούνται οι περιορισμοί των substactral συστημάτων, δηλαδή ένα δεδομένο με μη linear τύπο δεν μπορεί να περιέχει συστατικά με linear τύπο. Η αντίθετη μετάβαση, από μη linear σε linear τύπο, είναι ασφαλής σε κάθε περίπτωση.

Ένας linear δείκτης μαζί με την *swap*, την μοναδική εντολή που μπορεί με ασφάλεια να εφαρμοστεί πάνω στα περιεχόμενά του, ορίζεται εύκολα:

$$\text{LRef } \tau \equiv \exists c. (\text{Cap } \rho \tau \otimes !\text{Ptr } \rho)$$

```
lrswap  $\tau \equiv$ 
         $\lambda x : \text{LRef } \tau. \lambda x : \tau'.$ 
        let ' $\rho$ ,  $cp' = r$  in
        let  $< cp_0, p_0 > = cp$  in
        let  $< p_1, p_2 > = \text{dup } p_0$  in
        let  $!p'_2 = p_2$  in
        let  $< c_1, y > = \text{swap } p'_2 < c_0, x >$  in
         $< [\rho, < c_1, p_1 >], y >$ 
```

Όσον αφορά τα capabilities, ένας τελείως διαφορετικός μηχανισμός είναι απαραίτητος. Τα linear capabilities μετατρέπονται σε unrestricted, που πλέον λέγονται frozen, αν η θέση μνήμης την οποία ελέγχουν είναι unrestricted. Μ' αυτό τον τρόπο διατηρείται ο περιορισμός που επιβάλει unrestricted δεδομένα να μην περιέχουν linear δεδομένα. Φυσικά, ο τύπος της θέσης μνήμης στην οποία αναφέρεται το capability παραμένει σταθερός, παγωμένος. Τα frozen capabilities μπορούν ελεύθερα να διπλασιαστούν δημιουργώντας alias. Ένα frozen capability μπορεί να γίνει ξανά linear, που τώρα λέγεται thawed, αρκεί να αποκλειστεί η περίπτωση να υπάρχει άλλη linear, thawed capability για το ίδιο δεδομένο. Το thawed capability μπορεί να επανέλθει σε κατάσταση freeze. Η επαναφορά του όμως πρέπει να γίνει με τρόπο ώστε όλα τα αντίγραφα που παρέμειναν σε κατάσταση freeze να βρίσκονται ξανά σε συνεπή κατάσταση με τη μνήμη. Επομένως το thawed capability πρέπει να γίνει ξανά freeze υποχρεώνοντας τα δεδομένα της θέσης μνήμης στην οποία δείχνει να αποκτήσουν ξανά τον τύπο που είχαν πριν το capability γίνει thawed. Για την υλοποίηση των περιορισμών των παραπάνω λειτουργιών χρησιμοποιείται ένα σύνολο με linear συμπεριφορά που συγκεντρώνει τα capabilities που είναι σε κατάσταση thawed καθώς και τον τύπο τους πριν αποκτήσουν αυτή την κατάσταση. Το σύνολο αυτό διαδίδεται από τους κανόνες του συστήματος τύπων κατά τη ροή των δεδομένων.

Με την προσθήκη των unrestricted capabilities πλέον μπορούν να οριστούν και unrestricted δείκτες καθώς και οι πράξεις που τους συνοδεύουν (*read*, *write*):

$$\text{Ref } !\tau \equiv !\exists \rho. (!\text{Frzn } \rho !\tau \otimes !\text{Ptr } \rho)$$

```
read  $\equiv$ 
         $\lambda x : \text{LRef } \tau. \lambda t_0 : \text{Thwd } \bullet.$ 
        let ' $\rho, < f'_a, l = r$  in
        let  $< c_1, t_1 > = \text{thaw } < f_a, t_0 >$  voidrho in
        let  $< c_2, x > = \text{swap } l < c_1, \text{unit} >$  in
        let  $< c_3, \text{unit} > = \text{swap } l < c_1, x >$  in
        let  $< f_b, t_2 > = \text{refreeze } < c_3, t_1 >$  in
         $< f_b, t_2 >$ 
```

```

write ≡

$$\lambda x : \text{LRef } \tau. \lambda z : !\tau. \lambda t_0 : \text{Thwd} \bullet.$$


$$\text{let } [\rho, < f'_a, l = r \text{ in}$$


$$\text{let } < c_1, t_1 > = \text{thaw } < f_a, t_0 > \text{ void}_\text{ho} \text{ in}$$


$$\text{let } < c_2, x > = \text{swap } l < c_1, z > \text{ in}$$


$$\text{let } < f_b, t_2 > = \text{refreeze } < c_2, t_1 > \text{ in}$$


$$t_2$$


```

2.2 Προγραμματισμός με Αποδείξεις

Στον σύγχρονο κόσμο, όπου τόσο η παραγωγή όσο και η εκτέλεση του κώδικα γίνεται με κατανεμημένο και ανεξάρτητο τρόπο, η διασφάλιση της συμβατότητας των τμημάτων που απαρτίζουν μεγαλύτερα συστήματα και η αξιοπιστία κάθε ξεχωριστού τμήματος είναι κρίσιμη. Σε πολυάριθμες περιπτώσεις, προβλήματα ασφάλειας και συμβατότητας προγραμμάτων που έγιναν γνωστά, είχαν ως αποτέλεσμα να τεθεί σε κίνδυνο η λειτουργία μεγάλων συστημάτων και συνεπώς να επιφέρουν οικονομικά πλήγματα στους οργανισμούς που τα χρησιμοποιούσαν. Η αλματώδης ανάπτυξη του διαδικτύου και η μεταφορά εκτελέσιμων αρχείων χωρίς καμία δυνατότητα ελέγχου των προδιαγραφών τους κάνουν το πρόβλημα ακόμα πιο οξύ. Η ανάγκη για την εξασφάλιση της αξιοπιστίας του εκτελέσιμου κώδικα είναι, πλέον, επιτακτική.

2.2.1 Η Βιομηχανική Προσέγγιση

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 αντιστοιχθεί 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.

2.2.2 Η Ερευνητική Προσέγγιση

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 ελεγχτής 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 συνεχείᾳ 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 απόρ-

ρητο 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, εκφραζόμενες 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)[Howard9].

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. Εξυπακούεται ότι η ανάπτυξη ενός συστήματος πιστοποίησης εκτελέσιμου κώδικα προϋποθέτει ένα συστηματικό τρόπο να μπορεί κανείς να αποδεικνύει προτάσεις στη γλώσσα τύπων.

Οι Γλώσσες TIL και TAL

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 χρησιμο-

ποιηθούν για πιστοποίηση κώδικα. Παρ όλο που τα πλεονεκτήματα από τη χρήση ενδιάμεσης 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)

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 πληρού 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 σύγχριση με το TAL, το PCC χρησιμοποιεί μία γενικής-εφαρμογής κατηγορηματική λογική πρώτης τάξης και μπορεί να εκφράσει πιο πολύπλοκες ιδιότητες ασφαλείας. Από την άλλη μεριά το PCC χρησιμοποιεί ρητές αποδείξεις ασφαλείας 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 όλ' 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 κάπι 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 μεταθεωρίας

Το Σύστημα Foundational Proof-Carrying Code (FPCC)

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 τυμάτων 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.

To Σύστημα Flint

Η προσέγγιση των Shao *et al.* [Shao02] προτείνει ένα τυποθεωρητικό πλαίσιο για την ανάπτυξη πιστοποιημένου κώδικα. Παρόμοια προσέγγιση προτάθηκε ανεξάρτητα από τους Crary και Vanderwaart [Crar02]. Οι τυποθεωρητικές 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 άλιτα ως σήμερα, η προσέγγιση αυτή θεωρείται ελπιδοφόρα.

Το σύστημα Flint ορίζει μία ενδιάμεση γλώσσα οργανωμένη σε δύο στρώματα, την γλώσσα τύπων και τη γλώσσα υπολογισμών. Το σύστημα είναι ικανό να εκφράσει και να χειρίστει προτάσεις της κατηγορηματικής λογικής ανώτερης τάξης και τις αποδείξεις τους. Αυτό επιτυγχάνεται με την χρησιμοποίηση μίας παραλλαγής του λογισμού των επαγωγικών κατασκευών (CIC) σαν γλώσσα τύπων. Το CIC είναι ένα πανίσχυρο εργαλείο για την απόδειξη προτάσεων και θεωρημάτων.

Με βάση την κεντρική ιδέα της κατασκευαστικής λογικής, οι προτάσεις αντιστοιχούν σε εκφράσεις της γλώσσας τύπων και οι αποδείξεις αντιστοιχούν σε εκφράσεις στην γλώσσα υπολογισμών. Ο έλεγχος των αποδείξεων ανάγεται στον έλεγχο τύπων. Η απαραίτητη σύνδεση ανάμεσα στα δύο στρώματα, και συνεπώς ανάμεσα στις αποδείξεις και τις προτάσεις, γίνεται συνήθως με χρήση dependent τύπων [Xi99].

Όμως η ανάμειξη της γλώσσας τύπων και της γλώσσας υπολογισμού μέσω των dependent τύπων συνδέει τους στατικούς ελέγχους ορθότητας του προγράμματος με την εκτέλεση του. Η ανεπιθύμητη αυτή σύνδεση επιλύεται με την αναπαράσταση των αποδείξεων ως εκφράσεων της γλώσσας τύπων και των προτάσεων ως των τύπων αυτών των εκφράσεων. Πλέον, η σύνδεση ανάμεσα στη γλώσσα υπολογισμού και τη γλώσσα τύπων επιτυγχάνεται με τη χρήση singleton τύπων [Hayya91]. Οι singleton τύποι με την ένα-προς-ένα αντιστοιχία τους με τα αντικείμενα της γλώσσας υπολογισμού δίνουν τη δυνατότητα να εκφραστούν στη γλώσσα τύπων προτάσεις που περιγράφουν ιδιότητες του προγράμματος μαζί με τις αποδείξεις τους. Τώρα, ο στατικός έλεγχος είναι πλήρως αποσυνδεδεμένος από την εκτέλεση των προγραμμάτων.

Επιπλέον, η γλώσσα τύπων προσφέρει ένα πολύ δυνατό μηχανισμό επαγωγικών ορισμών το οποίο εξασφαλίζει ότι οι τύποι των όρων της γλώσσας υπολογισμού μπορούν να περιγραφούν σαν ένα επαγωγικά ορισμένο σύνολο, Ω . Συνεπώς η γλώσσα τύπων μπορεί να λειτουργήσει σαν ένα γενικότερο πλαίσιο και να συνδυαστεί με πολλές και διαφορετικές γλώσσες υπολογισμού.

Οι επαγωγικοί ορισμοί μπορούν να χρησιμοποιηθούν για τον ορισμό των τύπων δομών δεδομένων όπως μία λίστα:

```
Inductive List : Kind :=  
nil : List  
| cons :  $\Omega \rightarrow List \rightarrow List$ 
```

Μεσω των επαγωγικών ορισμών μπορούν να κωδικοποιηθούν και προτάσεις της λογικής:

```
Inductive LT : Nat → Nat → Kind  
:=
```

$$\begin{aligned} ltzs &: \Pi t:Nat. LT\ 0\ (\mathbf{succ}\ t) \\ ltss &: \Pi t:Nat. \Pi t':Nat. LT\ t\ t' \rightarrow LT\ (\mathbf{succ}\ t)\ (\mathbf{succ}\ t') \end{aligned}$$

Ακολουθεί ένα παράδειγμα όπου ορίζεται ένα διάνυσμα (*vector*) φυσικών αριθμών και στη συνέχεια δίνεται συνάρτηση που υπολογίζει το άθροισμά του:

$$\begin{aligned} vec &: Nat \rightarrow \Omega \rightarrow \Omega \\ vec &= \lambda t:Nat. \lambda t':\Omega. \mathbf{tup}\ t\ (nth\ (repeat\ t\ t')) \\ sumVec &: \forall t:Nat. snat\ t \rightarrow vec\ t\ nat \rightarrow nat \\ &\equiv \Lambda t:Nat \lambda n:snat\ t. \lambda v:vec\ t\ nat. \\ &\quad (\mathbf{fix}\ loop: nat \rightarrow nat \rightarrow nat, \\ &\quad \lambda i:nat. \lambda sum:nat. \\ &\quad \quad \mathbf{open}\ i\ \mathbf{as}\ < t', i' >\ \mathbf{in} \\ &\quad \quad \mathbf{if}\ [LTOrTrue\ t'\ t, ltPrf\ t'\ t] \\ &\quad \quad \quad (i' < n, \\ &\quad \quad \quad t_1. loop\ (add\ i\ 1)\ (add\ sum\ (\mathbf{sel}[t_1](v, i'))), \\ &\quad \quad \quad t_2. sum)) \end{aligned}$$

όπου

$$\begin{aligned} nth &: List \rightarrow Nat \rightarrow \Omega \\ nth\ nil &= \lambda t:Nat. \mathbf{void} \\ nth\ (cons\ t_1\ t_2) &= \lambda t:Nat. \mathbf{ifez}\ t\ \Omega\ t_1\ (nth\ t_2) \end{aligned}$$

$$\begin{aligned} repeat &: Nat \rightarrow \Omega \rightarrow List \\ repeat\ 0 &= \lambda t':\Omega. \mathbf{nil} \\ repeat\ (\mathbf{succ}\ t) &= \lambda t':\Omega. t' :: (repeat\ t)\ t' \end{aligned}$$

$$\begin{aligned} LTOrTrue &: Nat \rightarrow Nat \rightarrow Bool \rightarrow Kind \\ LTOrTrue &= \lambda t_1:Nat. \lambda t_2:Nat. \lambda t:Bool. \mathbf{Cond}\ t\ (LT\ t_1\ t_2)\ \mathbf{True} \end{aligned}$$

και open η πράξη αποσύνθεσης υπαρξιακών τύπων.

Κεφάλαιο 3

Περιγραφή και Ορισμός της Γλώσσας let!

Η έρευνα γύρω από τα substructural συστήματα τύπων έδειξε ότι μπορούν να χρησιμοποιηθούν για την ασφαλή εισαγωγή references και destructive update σε συναρτησιακές γλώσσες. Η μελέτη τους, όμως, ανέδειξε εκτός από τα πλεονεκτήματά τους και σημαντικά μειονεκτήματα. Οι linear τύποι είναι αρκετά δύσχρηστοι και μπορούν να μετατρέψουν σε πολύπλοκη και επίπονη διαδικασία τη συγγραφή προγραμμάτων. Ο συνδυασμός linear και unrestricted τύπων, αντίθετα, κάνει τον προγραμματισμό ευκολότερο όμως εγείρει ζητήματα ασφάλειας.

Για να μελετήσουμε τα substructural συστήματα τύπων και να εντοπίσουμε τις δυνατότητες και τις αδυναμίες τους ορίσαμε τη γλώσσα προγραμματισμού let!. Η γλώσσα let! διαθέτει ένα σύστημα τύπων με linear και unrestricted τύπους και υποστηρίζει references, destructive update και πολυμορφισμό. Ακόμα, προτείνει μία νέα ασφαλή μέθοδο συνδυασμού των linear και των unrestricted τύπων. Κατά τον σχεδιασμό της γλώσσας, έγινε προσπάθεια να εστιάσουμε στις ιδιότητες και τη λειτουργία των substructural συστημάτων τύπων αναζητώντας συγχρόνως όσο το δυνατόν πιο γενικές λύσεις. Η γλώσσα μας καταφέρνει να συνταιρίζει ασφάλεια και εκφραστικότητα ενώ η προσθήκη επιπλέον χαρακτηριστικών μπορεί να γίνει χωρίς ιδιαίτερες μετατροπές του αρχικού σκελετού.

3.1 Οι Τύποι της Γλώσσας

Οι τύποι της γλώσσας αποτελούνται από τρία τμήματα, τους pretypes και τους qualifiers και τα scopes. Ο συνδυασμός και των τριών στοιχείων οδηγεί στο σχηματισμό των πλήρων τύπων.

3.1.1 Pretypes

Οι pretypes μοιάζουν με τους τύπους των συνήθων συναρτησιακών γλωσσών. Κάθε pretype αντιστοιχεί και σε μία έκφραση της γλώσσας μέσω των κανόνων του συστήματος τύπων.

$$\phi ::= \alpha \mid b \mid \text{ref } \tau \mid \tau \xrightarrow{\vec{\eta}} \tau \mid \forall_\tau \alpha. \tau \mid \forall_s \rho. \tau$$

$$b ::= \text{unit} \mid \text{nat} \mid \text{bool}$$

3.1.2 Qualifiers

Το σύστημα τύπων της γλώσσας διαθέτει linear και unrestricted τύπους. Η διάκριση των δύο ειδών τύπων γίνεται μέσω των δύο αντίστοιχων qualifiers.

$$q ::= \mathbf{L} \mid \mathbf{U}$$

3.1.3 Scopes

Τα scopes χρησιμοποιούνται για τον έλεγχο της εγκυρότητας ενός δεδομένου. Παίζουν ιδιαίτερα σημαντικό ρόλο κατά τη μετατροπή ενός linear τύπου σε unrestricted και αντίστροφα.

$$\eta ::= \rho \mid \perp$$

3.1.4 Πλήρης Τύποι

Οι πλήρεις τύποι αποτελούν τους τύπους των εκφράσεων της γλώσσας.

$$\tau ::= \frac{q}{\eta} \phi$$

3.2 Η Σύνταξη της Γλώσσας

Οι εκφράσεις της γλώσσας μοιάζουν σε σύνταξη και ερμηνεία με τις εκφράσεις που συναντάμε συνήθως στις συναρτησιακές γλώσσες. Επιπλέον υπάρχουν εκφράσεις για την δημιουργία, την καταστροφή και χρήση των references και η γνωστή `let!` των γλωσσών με linear συστήματα τύπων.

$$\begin{aligned} e ::= & x \mid {}^q c \mid e \mid {}^q \text{unop } e \mid e_1 {}^q \text{binop } e_2 \\ & \mid {}^q \lambda^{\vec{\eta}} x:\tau.e \mid e_1 e_2 \mid \text{fix}(e) \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \\ & \mid {}^q \Lambda_{\tau} \alpha.e \mid {}^q \Lambda_s \rho.e \mid e [A] \mid e \{\eta\} \\ & \mid \text{new } e \mid \text{free } e \mid e_1 := e_2 \mid \text{deref } e \\ & \mid \text{at } \rho \text{ let!}(x) \ y = e_1 \text{ then } e_2 \end{aligned}$$

$$\text{unop} ::= \text{not}$$

$$\text{binop} ::= + \mid - \mid = \mid < \mid > \mid <= \mid >= \mid \text{and} \mid \text{or}$$

όπου	x, y	:	μεταβλητή των εκφράσεων της γλώσσας
	c	:	σταθερά της γλώσσας
	τ	:	πλήρης τύπος
	α	:	μεταβλητή pretype
	A	:	όρος pretype
	ρ, η	:	μεταβλητή scope
	q	\in	$\{\mathbf{U}, \mathbf{L}\}$

3.3 Λειτουργική Σημασιολογία

Για την καλύτερη κατανόηση της σύνταξης και της ερμηνείας των εκφράσεων της γλώσσας δίδεται πρώτα η λειτουργική σημασιολογία της γλώσσας.

3.3.1 Οι Τιμές της Γλώσσας

$$V ::= {}^q c \mid {}^q \lambda^{\vec{\eta}} x:\tau.e \mid {}^q \Lambda_{\tau} \alpha.e \mid {}^q \Lambda_s \rho.e \mid {}^q \text{loc } z$$

όπου	x, y	:	μεταβλητή των εκφράσεων της γλώσσας
	c	:	σταθερά της γλώσσας
	τ	:	πλήρης τύπος
	α	:	μεταβλητή pretype
	ρ, η	:	μεταβλητή scope
	q	\in	$\{\mathbf{U}, \mathbf{L}\}$

3.3.2 Εκτεταμένη Σύνταξη

Η λειτουργική σημασιολογία απαιτεί τον εμπλουτισμό της γλώσσας με επιπλέον εκφράσεις οι οποίες δεν είναι διαθέσιμες στον προγραμματιστή και εμφανίζονται μόνο κατά την διάρκεια των βημάτων της αποτίμησης.

$e ::= \dots | z | \text{at } \rho \text{ let!}(z) \ y = e_1 \text{ then } e_2$

όπου y : μεταβλητή των εκφράσεων της γλώσσας
 z : αυτόματη μεταβλητή των τιμών της γλώσσας
 ρ : μεταβλητή scope variable

3.3.3 Store και Scopes

Κατά την αποτίμηση των εκφράσεων είναι απαραίτητα δύο βοηθητικά περιβάλλοντα.

- Το περιβάλλον Σ των τιμών της γλώσσας.

$$\Sigma ::= \emptyset \mid \Sigma, \{z \mapsto {}^q_\eta v\}$$

όπου $q \in \{\mathbf{U}, \mathbf{L}\}$
 z : αυτόματη μεταβλητή των τιμών της γλώσσας
 η : μεταβλητή scope
 ${}^q_\eta v$: τιμή της γλώσσας

- Το περιβάλλον S των μεταβλητών scope. Το S περιέχει τα ενεργά scores κάθε στιγμή του προγράμματος. Μία τιμή είναι έγκυρη αν και μόνο αν το scope της ανήκει στο τρέχον S

$$S ::= \emptyset \mid S, \eta$$

όπου η : μεταβλητή scope

3.3.4 Βοηθητικές Συναρτήσεις

Τελεστής Ελέγχου Εγκυρότητας των Τιμών

Ο τελεστής ελέγχει αν η τιμή ${}^U_\eta v$ που αντιστοιχεί στη μεταβλητή z είναι έγκυρη. Μία τιμή χαρακτηρίζεται ως έγκυρη αν στο τρέχον Σ υπάρχει πληροφορία της μορφής $\{z \mapsto {}^U_\eta v\}$ και αν το scope η της τιμής ανήκει στο τρέχον S . Επιπλέον, ο τελεστής χειρίζεται διαφορετικά τις τιμές με linear και unrestricted χαρακτήρα. Οι τιμές με qualifier \mathbf{L} απομακρύνονται από το Σ μέσω του τελεστή εξασφαλίζοντας ότι ένα linear δεδομένο θα χρησιμοποιηθεί μόνο μία φορά. Από τα παραπάνω φαίνεται ότι είναι απαραίτητο οι qualifiers και τα scopes να επιβιώνουν μέχρι και το χρόνο εκτέλεσης.

$$\frac{\{z \mapsto {}^U_\eta v\} \in \Sigma \quad \eta \in S}{S; S; z \Downarrow \Sigma; {}^U_\eta v} \quad \frac{\{z \mapsto {}^L_\eta v\} \in \Sigma \quad \eta \in S \quad \Sigma' \equiv \Sigma / \{z \mapsto {}^L_\eta v\}}{S; S; z \Downarrow \Sigma'; {}^L_\eta v}$$

Τελεστής Μεταβολής των Περιεχομένων του Store

Ο τελεστής αφαιρεί την τιμή στην οποία δείχνει η μεταβλητή z στο Σ και στη θέση της τοποθετεί την τιμή ${}^q_\rho v$.

$$\Sigma | \{z \mapsto {}^q_\rho v\} \equiv \Sigma |_z, \{z \mapsto {}^q_\rho v\}$$

3.3.5 Σχέση Υπολογισμού

Στη συνέχεια δίδεται η σχέση υπολογισμού που προσομοιώνει τη λειτουργία της αφηρημένης μηχανής.

To Σχήμα Υπολογισμού

Ένα βήμα της αποτίμησης της έκφρασης e σε δοσμένο Σ και S οδηγεί στην έκφραση e' και σε ένα νέο store Σ' . Το S παραμένει αμετάβλητο.

$$\Sigma; e \xrightarrow{S} \Sigma'; e'$$

Τιμές

Η αυτόματη μηχανή αποθηκεύει όλες τις τιμές στο Σ . Αντιστοιχεί σε κάθε τιμή μία νέα αυτόματη μεταβλητή z και αντικαθιστά την εμφάνιση της τιμής στην έκφραση που αποτιμάται με τη μεταβλητή z .

$$\frac{z \text{ νέα μεταβλητή}}{\Sigma; {}^q v \xrightarrow{S} \Sigma, \{z \mapsto {}^q_{\perp} v\}; x}$$

Εναδικοί Τελεστές

$$\frac{\Sigma; e \xrightarrow{S} \Sigma'; e}{\Sigma; {}^q \text{unop } e \xrightarrow{S} \Sigma'; {}^q \text{unop } e'} \quad \frac{\Sigma; S; x \Downarrow \Sigma'; {}^q_{\eta} c \quad \| \text{unop } c \| \equiv c'}{\Sigma; {}^q' \text{unop } z \xrightarrow{S} \Sigma'; {}^q' c'}$$

Δυαδικοί Τελεστές

$$\frac{\Sigma; e_1 \xrightarrow{S} \Sigma'; e'_1}{\Sigma; e_1 \ {}^q \text{binop } e_2 \xrightarrow{S} \Sigma'; e'_1 \ {}^q \text{binop } e_2} \quad \frac{\Sigma; e_2 \xrightarrow{S} \Sigma'; e'_2}{\Sigma; z \ {}^q \text{binop } e_2 \xrightarrow{S} \Sigma'; z \ {}^q \text{binop } e'_2}$$

$$\frac{\Sigma; S; z_1 \Downarrow \Sigma'; {}^{q_1}_{\eta_1} c_1 \quad \Sigma; S'; z_2 \Downarrow \Sigma''; {}^{q_2}_{\eta_2} c_2 \quad \| \text{binop } c_1 \ c_2 \| \equiv c_3}{\Sigma; z_1 \ {}^{q_3} \text{binop } z_2 \xrightarrow{S} \Sigma''; {}^{q_3} c_3}$$

β-αναγωγή για Εκφράσεις

Κατά την κλήση μίας συνάρτησης είναι απαραίτητο να ελεγχθεί η εγκυρότητα εκτέλεσης του κώδικα που καλείται. Αυτό εξασφαλίζεται με τον έλεγχο της ύπαρξης των scopes που φέρει ο κώδικας στο τρέχον S .

$$\frac{\Sigma; e_1 \xrightarrow{S} \Sigma'; e'_1}{\Sigma; e_1 \ e_2 \xrightarrow{S} \Sigma'; e'_1 \ e_2} \quad \frac{\Sigma; e_2 \xrightarrow{S} \Sigma'; e'_2}{\Sigma; z \ e_2 \xrightarrow{S} \Sigma'; z \ e'_2}$$

$$\frac{\Sigma; S; z_1 \Downarrow \Sigma'; {}^q_{\eta} \lambda^{\vec{\eta}} \ x: \tau. e \quad \vec{\eta} \in S}{\Sigma; z_1 \ z_2 \xrightarrow{S} \Sigma'; e\{x \mapsto z_2\}}$$

β-αναγωγή για Τύπους

$$\frac{\Sigma; e_1 \xrightarrow{S} \Sigma'; e'_1}{\Sigma; e_1 \ [\tau] \xrightarrow{S} \Sigma'; e'_1 \ [\tau]} \quad \frac{\Sigma; S; z \Downarrow \Sigma'; {}^q_{\eta} \text{poly } \alpha: e.}{\Sigma; z \ [A] \xrightarrow{S} \Sigma'; e\{\alpha \mapsto A\}}$$

β-αναγωγή για Scopes

$$\frac{\Sigma; e_1 \xrightarrow{S} \Sigma'; e'_1}{\Sigma; e_1 \ \{\eta\} \xrightarrow{S} \Sigma'; e'_1 \ \{\eta\}} \quad \frac{\Sigma; S; z \Downarrow \Sigma'; {}^q_{\eta} \text{poly } + : \rho. e \quad \eta \in S}{\Sigma; z \ \{\eta\} \xrightarrow{S} \Sigma'; e\{\rho \mapsto \eta\}}$$

Τελεστής Fix

$$\frac{\Sigma; e \xrightarrow{S} \Sigma'; e'}{\Sigma; \text{fix}(e) \xrightarrow{S} \Sigma'; \text{fix}(e')} \quad \Sigma; \text{fix}(z) \xrightarrow{S} \Sigma; z \text{ fix}(z)$$

Αναφορές

- Δημιουργία Νέας Αναφοράς

$$\frac{\Sigma; e \xrightarrow{S} \Sigma'; e'}{\Sigma; \text{new } e \xrightarrow{S} \Sigma'; \text{new } e'} \quad \frac{\Sigma; S; z \Downarrow \Sigma'; \eta^q v \quad \text{fresh } z'}{\Sigma; \text{new } z \xrightarrow{S} \Sigma', \{z' \mapsto \text{Loc}_\perp z\}; z'}$$

- Καταστροφή Αναφοράς και Αποδέσμευση της Αντίστοιχης Θέσης Μνήμης

$$\frac{\Sigma; e \xrightarrow{S} \Sigma'; e'}{\Sigma; \text{free } e \xrightarrow{S} \Sigma'; \text{free } e'} \quad \frac{\Sigma; S; z \Downarrow \Sigma'; \eta^q \text{loc } z'}{\Sigma; \text{free } z \xrightarrow{S} \Sigma'|_{z'}; \text{U unit}}$$

- Εντολή Ανάθεσης

$$\frac{\Sigma; e_2 \xrightarrow{S} \Sigma'; e'_2}{\Sigma; e_1 := e_2 \xrightarrow{S} \Sigma'; e'_1 := e_2} \quad \frac{\Sigma; e_2 \xrightarrow{S} \Sigma'; e'_2}{\Sigma; M; H; z := e_2 \xrightarrow{S} \Sigma'; z := e'_2}$$

$$\frac{\Sigma; S; z_1 \Downarrow \Sigma'; \eta^q \text{loc } z_3 \quad \Sigma'; S; z_2 \Downarrow \Sigma''; \eta'^q v}{\Sigma; z_1 := z_2 \xrightarrow{S} \Sigma''|_{z_1}, \{z_1 \mapsto \text{Loc}_\perp z_2\}; z_1}$$

- Εντολή Ανάγνωσης

$$\frac{\Sigma; e \xrightarrow{S} \Sigma'; e'}{\Sigma; \text{deref } e \xrightarrow{S} \Sigma'; \text{deref } e'} \quad \frac{\Sigma; S; z \Downarrow \Sigma'; \eta^q \text{loc } z' \quad \Sigma'; S; z' \Downarrow \Sigma''; \eta'^q v}{\Sigma; \text{deref } z \xrightarrow{S} \Sigma''; z'}$$

Εντολή Ελέγχου με Συνθήκη if

$$\frac{\Sigma; e \xrightarrow{S} \Sigma'; e'}{\Sigma; \text{if } e \text{ then } e_1 \text{ else } e_2 \xrightarrow{S} \Sigma'; \text{if } e' \text{ then } e_1 \text{ else } e_2}$$

IF-true

$$\frac{\Sigma; S; x \Downarrow \Sigma'; \eta^q \text{true}}{\Sigma; \text{if } x \text{ then } e_1 \text{ else } e_2 \xrightarrow{S} \Sigma'; e_1}$$

IF-false

$$\frac{\Sigma; S; z \Downarrow \Sigma'; \eta^q \text{false}}{\Sigma; \text{if } z \text{ then } e_1 \text{ else } e_2 \xrightarrow{S} \Sigma'; e_2}$$

Ο 'Όρος auto!

Ο όρος `let!` μετατρέπει linear τιμές προσωρινά σε `unrestricted`. Είναι απαραίτητο να εξασφαλιστεί ότι όλα τα `unrestricted` αντίγραφα της τιμής πάνω στην οποία εφαρμόζεται ο όρος `let!` Ή αποτελεί ότι μόνο μέσα στην συντακτική εμβέλεια που ορίζει ο όρος. Διαφορετικά η ασφάλεια του συστήματος γίνεται διάτρητη καθώς συνυπάρχουν `linear` και `unrestricted` αντίγραφα για το ίδιο δεδομένο.

Η αποτίμηση του όρου `at ρ let!(x) y = e1 then e2` μπορεί να ξεκινήσει μόνο όταν η μεταβλητή x έχει αντικατασταθεί από κάποια αυτόματη μεταβλητή z μέσω β-αναγωγής. Τότε ο όρος αποκτά τη μορφή του όρου `auto!`: `at ρ let!(z) y = e1 then e2`. Κατά την αποτίμηση της έκφρασης e_1 , η `linear` τιμή στην οποία δείχνει η μεταβλητή z στο `store` μετατρέπεται σε `unrestricted` ενώ πλέον το `scope` της είναι ίσο με ρ . Στο S προστίθεται η μεταβλητή `scope` ρ ώστε να εξασφαλιστεί η εγκυρότητα της νέας τιμής της μεταβλητής z . Όταν αποτιμηθεί η έκφραση e_1 , ξεκινάει η αποτίμηση της έκφρασης e_2 με την μεταβλητή y να έχει αντικατασταθεί με την αυτόματη μεταβλητή z' η οποία δείχνει στην τιμή στην οποία κατέληξε η αποτίμηση της έκφρασης e_1 . Τώρα όμως η μεταβλητή z δείχνει στην αρχική `linear` τιμή της ενώ η μεταβλητή ρ δεν ανήκει στο S . Κάθε `unrestricted` τιμή με `scope` ίσο με ρ δεν είναι πλέον έγκυρη.

$$\frac{\Sigma, \{z \mapsto {}^U_\rho v\}; e_1 \xrightarrow{S, \rho} \Sigma'; e'_1}{\begin{aligned} & \Sigma, \{z \mapsto {}^L_\perp v\}; \text{at } \rho \text{ let!}(z) y = e_1 \text{ then } e_2 \xrightarrow{S} \\ & \Sigma'|_z, \{z \mapsto {}^L_\perp v\}; \text{at } \rho \text{ let!}(z) y = e'_1 \text{ then } e_2 \\ & \Sigma; S; z' \Downarrow \Sigma'; {}^q_\eta v \end{aligned}} \frac{}{\Sigma; \text{at } \rho \text{ let!}(z) y = z' \text{ then } e_2 \xrightarrow{S} \Sigma'; e_2\{y \mapsto z'\}}$$

3.4 Κανόνες του Συστήματος Τύπων

3.4.1 Περιβάλλοντα

- Το περιβάλλον Γ των μεταβλητών των εκφράσεων της γλώσσας.

$$\Gamma ::= \emptyset \mid \Gamma, x \triangleright \tau$$

όπου x : μεταβλητή των εκφράσεων της γλώσσας
 τ : πλήρης τύπος

Ο τελεστής `split` \oplus εξασφαλίζει ότι `linear` υποθέσεις δεν διπλασιάζονται ή αγνοούνται κατά τον τυποθεωρητικό έλεγχο των εκφράσεων.

$$\begin{array}{lll} \emptyset \oplus \Gamma & \equiv & \Gamma \\ (\Gamma_1, x \triangleright \tau) \oplus \Gamma_2 & \equiv & \Gamma_1 \oplus \Gamma_2, x \triangleright \tau \\ (\Gamma_1, x \triangleright \tau) \oplus \Gamma_2 & \equiv & \Gamma_1 \oplus \Gamma_2 \end{array} \quad \begin{array}{l} \text{αν } (x \triangleright \tau) \notin \Gamma_2 \\ \text{αν } (x \triangleright \tau) \in \Gamma_2 \text{ και } \tau \preceq \mathbf{U} \end{array}$$

- Το περιβάλλον Δ των μεταβλητών τύπων

$$\Delta ::= \emptyset \mid \Delta, \alpha$$

όπου α : μεταβλητή pretype

- Το περιβάλλον S των μεταβλητών scope. Το S περιέχει τα ενεργά scopes κάθε στιγμή του προγράμματος. Ένας τύπος είναι έγκυρος αν και μόνο αν το scope του ανήκει στο τρέχον S

$$S ::= \emptyset \mid S, \eta$$

όπου η : μεταβλητή scope

- Το περιβάλλον των τύπων των τιμών που ανήκουν στο store (*store typing*) M . Το περιβάλλον M είναι απαραίτητο μόνο κατά τον έλεγχο των εκφράσεων που έχουν προκύψει από τα ενδιάμεσα βήματα της αποτίμησης μίας έκφρασης δοσμένης από τον προγραμματιστή. Τα προγράμματα που δίδει ο χρήστης ελέγχονται με κενό M .

$$M ::= \emptyset \mid M, x \triangleright \tau$$

όπου x : αυτόματη μεταβλητή της γλώσσας
 τ : πλήρης τύπος

Ο τελεστής split \oplus :

$$\begin{array}{lll} \emptyset \oplus M & \equiv & M \\ (M_1, x \triangleright \tau) \oplus M_2 & \equiv & M_1 \oplus M_2, x \triangleright \tau \\ (M_1, x \triangleright \tau) \oplus M_2 & \equiv & M_1 \oplus M_2 \end{array} \quad \begin{array}{l} \text{αν } (x \triangleright \tau) \notin M_2 \\ \text{αν } (x \triangleright \tau) \in M_2 \text{ και } \tau \preceq U \end{array}$$

3.4.2 Μεταβλητές και Σταθερές

Μη Αυτόματες Μεταβλητές

$$\frac{q \in \{\mathbf{U}, \mathbf{L}\} \quad \Delta \vdash \Gamma \preceq \mathbf{U}}{\Delta; \Gamma \oplus (x \triangleright \perp^q \phi); M; S \vdash x \triangleright \perp^q \phi} \quad \frac{q \in \{\mathbf{U}, \mathbf{L}\} \quad \Delta \vdash \Gamma \preceq \mathbf{U}}{\Delta; \Gamma \oplus (x \triangleright \eta^q \phi); M; S, \eta \vdash x \triangleright \eta^q \phi}$$

Αυτόματες Μεταβλητές

$$\frac{q \in \{\mathbf{U}, \mathbf{L}\} \quad \Delta \vdash \Gamma \preceq \mathbf{U} \quad \Delta \vdash M \preceq \mathbf{U}}{\Delta; \Gamma; M \oplus (z \triangleright \perp^q \phi); S \vdash z \triangleright \perp^q \phi} \quad \frac{q \in \{\mathbf{U}, \mathbf{L}\} \quad \Delta \vdash \Gamma \preceq \mathbf{U} \quad \Delta \vdash M \preceq \mathbf{U}}{\Delta; \Gamma; M \oplus (z \triangleright \eta^q \phi); S, \eta \vdash z \triangleright \eta^q \phi}$$

Σταθερές

$$\frac{q \in \{\mathbf{U}, \mathbf{L}\} \quad \Delta \vdash \Gamma \preceq \mathbf{U} \quad \Delta \vdash M \preceq \mathbf{U}}{\Delta; \Gamma; M; S \vdash {}^q c \triangleright \perp^q b} \quad \frac{q \in \{\mathbf{U}, \mathbf{L}\} \quad \Delta \vdash \Gamma \preceq \mathbf{U} \quad \Delta \vdash M \preceq \mathbf{U}}{\Delta; \Gamma; M; S, \eta \vdash {}^q c \triangleright \eta^q b}$$

3.4.3 Εναδικοί και Διαδικοί Τελεστές

Αριθμητικοί Τελεστές

$$\frac{\begin{array}{c} \Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \mathbf{nat} \\ \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \mathbf{nat} \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \triangleright \perp^{q_3} \mathbf{nat}} \quad \frac{\begin{array}{c} \Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \mathbf{nat} \\ \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \mathbf{nat} \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \triangleright \perp^{q_3} \mathbf{nat}}$$

Σχεσιακοί Τελεστές

$$\begin{array}{c}
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{nat} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{nat}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{=} e_2 \triangleright \frac{q_3}{\perp} \text{bool}} \\[10pt]
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{nat} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{nat}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{<>} e_2 \triangleright \frac{q_3}{\perp} \text{bool}} \\[10pt]
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{nat} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{nat}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{<=} e_2 \triangleright \frac{q_3}{\perp} \text{bool}} \\[10pt]
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{nat} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{nat}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{>} e_2 \triangleright \frac{q_3}{\perp} \text{bool}} \\[10pt]
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{nat} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{nat}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{<} e_2 \triangleright \frac{q_3}{\perp} \text{bool}} \\[10pt]
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{nat} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{nat}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{>} e_2 \triangleright \frac{q_3}{\perp} \text{bool}}
 \end{array}$$

Λογικοί Τελεστές

$$\begin{array}{c}
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{bool} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{bool}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{\text{and}} e_2 \triangleright \frac{q_3}{\perp} \text{bool}} \\[10pt]
 \frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright \frac{q_1}{\eta_1} \text{bool} \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \frac{q_2}{\eta_2} \text{bool}}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 \stackrel{q_3}{\text{or}} e_2 \triangleright \frac{q_3}{\perp} \text{bool}} \\[10pt]
 \frac{\Delta; \Gamma; M; S \vdash e \triangleright \frac{q}{\eta} \text{bool}}{\Delta; \Gamma; M; S \vdash q' \text{not } e \triangleright \frac{q'}{\perp} \text{bool}}
 \end{array}$$

3.4.4 Συναρτήσεις

Ο qualifier μίας συνάρτησης καθορίζεται από τους qualifiers των υποθέσεων που χρησιμοποιείται. Αν στο περιβάλλον αποτίμησης της συνάρτησης υπάρχει κάποια linear υπόθεση τότε η συνάρτηση αποκτά qualifier **L**. Διαφορετικά, η συνάρτηση έχει qualifier **U**. Μ' αυτό τον τρόπο εξασφαλίζεται ότι μία linear υπόθεση θα χρησιμοποιηθεί ακριβώς μία φορά.

Επειδή οι συναρτήσεις μπορούν να περιέχουν σαν ελεύθερες μεταβλητές υποθέσεις οι οποίες δεν είναι ανιχνεύσιμες μέσω του τύπου των συναρτήσεων είναι απαραίτητο το σύστημα τύπων να υιοθετεί ελέγχους για να εξασφαλίζει την εγκυρότητα των χρησιμοποιούμενων υποθέσεων από τις συναρτήσεις. Το πρόβλημα εστιάζεται στις unrestricted υποθέσεις καθώς μία linear υπόθεση είναι πάντα έγκυρη. Πρέπει να επιβεβαιωθεί ότι μία unrestricted υπόθεση χρησιμοποιείται αν και μόνο αν το scope της είναι έγκυρο. Γι' αυτό τόσο στον τύπο των συναρτήσεων όσο και στην σύνταξή τους καταγράφονται όλα τα scopes των υποθέσεων που χρησιμοποιεί μία συνάρτηση.

Μ' αυτό τον τρόπο μπορεί να ελεγχθεί κάθε στιγμή αν η χρήση ενός τμήματος κώδικα είναι έγκυρη και ασφαλής.

$$\frac{\Delta; \emptyset \vdash \tau \quad \Delta; \Gamma, x \triangleright \tau; M; \emptyset \vdash e \triangleright \tau' \quad q \equiv \sqcup\{\Gamma, M\} \quad \vec{\eta} \equiv S}{\Delta; \Gamma; M; \emptyset \vdash {}^q \lambda^{\{x\}}. e \triangleright {}^q_{\perp}(\tau \xrightarrow{\vec{\eta}} \tau')}$$

$$\frac{\Delta; S \vdash \tau \quad \Delta; \Gamma, x \triangleright \tau; M; S \vdash e \triangleright \tau' \quad q \equiv \sqcup\{\Gamma, M\} \quad \vec{\eta} \equiv S}{\Delta; \Gamma; M; S \vdash {}^q \lambda^S. e \triangleright {}^q_{\vec{\eta}}(\tau \xrightarrow{\vec{\eta}} \tau')}$$

$$\frac{\Delta; \Gamma_1, x \triangleright \tau; M_1; S \vdash e_1 \triangleright {}^q_{\vec{\eta}}(\tau \xrightarrow{\vec{\eta}} \tau') \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \tau}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S, \vec{\eta} \vdash e_1 e_2 \triangleright \tau'}$$

$$\frac{\Delta, \alpha; \Gamma; M; S \vdash e \triangleright \tau \quad q \equiv \sqcup\{\Gamma, M\}}{\Delta; \Gamma; M; S \vdash {}^q \text{poly } \alpha : e. \triangleright {}^q_{\perp} \forall_{\tau} \alpha. \tau} \quad \frac{\Delta; \Gamma; M; S \vdash e \triangleright {}^q_{\eta} \forall_{\tau} \alpha. \tau'}{\Delta; \Gamma; M; S \vdash e [\tau] \triangleright \tau' \{\alpha \mapsto \tau\}}$$

$$\frac{\Delta; \Gamma; M; S, \rho \vdash e \triangleright \tau \quad q \equiv \sqcup\{\Gamma, M\}}{\Delta; \Gamma; M; S \vdash {}^q \text{poly } + : \rho. e \triangleright {}^q_{\perp} \forall_s \rho. \tau} \quad \frac{\Delta; \Gamma; M; S \vdash e \triangleright {}^q_{\eta} \forall_s \rho. \tau'}{\Delta; \Gamma; M; S \oplus \eta' \vdash e \{\eta'\} \triangleright \tau' \{\rho \mapsto \eta'\}}$$

$$\frac{\Delta; \Gamma; M; S \vdash e \triangleright {}^q_{\vec{\eta}}(\tau \xrightarrow{\vec{\eta}} \tau) \quad \mathbf{U} \equiv \text{getq}(\tau) \quad \vec{\eta} \equiv S}{\Delta; \Gamma; M; S, \vec{\eta} \vdash \text{fix}(e) \triangleright \tau}$$

3.4.5 Αναφορές

Οι αναφορές της γλώσσας `let!` μπορούν να δείχνουν μόνο σε unrestriced δεδομένα. Η επέκταση της γλώσσας ώστε να περιλαμβάνει και δείκτες που δείχνουν σε linear δεδομένα μπορεί να γίνει εύκολα. Προτιμήθηκε, εδώ, η πιο απλή εκδοχή.

$$\frac{\Delta; \Gamma; M; S \vdash e \triangleright \tau \quad \mathbf{U} \equiv \text{getq}(\tau)}{\Delta; \Gamma; M; S \vdash \text{new } e \triangleright {}^L_{\perp} \text{ref } \tau} \quad \frac{\Delta; \Gamma; M; S \vdash e \triangleright {}^L_{\eta} \text{ref } \tau}{\Delta; \Gamma; M; S \vdash \text{free } e \triangleright {}^U_{\perp} \text{unit}}$$

$$\frac{\Delta; \Gamma_1; M_1; S \vdash e_1 \triangleright {}^L_{\perp} \text{ref } \tau \quad \Delta; \Gamma_2; M_2; S \vdash e_2 \triangleright \tau \quad \mathbf{L} \equiv \text{getq}(\tau)}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2; S \vdash e_1 := e_2 \triangleright {}^L_{\perp} \text{ref } \tau'}$$

$$\frac{\Delta; \Gamma; M; S \vdash e \triangleright {}^U_{\eta} \text{ref } {}^U_{\eta'} \phi}{\Delta; \Gamma; M; S, \eta' \vdash \text{deref } e \triangleright {}^U_{\eta'} \phi}$$

3.4.6 Εντολή Ελέγχου με Συνθήκη if

$$\frac{\Delta; \Gamma; M; S \vdash e \triangleright {}^q_{\eta} \text{bool} \quad \Delta; \Gamma'; M'; S \vdash e_1 \triangleright \tau \quad \Delta; \Gamma'; M'; S \vdash e_2 \triangleright \tau}{\Delta; \Gamma \oplus \Gamma'; M \oplus M'; S \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \triangleright \tau}$$

3.4.7 Ο 'Ορος let!

Ο όρος `let!` μετατρέπει των linear τύπο δεδομένων προσωρινά σε unrestricted. Είναι απαραίτητο να εξασφαλιστεί ότι όλα τα unrestricted αντίγραφα του τύπου της μεταβλητής x , πάνω στην οποίο εφαρμόζεται ο όρος `at ρ let! (x) y = e1 then e2`, θα επιβιώσουν μόνο μέσα στην συντακτική εμβέλεια που ορίζει η έκφραση e_1 . Διαφορετικά η ασφάλεια του συστήματος γίνεται διάτρητη καθώς συνυπάρχουν linear και unrestricted αντίγραφα για το ίδιο δεδομένο στην έκφραση e_2 .

Για να επιτευχθεί ο στόχος της ασφάλειας, ο τύπος της μεταβλητής x αποκτά στο εσωτερικό της έκφρασης e_1 εκτός από διαφορετικό qualifier και ένα καινούργιο scope, ρ , που χαρακτηρίζει μοναδικά την τρέχουσα χρήση του όρου `let!`. Το νέο scope προστίθεται στο S μόνο κατά τον έλεγχο της έκφρασης e_1 και δεν υφίσταται κατά τον έλεγχο της έκφρασης e_2 . Αυτό έχει ως αποτέλεσμα να μην μπορούν να ελεγχθούν επιτυχώς τα unrestricted αντίγραφα που αποδρούν από την εμβέλεια του e_1 . Ο έλεγχος της έκφρασης e_2 γίνεται σε διευρυμένο περιβάλλον Γ με την προσθήκη της μεταβλητής y που φέρει τον τύπο τ' της έκφρασης e_1 και την μεταβλητή x με τον αρχικό της τύπο.

Αξίζει να σημειωθεί ότι δεν μπορεί να εφαρμοστεί ο όρος `let!` πάνω σε μεταβλητές με τύπο συνάρτησης. Ο περιορισμός επιβάλλεται γιατί μία linear συνάρτηση χρησιμοποιεί linear υποθέσεις. Η μετατροπή της σε unrestricted δίνει τη δυνατότητα να χρησιμοποιηθούν linear υποθέσεις κατά βούλησιν παραβιάζοντας τις αρχές των linear συστημάτων τύπων.

$$\frac{\Delta; \Gamma_1, x \triangleright_{\rho}^U b; M_1; S, \rho \vdash e_1 \triangleright \tau' \\ \Delta; S \vdash \tau' \quad \Delta; \Gamma_2, x \triangleright_{\perp}^L b, y \triangleright \tau'; M_2; S \vdash e_2 \triangleright \tau}{\Delta; \Gamma_1 \oplus \Gamma_2, x \triangleright_{\perp}^L b; M_1 \oplus M_2; S \vdash \text{at } \rho \text{ let! } (x) \ y = e_1 \text{ then } e_2 \triangleright \tau}$$

$$\frac{\Delta; \Gamma_1, x \triangleright_{\rho}^U \text{ref } \tau''; M_1; S, \rho \vdash e_1 \triangleright \tau' \\ \Delta; S \vdash \tau' \quad \Delta; \Gamma_2, x \triangleright_{\perp}^L \text{ref } \tau'', y \triangleright \tau'; M_2; S \vdash e_2 \triangleright \tau}{\Delta; \Gamma_1 \oplus \Gamma_2, x \triangleright_{\perp}^L \text{ref } \tau''; M_1 \oplus M_2; S \vdash \text{at } \rho \text{ let! } (x) \ y = e_1 \text{ then } e_2 \triangleright \tau}$$

3.4.8 Ο 'Ορος auto!

Ο όρος `auto!` εμφανίζεται μόνο στα ενδιάμεσα στάδια της αποτίμησης των εκφράσεων της γλώσσας και δεν είναι διαθέσιμος στον προγραμματιστή. Η λειτουργία του είναι παρόμοια με τη λειτουργία του όρου `let!` με τη διαφορά ότι εφαρμόζεται πάνω σε αυτόματη μεταβλητή z του συστήματος αντί της μη αυτόματης μεταβλητής x .

$$\frac{\Delta; \Gamma_1; M_1, z \triangleright_{\rho}^U b; S, \rho \vdash e_1 \triangleright \tau' \\ \Delta; S \vdash \tau' \quad \Delta; \Gamma_2, y \triangleright \tau'; M_2, z \triangleright_{\perp}^L b; S \vdash e_2 \triangleright \tau}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2, z \triangleright_{\perp}^L b; S \vdash \text{at } \rho \text{ let! } (z) \ y = e_1 \text{ then } e_2 \triangleright \tau}$$

$$\frac{\Delta; \Gamma_1; M_1, z \triangleright_{\rho}^U \text{ref } \tau''; S, \rho \vdash e_1 \triangleright \tau' \\ \Delta; S \vdash \tau' \quad \Delta; \Gamma_2, y \triangleright \tau'; M_2, z \triangleright_{\perp}^L \text{ref } \tau''; S \vdash e_2 \triangleright \tau}{\Delta; \Gamma_1 \oplus \Gamma_2; M_1 \oplus M_2, z \triangleright_{\perp}^L \text{ref } \tau''; S \vdash \text{at } \rho \text{ let! } (z) \ y = e_1 \text{ then } e_2 \triangleright \tau}$$

3.5 Μεταθεωρία της Γλώσσας let!

3.5.1 Η Σχέση Συμφωνίας Σ και M

Το περιβάλλον M περιέχει τους τύπους των τιμών της γλώσσας που ανήκουν στο τρέχον store του προγράμματος Σ . Είναι, λοιπόν, απαραίτητο να οριστεί πότε ένα περιβάλλον M είναι ορθά κατασκευασμένο, δηλαδή πότε είναι σε συμφωνία με το store Σ . Μπορούμε να ισχυριστούμε

ότι μία υπόθεση $z > \tau$ ανήκει ορθά στο M αν και μόνο αν υπάρχει στο store Σεγγραφή της μορφής $\{z \mapsto {}^q v\}$ και υπάρχουν Δ, Γ, S, M' τέτοια ώστε $\Delta; \Gamma; S; M' \vdash {}^q v > \tau$. Με τυπικό τρόπο η συμφωνία Σ και M περιγράφεται με τη σχέση:

$$\frac{\Delta; \Gamma_1; S \vdash \Sigma : M_1 \oplus M_2 \quad \Delta; \Gamma_2; S; M_2 \vdash {}^q v > \tau}{\Delta; \Gamma_1 \oplus \Gamma_2; S \vdash \Sigma, \{z \mapsto {}^q v\} : M_2, z > \tau}$$

3.5.2 Safety

Η γλώσσα `let!` είναι μία γλώσσα που εγγυάται την ασφάλεια δηλαδή εξασφαλίζει ότι κάθε τυποθεωρητικά ορθά διατυπωμένο πρόγραμμα αν τερματίζει, θα τερματίζει με ομαλό τρόπο. Διαφορετικά, κανένα ορθά διατυπωμένο πρόγραμμά δεν τερματίζει με μη ομαλό τρόπο. Το θεώρημα Safety διατυπώνει με τυπικό τρόπο τον παραπάνω ισχυρισμό

Θεώρημα 1 (Safety): Για κάθε έκφραση e , αν ισχύει ότι υπάρχουν τ και Σ' τέτοια ώστε $\emptyset; \emptyset; \emptyset; \emptyset \vdash e > \tau$ και $\emptyset; e \xrightarrow{*} \Sigma'; e'$ τότε η έκφραση e' είναι αυτόματη μεταβλητή z η οποία ανήκει στο $dom(\Sigma')$, όπου $\omega \xrightarrow{*}$ ορίζεται το μεταβατικό και ανακλαστικό κλείσιμο της σχέσης $\xrightarrow{\emptyset}$.

Για την απόδειξη του θεωρήματος safety χρειάζεται να αποδειχθούν τα θεωρήματα Preservation και Progress κατ' αντίστοιχα με την διαδικασία της απόδειξης των περισσότερων συναρτησιακών γλωσσών με ισχυρά συστήματα τύπων [Pier02].

Θεώρημα 2 (Preservation): Για κάθε έκφραση e , αν ισχύει ότι υπάρχουν $\tau, \Delta, \Gamma_1, S, M_1, M_2, \Sigma$ και Σ' τέτοια ώστε $\Delta; \Gamma_1; S; M_1 \vdash e > \tau$ και $\Delta; \Gamma_2; S \oplus S' \vdash \Sigma : M_1 \oplus M_2$ και $\Sigma; e \xrightarrow{S} \Sigma'; e'$ τότε υπάρχουν $\Gamma'_1, \Gamma'_2, M'_1$ τέτοια ώστε $\Gamma_1 \oplus \Gamma_2 \equiv \Gamma'_1 \oplus \Gamma'_2$ και $M' \equiv M'_1 \oplus M'_2$ και $\Delta; \Gamma'_1; S \vdash e' > \tau$ και $\Delta; \Gamma'_2; S \oplus S' \oplus S'' \vdash \Sigma' : M'_1 \oplus M_2$

Θεώρημα 3 (Progress): Για κάθε έκφραση e , αν ισχύει ότι υπάρχουν τ, S, M_1, M_2 και Σ τέτοια ώστε $\emptyset; \emptyset; S; M_1 \vdash e > \tau$ και $\emptyset; \emptyset; S \oplus S' \vdash \Sigma : M_1 \oplus M_2$ τότε η έκφραση e' είναι αυτόματη μεταβλητή z η οποία ανήκει στο $dom(\Sigma')$ ή υπάρχουν e', Σ' τέτοια ώστε $\Sigma; e \xrightarrow{S} \Sigma'; e'$

3.6 Παραδείγματα

Για την καλύτερη κατανόηση των δυνατοτήτων της γλώσσας `let!` δίδονται ορισμένα παραδείγματα. Σε κάποια από τα παραδείγματα χρησιμοποιείται μία επέκταση της `let!` που διαθέτει και πίνακες unrestricted δεδομένων δίνοντας τη δυνατότητα για τη δημιουργία πιο ενδιαφέροντων προγραμμάτων. Η εισαγωγή των πινάκων δεν έγινε στην αρχική μορφή της γλώσσας καθώς η παρουσία των πινάκων αυξάνει την πολυπλοκότητα της απόδειξης της μεταθεωρίας της γλώσσας χωρίς να προσφέρει στη μελέτη της σχέσης ανάμεσα στα substructural συστήματα τύπων και το destructive update, που είναι και ο στόχος της σχεδίασης της γλώσσας `let!`.

3.6.1 Increase

Η συνάρτηση (`incr`) αυξάνει κατά 1 το περιεχόμενο της αναφοράς σε φυσικούς αριθμούς που δίδεται ως παράμετρος.

$$\begin{aligned} \text{incr} &\triangleright \frac{}{\perp (\perp \text{ref } \perp \text{nat} \xrightarrow{\perp} \perp \text{ref } \perp \text{nat})} \\ &= \frac{}{U \lambda^\perp x : \perp \text{ref } \perp \text{nat}} \\ &\quad \text{at } \rho \text{ let!}(x) \ y = \text{deref } x \text{ then } x := y + 1 \end{aligned}$$

3.6.2 Swap

Η συνάρτηση (*swap*) ανταλλάσσει τα περιεχόμενα δύο θέσεων ενός πίνακα. Η συνάρτηση είναι πολυμορφική ως προς τον τύπο των περιεχομένων του πίνακα. Ο πίνακας και οι δείκτες(indexes) των στοιχείων που ανταλλάσσουν θέσεις δίδονται ως παράμετροι.

$$\begin{aligned} \text{swap} &\triangleright \frac{\mathbf{U}\forall_s\rho.\mathbf{U}\forall_\tau\alpha.}{\perp(\mathbf{U}\mathbf{nat}\xrightarrow[\rho]{\perp}\mathbf{U}(\mathbf{nat}\xrightarrow[\rho]{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha\xrightarrow[\rho]{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha)))} \\ &= \mathbf{U}\mathbf{poly} + : \rho. \mathbf{U}\mathbf{poly} \alpha : \\ &\quad \mathbf{U}\lambda^\rho n_1:\mathbf{U}\mathbf{nat}. \mathbf{U}\lambda^\rho n_2:\mathbf{U}\mathbf{nat}. \mathbf{U}\lambda^\rho x:\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha. \\ &\quad \text{at } \rho' \text{ let!}(x) y_1 = \text{read } x \text{ } n_1 \text{ then} \\ &\quad \text{at } \rho'' \text{ let!}(x) y_2 = \text{read } x \text{ } n_2 \text{ then} \\ &\quad \text{write (write } x \text{ } n_1) \text{ } n_2. \end{aligned}$$

3.6.3 Reverse

Η συνάρτηση (*reverse*) αντιστρέφει τα περιεχόμενα ενός πίνακα. Η συνάρτηση είναι πολυμορφική ως προς τον τύπο των περιεχομένων του πίνακα. Ο πίνακας και το μέγεθος του δίδονται ως παράμετροι.

$$\begin{aligned} \text{reverse} &\triangleright \frac{\mathbf{U}\forall_s\rho.\mathbf{U}\forall_s\rho'.\mathbf{U}\forall_\tau\alpha.}{\perp(\mathbf{U}\mathbf{nat}\xrightarrow[\rho,\rho']{\perp}\mathbf{U}(\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha\xrightarrow[\rho,\rho']{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha))} \\ &= \mathbf{U}\mathbf{poly} + : \rho. \mathbf{U}\mathbf{poly} + : \rho'. \mathbf{U}\mathbf{poly} \alpha : \\ &\quad \mathbf{U}\lambda^{\rho,\rho'} n:\mathbf{U}\mathbf{nat}. \\ &\quad \text{fix}(\mathbf{U}\lambda^{\rho,\rho'} \text{ walk}: \\ &\quad \frac{\mathbf{U}(\mathbf{U}\mathbf{nat}\xrightarrow[\rho,\rho']{\perp}\mathbf{U}(\mathbf{U}\mathbf{nat}\xrightarrow[\rho,\rho']{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha\xrightarrow[\rho,\rho']{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha))). \\ &\quad \mathbf{U}\lambda^{\rho,\rho'} i:\mathbf{U}\mathbf{nat}. \mathbf{U}\lambda^{\rho,\rho'} j:\mathbf{U}\mathbf{nat}. \mathbf{U}\lambda^{\rho,\rho'} \text{ arr}:\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha. \\ &\quad \text{if } i \geq j \text{ then} \\ &\quad \quad \text{arr} \\ &\quad \text{else} \\ &\quad \quad \text{walk (i + 1) (j - 1) (swap }\{\rho\} [\alpha] \text{ arr) i j) \\ &\quad 0 \text{ (n - 1).} \end{aligned}$$

3.6.4 Map

Η συνάρτηση *map* εφαρμόζει τη συνάρτηση *m* στα περιεχόμενα του πίνακα *arr*. Η συνάρτηση *map* είναι πολυμορφική ως προς τον τύπο των περιεχομένων του πίνακα. Ο πίνακας *arr*, το μέγεθος *n* του πίνακα και η συνάρτηση *m* δίδονται ως παράμετροι.

$$\begin{aligned} \text{map} &\triangleright \frac{\mathbf{U}\forall_s\rho.\mathbf{U}\forall_s\rho'.\mathbf{U}\forall_\tau\alpha.}{\perp(\mathbf{U}(\mathbf{U}(\rho\alpha\xrightarrow[\rho,\rho']{\perp}\rho\alpha)\xrightarrow[\rho,\rho']{\perp}\mathbf{U}(\mathbf{U}\mathbf{nat}\xrightarrow[\rho,\rho']{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha\xrightarrow[\rho,\rho']{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha)))} \\ &= \mathbf{U}\mathbf{poly} + : \rho. \mathbf{U}\mathbf{poly} + : \rho'. \mathbf{U}\mathbf{poly} \alpha : \\ &\quad \mathbf{U}\lambda^{\rho,\rho'} m:\mathbf{U}(\rho\alpha\xrightarrow[\rho]{\perp}\rho\alpha). \\ &\quad \mathbf{U}\lambda^{\rho,\rho'} n:\mathbf{U}\mathbf{nat}. \mathbf{U}\lambda^{\rho,\rho'} \text{ arr}:\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha. \\ &\quad \text{fix}(\mathbf{U}\lambda^{\rho,\rho'} \text{ walk}:\mathbf{U}(\mathbf{U}\mathbf{nat}\xrightarrow[\rho,\rho']{\perp}\mathbf{U}(\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha\xrightarrow[\rho,\rho']{\perp}\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha)). \\ &\quad \mathbf{U}\lambda^{\rho,\rho'} i:\mathbf{U}\mathbf{nat}. \mathbf{U}\lambda^{\rho,\rho'} x:\mathbf{L}\mathbf{array}_\rho^\mathbf{U}\alpha. \\ &\quad \text{if } i = n \text{ then} \\ &\quad \quad x \text{ else} \\ &\quad \quad \text{at } \rho'' \text{ let!}(x) y = \text{read } x \text{ } i \text{ then} \\ &\quad \quad \text{walk (i + 1) (write } x \text{ } i) \text{) 0 arr.} \end{aligned}$$

Κεφάλαιο 4

Περιγραφή της Γλώσσας Τύπων του Συστήματος NFlint

Το σύστημα NFlint είναι μια παραλλαγή του συστήματος Flint [Shao02] και ορίζει μια συναρτησιακή γλώσσα με ισχυρό σύστημα τύπων. Η NFlint χωρίζεται λογικά σε δύο επιμέρους γλώσσες: τη γλώσσα τύπων και την γλώσσα υπολογισμών.

Η γλώσσα τύπων στηρίζεται στον λογισμό των επαγωγικών κατασκευαστών (CIC) και σε αυτήν περιγράφονται οι τύποι των εκφράσεων της γλώσσας υπολογισμών. Κάθε πρόγραμμα εκφρασμένο στην γλώσσα υπολογισμών έχει και έναν αντίστοιχο τύπο, εκφρασμένο στην γλώσσα τύπων. Επιπλέον, η γλώσσα τύπων χρησιμοποιείται για τη περιγραφή ιδιοτήτων του προγράμματος και την απόδειξη της ισχύος τους.

4.1 Αναλυτική Σύνταξη

Η γραμματική της γλώσσας είναι οργανωμένη σε σε τέσσερα επίπεδα τύπων: **External**, **Kind schemata**, **Kinds** και **Types**

Πίνακας 4.1: Η γλώσσα NFlint.

External (σ)	Kind schemata (u)	Kinds (κ)	Types (τ)
Constants			
Type	Set		
Variables			
	z	k	t
Products			
$\Pi k:u. \sigma$	$\Pi z:\sigma. u$	$\Pi z:\sigma. \kappa$	
$\Pi t:\kappa. \sigma$	$\Pi k:u. u'$	$\Pi k:u. \kappa$	
	$\Pi t:\kappa. u$	$\Pi t:\kappa. \kappa'$	
Product introduction (abstraction)			
	$\lambda k:u. \sigma$	$\lambda z:\sigma. \kappa$	$\lambda z:\sigma. \tau$
	$\lambda t:\kappa. \sigma$	$\lambda k:u. \kappa$	$\lambda k:u. \tau$
		$\lambda t:\kappa. \kappa'$	$\lambda t:\kappa. \tau$
Product elimination (application)			
	$u \kappa$	κu	τu
	$u \tau$	$\kappa \tau$	$\tau \tau'$
Inductive definitions			
		$\text{Ind}(k:u)\{\vec{\kappa}\}$	$\text{Constr}(n, \kappa)$
		$\text{Elim}[u'](\tau:\kappa \vec{q})\{\vec{\kappa}\}$	$\text{Elim}[\kappa'](\tau:\kappa \vec{q})\{\vec{\tau}\}$

'Όπου q είναι κ ή τ και \vec{x} είναι ένα διάνυσμα (λίστα) από x .

4.2 Συνεπυγμένη Σύνταξη (PTS)

Η αφηρημένη γραμματική σύνταξη της γλώσσας των τύπων μπορεί να γραφεί και σε συνεπυγμένη BNF μορφή.

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

Το s αναπαριστά τις σταθερές, το X αναπαριστά τις μεταβλητές και τα A, B τους όρους της γλώσσας.

4.3 Απλοποιημένη Σύνταξη

Όταν μία συνάρτηση δεν είναι εξαρτημένη από τύπο, ο τύπος της μπορεί να γραφεί και ως:

$$A \rightarrow B \equiv \Pi X : A. B \quad \text{αν } X \notin \text{FV}(B)$$

4.4 Βοηθητικοί Κανόνες

4.4.1 Ελεύθερες Μεταβλητές

Οι ελεύθερες μεταβλητές εκφράσεων της γλώσσας τύπων μπορούν να βρεθούν ακολουθώντας τους παρακάτω κανόνες.

$$\begin{aligned} \text{FV}(s) &= \emptyset \\ \text{FV}(X) &= \{X\} \\ \text{FV}(\Pi X : A. B) &= \text{FV}(A) \cup (\text{FV}(B) - \{X\}) \\ \text{FV}(\lambda X : A. B) &= \text{FV}(A) \cup (\text{FV}(B) - \{X\}) \\ \text{FV}(A B) &= \text{FV}(A) \cup \text{FV}(B) \\ \text{FV}(\text{Ind}(X : A) \{\vec{A}\}) &= \text{FV}(A) \cup (\text{FV}(\vec{A}) - \{X\}) \\ \text{FV}(\text{Constr}(n, A)) &= \text{FV}(\vec{A}) \\ \text{FV}(\text{Elim}[A'](A : B \vec{B}) \{\vec{A}\}) &= \text{FV}(A') \cup \text{FV}(A) \cup \text{FV}(B) \cup \text{FV}(\vec{B}) \cup \text{FV}(\vec{A}) \\ \\ \text{FV}(\epsilon) &= \emptyset \\ \text{FV}(A; \vec{A}) &= \text{FV}(A) \cup \text{FV}(\vec{A}) \end{aligned}$$

4.4.2 Αντικαταστάσεις

Παρατίθενται οι αναδρομικοί κανόνες για την πράξη της αντικατάστασης στην γλώσσα τύπων τουNFLint.

$$\begin{aligned} s\{\vec{Z} \mapsto \vec{R}\} &\equiv s \\ X\{\vec{Z} \mapsto \vec{R}\} &\equiv \begin{cases} X & \text{αν } X \neq Z_i, \text{ για κάθε } i \\ R_i & \text{αν } X = Z_i, \text{ για κάποιο } i \end{cases} \\ (\Pi X : A. B)\{\vec{Z} \mapsto \vec{R}\} &\equiv \begin{cases} \Pi X : A\{\vec{Z} \mapsto \vec{R}\}. B\{\vec{Z} \mapsto \vec{R}\} \\ , \text{ περίπτωση (i)} \\ \Pi Y : A\{\vec{Z} \mapsto \vec{R}\}. B\{X \mapsto Y\}\{\vec{Z} \mapsto \vec{R}\} \\ , \text{ περίπτωση (ii)} \end{cases} \end{aligned}$$

$$\begin{aligned}
(\lambda X:A. B)\{\vec{Z} \mapsto \vec{R}\} &\equiv \begin{cases} \lambda X:A\{\vec{Z} \mapsto \vec{R}\}. B\{\vec{Z} \mapsto \vec{R}\} \\ , \text{ περίπτωση (i)} \\ \lambda Y:A\{\vec{Z} \mapsto \vec{R}\}. B\{X \mapsto Y\}\{\vec{Z} \mapsto \vec{R}\} \\ , \text{ περίπτωση (ii)} \end{cases} \\
(A B)\{\vec{Z} \mapsto \vec{R}\} &\equiv A\{\vec{Z} \mapsto \vec{R}\} B\{\vec{Z} \mapsto \vec{R}\} \\
(\text{Ind}(X:A)\{\vec{A}\})\{\vec{Z} \mapsto \vec{R}\} &\equiv \begin{cases} \text{Ind}(X:A\{\vec{Z} \mapsto \vec{R}\})\{\vec{A}\{\vec{Z} \mapsto \vec{R}\}\} \\ , \text{ περίπτωση (i)} \\ \text{Ind}(Y:A\{\vec{Z} \mapsto \vec{R}\})\{\vec{A}\{X \mapsto Y\}\{\vec{Z} \mapsto \vec{R}\}\} \\ , \text{ περίπτωση (ii)} \end{cases} \\
(\text{Constr}(n, A))\{\vec{Z} \mapsto \vec{R}\} &\equiv \text{Constr}(n, A\{\vec{Z} \mapsto \vec{R}\}) \\
(\text{Elim}[A'](A:B\vec{B})\{\vec{A}\})\{\vec{Z} \mapsto \vec{R}\} &\equiv \text{Elim}[A'\{\vec{Z} \mapsto \vec{R}\}] \\
&\quad (A\{\vec{Z} \mapsto \vec{R}\}:B\{\vec{Z} \mapsto \vec{R}\}\vec{B}\{\vec{Z} \mapsto \vec{R}\}) \\
&\quad \{\vec{A}\{\vec{Z} \mapsto \vec{R}\}\}
\end{aligned}$$

όπου: “περίπτωση (i)”

σημαίνει αν $X \neq Z_i$, για κάθε i , και $X \notin \text{FV}(\vec{R})$

“περίπτωση (ii)”

σημαίνει αν $X = Z_i$, για κάποιο i , ή $X \in \text{FV}(\vec{R})$

Y είναι μία νέα μεταβλητή

$$\begin{array}{rcl}
\epsilon\{\vec{Z} \mapsto \vec{R}\} & \equiv & \epsilon \\
(A; \vec{A})\{Z \mapsto R\} & \equiv & A\{\vec{Z} \mapsto \vec{R}\}; \vec{A}\{\vec{Z} \mapsto \vec{R}\}
\end{array}$$

4.4.3 Θετικές Εμφανίσεις

Λέμε ότι το X έχει θετική εμφάνιση (positive occurrence) μέσα στο $\Pi \vec{Y}:\vec{A}. X \vec{B}$ και γράφουμε ότι $\text{pos}_X(\Pi \vec{Y}:\vec{A}. X \vec{B})$ όταν ισχύει ο κανόνας:

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

4.4.4 Καλά Ορισμένοι Κατασκευαστές

Ένας κατασκευαστής (constructor) ονομάζεται καλά ορισμένος (well-founded) για έναν όρο A και συμβολίζεται $\text{wfc}_X(A)$ όταν ισχύουν:

$$\begin{array}{c}
\frac{X \notin \text{FV}(A_i) \quad (\text{για κάθε } i)}{\text{wfc}_X(X \vec{A})} \\
\\
\frac{X \neq X' \quad X \notin \text{FV}(A') \quad \text{wfc}_X(B)}{\text{wfc}_X(\Pi X':A'. B)} \quad \frac{\text{pos}_X(A) \quad \text{wfc}_X(B)}{\text{wfc}_X(A \rightarrow B)}
\end{array}$$

4.4.5 Μικροί Τύποι Κατασκευαστών

Οι τύποι λέγονται μικρών κατασκευαστών (small constructor types) όταν όλοι οι κατασκευαστές τους εξαρτώνται μόνο από **Kinds**. Δηλαδή όταν όλες οι παράμετροι τους έχουν τύπο **Set**.

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

4.4.6 Εφαρμογή της Επαγωγής στους Επαγωγικούς Ορισμούς

Ο τύπος της εφαρμογής της επαγωγής (κανόνας **Elim**) δίνεται από τον τελεστή Ψ_X και ορίζεται αναδρομικά από τους κανόνες:

$$\begin{aligned}\Psi_X(X \vec{A}', A, B) &\equiv A \vec{A}' B \\ \Psi_X(\Pi X': A'. B', A, B) &\equiv \Pi X': A'. \Psi_X(B', A, B X') \\ \Psi_X((\Pi \vec{X}': \vec{A}'. X \vec{B}') \rightarrow B', A, B) &\equiv \Pi Y: (\Pi \vec{X}': \vec{A}'. X \vec{B}'). \Pi \vec{X}': \vec{A}' \\ &\quad A \vec{B}' (Y \vec{X}') \rightarrow \Psi_X(B', A, B Y)\end{aligned}$$

Η ίδια η εφαρμογή της επαγωγής συμβολίζεται με τον τελεστή Φ_X και γίνεται σύμφωνα με τους παρακάτω κανόνες:

$$\begin{aligned}\Phi_X(X \vec{A}', A, B) &\equiv A \\ \Phi_X(\Pi X': A'. B', A, B) &\equiv \lambda X': A'. \Phi_X(B', A X', B) \\ \Phi_X((\Pi \vec{X}': \vec{A}'. X \vec{B}') \rightarrow B', A, B) &\equiv \lambda Y: (\Pi \vec{X}': \vec{A}'. X \vec{B}'). \\ &\quad \Phi_X(B', A Y (\lambda \vec{X}': \vec{A}'. B \vec{B}' (Y \vec{X}'))), B)\end{aligned}$$

4.5 Λειτουργική Σημασιολογία

Έστω ότι \sim είναι μια σχέση πάνω σε ένα σύνολο όρων της γλώσσας τύπων. Τότε η σχέση \sim^* πάνω στο σύνολο των διανυσμάτων των όρων της γλώσσας τύπων μπορεί να ορισθεί σαν τη μικρότερη σχέση η οποία ικανοποιεί τις ακόλουθες ιδιότητες για όλους τους όρους A, A' και για όλα τα διανύσματα όρων \vec{B}, \vec{B}' της γλώσσας τύπων:

$$\begin{aligned}A \sim A' &\Rightarrow A; \vec{B} \sim^* A'; \vec{B} \\ \vec{B} \sim \vec{B}' &\Rightarrow A; \vec{B} \sim^* A; \vec{B}'\end{aligned}$$

Μια σχέση \sim πάνω σε ένα σύνολο από όρους της γλώσσας τύπων ονομάζεται συμβατή (compatible) αν για κάθε μεταβλητή τύπου X , για κάθε όρο A, B, A', B', Q, Q' της γλώσσας τύπων και για κάθε διάνυσμα όρων \vec{A}, \vec{B} της γλώσσας τύπων ικανοποιούνται οι ακόλουθοι κανόνες:

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

4.5.1 Αναγωγές

Έστω $\rightarrow_\beta, \rightarrow_\eta$ και \rightarrow_ι , οι μικρότερες συμβατές σχέσεις που ικανοποιούν τους κανόνες:

$$\begin{aligned}
& (\lambda X : A. B) A' \rightarrow_{\beta} B\{X \mapsto A'\} \\
& \lambda X : A. B X \rightarrow_{\eta} B \\
& \quad \text{αν } X \notin \text{FV}(B) \\
& \text{Elim}[A'](\text{Constr}(i, B) \vec{A}' : B \vec{B}) \{\vec{A}\} \rightarrow_{\iota} \\
& \quad \Phi_X(A'_i, A_i, \lambda \vec{X} : \vec{B}'. \lambda Y : B \vec{X}. \text{Elim}[A'](Y : B \vec{X}) \{\vec{A}\}) \vec{A}' \\
& \text{όπου } B = \text{Ind}(X : B') \{\vec{A}'\} \text{ και } B' = \Pi \vec{X} : \vec{B}'. \text{Set}
\end{aligned}$$

$\Omega \varsigma \rightarrow_{\chi}$ ορίζεται το ανακλαστικό και μεταβατικό κλείσιμο του \rightarrow_{χ} , για $\chi \in \{\beta, \eta, \iota\}$. Έστω $\rightarrow_{\beta\eta\iota}$ η ένωση των $\rightarrow_{\beta}, \rightarrow_{\eta}$ και \rightarrow_{ι} . Ορίζεται ο τελεστής $\rightarrow_{\beta\eta\iota}$ ως το ανακλαστικό και μεταβατικό κλείσιμο του $\rightarrow_{\beta\eta\iota}$. Έστω \rightarrow και \Rightarrow οι συντμήσεις των $\rightarrow_{\beta\eta\iota}$ και $\rightarrow_{\beta\eta\iota}$ αντίστοιχα.

4.5.2 Ισότητες

Έστω $=_{\chi}$ η σχέση ισότητας που παράγεται από το \rightarrow_{χ} , για $\chi \in \{\beta, \eta, \iota\}$. Ορίζεται ο τελεστής $=_{\beta\eta\iota}$, οποίος αναγράφεται και ως $=$, ως η σχέση ισότητας που παράγεται από το \rightarrow .

4.6 Κανόνες Τύπων

Παρακάτω αναγράφονται οι κανόνες που προδιαγράφουν τους τύπους των διαφόρων όρων της γλώσσας τύπων.

4.6.1 Μεταβλητές και Σταθερές

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

4.6.2 Μετατροπή

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

4.6.3 Γινόμενα, Αφαίρεση και Εφαρμογή

$$\begin{array}{c}
\frac{\Delta \vdash A : s_1 \quad \Delta, X : A \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Delta \vdash \Pi X : A. B : s_2} \\
\\
\frac{\Delta, X : A \vdash B : B' \quad \Delta \vdash \Pi X : A. B' : s}{\Delta \vdash \lambda X : A. B : \Pi X : A. B'} \qquad \frac{\Delta \vdash A : \Pi X : B'. A' \quad \Delta \vdash B : B'}{\Delta \vdash AB : A'\{X \mapsto B\}}
\end{array}$$

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

4.6.4 Επαγωγικοί Ορισμοί

$$\begin{array}{c}
 \frac{\Delta \vdash A : \text{Type} \quad A = \Pi \vec{X} : \vec{B}. \text{Set}}{\Delta, X : A \vdash A_i : \text{Set} \quad \text{wfc}_X(A_i) \quad (\gamma\text{ια κάθε } i)} \quad \frac{A = \text{Ind}(X : A')\{\vec{A}\} \quad \Delta \vdash A : A'}{\Delta \vdash \text{Constr}(n, A) : A_n\{X \mapsto A\}}
 \\[10pt]
 \frac{B = \text{Ind}(X : B')\{\vec{A}'\} \quad B' = \Pi \vec{X} : \vec{B}'. \text{Set}}{\Delta \vdash \vec{B} : (\vec{X} : \vec{B}')} \quad \frac{\Delta \vdash A : B \vec{B} \quad \Delta \vdash A' : \Pi \vec{X} : \vec{B}'. B \vec{X} \rightarrow \text{Type}}{\Delta \vdash A_i : \Psi_X(A'_i, A', \text{Constr}(i, B)) \quad (\gamma\text{ια κάθε } i)} \quad \frac{\Delta \vdash \text{Elim}[A'](A : B \vec{B})\{\vec{A}\} : A' \vec{B} A}{\Delta \vdash \text{Elim}[A'](A : B \vec{B})\{\vec{A}\} : A' \vec{B} A}
 \\[10pt]
 \frac{B = \text{Ind}(X : B')\{\vec{A}'\} \quad B' = \Pi \vec{X} : \vec{B}'. \text{Set}}{\Delta \vdash \vec{B} : (\vec{X} : \vec{B}')} \quad \frac{\Delta \vdash A : B \vec{B} \quad \Delta \vdash A' : \Pi \vec{X} : \vec{B}'. B \vec{X} \rightarrow \text{Type}}{\Delta \vdash A_i : \Psi_X(A'_i, A', \text{Constr}(i, B)) \quad (\gamma\text{ια κάθε } i)} \quad \frac{\Delta, X : B' \vdash \text{small}_X(A'_i) \quad (\gamma\text{ια κάθε } i)}{\Delta \vdash \text{Elim}[A'](A : B \vec{B})\{\vec{A}\} : A' \vec{B} A}
 \end{array}$$

4.6.5 Πολλαπλές Εκφράσεις, Εξάρτηση

$$\frac{}{\Delta \vdash \epsilon : (\epsilon : \epsilon)} \quad \frac{\Delta \vdash A : B \quad \Delta \vdash \vec{A} : (\vec{X} : \vec{B}\{X \mapsto A\})}{\Delta \vdash A; \vec{A} : (X; \vec{X} : B; \vec{B})}$$

4.7 Παραγόμενες Θεωρίες

Με βάση την θεωρία και τους κανόνες που αναφέρθηκαν μπορούν να ορισθούν τα παρακάτω στοιχεία της γλώσσας τύπων, τα οποία αποτελούν μια βάση για την υλοποίηση των απαραίτητων τύπων που χρησιμοποιούνται στην ενδιάμεση γλώσσα.

4.7.1 Τιμές Αληθείας

$$\begin{array}{ll}
 \text{Bool} & : \text{Set} \\
 & \equiv \text{Ind}(X : \text{Set})\{X; X\} \\
 \text{true} & : \text{Bool} \\
 & \equiv \text{Constr}(0, \text{Bool}) \\
 \text{false} & : \text{Bool} \\
 & \equiv \text{Constr}(1, \text{Bool}) \\
 \\
 \text{not} & : \text{Bool} \rightarrow \text{Bool} \\
 & \equiv \lambda b : \text{Bool}. \text{Elim}[\lambda b : \text{Bool}. \text{Bool}](b : \text{Bool})\{\text{false}; \text{true}\} \\
 \text{and} & : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
 & \equiv \lambda b_1 : \text{Bool}. \text{Elim}[\lambda b_1 : \text{Bool}. \text{Bool} \rightarrow \text{Bool}](b_1 : \text{Bool})\{\lambda b_2 : \text{Bool}. b_2; \lambda b_2 : \text{Bool}. \text{false}\} \\
 \text{or} & : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
 & \equiv \lambda b_1 : \text{Bool}. \text{Elim}[\lambda b_1 : \text{Bool}. \text{Bool} \rightarrow \text{Bool}](b_1 : \text{Bool})\{\lambda b_2 : \text{Bool}. \text{true}; \lambda b_2 : \text{Bool}. b_2\} \\
 \text{Cond}_{\text{Set}} & : \Pi k : \text{Set}. \text{Bool} \rightarrow k \rightarrow k \rightarrow k \\
 & \equiv \lambda k : \text{Set}. \lambda b : \text{Bool}. \lambda x : k. \lambda y : k. \text{Elim}[\lambda b : \text{Bool}. k](b : \text{Bool})\{x; y\} \\
 \text{Cond}_{\text{Type}} & : \Pi z : \text{Type}. \text{Bool} \rightarrow z \rightarrow z \rightarrow z \\
 & \equiv \lambda z : \text{Type}. \lambda b : \text{Bool}. \lambda x : z. \lambda y : z. \text{Elim}[\lambda b : \text{Bool}. z](b : \text{Bool})\{x; y\}
 \end{array}$$

4.7.2 Θετικοί Φυσικοί Αριθμοί

$$\begin{array}{ll}
 \text{PNat} & : \text{Set} \\
 & \equiv \text{Ind}(X : \text{Set})\{X, X \rightarrow X, X \rightarrow X\}
 \end{array}$$

```

pnat1    : PNat
         ≡ Constr(0, PNat)
pnat2x   : PNat → PNat
         ≡ Constr(1, PNat)
pnat2x1  : PNat → PNat
         ≡ Constr(2, PNat)

```

4.7.3 Φυσικοί Αριθμοί

```

Nat     : Set
         ≡ Ind(X:Set){X, PNat → X}
nat0   : Nat
         ≡ Constr(0, Nat)
natP   : PNat → Nat
         ≡ Constr(1, Nat)

```

n : Nat αναπαράσταση του φυσικού αριθμού $n \in \mathbb{N}$

$+_{\text{Nat}}, \ominus_{\text{Nat}}, *_{\text{Nat}}$	$: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$
$\text{eq}_{\text{Nat}}, \text{ne}_{\text{Nat}}, \text{lt}_{\text{Nat}}, \text{gt}_{\text{Nat}}, \text{le}_{\text{Nat}}, \text{ge}_{\text{Nat}}$	$: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool}$
$=_{\text{Nat}}, \neq_{\text{Nat}}, <_{\text{Nat}}, >_{\text{Nat}}, \leq_{\text{Nat}}, \geq_{\text{Nat}}$	$: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Set}$

4.7.4 Ακέραιοι Αριθμοί

```

Int     : Set
         ≡ Ind(X:Set){X, PNat → X, PNat → X}
int0   : Int
         ≡ Constr(0, Int)
intP   : PNat → Int
         ≡ Constr(1, Int)
intN   : PNat → Int
         ≡ Constr(2, Int)

```

\bar{n} : Int αναπαράσταση του ακέραιου αριθμού $n \in \mathbb{Z}$

neg_{Int}	$: \text{Int} \rightarrow \text{Int}$
$+_{\text{Int}}, -_{\text{Int}}, *_{\text{Int}}, /_{\text{Int}}, \%_{\text{Int}}$	$: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
$\text{eq}_{\text{Int}}, \text{ne}_{\text{Int}}, \text{lt}_{\text{Int}}, \text{gt}_{\text{Int}}, \text{le}_{\text{Int}}, \text{ge}_{\text{Int}}$	$: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$
$=_{\text{Int}}, \neq_{\text{Int}}, <_{\text{Int}}, >_{\text{Int}}, \leq_{\text{Int}}, \geq_{\text{Int}}$	$: \text{Int} \rightarrow \text{Int} \rightarrow \text{Set}$

4.7.5 Προτασιακός Λογισμός

```

True    : Set
False   : Set
truePrf : True
¬       : Set → Set
∧       : Set → Set → Set
∨       : Set → Set → Set

```

4.7.6 Λίστες

$\text{List} : \text{Set} \rightarrow \text{Set} \equiv \lambda A : \text{Set}. \text{Ind}(X : \text{Set}) \{ X, A \rightarrow X \rightarrow X \}$
 $\text{nil} : \prod k : \text{Set}. \text{List } k \equiv \lambda A : \text{Set}. \text{Constr}(0, \text{List } A)$
 $\text{cons} : \prod k : \text{Set}. k \rightarrow \text{List } k \rightarrow \text{List } k \equiv \lambda A : \text{Set}. \text{Constr}(1, \text{List } A)$
 $\text{nth} : \prod k : \text{Set}. k \rightarrow \text{List } k \rightarrow \text{Nat} \rightarrow k$
 $\text{repeat} : \prod k : \text{Set}. \text{Nat} \rightarrow k \rightarrow \text{List } k$

ή με μήκος:

$\text{List} : \text{Set} \rightarrow \text{Nat} \rightarrow \text{Set}$
 $\text{nil} : \prod k : \text{Set}. \text{List } k 0$
 $\text{cons} : \prod k : \text{Set}. \prod n : \text{Nat}. k \rightarrow \text{List } k n \rightarrow \text{List } k(n+1)$
 $\text{nth} : \prod k : \text{Set}. \prod n : \text{Nat}. \text{List } k n \rightarrow \prod i : \text{Nat}. (i <_{\text{Nat}} n) \rightarrow k$
 $\text{repeat} : \prod k : \text{Set}. \prod n : \text{Nat}. k \rightarrow \text{List } k n$

4.7.7 Χαρακτήρες

$\text{Char} : \text{Set}$

4.8 Μεταθεωρία της Γλώσσας Τύπων

Τα παρακάτω θεωρήματα μπορούν να αποδειχθούν για τη γλώσσα των τύπων. Οι αποδείξεις τους για το λογισμό των επαγγειακών 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 αποχρισμότητα του ελέγχου τύπων και η μοναδικότητα των τύπων είναι απαραίτητες προϋποθέσεις για τον καλό ορισμό του συστήματος τύπων του NFlint. Προφανώς, οι τύποι είναι μοναδικοί ως προς $=_{\alpha\beta\eta}$.

Θεώρημα 4 (Αποχρισμότητα 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 5 (Μοναδικότητα των Τύπων): $A \nu \Delta \vdash A : B$ 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 6 (Συμβολή): *Εστω $\Delta \vdash A : B$. $A \nu A \twoheadrightarrow A_1$ και $A \twoheadrightarrow A_2$, τότε υπάρχει ένας όρος A' τέτοιος ώστε $A_1 \twoheadrightarrow A'$ και $A_2 \twoheadrightarrow A'$.*

Θεώρημα 7 (Church-Rosser): *Εστω $\Delta \vdash A_1 : B$ και $\Delta \vdash A_2 : B$. $A \nu A_1 = A_2$, τότε υπάρχει ένας όρος A' τέτοιος ώστε $A_1 \twoheadrightarrow A'$ και $A_2 \twoheadrightarrow A'$.*

Θεώρημα 8 (Ορθότητα των Αναγωγών): $A \nu \Delta \vdash A : B$ και $A \twoheadrightarrow A'$, τότε $\Delta \vdash A' : B$.

Ένας όρος A λέγεται κανονική μορφή αν δεν υπάρχει όρος A' τέτοιος ώστε $A \rightarrow A'$. Ένας όρος A λέγεται κανονικοποιησμός αν υπάρχει κανονική μορφή B τέτοια ώστε $A \Rightarrow B$. Ένας όρος A λέγεται ισχυρά κανονικοποιησμός αν είναι κανονικοποιησμός και δεν υπάρχει άπειρη ακολουθία \rightarrow -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 9 (Ισχυρή Κανονικοποίηση): $\text{Αν } \Delta \vdash A : B, \text{ τότε ο όρος } A \text{ είναι ισχυρά κανονικοποιησμός.}$

Κεφάλαιο 5

Περιγραφή και Ορισμός του Συστήματος linFlint

Στα συστήματα προγραμματισμού με αποδείξεις χρησιμοποιούνται ισχυρά συστήματα τύπων στα οποία μπορούν να περιγραφούν ιδιότητες των προγραμμάτων. Τα παραδοσιακά συστήματα τύπων δεν μπορούν να περιγράψουν την κατάσταση της μνήμης και τις μεταβολές της. Η αδυναμία τους αυτή αποτρέπει από το να χρησιμοποιηθούν οι ιδέες τους ως βάση για ένα σύστημα προγραμματισμού με αποδείξεις που υποστηρίζει references και destructive update.

Η έρευνα, όμως, γύρω από τα substructural συστήματα τύπων έδειξε ότι μπορούν να χρησιμοποιηθούν για την περιγραφή της κατάστασης της μνήμης και των μεταβολών της με τυποθεωρητικά ασφαλή τρόπο.

Από το συνδυασμό των ιδεών του προγραμματισμού με αποδείξεις και των substructural συστημάτων τύπων μπορεί να προκύψει σύστημα προγραμματισμού με αποδείξεις που υποστηρίζει references και destructive update.

Για να μελετήσουμε τη δυνατότητα κατασκευής ενός τέτοιου συστήματος ορίσαμε το σύστημα linFlint. Το σύστημα linFlint διαθέτει μία πολύ εκφραστική γλώσσα τύπων στα πρότυπα του προγραμματισμού με αποδείξεις. Παράλληλα οι τύποι των όρων της γλώσσας υπολογισμού συνιστούν ένα πλήρες substructural σύστημα τύπων, με linear, affine, relevant και unrestricted τύπους, κωδικοποιημένο στη γλώσσα τύπων του συστήματος. Το σύστημα υποστηρίζει ρητή διαχείριση της μνήμης σε επίπεδο regions και πρόσβαση στα περιεχόμενα της με δείκτες σε πλειάδες δεδομένων. Τα περιεχόμενα της μνήμης μπορούν να μεταβληθούν μέσω ενός συνόλου εντολών ανάθεσης. Η γλώσσα τύπων του συστήματος επιτρέπει τη διατύπωση πολύπλοκων λογικών προτάσεων που περιγράφουν τις ιδιότητες προγραμμάτων με references και destructive update και μπορούν να χρησιμοποιηθούν για την παραγωγή πιστοποιημένου κώδικα.

5.1 Η Γλώσσα Τύπων του Συστήματος linFlint

Το σύστημα linFlint χρησιμοποιεί ως γλώσσα τύπων μία παραλλαγή της γλώσσας τύπων του συστήματοςNFLint. Η μόνη διαφορά έγκειται στο ότι η γλώσσα τύπων του συστήματος linFlint υποστηρίζει επιπλέον αμοιβαία επαγωγικούς ορισμούς.

Η προσθήκη δεν επηρεάζει τις ιδιότητες και τη μεταθεωρία της γλώσσας τύπων καθώς οι αμοιβαία επαγωγικοί ορισμοί υποστηρίζονται από το λογισμό των επαγωγικών κατασκευών όπως χρησιμοποιείται από το σύστημα Coq. Οι αποδείξεις της μεταθεωρίας του λογισμού των επαγωγικών κατασκευών περιέχονται στη διδακτορική διατριβή του Benjamin Werner [Wern94]. Η προσαρμογή των αποδείξεων για την παραλλαγή της γλώσσας τύπων του συστήματος NFLint που χρησιμοποιεί το σύστημα linFlint μπορεί να γίνει με τρόπο παρόμοιο με τη διαδικασία που ακολουθείται στην εργασία των Shao *et al.* [Shao02].

5.2 Η Γλώσσα Υπολογισμών του Συστήματος linFlint

Η γλώσσα υπολογισμών του συστήματος linFlint μπορεί να χρησιμοποιηθεί σαν ενδιάμεση γλώσσα ενός μεταγλωττιστή που παράγει πιστοποιημένο κώδικα.

5.2.1 Ορισμός Βοηθητικών Τύπων της Γλώσσας Υπολογισμών

Οι τύποι των εκφράσεων της γλώσσας υπολογισμών συνιστούν ένα πλήρες substructural σύστημα τύπων. Για την κωδικοποίηση του στη γλώσσα τύπων του συστήματος απαιτείται ο ορισμός βοηθητικών τύπων. Επίσης, ορίζονται βοηθητικοί τύποι για την αναπαράσταση της μνήμης στη γλώσσα τύπων.

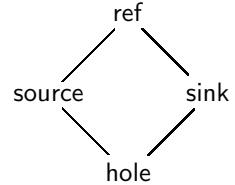
- Το Set FMode ελέγχει τον τρόπο πρόσβασης σε μία περιοχή της μνήμης.

$$\begin{array}{ll} \text{FMode} & : \text{Set} \\ \text{source, sink, ref, hole} & : \text{FMode} \end{array}$$

Οι χαρακτηρισμοί sink και source δήλωναν ότι μία περιοχή μνήμης μπορεί να χρησιμοποιηθεί μόνο για εγγραφή ή ανάγνωση αντίστοιχα. Ο χαρακτηρισμός ref επιτρέπει κάθε είδους πρόσβαση ενώ ο χαρακτηρισμός hole δεν δίνει τη δυνατότητα για οποιαδήποτε πρόσβαση στην περιοχή μνήμης.

Τα αντικείμενα του Set FMode σχετίζονται με μία τυποθεωρητική σχέση ιεραρχίας (*subtyping*).

$$\begin{array}{l} \text{source} \preceq \text{ref} \\ \text{sink} \preceq \text{ref} \\ \text{hole} \preceq \text{source} \\ \text{hole} \preceq \text{sink} \end{array}$$

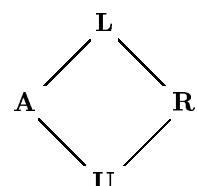


- Το Set Qual αντιστοιχεί στο σύνολο των qualifiers όπως είναι γνωστό από τα substructural συστήματα τύπων. Διατηρούνται οι συνήθεις ερμηνείες των συμβόλων.

$$\begin{array}{ll} \text{Qual} & : \text{Set} \\ \mathbf{U}, \mathbf{L}, \mathbf{A}, \mathbf{R} & : \text{Qual} \end{array}$$

Διατηρείται, επίσης, η σχέση ιεραρχίας των qualifiers των substructural συστημάτων τύπων.

$$\begin{array}{l} \mathbf{A} \preceq \mathbf{L} \\ \mathbf{R} \preceq \mathbf{L} \\ \mathbf{U} \preceq \mathbf{A} \\ \mathbf{U} \preceq \mathbf{R} \end{array}$$



- Το μη προσδιορισμένο Set Sc κατά αναλογία με τα scopes της γλώσσας let!.
- Το μη προσδιορισμένο Set Rgn των Regions αποτελεί την αναπαράσταση στη γλώσσα τύπων των regions που αποτελούν τη βάση διαχείρισης της μνήμης .

$$\text{Rgn} : \text{Set}$$

- Το Set Loc των Locations αποτελεί την αναπαράσταση στη γλώσσα τύπων των θέσεων της μνήμης.

$$\text{Loc} : \text{Rgn} \rightarrow \text{Set}$$

$$\text{subloc} : \Pi \rho : \text{Rgn}. \text{Loc} \rho \rightarrow \text{Nat} \rightarrow (\text{Nat} \rightarrow \Omega \times \text{FMode}) \rightarrow \text{Loc} \rho$$

- To Set Ξ των unqualified capabilities. Υπάρχουν δύο κατασκευαστές Ξ . Ο ένας αντιστοιχεί σε regions και ο άλλος σε πλειάδες.

$$\begin{aligned}\Xi &: \text{Set} \\ \text{rgn} &: \text{Rgn} \rightarrow \Xi \\ \text{typ} &: \prod_{\rho:\text{Rgn}} \text{Loc } \rho \rightarrow \text{Nat} \rightarrow (\text{Nat} \rightarrow \Omega \times \text{FMode}) \rightarrow \Xi\end{aligned}$$

- To Set C των capabilities. Τα capabilities χρησιμοποιούνται για τον έλεγχο της χρήσης των αναφορών σε regions και locations και σχηματίζονται από το Set C με την προσθήκη κατάλληλων qualifiers.

$$\begin{aligned}C &: \text{Set} \\ \text{qualc} &: \text{Qual} \rightarrow \text{Sc} \rightarrow \Xi \rightarrow C\end{aligned}$$

5.2.2 Απλοποιημένη Σύνταξη για τους Βοηθητικούς Τύπους

Απλοποιημένη σύνταξη για unqualified Capabilities:

$$l @ \rho \rightsquigarrow \langle \mu_0 \tau_0, \dots, \mu_{p-1} \tau_{p-1} \rangle \equiv \begin{array}{r} \text{typ } \rho \ l \ p \\ (\text{nth}(\Omega \times \text{FMode}) ((^u \text{unit}, \text{hole})) (\text{cons}(\Omega \times \text{FMode}) \langle \ \mu_0, \tau_0 \ \rangle \dots \\ (\text{cons}(\Omega \times \text{FMode}) \langle \ \mu_{p-1}, \tau_{p-1} \ \rangle \\ (\text{nil}(\Omega \times \text{FMode})) \dots)) \end{array} \quad \begin{array}{l} \rho : \text{Rgn}, l : \text{Loc} \\ \mu_0, \dots, \mu_{p-1} : \text{FMode} \\ \tau_0, \dots, \tau_{p-1} : \Omega \end{array}$$

Απλοποιημένη σύνταξη για Capabilities:

$$\frac{q \xi \equiv \text{qualc } q \eta \xi}{q : \text{Qual}, \eta : \text{Sc}, \xi : \Xi}$$

5.2.3 Pretypes της Γλώσσας Υπολογισμών

Οι τύποι των εκφράσεων της γλώσσας υπολογισμών αποτελούνται από τρία τμήματα, τους pretypes και τους qualifiers και τα scopes. Ο συνδυασμός και των τριών στοιχείων οδηγεί στο σχηματισμό των πλήρη τύπων.

To Set Φ των pretypes της γλώσσας υπολογισμού ορίζεται επαγωγικά στη γλώσσα τύπων.

$$\begin{aligned}\Phi &: \text{Set} \\ \text{unit} &: \Phi \\ \text{snat} &: \text{Nat} \rightarrow \Phi \\ \text{sint} &: \text{Int} \rightarrow \Phi \\ \text{schar} &: \text{Char} \rightarrow \Phi \\ \text{sbool} &: \text{Bool} \rightarrow \Phi \\ \text{arrow} &: \Omega \rightarrow \text{List Sc} \rightarrow \Omega \rightarrow \Phi \\ \text{tuple} &: \prod_{\rho:\text{Rgn}} \text{Loc } \rho \rightarrow \Phi \\ \text{forall}_{\text{Set}} &: \prod A:\text{Set}. (A \rightarrow \Omega) \rightarrow \Phi \\ \text{exists}_{\text{Set}} &: \prod A:\text{Set}. (A \rightarrow \Omega) \rightarrow \Phi \\ \text{forall}_{\text{Type}} &: \prod A:\text{Type}. (A \rightarrow \Omega) \rightarrow \Phi \\ \text{exists}_{\text{Type}} &: \prod A:\text{Type}. (A \rightarrow \Omega) \rightarrow \Phi \\ \text{forall}_{\text{C}} &: \text{C} \rightarrow \Omega \rightarrow \Phi \\ \text{exists}_{\text{C}} &: \text{C} \rightarrow \Omega \rightarrow \Phi \\ \text{forall}_{\text{Sc}} &: (\text{Sc} \rightarrow \Omega) \rightarrow \Phi \\ \text{handle} &: \text{Rgn} \rightarrow \Phi\end{aligned}$$

5.2.4 Τύποι των Εκφράσεων της Γλώσσας Υπολογισμών

To Set Ω των τύπων της γλώσσας υπολογισμών ορίζεται σαν Set των qualified pretypes.

$$\begin{aligned}\Omega &: \text{Set} \\ \text{qual} &: \text{Qual} \rightarrow \Phi \rightarrow \Omega\end{aligned}$$

5.2.5 Απλοποιημένη Σύνταξη για τους Pretypes της Γλώσσας Υπολογισμών

$$\begin{array}{lll}\tau_1 \xrightarrow[\vec{\eta}]{} \tau_2 & \equiv & \text{arrow } \tau_1 \vec{\eta} \tau_2 & \tau_1, \tau_2 : \Omega, \eta : \text{Sc} \\ \forall_{\text{Set}} t : \kappa. \tau & \equiv & \text{forall}_{\text{Set}} \kappa (\lambda t : \kappa. \tau) & \kappa : \text{Set}, \tau : \Omega \\ \exists_{\text{Set}} t : \kappa. \tau & \equiv & \text{exists}_{\text{Set}} \kappa (\lambda t : \kappa. \tau) & \kappa : \text{Set}, \tau : \Omega \\ \forall_{\text{Type}} k : u. \tau & \equiv & \text{forall}_{\text{Type}} u (\lambda k : u. \tau) & u : \text{Type}, \tau : \Omega \\ \exists_{\text{Type}} k : u. \tau & \equiv & \text{exists}_{\text{Type}} u (\lambda k : u. \tau) & u : \text{Type}, \tau : \Omega \\ \forall_C c. \tau & \equiv & \text{forall}_C c \tau & c : C, \tau : \Omega \\ \exists_C c. \tau & \equiv & \text{exists}_C c \tau & c : C, \tau : \Omega \\ \forall_{\text{Sc}} \eta. \tau & \equiv & \text{forall}_{\text{Sc}} (\lambda \eta : \text{Sc}. \tau) & \tau : \Omega, \eta : \text{Sc}\end{array}$$

5.2.6 Απλοποιημένη Σύνταξη για τους Τύπους της Γλώσσας Υπολογισμών

$$\begin{array}{lll}\tau_1 \xrightarrow[\vec{\eta}]{}^q \tau_2 & \equiv & {}^q(\tau_1 \xrightarrow[\vec{\eta}]{} \tau_2) & \tau_1, \tau_2 : \Omega, q : \text{Qual}, \eta : \text{Sc} \\ {}^q \phi & \equiv & \text{qual } q \phi & q : \text{Qual}, \phi : \Phi\end{array}$$

5.2.7 Η Σύνταξη της Γλώσσας

$$\begin{array}{ll}e ::= & x \mid \text{cnat}[n] \mid \text{cint}[i] \mid \text{cchar}[c] \mid \text{cbool}[b] \mid op \mid op\ e \mid e_1\ op\ e_2 \\ & \mid {}^q\text{lambda}^{\vec{\eta}}\ x : \tau. \ e \mid e_1\ e_2 \mid {}^q\text{poly}\ X : A. \ e \mid e\ [A] \\ & \mid {}^q\Rightarrow\ \eta. \ e \mid e\ [\eta] \mid \text{letrec}\ [\vec{A}]\ \vec{x} : \vec{e} = \vec{e}_1 \text{ in } e_2 \\ & \mid {}^q\text{pack}(X : B = A, e : \tau) \mid \text{unpack}\ e_1 \text{ as } (X : B, x : \tau) \text{ in } e_2 \\ & \mid {}^q\langle\ \mu_0 e_0, \dots, \mu_{p-1} e_{p-1}\ \rangle \mid \text{if}[B, A](e, X_1. \ e_1, X_2. \ e_2) \\ & \mid \text{at } \eta \text{ let! } [A, B, X]\ Y = e_1 \text{ in } e_2 \mid \text{qcast}[A](e) \mid \text{knowing } [A] \text{ for } X \text{ in } e \\ \\ op ::= & \text{u+} \mid \text{u-} \mid \text{not} \mid + \mid - \mid * \mid / \mid \% \mid = \mid \text{<>} \mid < \mid > \mid \leq \mid \geq \\ & \mid \text{and} \mid \text{or} \mid \langle \rangle \mid \text{newregion} \mid \text{freeregion} \mid \text{read} \mid \text{assign}_+ \mid \text{assign}_- \\ & \mid \text{swap}_+ \mid \text{swap}_- \mid \text{split} \mid \text{join}\end{array}$$

x	: μεταβλητή της γλώσσας υπολογισμού
X, Y	: μεταβλητή της γλώσσας τύπων
A, B	: όροι της γλώσσας τύπων
n, i, c, b	: κλειστοί όροι της γλώσσας τύπων
η	: μεταβλητή με τύπο Sc
q	$\in \{\mathbf{L}, \mathbf{A}, \mathbf{R}, \mathbf{U}\}$
μ_k	$\in \{\text{ref}, \text{source}, \text{sink}, \text{hole}\}$
p	$\in \mathbb{N}$

5.2.8 Περιβάλλοντα

Κατά τον τυποθεωρητικό έλεγχο ορθότητας των εκφράσεων της γλώσσας υπολογισμού χρησιμοποιούνται τέσσερα περιβάλλοντα υποθέσεων.

- Το περιβάλλον Γ των μεταβλητών της γλώσσας υπολογισμού

$$\Gamma ::= \emptyset \mid \Gamma, x_q \triangleright \tau$$

όπου

$$\begin{aligned} x &: \text{μεταβλητή της γλώσσας υπολογισμού} \\ \tau &: \Omega \end{aligned}$$

Ο τελεστής $\text{split } \oplus$ εξασφαλίζει ότι linear υποθέσεις δεν διπλασιάζονται ή αγνοούνται κατά τον τυποθεωρητικό έλεγχο των εκφράσεων.

$$\begin{array}{lll} \emptyset \oplus \Gamma & \equiv & \Gamma \\ (\Gamma_1, x_{q'} \triangleright {}^q\phi) \oplus \Gamma_2 & \equiv & \Gamma_1 \oplus \Gamma_2, x_{q'} \triangleright {}^q\phi \\ (\Gamma_1, x_{q'} \triangleright {}^q\phi) \oplus \Gamma_2 & \equiv & \Gamma_1 \oplus \Gamma_2 \end{array} \quad \begin{array}{l} \text{αν } (x_{q'} \triangleright {}^q\phi) \notin \Gamma_2 \\ \text{αν } (x_{q'} \triangleright {}^q\phi) \in \Gamma_2 \text{ και } q' \preceq \mathbf{R} \end{array}$$

- Το περιβάλλον C των μεταβλητών του Set C

$$C ::= \emptyset \mid C, X_q \triangleright C$$

όπου X : μεταβλητή της γλώσσας τύπων
 c : C

Ο τελεστής $\text{split } \oplus$:

$$\begin{array}{lll} \emptyset \oplus C & \equiv & C \\ (C_1, X_{q'} \triangleright {}^q\eta\xi) \oplus C_2 & \equiv & C_1 \oplus C_2, X_{q'} \triangleright {}^q\eta\xi \\ (C_1, X_{q'} \triangleright {}^q\eta\xi) \oplus C_2 & \equiv & C_1 \oplus C_2 \end{array} \quad \begin{array}{l} \text{αν } (X_{q'} \triangleright {}^q\eta\xi) \notin C_2 \\ \text{αν } (X_{q'} \triangleright {}^q\eta\xi) \in C_2 \text{ και } q' \preceq \mathbf{R} \end{array}$$

- Το περιβάλλον Δ των μεταβλητών τύπου Set της γλώσσας τύπων.

$$\Delta ::= \emptyset \mid \Delta, X : A$$

όπου X : μεταβλητή της γλώσσας τύπων
 A : όρος τύπου Set

- Το περιβάλλον S των μεταβλητών Sc της γλώσσας τύπων.

$$S ::= \emptyset \mid S, \eta$$

όπου η : μεταβλητή της γλώσσας τύπων

5.2.9 Οι Κανόνες του Συστήματος Τύπων

Μεταβλητές και Σταθερές

$$\Delta; (x \ q \triangleright \ \tau); \emptyset; S \vdash x \triangleright \ \tau$$

$$\frac{\Delta; \Gamma_1; C; S \vdash e \triangleright \ \tau \quad \Delta \vdash \Gamma_2 \preceq A}{\Delta; \Gamma_1 \oplus \Gamma_2; C; S \vdash e \triangleright \ \tau}$$

$$\frac{\Delta; \Gamma; C_1; S \vdash e \triangleright \ \tau \quad \Delta \vdash C_2 \preceq A}{\Delta; \Gamma; C_1 \oplus C_2; S \vdash e \triangleright \ \tau}$$

$$\frac{\emptyset \vdash n : Nat}{\Delta; \emptyset; \emptyset; S \vdash cnat[n] \triangleright \text{^snat } n}$$

$$\frac{\emptyset \vdash n : Int}{\Delta; \emptyset; \emptyset; S \vdash cint[n] \triangleright \text{^sint } n}$$

$$\frac{\emptyset \vdash c : Char}{\Delta; \emptyset; \emptyset; S \vdash cchar[c] \triangleright \text{^schar } c}$$

$$\frac{\emptyset \vdash b : Bool}{\Delta; \emptyset; \emptyset; S \vdash cbool[b] \triangleright \text{^sbool } b}$$

Προσαρμογή (casting) των Qualifiers

Ο έλεγχος πρόσβασης στη μνήμη γίνεται μέσω των capabilities. Επομένως, οι qualifiers, όλων των τύπων πλην των capabilities, χρησιμοποιούνται μόνο κατά την ανάλυση ροής των δεδομένων όπως και σε όλα τα συστήματα που συνδυάζουν capabilities και substuctural συστήματα τύπων. Η αλλαγή του qualifier του τύπου ενός δεδομένου με αντίθετη φορά από αυτή που ορίζει η ιεραρχία των qualifiers είναι τυποθεωρητικά ασφαλής.

$$\frac{\Delta; \Gamma; C; S \vdash e \triangleright {}^q \phi \quad \Delta \vdash A : (q \preceq q')}{\Delta; \Gamma; C; S \vdash qcast[A](e) \triangleright {}^{q'} \phi}$$

Τελεστής Γνωστοποίησης Πληροφορίας

Η γλώσσα υπολογισμού διαθέτει ένα πανίσχυρο σύστημα πολυμορφισμού με αποτέλεσμα να είναι πολυμορφική και ως προς τους qualifiers. Ο ρόλος των qualifiers είναι ιδιαίτερα σημαντικός για τον τρόπο που χρησιμοποιούνται οι μεταβλητές. Στην περίπτωση που ο qualifier ενός αντικειμένου της γλώσσας τύπων είναι μεταβλητή τότε ο ασφαλέστερος τρόπος είναι να αντιμετωπιστεί ο τύπος σαν linear. Όμως, ο προγραμματιστής μπορεί να διαθέτει κάποια πληροφορία σχετικά με τη φύση του qualifier-μεταβλητής. Ο τελεστής γνωστοποίησης πληροφορίας χρησιμοποιείται για τη διάδοση τέτοιας πληροφορίας με τη μορφή απόδειξης.

$$\frac{\Delta; \Gamma, (x \ q'' \triangleright {}^q \phi); C; S \vdash e \triangleright \ \tau \quad \Delta \vdash A : (q \preceq q'')}{\Delta; \Gamma, (x \ q' \triangleright {}^q \phi); C; S \vdash \text{knowing } [A] \text{ for } x \text{ in } e \triangleright \ \tau}$$

$$\frac{\Delta; \Gamma; C, (X \ q'' \triangleright {}^q \xi); S \vdash e \triangleright \ \tau \quad \Delta \vdash A : (q \preceq q'')}{\Delta; \Gamma; C, (X \ q' \triangleright {}^q \xi); S \vdash \text{knowing } [A] \text{ for } X \text{ in } e \triangleright \ \tau}$$

Τελεστές

Πρωταρχικοί Τελεστές Οι πρωταρχικοί τελεστές είναι ενσωματωμένοι στη γλώσσα και χρησιμοποιούνται για τη διαχείριση μνήμης και την ανάγνωση και τροποποίηση των περιεχομένων της μνήμης.

- Ο τελεστής $\langle \rangle$ κατασκευάζει αντικείμενα της γλώσσας υπολογισμών τύπου $^U\text{unit}$

$$\langle \rangle \triangleright ^U\text{unit}$$

- Ο τελεστής `newregion` κατασκευάζει ένα linear πακέτο μέσα στο οποίο περιέχονται το linear capability για ένα νέο region και ο αντίστοιχος unrestricted handler. Είναι ο τελεστής μέσω του οποίου γίνεται η δέσμευση μνήμης. Η πρόσβαση στα περιεχόμενα ενός region απαιτούν και τις δύο παραπάνω οντότητες. Ο handler είναι unrestricted για να επιτρέπει τη δημιουργία αντιγράφων του ενώ το capability είναι linear για να εξασφαλίζει την ασφαλή πρόσβαση στο region.

$$\text{newregion} \triangleright ^U\text{unit} \xrightarrow{^U} L_{\exists_{\text{Set}} \rho : \text{Rgn}. L_{\exists_C L_{\perp} \text{rgn} \rho}. ^U\text{handle} \rho}$$

- Ο τελεστής `freeregion` αποδεσμεύει τη μνήμη που αντιστοιχεί σε ένα region. Για να γίνει αυτό απαιτείται να δοθεί ένα linear capability για το region που εξασφαλίζει ότι δεν θα υπάρξουν μελλοντικές προσπάθειες χρήσης του. Επίσης ζητείται και ο handler του region.

$$\text{freeregion} \triangleright ^U\forall_{\text{Set}} \rho : \text{Rgn}. ^U\forall_C L_{\perp} \text{rgn} \rho. L_{\text{handle} \rho} \xrightarrow{\perp} ^U\text{unit}$$

- Ο τελεστής `read` χρησιμοποιείται για την ανάγνωση μόνο unrestricted ή relevant δεδομένων που περιέχονται σε μία πλειάδα. Ο τελεστής διπλασιάζει τις αναφορές πάνω στα δεδομένα όπου εφαρμόζεται και γι' αυτό δεν μπορεί να εφαρμοστεί πάνω σε linear ή affine δεδομένα. Η χρήση του τελεστή δεν αλλάζει την κατάσταση της μνήμης και γι' αυτό δεν απαιτείται αποκλειστική πρόσβαση σε αυτή. Για να εφαρμοστεί, χρειάζεται απόδειξή ότι το πεδίο της πλειάδας είναι αναγνώσιμο.

$$\begin{aligned} \text{read} \triangleright & \quad ^U\forall_{\text{Set}} \rho : \text{Rgn}. \\ & ^U\forall_{\text{Set}} l : \text{Loc} \rho. \\ & ^U\forall_{\text{Set}} n : \text{Nat}. \\ & ^U\forall_{\text{Set}} f : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\ & ^U\forall_{\text{Set}} i : \text{Nat}. \\ & ^U\forall_{\text{Set}} p_1^* : (i <_{\text{Nat}} n). \\ & ^U\forall_{\text{Set}} p_2^* : (\text{source} \preceq \text{snd} (f i)). \\ & ^U\forall_{\text{Set}} p_3^* : (\text{fst} (f 0) \preceq \mathbf{R}). \\ & ^U\forall_{\text{Sc}} \eta_1. ^U\forall_{\text{Sc}} \eta_2. \\ & ^U\forall_{\text{Set}} q_1 : \text{Qual}. ^U\forall_{\text{Set}} q_2 : \text{Qual}. \\ & ^U\forall_C \frac{q_1}{\eta_1} \text{rgn} \rho. \\ & q_1 \forall_C \frac{q_2}{\eta_2} \text{typ} \rho l n f. \\ & L_{\text{tuple} \rho} l \xrightarrow[\eta_1, \eta_2]{\sqcup \{q_1, q_2\}} \text{fst} (f i) \end{aligned}$$

- Ο τελεστής assign_- χρησιμοποιείται για την εγγραφή δεδομένων σε θέση δοσμένης πλειάδας που περιέχει μόνο unrestricted ή affine δεδομένα ίδιου τύπου. Ο τελεστής αγνοεί τα αρχικά περιεχόμενα της πλειάδας στην οποία εφαρμόζεται και γι' αυτό δεν μπορεί να εφαρμοστεί πάνω σε linear ή relevant δεδομένα. Η χρήση του τελεστή δεν αλλάζει την κατάσταση της μνήμης και γι' αυτό δεν απαιτείται αποκλειστική πρόσβαση σε αυτή. Για να εφαρμοστεί χρειάζεται απόδειξη ότι το πεδίο της πλειάδας είναι εγγράψιμο.

$$\begin{aligned}
\text{assign}_- \triangleright & \quad \mathbf{U}_{\text{Set}} \rho : \text{Rgn}. \\
& \mathbf{U}_{\text{Set}} l : \text{Loc } \rho. \\
& \mathbf{U}_{\text{Set}} n : \text{Nat}. \\
& \mathbf{U}_{\text{Set}} f : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U}_{\text{Set}} i : \text{Nat}. \\
& \mathbf{U}_{\text{Set}} p_1^* : (i <_{\text{Nat}} n). \\
& \mathbf{U}_{\text{Set}} p_2^* : (\text{sink} \preceq \text{snd}(f i)). \\
& \mathbf{U}_{\text{Set}} p_3^* : (\text{fst}(f i) \preceq \mathbf{A}). \\
& \mathbf{U}_{\text{Sc}} \eta_1. \mathbf{U}_{\text{Sc}} \eta_2. \\
& \mathbf{U}_{\text{Set}} q_1 : \text{Qual}. \mathbf{U}_{\text{Set}} q_2 : \text{Qual}. \\
& \mathbf{U}_{\mathcal{C} \eta_1} \text{rgn } \rho. \\
& {}^{q_1} \forall_{\mathcal{C} \eta_2} {}^{q_2} \text{typ } \rho l n f. \\
& \mathbf{L}_{\text{tuple}} \rho l \xrightarrow[\eta_1, \eta_2]{\sqcup\{q_1, q_2\}} \text{fst}(f i) \xrightarrow[\eta_1, \eta_2]{\sqcup\{q_1, q_2\}} \mathbf{U}_{\text{unit}}
\end{aligned}$$

- Ο τελεστής assign_+ χρησιμοποιείται για την εγγραφή δεδομένων σε θέση δοσμένης πλειάδας που περιέχει μόνο unrestricted ή affine δεδομένα συμβατού τύπου. Ο τελεστής αγνοεί τα αρχικά περιεχόμενα της πλειάδας στην οποία εφαρμόζεται και γι' αυτό δεν μπορεί να εφαρμοστεί πάνω σε linear ή relevant δεδομένα. Η χρήση του τελεστή αλλάζει την κατάσταση της μνήμης και γι' αυτό απαιτείται αποκλειστική πρόσβαση σε αυτή. Για να εφαρμοστεί χρειάζεται απόδειξη ότι το πεδίο της πλειάδας είναι εγγράψιμο.

$$\begin{aligned}
\text{assign}_+ \triangleright & \quad \mathbf{U}_{\text{Set}} \rho : \text{Rgn}. \\
& \mathbf{U}_{\text{Set}} l : \text{Loc } \rho. \\
& \mathbf{U}_{\text{Set}} n : \text{Nat}. \\
& \mathbf{U}_{\text{Set}} f : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U}_{\text{Set}} \tau : \Omega. \\
& \mathbf{U}_{\text{Set}} i : \text{Nat}. \\
& \mathbf{U}_{\text{Set}} p_1^* : (i <_{\text{Nat}} n). \\
& \mathbf{U}_{\text{Set}} p_2^* : (\text{sink} \preceq \text{snd}(f i)). \\
& \mathbf{U}_{\text{Set}} p_3^* : (\text{fst}(f i) \preceq \mathbf{A}). \\
& \mathbf{U}_{\text{Set}} p_4^* : (\text{compatible } \tau(\text{fst}(f i))). \\
& \mathbf{U}_{\text{Sc}} \eta. \mathbf{U}_{\text{Set}} q_1 : \text{Qual}. \mathbf{U}_{\text{Set}} q_2 : \text{Qual}. \\
& \mathbf{U}_{\text{Set}} p_4^* : (\mathbf{A} \preceq q_2). \\
& \mathbf{U}_{\mathcal{C} \eta} \text{rgn } \rho. \\
& {}^{q_1} \forall_{\mathcal{C} \perp} {}^{q_2} \text{typ } \rho l n f. \\
& \mathbf{L}_{\text{tuple}} \rho l \xrightarrow[\eta]{\sqcup\{q_1, q_2\}} \tau \xrightarrow[\eta]{\sqcup\{q_1, q_2\}} \\
& \quad \sqcup\{q_2, \text{getq}(\tau)\} \exists_{\mathcal{C} \perp} \sqcup\{q_2, \text{getq}(\tau)\} \text{typ } \rho l n (\text{update } i(\tau, \text{snd}(f i)) f). \mathbf{U}_{\text{unit}}
\end{aligned}$$

- Ο τελεστής swap_- χρησιμοποιείται για την ανταλλαγή δεδομένου με δεδομένο ίδιου τύπου που βρίσκεται σε θέση δοσμένης πλειάδας. Ο τελεστής διατηρεί τις αναφορές πάνω στα δεδομένα που εφαρμόζεται και γι' αυτό μπορεί να εφαρμοστεί πάνω σε κάθε είδους δεδομένο. Η χρήση του τελεστή δεν αλλάζει την κατάσταση της μνήμης και γι' αυτό δεν απαιτείται αποκλειστική πρόσβαση σε αυτή. Για να εφαρμοστεί χρειάζεται απόδειξη ότι το πεδίο της πλειάδας είναι εγγράψιμο και αναγνώσιμο.

$$\begin{aligned}
\text{swap}_- \triangleright & \quad \mathbf{U} \forall_{\text{Set}} \rho : \text{Rgn}. \\
& \mathbf{U} \forall_{\text{Set}} l : \text{Loc } \rho. \\
& \mathbf{U} \forall_{\text{Set}} n : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} f : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U} \forall_{\text{Set}} i : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} p_1^* : (i <_{\text{Nat}} n). \\
& \mathbf{U} \forall_{\text{Set}} p_2^* : (\text{ref} \preceq \text{snd} (f i)). \\
& \mathbf{U} \forall_{\text{Sc}} \eta_1. \mathbf{U} \forall_{\text{Sc}} \eta_2. \\
& \mathbf{U} \forall_{\text{Set}} q_1 : \text{Qual}. \mathbf{U} \forall_{\text{Set}} q_2 : \text{Qual}. \\
& \mathbf{U} \forall_{\text{C}} \eta_1^{q_1} \text{rgn } \rho. \\
& {}^{q_1} \forall_{\text{C}} \eta_2^{q_2} \text{typ } \rho \text{ } l \text{ } n \text{ } f. \\
& \mathbf{L} \text{tuple } \rho \text{ } l \xrightarrow[\eta_1, \eta_2]{\sqcup\{q_1, q_2\}} \text{fst } (f i) \xrightarrow[\eta_1, \eta_2]{\sqcup\{q_1, q_2\}} \text{fst } (f i)
\end{aligned}$$

- Ο τελεστής swap_+ χρησιμοποιείται για την ανταλλαγή δεδομένου με δεδομένο συμβατού τύπου που βρίσκεται σε θέση δοσμένης πλειάδας. Ο τελεστής διατηρεί τις αναφορές πάνω στα δεδομένα που εφαρμόζεται και γι' αυτό μπορεί να εφαρμοστεί πάνω σε κάθε είδους δεδομένο. Η χρήση του τελεστή αλλάζει την κατάσταση της μνήμης και γι' αυτό απαιτείται αποκλειστική πρόσβαση σε αυτή. Για να εφαρμοστεί χρειάζεται απόδειξη ότι το πεδίο της πλειάδας είναι εγγράψιμο και αναγνώσιμο.

$$\begin{aligned}
\text{swap}_+ \triangleright & \quad \mathbf{U} \forall_{\text{Set}} \rho : \text{Rgn}. \\
& \mathbf{U} \forall_{\text{Set}} l : \text{Loc } \rho. \\
& \mathbf{U} \forall_{\text{Set}} n : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} f : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U} \forall_{\text{Set}} \tau : \Omega. \\
& \mathbf{U} \forall_{\text{Set}} i : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} p_1^* : (i <_{\text{Nat}} n). \\
& \mathbf{U} \forall_{\text{Set}} p_2^* : (\text{ref} \preceq \text{snd} (f i)). \\
& \mathbf{U} \forall_{\text{Set}} p_3^* : (\text{compatible } \tau (\text{fst } (f 0))). \\
& \mathbf{U} \forall_{\text{Sc}} \eta. \mathbf{U} \forall_{\text{Set}} q_1 : \text{Qual}. \mathbf{U} \forall_{\text{Set}} q_2 : \text{Qual}. \\
& \mathbf{U} \forall_{\text{Set}} p_3^* : (\mathbf{A} \preceq q_2). \\
& \mathbf{U} \forall_{\text{C}} \eta^{q_1} \text{rgn } \rho. \\
& {}^{q_1} \forall_{\text{C}} \perp^{q_2} \text{typ } \rho \text{ } l \text{ } n \text{ } f. \\
& \mathbf{L} \text{tuple } \rho \text{ } l \xrightarrow[\eta]{\sqcup\{q_1, q_2\}} \tau \xrightarrow[\eta]{\sqcup\{q_1, q_2\}} \\
& \quad \sqcup\{q_2, \text{getq}(\tau)\} \exists_{\text{C}} \perp^{\sqcup\{q_2, \text{getq}(\tau)\}} \text{typ } \rho \text{ } l \text{ } n \text{ (update } 0 (\tau, \text{snd } (f i)) \text{ } f) \text{. fst } (f i)
\end{aligned}$$

- Ο τελεστής `split` δεν έχει υπολογιστική αξία. Χρησιμοποιείται για την αλλαγή του τρόπου οργάνωσης των δεδομένων της μνήμης. Δέχεται ένα capability μίας πλειάδας και μία αναφορά πλειάδα και επιστρέφει δύο αναφορές σε δύο τμήματα της αρχικής πλειάδας και τα αντίστοιχα capabilities.

$$\begin{aligned}
\text{split} \triangleright & \quad \mathbf{U} \forall_{\text{Set}} \rho : \text{Rgn}. \\
& \mathbf{U} \forall_{\text{Set}} l : \text{Loc } \rho. \\
& \mathbf{U} \forall_{\text{Set}} n : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} f : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U} \forall_{\text{Set}} P : \text{Nat} \rightarrow \text{Bool}. \\
& \mathbf{U} \forall_{\text{Set}} i : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} p_1^* : (i <_{\text{Nat}} n). \\
& \mathbf{U} \forall_{\text{Sc}} \eta_1 \cdot \mathbf{U} \forall_{\text{Sc}} \eta_1. \\
& \mathbf{U} \forall_{\text{Set}} q_1 : \text{Qual}. \mathbf{U} \forall_{\text{Set}} q_2 : \text{Qual}. \\
& \mathbf{U} \forall_{\mathcal{C}} \eta_1^{q_1} \text{rgn } \rho. \\
& q_1 \forall_{\mathcal{C}} \eta_2^{q_2} \text{typ } \rho \mid n \ f. \\
& \mathbf{L} \text{tuple } \rho l \xrightarrow[\eta_1, \eta_2]{\sqcup\{q_1, q_2\}} \mathbf{L} \text{snat } i \xrightarrow[\eta_1, \eta_2]{\sqcup\{q_1, q_2\}} \\
& q_2 \exists_{\mathcal{C}} \eta_2^{q_2} \text{typ } \rho \mid n \ (\text{except } P \ f). \\
& q_2 \exists_{\mathcal{C}} \eta_2^{q_2} \text{typ } \rho \ (\text{subloc } \rho \mid i \ f) \ (n \ominus_{\text{Nat}} i) \ (\text{shift } i \ (\text{only } P \ f)). \\
& \mathbf{U} \text{tuple } \rho \ (\text{subloc } \rho \mid i \ f)
\end{aligned}$$

- Ο τελεστής `join` δεν έχει επίσης υπολογιστική αξία και χρησιμοποιείται για την αλλαγή του τρόπου οργάνωσης των δεδομένων της μνήμης. Είναι ο αντιστροφος του τελεστή `split`. Δέχεται δύο capability και δύο αναφορές σε δύο τμήματα μίας πλειάδας και επιστρέφει μία αναφορά στην πλειάδα που προκύπτει από την ένωση των δύο τμήμα και το αντίστοιχο capability. Η συνένωση των τμημάτων μπορεί να γίνει αν τα capabilities των τμημάτων δεν έχουν υποστεί αλλαγή του qualifier τους. Στην αντίθετη περίπτωση μπορούμε να οδηγηθούμε στην συνύπαρξη linear και unrestricted capabilities που δίνουν δικαίωμα πρόσβασης σε κοινά τμήματα μνήμης.

$$\begin{aligned}
\text{join} \triangleright & \quad \mathbf{U} \forall_{\text{Set}} \rho : \text{Rgn}. \\
& \mathbf{U} \forall_{\text{Set}} l : \text{Loc } \rho. \\
& \mathbf{U} \forall_{\text{Set}} n : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} f : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U} \forall_{\text{Set}} P : \text{Nat} \rightarrow \text{Bool}. \\
& \mathbf{U} \forall_{\text{Set}} i : \text{Nat}. \\
& \mathbf{U} \forall_{\text{Set}} p_1^* : (i <_{\text{Nat}} n). \\
& \mathbf{U} \forall_{\text{Set}} q_1 : \text{Qual}. \mathbf{U} \forall_{\text{Set}} q_2 : \text{Qual}. \mathbf{U} \forall_{\text{Set}} q_3 : \text{Qual}. \\
& \mathbf{U} \forall_{\text{Set}} g : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U} \forall_{\text{Set}} h : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \mathbf{U} \forall_{\text{Set}} p_2^* : (\Pi k : \text{Nat}. P \ k \rightarrow \text{snd}(g \ k) \preceq \text{hole}). \\
& \mathbf{U} \forall_{\text{Set}} p_3^* : (\Pi k : \text{Nat}. \neg P \ k \rightarrow \text{snd}(h \ k) \preceq \text{hole}). \\
& \mathbf{U} \forall_{\mathcal{C}} \eta_1^{q_1} \text{rgn } \rho.
\end{aligned}$$

$$\begin{aligned}
& \quad q_1 \forall_{C \perp} \text{typ } \rho \ l \ n \ g. \\
& \quad \sqcup \{q_1, q_2\} \forall_{C \perp} \text{typ } \rho \ (\text{subloc } \rho \ l \ i \ f) (n \ominus_{\text{Nat}} i) \ h. \\
& \quad \mathbf{L}_{\exists_{C \perp}} \sqcup \{q_2, q_3\} \text{typ } \rho \ l \ n \ (\text{choose } P \ (\text{unshift } i \ h) \ g). \ \mathbf{U}_{\text{unit}}
\end{aligned}$$

Αριθμητικοί Τελεστές Χρειάζονται αρχετοί υπερφορτωμένοι τελεστές, όπως ο τελεστής $+$. Για λόγους απλότητας δίδονται οι κανόνες για τους τελεστές ακεραίων.

$$\begin{array}{c}
\dfrac{\Delta; \Gamma; C; S \vdash e \triangleright \text{qint } n}{\Delta; \Gamma; C; S \vdash u+e \triangleright \text{qint } n} \\[10pt]
\dfrac{\Delta; \Gamma; C; S \vdash e \triangleright \text{qint } n}{\Delta; \Gamma; C; S \vdash u-e \triangleright \text{qint } (\text{neg}_{\text{Int}} n)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 + e_2 \triangleright \text{qint } (n_1 +_{\text{Int}} n_2)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 - e_2 \triangleright \text{qint } (n_1 -_{\text{Int}} n_2)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 * e_2 \triangleright \text{qint } (n_1 *_{\text{Int}} n_2)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 / e_2 \triangleright \text{qint } (n_1 /_{\text{Int}} n_2)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 \% e_2 \triangleright \text{qint } (n_1 \%_{\text{Int}} n_2)}
\end{array}$$

Σχεσιακοί Τελεστές

$$\begin{array}{c}
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 = e_2 \triangleright \text{sbool } (\text{eq}_{\text{Int}} n_1 n_2)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 <> e_2 \triangleright \text{sbool } (\text{ne}_{\text{Int}} n_1 n_2)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 < e_2 \triangleright \text{sbool } (\text{lt}_{\text{Int}} n_1 n_2)} \\[10pt]
\dfrac{\begin{array}{c} \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \text{qint } n_1 \\ \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \text{qint } n_2 \end{array}}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 <= e_2 \triangleright \text{sbool } (\text{le}_{\text{Int}} n_1 n_2)}
\end{array}$$

$$\begin{array}{c}
 \frac{\Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright {}^q \text{sint } n_1}{\Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright {}^q \text{sint } n_2} \\
 \hline
 \Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 > e_2 \triangleright {}^q \text{sbool} (\text{gt}_{\text{Int}} n_1 n_2)
 \end{array}$$

$$\begin{array}{c}
 \frac{\Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright {}^q \text{sint } n_1}{\Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright {}^q \text{sint } n_2} \\
 \hline
 \Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 >= e_2 \triangleright {}^q \text{sbool} (\text{ge}_{\text{Int}} n_1 n_2)
 \end{array}$$

Λογικοί Τελεστές

$$\begin{array}{c}
 \frac{\Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright {}^q \text{sbool } b_1}{\Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright {}^q \text{sbool } b_2} \\
 \hline
 \Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 \text{ and } e_2 \triangleright {}^q \text{sbool} (\text{and } b_1 b_2)
 \end{array}$$

$$\begin{array}{c}
 \frac{\Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright {}^q \text{sbool } b_1}{\Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright {}^q \text{sbool } b_2} \\
 \hline
 \Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 \text{ and } e_2 \triangleright {}^q \text{sbool} (\text{or } b_1 b_2)
 \end{array}$$

$$\begin{array}{c}
 \frac{\Delta; \Gamma; C; S \vdash e \triangleright {}^q \text{sbool } b}{\Delta; \Gamma; C; S \vdash \text{not } e \triangleright {}^q \text{sbool} (\text{not } b)}
 \end{array}$$

Συναρτήσεις

Ο qualifier μίας συνάρτησης καθορίζεται από τους qualifiers των υποθέσεων που χρησιμοποιούνται. Η συνάρτηση πρέπει να έχει qualifier ίσο ή σε υψηλότερο επίπεδο της ιεραρχίας με το κατώτερο άνω όριο του περιβάλλοντος στο οποίο αποτιμάται. Αν π.χ. στο περιβάλλον αποτίμησης της συνάρτησης υπάρχει κάποια linear υπόθεση τότε η συνάρτηση αποκτά qualifier **L**. Μ' αυτό τον τρόπο εξασφαλίζεται ότι πληρούνται οι περιορισμοί που θέτουν τα substructural συστήματα τύπων για κάθε qualifier.

Επειδή οι συναρτήσεις μπορούν να περιέχουν σαν ελεύθερες μεταβλητές υποθέσεις οι οποίες δεν είναι ανιχνεύσιμες μέσω του τύπου των συναρτήσεων, είναι απαραίτητο το σύστημα τύπων να υιοθετεί ελέγχους για να εξασφαλίζει την εγκυρότητα των χρησιμοποιούμενων υποθέσεων από τις συναρτήσεις. Το πρόβλημα εστιάζεται στις unrestricted και relevant υποθέσεις καθώς μία linear ή affine υπόθεση είναι πάντα έγκυρη. Πρέπει να επιβεβαιωθεί ότι μία unrestricted υπόθεση χρησιμοποιείται αν και μόνο αν το scope της είναι έγκυρο. Γι' αυτό τόσο στον τύπο των συναρτήσεων όσο και στην σύνταξή τους καταγράφονται όλα τα scopes των υποθέσεων που χρησιμοποιούνται μία συνάρτηση. Μ' αυτό τον τρόπο μπορεί να ελεγχθεί κάθε στιγμή αν η χρήση ενός τμήματος κώδικα είναι έγκυρη και ασφαλής.

$$\begin{array}{c}
 \frac{\Delta; \emptyset \vdash \tau : \Omega \quad \sqcup \Gamma, C \equiv q}{\Delta; \Gamma, x \triangleright \tau; C; \emptyset \vdash e \triangleright \tau'} \\
 \hline
 \Delta; \Gamma; C; \emptyset \vdash {}^q \text{lambda}^{\{\}} x:\tau. e \triangleright \tau \frac{q}{\perp} \tau' \\
 \hline
 \Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright \tau \frac{q}{\vec{\eta}} \tau' \quad \Delta; \Gamma_2; C_2; S \vdash e_2 \triangleright \tau \quad S \vdash \vec{\eta} \\
 \hline
 \Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash e_1 e_2 \triangleright \tau'
 \end{array}$$

$$\begin{array}{c}
 \frac{\Delta; S \vdash \tau : \Omega \quad \sqcup \{\Gamma, C\} \equiv q}{\Delta; \Gamma, x \triangleright \tau; C; S \vdash e \triangleright \tau'} \\
 \hline
 \Delta; \Gamma; C; S \vdash {}^q \text{lambda}^S x:\tau. e \triangleright \tau \frac{q}{S} \tau'
 \end{array}$$

Η γλώσσα διαθέτει ένα πανίσχυρο σύστημα πολυμορφισμού πάνω σε αντικείμενα της γλώσσας τύπων με τύπο Set, Ω , C και Sc . Για κάθε τύπο δεδομένων υπάρχει ο ανάλογος τελεστής που αναλαμβάνει να τοποθετήσει τις νέες υποθέσεις στο κατάλληλο περιβάλλον.

$$\begin{array}{c}
\frac{\Delta; S \vdash A : s \quad \sqcup \Gamma, C \equiv q}{\Delta, X : A; \Gamma; \emptyset; S \vdash e \triangleright \tau \quad s \in \{\text{Set}, \text{Type}\}} \quad \frac{\Delta; \Gamma; C \vdash e \triangleright {}^q\text{forall}_s A' \tau \quad \Delta \vdash A : A'}{\Delta; \Gamma; C \vdash e [A] \triangleright \tau A} \\
\\
\frac{\Delta; S \vdash A : C \quad \sqcup \Gamma, C \equiv q}{\Delta; \Gamma; X : A; S \vdash e \triangleright \tau} \quad \frac{\Delta; \Gamma; C; S \vdash e \triangleright {}^q\text{poly } X : A. e \triangleright {}^q\forall_C X : A. \tau}{\Delta; \Gamma; C; S \vdash e \triangleright \eta. e \triangleright {}^q\forall_C \eta. \tau} \\
\\
\frac{\Delta; \Gamma; S, \eta \vdash e \triangleright \tau \quad \sqcup \Gamma, C \equiv q}{\Delta; \Gamma; C; S \vdash e \triangleright {}^q \Rightarrow \eta. e \triangleright {}^q\forall_{S_c} \eta. \tau} \quad \frac{\Delta; \Gamma; C; S \vdash e \triangleright {}^q\forall_{S_c} \eta. \tau \quad S \vdash \eta}{\Delta; \Gamma; C; S \vdash e [\eta] \triangleright \tau}
\end{array}$$

Η γλώσσα δίδει τη δυνατότητα ορισμού αναδρομικών συναρτήσεων με τον περιορισμό να έχουν unrestricted τύπο γιατί η κλήση μίας αναδρομικής συνάρτησης μπορεί να οδηγήσει σε μία ή περισσότερες κλήσεις της.

$$\frac{\Delta; \Gamma_i, \vec{x} \triangleright \vec{\tau}; C_i; S \vdash e_i \triangleright \tau_i \quad \Delta; \Gamma, \vec{x} \triangleright \vec{\tau}; C; S \vdash e' \triangleright \tau}{\Delta \vdash A_i : (\tau_i \preceq \mathbf{U}) \quad \Delta; S \vdash \tau_i : \Omega} \quad \frac{}{\Delta; \vec{\Gamma} \oplus \Gamma, \vec{C} \oplus C; S \vdash \text{letrec } [\vec{A}] \vec{x} : \vec{\tau} = \vec{e_i} \text{ in } e' \triangleright \tau}$$

Υπαρξιακοί Τύποι

Ο qualifier ενός πακέτου καθορίζεται από τους qualifiers των δεδομένων που περιέχει. Το πακέτο πρέπει να έχει qualifier ίσο ή σε υψηλότερο επίπεδο της ιεραρχίας με το κατώτερο άνω όριο των qualifiers των περιεχομένων του. Αν π.χ. ένα πακέτο περιέχει κάποιο linear δεδομένο τότε αποκτά qualifier **L**. Μ' αυτό τον τρόπο εξασφαλίζεται ότι το πακέτο θα μπορέσει να χρησιμοποιηθεί μόνο όσες φορές επιτρέπεται από τους περιορισμούς που θέτουν τα substructural συστήματα τύπων για κάθε qualifier. Ένα linear πακέτο πρέπει να ανοιχτεί μία φορά. Αν δεν ανοιχτεί αγνοούνται τα linear περιεχόμενά του ενώ αν ανοιχτεί περισσότερες από μία, κατασκευάζονται αντίγραφα για linear δεδομένα. Η γλώσσα δίνει δυνατότητα κατασκευής πακέτων με αντικείμενα της γλώσσας τύπων κάθε τύπου. Εξαιρούνται τα αντικείμενα τύπου **Sc**. Αυτά είναι άμεσα συνδεδεμένα με την συντακτική εμβέλεια στην οποία εμφανίζονται και δεν πρέπει να μπορούν να αποδράσουν από αυτήν.

$$\begin{array}{c}
\frac{\Delta \vdash A : B \quad \Delta \vdash B : s \quad s \in \{\text{Set}, \text{Type}\} \quad \text{getq}(\tau \{X \mapsto A\}) \equiv q}{\Delta; \Gamma; C; S \vdash e \triangleright \tau \{X \mapsto A\}} \quad \frac{\Delta; \Gamma; C_1; S \vdash e_1 \triangleright {}^q\text{exists}_s B \tau'' \quad \Delta, X : B; S \vdash \tau : \Omega \quad \Delta; S \vdash \tau' : \Omega}{\Delta, X : B; \Gamma_2, x \triangleright \tau'' X, C_2; S \vdash e_2 \triangleright \tau'} \quad \frac{\Delta; S \vdash A : C \quad \Delta; \Gamma; C; S \vdash e \triangleright \tau \quad \sqcup \{A, \tau\} \equiv q}{\Delta; \Gamma; C, Y : A; S \vdash {}^q\text{pack}(X : A = Y, e : \tau) \triangleright {}^q\exists_C A. \tau} \\
\\
\frac{\Delta; \Gamma_1; C_1; S \vdash e_1 \triangleright {}^q\exists_C A. \tau \quad \Delta; S \vdash A : C \quad \Delta; S \vdash \tau : \Omega \quad \Delta; S \vdash \tau' : \Omega}{\Delta; \Gamma_2, x \triangleright \tau; C_2, X : A; S \vdash e_2 \triangleright \tau'} \quad \frac{}{\Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash \text{unpack } e_1 \text{ as } (X : A, x : \tau) \text{ in } e_2 \triangleright \tau'}
\end{array}$$

Πλειάδες

Για την κατασκευή μίας πλειάδας απαιτείται να δοθούν το region μέσα στο οποίο θα τοποθετηθεί, το μέγεθός της, τα αρχικά της περιεχόμενα και τα δικαιώματα πρόσβασης σε κάθε θέση της.

Κατά την κατασκευή της επιστρέφεται το capability που ελέγχει την πρόσβαση στα περιεχόμενα της πλειάδας και ένας δείκτης στο πρώτα στοιχείο της. Ο δείκτης είναι unrestricted ώστε να μπορεί να διπλασιαστεί κατά βούληση. Το capability έχει qualifier ίσο ή σε υψηλότερο επίπεδο της ιεραρχίας με το κατώτερο άνω όριο των qualifiers των περιεχομένων της πλειάδας. Μ' αυτο τον τρόπο εξασφαλίζεται ότι ο διπλασιασμός του δείκτη της πλειάδας δεν μπορεί να οδηγήσει σε διπλασιασμό linear περιεχομένων της πλειάδας. Επίσης απαιτείται τα περιεχόμενα της πλειάδας να είναι τουλάχιστον αναγνώσιμα ή εγγράψιμα. Η δέσμευση χώρου για μη αναγνώσιμα ή εγγράψιμα δεδομένα οδηγεί σε κακοδιαχείριση της μνήμης.

$$\frac{\Delta; \Gamma_i; C_i; S \vdash e_i \triangleright \tau_i \quad \Delta \vdash A_i : (\text{hole} \preceq \mu_i) \quad \Delta \vdash B_i : (\tau \preceq q)}{\Delta; \vec{\Gamma}; \vec{C}; S \vdash {}^q\langle [A_0, B_0]_{\mu_0 e_0}, \dots, [A_{p-1}, B_{p-1}]_{\mu_{p-1} e_{p-1}} \rangle \triangleright \\ \mathbf{L}_{\forall_{\text{Set}} \rho : \text{Rgn.}} \mathbf{L}_{\forall_{\text{Sc}} \eta.} \mathbf{L}_{\forall_{\mathcal{C}} \eta} \text{rgn } \rho. \\ \mathbf{L}_{\exists_{\text{Set}} l : \text{Loc } \rho.} \mathbf{L}_{\exists_{\mathcal{C}} l @ \rho \rightsquigarrow \langle \mu_0 \tau_0, \dots, \mu_{p-1} \tau_{p-1} \rangle.} \mathbf{U}_{\text{tuple } \rho} l}$$

Η Δομή Ελέγχου με Συνθήκη if

Η δομή if δίνει τη δυνατότητα για εκμετάλευση στο χρόνο μεταγλώττισης υπό τη μορφή αποδίξεων, πληροφορίας που συνήθως αποκτάται κατά το χρόνο εκτέλεσης. Οι δύο κλάδοι της δομής είναι παραμετρικοί ως προς τις αποδίξεις. Ο όρος της γλώσσας τύπων A αποτελεί απόδιξη της λογικής πρότασης που αναπαρίσταται είτε με τον όρο της γλώσσας τύπων $B \text{ true}$ ή είτε με τον όρο $B \text{ false}$. Η σύνδεση ανάμεσα στην τιμή της συνθήκης e και τον τύπο του A θεμελιώνεται με τη χρήση boolean singleton τύπων.

$$\frac{\Delta \vdash B : \text{Bool} \rightarrow \text{Set} \quad \Delta; \Gamma_1; C_1; S \vdash e \triangleright \text{sbool } b \quad \Delta \vdash A : B b}{\Delta, X_1 : B \text{ true}; \Gamma_2; C_2; S \vdash e_1 \triangleright \tau \quad \Delta, X_2 : B \text{ false}; \Gamma_2; C_2; S \vdash e_2 \triangleright \tau} \\ \Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2; S \vdash \text{if}[B, A](e, X_1.e_1, X_2.e_2) \triangleright \tau$$

let!

Η δομή let! επιτρέπει την προσωρινή αλλαγή του qualifier ενός capability.

Είναι απαραίτητο να εξασφαλιστεί ότι όλα τα νέα αντίγραφα του capability X , πάνω στην οποίο εφαρμόζεται ο όρος, θα επιβιώσουν μόνο μέσα στην συντακτική εμβέλεια που ορίζει η έκφραση e_1 . Διαφορετικά η ασφάλεια του συστήματος γίνεται διάτρητη καθώς συνυπάρχουν linear και unrestricted αντίγραφα για το ίδιο capability στην έκφραση e_2 . Για να επιτευχθεί ο στόχος της ασφάλειας, ο τύπος του capability x αποκτά στο εσωτερικό της έκφρασης e_1 εκτός από διαφορετικό qualifier και ένα καινούργιο scope, ρ , που χαρακτηρίζει μοναδικά την τρέχουσα χρήση του όρου let!. Το νέο scope προστίθεται στο S μόνο κατά τον έλεγχο της έκφρασης e_1 και δεν υφίσταται κατά τον έλεγχο της έκφρασης e_2 . Αυτό έχει ως αποτέλεσμα να μην μπορούν να ελεγχθούν επιτυχώς τα unrestricted αντίγραφα που αποδούν από την εμβέλεια του e_1 . Ο έλεγχος της έκφρασης e_2 γίνεται σε διευρυμένο περιβάλλον Γ με την προσθήκη της μεταβλητής y που φέρει τον τύπο τ' της έκφρασης e_1 και το capability X με τον αρχικό της τύπο.

Αξίζει να σημειωθεί ότι δεν μπορεί να εφαρμοστεί ο όρος let! χωρίς περιορισμούς. Η αλλαγή του qualifier είναι ασφαλής μόνο όταν γίνεται από τα ανώτερα επίπεδα της ιεραρχίας προς τα κατώτερα. Διαφορετικά ένα π.χ. unrestricted capability θα μπορούσε να μετατραπεί σε linear και να χρησιμοποιηθεί στο e_1 για να αλλάξει η κατάσταση της μνήμης. Στο e_2 το capability επανέρχεται στην αρχική του μορφή και δεν είναι συνεπές με την τρέχουσα κατάσταση της μνήμης. Το ίδιο πρόβλημα μπορεί να προκύψει και κατά τη μετατροπή linear capability σε affine ακόμα και αν είναι σύμφωνη με την ασφαλή φορά μετατροπής. Η ισχύς των περιορισμών ελέγχεται μέσω αποδίξεων που ζητούνται ώστε να εφαρμοστεί όρος.

$$\frac{\Delta \vdash A : (q' \preceq q) \quad \Delta \vdash B : (q' \preceq \mathbf{R})}{\Delta; \Gamma_1; C_1, X : {}^q'_\eta \xi; S, \eta' \vdash e_1 \triangleright \tau_1 \quad \Delta; \Gamma_2, x \triangleright \tau_1; C_2, X : {}^q_\eta \xi; S \vdash e_2 \triangleright \tau_2} \\ \Delta; \Gamma_1 \oplus \Gamma_2; C_1 \oplus C_2, X : {}^q_\eta \xi; S \vdash \text{at } \eta \text{ let! } [A, B, X] \ x = e_1 \text{ in } e_2 \triangleright \tau_2$$

5.3 Παραδείγματα

Για την καλύτερη κατανόησή του συστήματος και των δυνατοτήτων του δίδεται, ως παράδειγμα, η συνάρτηση `xchange`. Η συνάρτηση ανταλλάσσει τα περιεχόμενα της θέσης i_1 της πλειάδας x_1 μεγέθους n_1 στοιχείων με τα περιεχόμενα της θέσης i_2 της πλειάδας x_2 μεγέθους n_2 στοιχείων. Η ισχύς των προδιαγραφών που πρέπει να πληρούν τα δεδομένα, που δίδονται ως παράμετροι, εξασφαλίζεται μέσω κατάλληλων αποδείξεων που δίδει ο χρήστης ως παραμέτρους της συνάρτησης.

$$\begin{aligned}
\text{xchange} & \triangleright \begin{array}{l} \mathbf{U} \forall_{\text{Set}} \rho_1 : \text{Rgn}. \mathbf{U} \forall_{\text{Set}} l_1 : \text{Loc } \rho. \\ \mathbf{U} \forall_{\text{Set}} \rho_2 : \text{Rgn}. \mathbf{U} \forall_{\text{Set}} l_2 : \text{Loc } \rho. \\ \mathbf{U} \forall_{\text{Set}} n_1 : \text{Nat}. \mathbf{U} \forall_{\text{Set}} n_2 : \text{Nat}. \\ \mathbf{U} \forall_{\text{Set}} i_1 : \text{Nat}. \mathbf{U} \forall_{\text{Set}} i_2 : \text{Nat}. \\ \mathbf{U} \forall_{\text{Set}} f_1 : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\ \mathbf{U} \forall_{\text{Set}} f_2 : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\ \mathbf{U} \forall_{\text{Set}} q_1 : \text{Qual}. \mathbf{U} \forall_{\text{Set}} q_2 : \text{Qual}. \\ \mathbf{U} \forall_{\text{Set}} q_3 : \text{Qual}. \mathbf{U} \forall_{\text{Set}} q_4 : \text{Qual}. \\ \mathbf{U} \forall_{\text{Sc}} \eta_1. \mathbf{U} \forall_{\text{Sc}} \eta_2. \\ \mathbf{U} \forall_{\text{Set}} p_1^* : (\text{ref} \preceq \text{snd}(f_1 i_1)). \\ \mathbf{U} \forall_{\text{Set}} p_2^* : (\text{ref} \preceq \text{snd}(f_2 i_2)). \\ \mathbf{U} \forall_{\text{Set}} p_3^* : (\text{compatible}(\text{fst}(f_1 i_1))(\text{fst}(f_2 i_2))). \\ \mathbf{U} \forall_{\text{Set}} p_4^* : (\text{fst}(f_1 i_1) \preceq \mathbf{U}). \\ \mathbf{U} \forall_{\text{Set}} p_5^* : (\text{fst}(f_2 i_2) \preceq \mathbf{U}). \\ \mathbf{U} \forall_{\text{Set}} p_6^* : (\mathbf{A} \preceq q_3). \\ \mathbf{U} \forall_{\text{Set}} p_7^* : (\mathbf{A} \preceq q_4). \\ \mathbf{U} \forall_{\text{Set}} p_8^* : (i_1 <_{\text{Nat}} n). \\ \mathbf{U} \forall_{\text{Set}} p_9^* : (i_2 <_{\text{Nat}} n). \\ \mathbf{U} \forall_{\mathcal{C}} \eta_1^{q_1} \text{rgn } \rho_1. \eta_1^{q_1} \forall_{\mathcal{C}} \eta_2^{q_2} \text{rgn } \rho_2. \\ \sqcup \{q_1, q_2\} \forall_{\mathcal{C}} \eta_1^{q_3} \text{typ } \rho_1 l_1 n_1 f_1. \\ \sqcup \{q_1, q_2, q_3\} \forall_{\mathcal{C}} \eta_2^{q_4} \text{typ } \rho_2 l_2 n_2 f_2. \\ \mathbf{U} \text{tuple } \rho_1 l_1 \xrightarrow[\eta_1, \eta_2]{\sqcup q_1, q_2} \mathbf{U} \text{tuple } \rho_2 l_2 \xrightarrow[\eta_1, \eta_2]{\sqcup q_1, q_2} \\ \sqcup q_3, q_4 \exists_{\mathcal{C}} \eta_3^{q_3} \text{typ } \rho_1 l_1 n_1 (\text{update } i_1 (\text{fst}(f_2 i_2) \times \text{snd}(f_1 i_1)) f_1). \\ q_4 \exists_{\mathcal{C}} \eta_4^{q_4} \text{typ } \rho_2 l_2 n_2 (\text{update } i_2 (\text{fst}(f_1 i_1) \times \text{snd}(f_2 i_2)) f_2). \\ \mathbf{U} \text{unit} \end{array} \\
& = \mathbf{U} \text{poly } \rho_1 : \text{Rgn}. \mathbf{U} \text{poly } l_1 : \text{Loc } \rho. \\
& \quad \mathbf{U} \text{poly } \rho_2 : \text{Rgn}. \mathbf{U} \text{poly } l_2 : \text{Loc } \rho. \\
& \quad \mathbf{U} \text{poly } n_1 : \text{Nat}. \mathbf{U} \text{poly } n_2 : \text{Nat}. \\
& \quad \mathbf{U} \text{poly } i_1 : \text{Nat}. \mathbf{U} \text{poly } i_2 : \text{Nat}. \\
& \quad \mathbf{U} \text{poly } f_1 : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \quad \mathbf{U} \text{poly } f_2 : \text{Nat} \rightarrow (\Omega \times \text{FMode}). \\
& \quad \mathbf{U} \text{poly } q_1 : \text{Qual}. \mathbf{U} \text{poly } q_2 : \text{Qual}. \\
& \quad \mathbf{U} \text{poly } q_3 : \text{Qual}. \mathbf{U} \text{poly } q_4 : \text{Qual}. \\
& \quad \mathbf{U} \Rightarrow \eta_1. \mathbf{U} \Rightarrow \eta_2. \\
& \quad \mathbf{U} \text{poly } p_1^* : (\text{ref} \preceq \text{snd}(f_1 0)). \\
& \quad \mathbf{U} \text{poly } p_2^* : (\text{ref} \preceq \text{snd}(f_2 0)). \\
& \quad \mathbf{U} \text{poly } p_3^* : (\text{compatible}(\text{fst}(f_1 0))(\text{fst}(f_2 0))). \\
& \quad \mathbf{U} \text{poly } p_4^* : (\text{fst}(f_1 0) \preceq \mathbf{R}). \\
& \quad \mathbf{U} \text{poly } p_5^* : (\text{fst}(f_2 0) \preceq \mathbf{R}). \\
& \quad \mathbf{U} \text{poly } p_6^* : (\mathbf{A} \preceq q_3). \\
& \quad \mathbf{U} \text{poly } p_7^* : (\mathbf{A} \preceq q_4). \\
& \quad \mathbf{U} \text{poly } p_8^* : (i_1 <_{\text{Nat}} n).
\end{aligned}$$

```


$$\begin{aligned}
& \mathbf{U}_{\mathbf{poly}} p_9^*: (i_2 <_{\mathbf{Nat}} n). \\
& \mathbf{U}_{\mathbf{poly}} X_1 :_{\eta_1}^{q_1} \mathbf{rgn} \rho_1. {}^{q_1} \mathbf{poly} X_2 :_{\eta_2}^{q_2} \mathbf{rgn} \rho_2. \\
& \sqcup_{\{q_1, q_2\}} \mathbf{poly} Y_1 :_{\perp}^{q_3} \mathbf{typ} \rho_1 l_1 n_1 f_1. \\
& \sqcup_{\{q_1, q_2, q_3\}} \mathbf{poly} Y_2 :_{\perp}^{q_4} \mathbf{typ} \rho_2 l_2 n_2 f_2. \\
& \sqcup_{\{q_1, q_2, q_3\}} \mathbf{lambda}^{\eta_1, \eta_2} \mathbf{x}_1 : \mathbf{U}_{\mathbf{tuple}} \rho_1 l_1. \sqcup_{\{q_1, q_2, q_3\}} \mathbf{lambda}^{\eta_1, \eta_2} \mathbf{x}_2 : \mathbf{U}_{\mathbf{tuple}} \rho_2 l_2. \\
& \mathbf{at} \eta_{a_1} \mathbf{let!} [(q\_pr \mathbf{U} q_1), (q\_pr \mathbf{U} \mathbf{R}), X_1] \\
& \quad y_{a_1} = \\
& \quad \mathbf{at} \eta_{a_2} \mathbf{let!} [(q\_pr \mathbf{U} q_3), (q\_pr \mathbf{U} \mathbf{R}), Y_1] \\
& \quad \quad y_{a_2} = \mathbf{read} [\rho_1] [l_1] [n_1] [f_1] [i_1] \\
& \quad \quad \quad [p_8^*] [\mathbf{fst}(\mathbf{weaken}_{\mathbf{Fmode}} p_1^*)] [p_4^*] [\eta_1] \\
& \quad \quad \quad [\perp][q_1] [q_3][X_1] [Y_1] \mathbf{x}_1 \\
& \quad \quad \mathbf{in} \\
& \quad \quad y_{a_2} \\
& \quad \mathbf{in} \\
& \quad \mathbf{at} \eta_{b_1} \mathbf{let!} [(q\_pr \mathbf{U} q_1), (q\_pr \mathbf{U} \mathbf{R}), X_2] \\
& \quad \quad y_{b_1} = \\
& \quad \quad \mathbf{at} \eta_{b_2} \mathbf{let!} [(q\_pr \mathbf{U} q_3), (q\_pr \mathbf{U} \mathbf{R}), Y_2] \\
& \quad \quad \quad y_{b_2} = \mathbf{read} [\rho_2] [l_2] [n_2] [f_2] [i_2] \\
& \quad \quad \quad \quad [p_9^*] [\mathbf{fst}(\mathbf{weaken}_{\mathbf{Fmode}} p_2^*)] [p_5^*] [\eta_2] \\
& \quad \quad \quad \quad [\perp] [q_2] [q_4][X_2] [Y_2] \mathbf{x}_2 \\
& \quad \quad \mathbf{in} \\
& \quad \quad y_{b_2} \\
& \quad \mathbf{in} \\
& \quad \quad \mathbf{unpack} (\mathbf{assign}_+ [\rho_1] [l_1] [n_1] [f_1] [(f_2 i_2)] \\
& \quad \quad \quad [p_8^*] [\mathbf{snd}(\mathbf{weaken}_{\mathbf{Fmode}} p_1^*)] \\
& \quad \quad \quad [p_4^*] [p_3^*] [\eta_1] [\perp] [q_1] [q_3] [p_6^*] [X_1] [Y_1] [x_1] [y_{a_1}] \\
& \quad \quad \mathbf{as} (Z_1 :^{q_3} \mathbf{typ} \rho_2 l_2 n_2 (\mathbf{update} i_1 (\mathbf{fst} (f_2 i_2) \times \mathbf{snd} (f_1 i_1)) f_1), \\
& \quad \quad \quad z_1 : \mathbf{U}_{\mathbf{unit}} \\
& \quad \quad ) \mathbf{in} \\
& \quad \quad \quad \mathbf{unpack} (\mathbf{assign}_+ [\rho_2] [l_2] [n_2] [f_2] [(f_1 i_1)] \\
& \quad \quad \quad \quad [p_9^*] [\mathbf{snd}(\mathbf{weaken}_{\mathbf{Fmode}} p_2^*)] \\
& \quad \quad \quad \quad [p_5^*] [p_3^*] [\eta_1] [\perp] [q_2] [q_4] [p_7^*] \\
& \quad \quad \mathbf{as} (Z_2 :^{q_3} \mathbf{typ} \rho_2 l_2 n_2 (\mathbf{update} i_2 (\mathbf{fst} (f_1 i_1) \times \mathbf{snd} (f_2 i_2)) f_2), \\
& \quad \quad \quad z_2 : \mathbf{U}_{\mathbf{unit}} \\
& \quad \quad ) \mathbf{in} \\
& \quad \quad \quad \sqcup^{q_3, q_4} \mathbf{pack} (X : \mathbf{A} \mathbf{typ} \rho_1 l_1 n_1 (\mathbf{update} i_1 (\mathbf{fst} (f_2 i_2) \times \mathbf{snd} (f_1 i_1)) f_1) = Z_1, \\
& \quad \quad \quad {}^{q_4} \mathbf{pack} (Y : \mathbf{A} \mathbf{typ} \rho_2 l_2 n_2 (\mathbf{update} i_2 (\mathbf{fst} (f_1 i_1) \times \mathbf{snd} (f_2 i_2)) f_2) = Z_2, \\
& \quad \quad \quad \langle \rangle : \mathbf{U}_{\mathbf{unit}}) : \mathbf{L}_{\exists c} \mathbf{A} \mathbf{typ} \rho_2 l_2 n_2 (\mathbf{update} i_2 (\mathbf{fst} (f_1 i_1) \times \mathbf{snd} (f_2 i_2)) f_2).
& \quad \quad \quad \mathbf{U}_{\mathbf{unit}})
\end{aligned}$$


```

Η περιγραφή των συναρτήσεων της γλώσσας τύπων, που χρησιμοποιούνται στο παράδειγμα για την κατασκευή των αποδείξεων που απαιτούνται, παραλείπεται για λόγους απλότητας.

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

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

- Σχεδιάστηκε πλήρως μία γλώσσα προγραμματισμού με substructural σύστημα τύπων με linear και unrestricted τύπους. Η γλώσσα υποστηρίζει πολυμορφισμό και προτείνει μία μέθοδο για την τυποθεωρητικά ασφαλή μετατροπή linear τύπων σε unrestricted και αντίστροφα. Ορίσθηκε η σημασιολογία της γλώσσας και διατυπώθηκε η μεταθεωρία της.
- Σχεδιάστηκε ένα σύστημα προγραμματισμού με αποδείξεις που υποστηρίζει αναφορές και ανάθεση. Στην γλώσσα τύπων του συστήματος μπορούν να εκφραστούν και να αποδειχθούν λογικές προτάσεις για την κατάσταση της μνήμης ενώ στη γλώσσα υπολογισμού μπορούν να γραφούν προγράμματα που μεταβάλουν με ασφάλεια την κατάσταση της μνήμης. Για τον σχεδιασμό του συστήματος απαιτήθηκε η κωδικοποίηση στη γλώσσα τύπων του συστήματος ενός substructural συστήματος τύπων με unrestricted, linear, affine και relevant τύπους. Η διαχείριση μνήμης γίνεται σε επίπεδο regions.
- Από τα αποτελέσματα που παρατηρήθηκαν, διαπιστώθηκε η χρησιμότητα των substructural συστημάτων τύπων για την αναπαράσταση σε τυποθεωρητικό επίπεδο των καταστάσεων της μνήμης και των μεταβολών της.
- Διαπιστώθηκε, επίσης, ότι τα substructural συστημάτων τύπων μπορούν να χρησιμοποιηθούν για το σχεδιασμό ισχυρά τυποποιημένων γλώσσων προγραμματισμού με αναφορές και ανάθεση που εγγυώνται την ασφάλεια. Ιδιαίτερα η απόδειξη της δυνατότητας ασφαλούς μετατροπής linear τύπων σε unrestricted και αντίστροφα προσδίδει στις υπό σχεδιαση γλώσσες ιδιαίτερη εκφραστικότητα και ευελιξία.
- Διαπιστώθηκε, επίσης, ότι τα substructural συστημάτων τύπων μπορούν να συνδυαστούν με επιτυχία με τον προγραμματισμό με αποδείξεις. Από το συνδιασμό μπορεί να προκύψει ένα σύστημα που παράγει πιστοποιημένο κώδικα για προγράμματα που μεταβάλουν την κατάσταση της μνήμης.

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:

- *Tυπικότητα:* 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 πρέπει να αυξάνει δυσανάλογα.
- Το σύστημα προγραμματισμού με αποδείξεις που σχεδιάστηκε εξυπηρετεί εν μέρει το μακροπρόθεσμο στόχο. Στο πλαίσιο της μελλοντικής εργασίας για την ανάπτυξη ενός τέτοιου συστήματος πιστοποίησης κώδικα, είναι αναγκαίο:
- Να αποδειχθεί η μεταθεωρία της γλώσσας υπολογισμών του συστήματος `let!`. Η απόδειξη δεν έχει ακόμα ολοκληρωθεί
 - Να οριστεί η λειτουργική σημασιολογία της γλώσσας υπολογισμών του συστήματος `LiNFlint`.
 - Να διατυπωθεί και να αποδειχθεί η μεταθεωρία της γλώσσας υπολογισμών του συστήματος `LiNFlint`.
 - Να υλοποιηθεί η γλώσσα `Túpaw` και η γλώσσα υπολογισμών του συστήματος `LiNFlint` στη γλώσσα `Gallina` του συστήματος `Coq`.
 - Να μελετηθεί η δυνατότητα χρήσης των βιβλιοθηκών με κωδικοποιημένες μαθηματικές Θεωρίες του συστήματος `Coq`. Η κωδικοποίηση των θεωριών από τα μαθηματικά είναι απαραίτητη για την αποδεικτική διαδικασία προτάσεων που χρησιμοποιούνται στις προδιαγραφές ασφαλείας.
 - Να οριστεί και να υλοποιηθεί κατάλληλα μια γλώσσα προγραμματισμού υψηλού επιπέδου, η οποία θα εκμεταλλεύεται τη γλώσσα `Túpaw` και θα μεταφράζεται στη γλώσσα υπολογισμών.

Βιβλιογραφία

- [Ahme05] A. Ahmed, M. Fluet and G. Morrisett, “A step-indexed model of substructural state”, in *ICFP ’05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pp. 78–91, New York, NY, USA, 2005, ACM Press.
- [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.
- [Crar99] K. Crary, D. Walker and G. Morrisett, “Typed memory management in a calculus of capabilities”, in *POPL ’99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 262–275, New York, NY, USA, 1999, ACM Press.
- [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.
- [Flue06] M. Fluet, G. Morrisett and A. J. Ahmed, “Linear Regions Are All You Need.”, in *ESOP*, pp. 7–21, 2006.
- [Gira90] J.-Y. Girard, “La Logique Linéaire”, *Pour La Science, Edition Française de ‘Scientific American’*, vol. 150, pp. 74–85, April 1990.
- [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.
- [Haya91] S. Hayashi, “Singleton, Union and Intersection Types for Program Extraction.”, in *TACS*, pp. 701–730, 1991.
- [Howa69] W. A. Howard, “To H.B. Curry: The Formulae-as-types Notion of Construction”, in J. Hindley and J. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1969.
- [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.
- [Morr05] G. Morrisett, A. J. Ahmed and M. Fluet, “L³: A Linear Language with Locations.”, in *TLCA*, pp. 293–307, 2005.

- [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.
- [Oder92] M. Odersky, “Observers for Linear Types”, in B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pp. 390–407, New York, 1992, Springer-Verlag.
- [Pier02] B. C. Pierce, *Types and programming languages*, MIT Press, Cambridge, MA, USA, 2002.
- [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.
- [Smit00] F. Smith, D. Walker and G. Morrisett, “Alias Types”, *Lecture Notes in Computer Science*, vol. 1782, pp. 366–??, 2000.
- [Toft94] M. Tofte and J.-P. Talpin, “Implementation of the typed call-by-value λ -calculus using a stack of regions”, in *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 188–201, New York, NY, USA, 1994, ACM Press.
- [Toft97] M. Tofte and J.-P. Talpin, “Region-Based Memory Management”, *Information and Computation*, 1997.
- [Toft98] M. Tofte and L. Birkedal, “A region inference algorithm”, *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 4, pp. 724–767, 1998.
- [Wadl90] P. Wadler, “Linear types can change the world!”, in M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pp. 347–359, North Holland, 1990.
- [Walk01a] D. Walker and G. Morrisett, “Alias Types for Recursive Data Structures”, *Lecture Notes in Computer Science*, vol. 2071, pp. 177–??, 2001.
- [Walk01b] D. Walker and K. Watkins, “On regions and linear types (extended abstract)”, in *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pp. 181–192, New York, NY, USA, 2001, ACM Press.
- [Walk05] D. Walker, “Substructural Type Systems”, in B. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pp. 3–43, MIT Press, 2005.
- [Wern94] B. Werner, *Une Théorie des Constructions Inductives*, Ph.D. thesis, Université Paris VII, Paris, France, May 1994.
- [Xi99] H. Xi and F. Pfenning, “Dependent types in practical programming”, in *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 214–227, New York, NY, USA, 1999, ACM Press.

- [Xi04] H. Xi, “Applied Type System (extended abstract)”, in *post-workshop Proceedings of TYPES 2003*, pp. 394–408, Springer-Verlag LNCS 3085, 2004.
- [Zhu05] D. Zhu and H. Xi, “Safe Programming with Pointers through Stateful Views”, in *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pp. 83–97, Long Beach, CA, January 2005, Springer-Verlag LNCS vol. 3350.