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

Συστήματα Τύπων με Γραμμικά Δικαιώματα Πρόσβασης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΗ ΒΑΖΟΥ

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

Αθήνα, Οκτώβριος 2010

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

Συστήματα Τύπων με Γραμμικά Δικαιώματα Πρόσβασης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΗ ΒΑΖΟΥ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22η Οκτωβρίου 2010.

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

Αθήνα, Οκτώβριος 2010

....

Νίκη Βάζου

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

Copyright © Νίκη Βάζου, 2010.
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 των συστημάτων τύπων με γραμμικά δικαιώματα πρόσβασης και η εφαρμογή τους για την ασφάλεια τύπων σε δύο γλώσσες προγραμματισμού. Η πρώτη είναι μια ακολουθιακή γλώσσα με αναφορές, ισχυρή ανάθεση (strong update) και ρητή αποδέσμευση, όπου το σύστημα τύπων εγγυάται την ασφάλεια της πρόσβασης στη μνήμη. Η δεύτερη είναι μια γλώσσα ταυτόχρονου προγραμματισμού, όπου το σύστημα τύπων εγγυάται αφενός την ασφάλεια της πρόσβασης στη μνήμη, αφετέρου την απουσία συνθηκών ανταγωνισμού (data races).

Στο πλαίσιο της εργασίας, αρχικά τροποποιήθηκε η γλώσσα let! ώστε να υποστηρίζει γραμμικά δικαιώματα πρόσβασης (linear capabilities). Η γλώσσα let!, όπως και η τροποποίησή της, είναι συναρτησιακές γλώσσες με ισχυρό σύστημα τύπων που υποστηρίζουν αναφορές. Η εισαγωγή των αναφορών σε γλώσσες με ισχυρό σύστημα τύπων δημιουργεί πρόβλημα, αφού λόγω του aliasing δεν μπορούν να γίνουν ασφαλώς η ισχυρή ανάθεση (strong update) και η αποδέσμευση της μνήμης. Στη γλώσσα let! ο έλεγχος του aliasing γίνεται αντιμετωπίζοντας τις αναφορές με γραμμικό τρόπο, ενώ η έκφραση του let! επιτρέπει ελεγχόμενη μετατροπή των linear τιμών σε unrestricted. Στην ακολουθιακή γλώσσα που παρουσιάζεται σε αυτήν την εργασία τα γραμμικά αντικείμενα που ελέγχουν το aliasing είναι τα δικαιώματα πρόσβασης. Επεκτείνοντας τη γλώσσα ώστε να υποστηρίζει ταυτόχρονο προγραμματισμό, τα ίδια δικαιώματα πρόσβασης μπορούν να χρησιμοποιηθούν για την αποφυγή των συνθηκών ανταγωνισμού.

Για την ακολουθιακή και την ταυτόχρονη γλώσσα παρουσιάζονται η σύνταξή τους, το σύστημα τύπων τους και η λειτουργική σημασιολογία τους. Επίσης, έχουν κατασκευαστεί και για τις δύο ελεγκτές τύπων και διερμηνείς.

Λέξεις κλειδιά

Γραμμικά συστήματα τύπων, δικαιώματα πρόσβασης, letbang, ασφάλεια μνήμης, ταυτόχρονος προγραμματισμός, συνθήκες ανταγωνισμού.

Abstract

The purpose of this diploma dissertation is to study type systems with linear capabilities and to apply them to obtain type safety for two programming languages. The first is a sequential language with references, strong update and explicit deallocation, where the type system guarantees memory safety. The second is a concurrent language, where the type system guarantees memory safety, as well as absence of data races.

As part of this dissertation, initially the language let! was modified so as to support linear capabilities. The language let! and its modification are strongly typed functional languages that support references. The introduction of references in languages with a strong type system is problematic because, due to aliasing, strong updates and explicit memory deallocation can not be done safely. In the language let! the tracking of aliases is achieved by treating references in a linear way, while the let! construct enables a temporary conversion from linear values to unrestricted. In the sequential language presented in this work, the linear objects that are used to track aliases are capabilities. By extending the language so as to support concurrent programming, capabilities can be used to prevent data races.

Both for the sequential and the concurrent language, we present their syntax, type system and operational semantics. We also provide for each one a type checker and an interpreter.

Key words

Linear type systems, capabilities, letbang, memory safety, parallel programming, data races.

Ευχαριστίες

Ευχαριστώ τους γονείς μου και την αδερφή μου για την στήριξη που μου παρείχαν καθ'όλη τη διάρκεια των σπουδών μου και τον καθηγητή μου κ. Παπασπύρου και τον διδακτορικό του σπουδαστή Μιχάλη Παπακυριάκου για την αγάπη που μου μετέδωσαν για την Επιστήμη των Υπολογιστών και το χρόνο που διέθεσαν για τη συγκεκριμένη εργασία.

Νίκη Βάζου,

Αθήνα, 22η Οκτωβρίου 2010

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-3-10, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2010.

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

Περιεχόμενα

Περίληψη . . . 5

Abstract . . . 7

Ευχαριστίες . . . 9

Περιεχόμενα . . . 11

Σχήματα . . . 13

1. Εισαγωγή . . . 15
- 1.1 Σκοπός της Εργασίας . . . 15
- 1.2 Παρακίνηση για την Εργασία . . . 15
- 1.3 Σύνοψη της εργασίας . . . 16

2. Γραμμικά Συστήματα Τύπων . . . 17
- 2.1 Γλώσσες Προγραμματισμού της οικογένειας ML και Αναφορές . . . 17
- 2.2 Γραμμικά Συστήματα Τύπων . . . 18
  - 2.2.1 Substructural Συστήματα Τύπων . . . 18
  - 2.2.2 Μεικτά Συστήματα Τύπων για Χειρισμό Αναφορών . . . 20
- 2.3 Μετατροπή Linear Τιμής σε Unrestricted . . . 20
  - 2.3.1 Το αρχικό let! . . . 21
  - 2.3.2 Τύποι Observer . . . 22
  - 2.3.3 let! με scopes . . . 22

3. Δικαιώματα Πρόσβασης . . . 25
- 3.1 Τι είναι τα Δικαιώματα Πρόσβασης . . . 25
- 3.2 Δικαιώματα Πρόσβασης στις Γλώσσες Προγραμματισμού . . . 25
  - 3.2.1 Ο λογισμός των regions . . . 25
  - 3.2.2 Ο λογισμός των Capabilities . . . 26
- 3.3 Γραμμικά Δικαιώματα Πρόσβασης . . . 28
  - 3.3.1 Linear Regions . . . 28
  - 3.3.2 L^3 . . . 29

4. Εφαρμογή σε ιδιότητες ασφάλειας μνήμης . . . 31
- 4.1 Σύνταξη . . . 31
  - 4.1.1 References . . . 31
  - 4.1.2 Qualifiers - Scopes . . . 32
  - 4.1.3 Τύποι . . . 32
  - 4.1.4 Εκφράσεις . . . 32
- 4.2 Κανόνες Συστήματος Τύπων . . . 32
  - 4.2.1 Περιβάλλοντα . . . 33
  - 4.2.2 Χρήσιμες Συναρτήσεις . . . 34

4.2.3 Οι κανόνες . . . 34
4.3 Λειτουργική Σημασιολογία . . . 37
4.3.1 Περιβάλλοντα . . . 37
4.3.2 Οι κανόνες . . . 38
4.4 Επεκτάσεις . . . 39
4.4.1 Σύνταξη . . . 39
4.4.2 Κανόνες Συστήματος Τύπων . . . 39
4.4.3 Λειτουργική Σημασιολογία . . . 39
4.5 Η υλοποίηση . . . 40
4.5.1 Ελεγκτής Τύπων . . . 42
4.5.2 Διερμηνέας . . . 43
4.6 Παραδείγματα . . . 43
4.6.1 Αφελής προσπάθεια διαφυγής scope . . . 43
4.6.2 Προσπάθεια διαφυγής scope μέσω συνάρτησης . . . 44
4.6.3 Προσπάθεια διαφυγής μέσω Ανάθεσης στην Μνήμη . . . 44
4.6.4 Ανάθεση σε θέση μνήμης γραμμικό περιεχόμενο . . . 44
4.6.5 Συνάρτηση για ανάγνωση περιεχομένου θέσης μνήμης . . . 45
4.6.6 Fibonacci . . . 45

5. Εφαρμογή σε γλώσσες ταυτόχρονου προγραμματισμού . . . 47
5.1 Εισαγωγή . . . 47
5.2 Σύνταξη . . . 47
5.2.1 Οι καταστάσεις s . . . 48
5.2.2 Οι qualifiers q . . . 48
5.2.3 Οι εκφράσεις e . . . 48
5.3 Κανόνες Συστήματος Τύπων . . . 49
5.3.1 Διάσπαση Περιβάλλοντος Γ . . . 49
5.3.2 Οι κανόνες . . . 50
5.4 Λειτουργική Σημασιολογία . . . 53
5.4.1 Το περιβάλλον t . . . 53
5.4.2 Οι κανόνες . . . 53
5.5 Η Υλοποίηση . . . 56
5.5.1 Ελεγκτής Τύπων . . . 56
5.5.2 Διερμηνέας . . . 59
5.6 Παραδείγματα . . . 59
5.6.1 Συνθήκη Ανταγωνισμού: Εγγραφή με Εγγραφή . . . 60
5.6.2 Συνθήκη Ανταγωνισμού: Εγγραφή με Ανάγνωση . . . 60
5.6.3 Συνθήκες Ανταγωνισμού και Συναρτήσεις . . . 60
5.6.4 Αποφυγή “ψεύτικου” αδιεξόδου . . . 61
5.6.5 Συγχρονισμένη Εγγραφή στην Ίδια Θέση Μνήμης . . . 62
5.6.6 Fibonacci . . . 63

6. Συμπεράσματα . . . 63
6.1 Συνεισφορά . . . 63
6.2 Μελλοντική Έρευνα . . . 63

Βιβλιογραφία . . . 65

Σχήματα

2.1 Ιεραρχία των Substructural Συστημάτων Τύπων 19
2.2 Κανόνες Τύπων για Αναφορές σε Μεικτό Σύστημα Τύπων 20

4.1 Σύνταξη 31
4.2 Διάσπαση Περιβάλλοντος Γ 33
4.3 Έλεγχος εγκυρότητας scope 33
4.4 Έλεγχος εγκυρότητας reference 34
4.5 Κατηγόρημα ελέγχου qualifier 34
4.6 Ελεύθερες reference μεταβλητές (FRV) 34
4.7 Κανόνες τύπων: Βασικές Εκφράσεις 35
4.8 Κανόνες τύπων: Σειριακή Εκτέλεση 35
4.9 Κανόνες τύπων: Αφαίρεση και Εφαρμογή 35
4.10 Κανόνες τύπων: Ζεύγη και Πακέτα 36
4.11 Κανόνες Τύπων: Capabillities και References 36
4.12 Κανόνες Τύπων: Διαχείριση References 36
4.13 Κανόνες τύπων: let! 37
4.14 Συνάρτηση αναζήτησης S 37
4.15 Κανόνες αποτίμησης: Τιμή 38
4.16 Κανόνες αποτίμησης: Ζεύγη και Πακέτα 38
4.17 Κανόνες αποτίμησης: Εφαρμογή - Σειριακή Αποτίμηση 39
4.18 Κανόνες αποτίμησης: Διαχείριση Μνήμης 40
4.19 Κανόνες αποτίμησης: Let! 41
4.20 Επεκτάσεις: Σύνταξη 41
4.21 Επεκτάσεις: Κανόνες Συστήματος Τύπων 41
4.22 Επεκτάσεις: Κανόνες Αποτίμησης 42

5.1 Σύνταξη 48
5.2 Σχέση των καταστάσεων 48
5.3 Διάσπαση Περιβάλλοντος Γ 49
5.4 Διάσπαση Περιβάλλοντος Γ για παράλληλες υποεκφράσεις 50
5.5 Έλεγχος εγκυρότητας cl 50
5.6 Κανόνες τύπων: Βασικές Εκφράσεις 51
5.7 Κανόνες Τύπων: Παράλληλος Υπολογισμός 51
5.8 Κανόνες τύπων: Αφαίρεση 51
5.9 Κανόνες Τύπων: Αλλαγή Κατάστασης Capability 52
5.10 Κανόνες τύπων: let! 52
5.11 Κανόνες Τύπων: Ανάγνωση Περιεχομένου Θέσης Μνήμης 53
5.12 Κανόνες Τύπων: Εγγραφή Περιεχομένου Θέσης Μνήμης 53
5.13 Κανόνες αποτίμησης: Αρχικός Κανόνας 54
5.14 Κανόνες αποτίμησης: Παράλληλος Υπολογισμός 54
5.15 Κανόνες αποτίμησης: Αλλαγή Κατάστασης σε T 55
5.16 Κανόνες αποτίμησης: Αλλαγή Κατάστασης σε R 56
5.17 Κανόνες αποτίμησης: Let! 57

5.18 Κανόνες αποτίμησης: Ανάγνωση Περιεχομένου Θέσης Μνήμης 57
5.19 Κανόνες αποτίμησης: Εγγραφή Περιεχομένου Θέσης Μνήμης 58

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός της Εργασίας

Σκοπός της εργασίας είναι η μελέτη των συστημάτων τύπων με γραμμικά δικαιώματα πρόσβασης και η εφαρμογή τους για την ασφάλεια τύπων σε δύο γλώσσες προγραμματισμού. Η πρώτη είναι μια ακολουθιακή γλώσσα με αναφορές, ισχυρή ανάθεση (strong update) και ρητή αποδέσμευση, όπου το σύστημα τύπων εγγυάται την ασφάλεια της πρόσβασης στη μνήμη. Η δεύτερη είναι μια γλώσσα ταυτόχρονου προγραμματισμού, όπου το σύστημα τύπων εγγυάται αφενός την ασφάλεια της πρόσβασης στη μνήμη, αφετέρου την απουσία συνθηκών ανταγωνισμού (data races).

Στο πλαίσιο της εργασίας, αρχικά τροποποιήθηκε η γλώσσα let! ώστε να υποστηρίζει γραμμικά δικαιώματα πρόσβασης (linear capabilities). Η γλώσσα let!, όπως και η τροποποίησή της, είναι συναρτησιακές γλώσσες με ισχυρό σύστημα τύπων που υποστηρίζουν αναφορές. Η εισαγωγή των αναφορών σε γλώσσες με ισχυρό σύστημα τύπων δημιουργεί πρόβλημα, αφού λόγω του aliasing δεν μπορούν να γίνουν ασφαλώς η ισχυρή ανάθεση (strong update) και η αποδέσμευση της μνήμης. Στη γλώσσα let! ο έλεγχος του aliasing γίνεται αντιμετωπίζοντας τις αναφορές με γραμμικό τρόπο, ενώ η έκφραση του let! επιτρέπει ελεγχόμενη μετατροπή των linear τιμών σε unrestricted. Στην ακολουθιακή γλώσσα που παρουσιάζεται σε αυτήν την εργασία τα γραμμικά αντικείμενα που ελέγχουν το aliasing είναι τα δικαιώματα πρόσβασης. Επεκτείνοντας τη γλώσσα ώστε να υποστηρίζει ταυτόχρονο προγραμματισμό, τα ίδια δικαιώματα πρόσβασης μπορούν να χρησιμοποιηθούν για την αποφυγή των συνθηκών ανταγωνισμού.

Για την ακολουθιακή και την ταυτόχρονη γλώσσα παρουσιάζονται η σύνταξή τους, το σύστημα τύπων τους και η λειτουργική σημασιολογία τους. Επίσης, έχουν κατασκευαστεί και για τις δύο ελεγκτές τύπων και διερμηνείς.

1.2 Παρακίνηση για την Εργασία

Οι γλώσσες με ισχυρό σύστημα τύπων υπερτερούν έναντι αυτών με ασθενές αφού μπορούν στατικά να εξασφαλίσουν ότι τα προγράμματα δεν θα εμφανίσουν σφάλμα κατά τον χρόνο εκτέλεσης. Το γεγονός αυτό είναι πολύ σημαντικό στις μέρες μας που οι απαιτήσεις για την ασφάλεια του κώδικα είναι ιδιαίτερα αυξημένες.

Οι προσπάθειες για τον ορισμό ισχυρών συστημάτων τύπων επικεντρώθηκαν αρχικά σε γλώσσες με συναρτησιακά χαρακτηριστικά. Οι γλώσσες αυτές όμως δεν υποστηρίζουν λειτουργίες όπως η ισχυρή ανάθεση ή η αποδέσμευση της μνήμης. Οι λειτουργίες αυτές είναι πολύ χρήσιμες στον προγραμματιστή, αφού η ισχυρή ανάθεση δίνει την δυνατότητα άμεσης διαμόρφωσης της μνήμης και ελέγχου της ροής δεδομένων στα διάφορα σημεία του προγράμματος. Ταυτόχρονα, η ρητή αποδέσμευση της μνήμης και η εξασφάλιση πλήρους αποδέσμευσης της μνήμης που χρησιμοποιείται από το πρόγραμμα καθιστά περιττό τον συλλέκτη σκουπιδιών που καταναλώνει πόρους κατά τον χρόνο εκτέλεσης του προγράμματος.

Τα γραμμικά συστήματα τύπων προσφέρουν την δυνατότητα παρακολούθησης της ροής των δεδομένων του προγράμματος. Άρα παρακολουθώντας τις αναφορές ή ακόμα καλύτερα τα δικαιώματα πρόσβασης, αφού είναι οντότητες άχρηστες κατά τον χρόνο εκτέλεσης, μπορεί να παρακολουθεί-

ται και η κατάσταση της μνήμης και οι μεταβολές της. Συνεπώς τα γραμμικά συστήματα καθιστούν δυνατή την κατασκευή γλωσσών προγραμματισμού με ισχυρό σύστημα τύπων που υποστηρίζουν ισχυρές αναθέσεις και ρητή αποδέσμευση μνήμης.

Το σημαντικότερο κίνητρο για την εργασία όμως υπήρξε η δημιουργία μίας ταυτόχρονης γλώσσας, το σύστημα τύπων της οποίας θα εγγυάται την απουσία συνθηκών ανταγωνισμού. Είναι γεγονός ότι στις μέρες μας το ενδιαφέρον για τον ταυτόχρονο προγραμματισμό είναι μεγάλο, κάτι που οφείλεται εν μέρει στην διαθεσιμότητα φθηνών επεξεργαστών και εν μέρει στην εξάπλωση των εφαρμογών γραφικών, πολυμέσων και Διαδικτύου, οι οποίες όλες αντιπροσωπεύονται φυσικά από ταυτόχρονα νήματα ελέγχου.

Ο ταυτόχρονος προγραμματισμός εισάγει έναν αριθμό προβλημάτων όπως οι συνθήκες ανταγωνισμού και τα αδιέξοδα. Η συνθήκη ανταγωνισμού είναι μία κατάσταση κατά την οποία δύο νήματα επεξεργάζονται ένα δεδομένο ταυτόχρονα και χωρίς συγχρονισμό. Οι συνθήκες ανταγωνισμού είναι συχνά και ύπουλα λάθη στις ταυτόχρονες γλώσσες και συχνά οδηγούνε σε μη-ντετερμινισμό και λάθος αποτελέσματα. Επιπλέον, αφού εξαρτώνται από τον χρόνο είναι δύσκολο να εντοπιστούν από τον προγραμματιστή.

Ο στατικός εντοπισμός συνθηκών ανταγωνισμού, σε γλώσσες με ασθενές σύστημα τύπων είναι δύσκολος. Συγκεκριμένα έχει δειχθεί ότι ο στατικός εντοπισμός σε προγράμματα που χρησιμοποιούν πολλαπλούς σηματοφορείς είναι NP-hard [Netz90] και συνεπώς μια αποδοτική λύση δεν είναι δυνατόν να βρεθεί.

Η χρήση γραμμικών δικαιωμάτων πρόσβασης σε γλώσσες ταυτόχρονου προγραμματισμού επιτρέπει τόσο την σωστή διαχείριση της μνήμης όσο και την στατική ανίχνευση συνθηκών ανταγωνισμού γεγονός που διευκολύνει την δουλειά των προγραμματιστών.

1.3 Σύνοψη της εργασίας

Η διπλωματική εργασία έχει την ακόλουθη δομή:

Κεφάλαιο 2: Εισαγωγή στα γραμμικά συστήματα τύπων. Παρουσίαση των κυριότερων συστημάτων και της γλώσσας let!.

Κεφάλαιο 3: Παρουσίαση της θεωρίας των δικαιωμάτων πρόσβασης. Παρουσίαση των κυριότερων συστημάτων.

Κεφάλαιο 4: Ορισμός και περιγραφή μιας ακολουθιακής γλώσσας με γραμμικά δικαιώματα πρόσβασης, με σκοπό την εξασφάλιση των ιδιοτήτων ασφάλειας μνήμης.

Κεφάλαιο 5: Ορισμός και περιγραφή μιας γλώσσας ταυτόχρονου προγραμματισμού με γραμμικά δικαιώματα πρόσβασης, με σκοπό την αποτροπή συνθηκών ανταγωνισμού.

Κεφάλαιο 6: Συμπεράσματα και προτάσεις μελλοντικής έρευνας.

Κεφάλαιο 2

Γραμμικά Συστήματα Τύπων

2.1 Γλώσσες Προγραμματισμού της οικογένειας ML και Αναφορές

Τα συστήματα τύπων είναι συντακτικές μέθοδοι πολυωνυμικού χρόνου για την ταξινόμηση των τμημάτων ενός προγράμματος ανάλογα με τις τιμές που αυτά υπολογίζουν, με σκοπό να αποδείξουν την απουσία ορισμένων ανεπιθύμητων συμπεριφορών κατά την εκτέλεσή του [Pier02]. Συνεπώς, οι γλώσσες προγραμματισμού που χαρακτηρίζονται από ισχυρό σύστημα τύπων εγγυώνται την ασφάλεια των προγραμμάτων ως προς τα χαρακτηριστικά που ελέγχει το σύστημα τύπων: Όταν ένα πρόγραμμα περάσει από τον στατικό έλεγχο τύπων (type-checker) είναι εγγυημένη η ασφάλειά του, δηλαδή η αποδεδειγμένη απουσία ορισμένων ανεπιθύμητων συμπεριφορών κατά την εκτέλεσή του.

Οι γλώσσες προγραμματισμού της οικογένειας ML (ML family programming languages, όπως Standard ML (SML), Caml και άλλες) χαρακτηρίζονται από ισχυρά συστήματα τύπων. Η εισαγωγή των αναφορών σε αυτές τις γλώσσες είναι χρή: Στην ML οι περισσότερες δομές δεδομένων, αφού δημιουργηθούν και αρχικοποιηθούν είναι immutable δηλαδή, δεν μπορούν να αλλάξουν, να ενημερωθούν ή να αποθηκευτούν. Κατά την υλοποίηση αλγορίθμων όμως, η ανάθεση νέας τιμής σε κάποια δομή (destructive update) είναι απαραίτητη και ο μόνος τρόπος να επιτευχθεί είναι με την εισαγωγή των αναφορών. Έτσι στην ML έχουν εισαχθεί οι αναφορές και συγκεκριμένα υποστηρίζονται:

- δέσμευση μνήμης : **let** r = **ref** 7 **in** ... που επιστρέφει έναν δείκτη σε μία νέα θέση μνήμης
- ανάθεση : r := 8 ... που αλλάζει το περιεχόμενο της θέσης μνήμης που δείχνει ο r (destructive update)
- ανάγνωση ... !r ... που ανακαλεί το περιεχόμενο της θέσης μνήμης που δείχνει ο r

Η εισαγωγή αυτή όμως δημιουργεί και κάποια προβλήματα.

Το κυριότερο πρόβλημα είναι ότι η ML δεν υποστηρίζει αποδέσμευση μνήμης. Αν η αποδέσμευση μνήμης υποστηριζόταν, τότε θα μπορούσε να γραφτεί ένα πρόγραμμα σαν το ακόλουθο:

Παράδειγμα 1
let r = **ref** 7 **in**
let a = r **in**
  free r; !a

Στο πρόγραμμα αυτό γίνεται ανάγνωση από έναν ξεκρέμαστο δείκτη (dangling pointer) που οδηγεί σε σφάλμα κατά τον χρόνο εκτέλεσης. Άρα η ρητή αποδέσμευση δεν μπορεί να υποστηριχτεί από την ML και πρέπει να χρησιμοποιηθεί garbage collector που μειώνει την απόδοση των προγραμμάτων.

Ένα άλλο πρόβλημα είναι ότι η ML δεν υποστηρίζει ισχυρές αναθέσεις (strong updates). Οι ισχυρές αναθέσεις επιτρέπουν την ανάθεση στο περιεχόμενο μίας θέσης μνήμης τιμής διαφορετικού τύπου, σε αντίθεση με τις ασθενείς αναθέσεις (weak updates) που δεν μπορούν να αλλάξουν τον τύπο του περιεχομένου μιας θέσης μνήμης. Η ύπαρξη των ασθενών αναθέσεων είναι αναγκαία όταν οι αναφορές αλληλεπιδρούν με singleton types.

Οι singleton types εισήχθηκαν από τον και Hayashi [Haya91] και χρησιμοποιήθηκαν πρώτη φορά από τον Xi σε προγραμματισμό με dependent types [Xi98]. Ο τύπος μίας μεταβλητής καθορίζει ένα

σύνολο επιτρεπτών τιμών που μπορεί η μεταβλητή να πάρει. Για παράδειγμα αν $x : Int$ τότε μπορεί να πάρει οποιαδήποτε ακέραια τιμή. Στους singeton τύπους το σύνολο αυτό αποτελείται από ακριβώς ένα στοιχείο. Για παράδειγμα αν $x : sint\ \widehat{0}$ τότε μπορεί να μόνο την τιμή 0.

Αν οι ισχυρές αναθέσεις υποστηρίζονταν, τότε θα μπορούσε να γραφτεί ένα πρόγραμμα σαν το ακόλουθο:

Παράδειγμα 2
let r = **ref** 7 **in** $(*r : sint\ \widehat{7}*)$
…r := 42 $(*42 : sint\ \widehat{42}*)$

Το οποίο οδηγεί σε σφάλμα τύπου αφού ο τύπος του περιεχομένου της θέσης μνήμης r είναι $sint\ \widehat{7}$.

Ακόμα και αν οι singeton types δεν υπήρχαν, η ισχυρή ανάθεση θα μπορούσε να οδηγήσει σε σφάλμα κατά τον χρόνο εκτέλεσης σε ένα πρόγραμμα σαν το ακόλουθο:

Παράδειγμα 3
let r = **ref** 7 **in**
let a = r **in**
a:=true; !r+5

Στο πρόγραμμα αυτό απαιτείται η πρόσθεση δύο μη συμβατών τιμών και εμφανίζεται σφάλμα κατά τον χρόνο εκτέλεσης.

Όπως παρατηρείται το στοιχείο της γλώσσας που εισάγει και τα δύο προβλήματα είναι το aliasing: ένας δείκτης σε μία θέση μνήμης μπορεί να ανατεθεί σε άλλες μεταβλητές και να βρίσκεται σε πολλά σημεία του προγράμματος. Η αντιμετώπιση των αναφορών σαν γραμμικά αντικείμενα μπορεί να λύσει αυτό το πρόβλημα. Γραμμικά συστήματα τύπων χρησιμοποιήθηκαν για ασφαλείς αναθέσεις τιμών [Wadl90, Oder92], για την υλοποίηση μιας διαλέκτου της Lisp χωρίς συλλέκτη σκουπιδιών [Bake92] και για την υλοποίηση μιας ασφαλούς διαλέκτου της C με unique pointers, την Cyclone [Hick03, Swam06].

2.2 Γραμμικά Συστήματα Τύπων

Με την εισαγωγή της γραμμικής λογικής από τον Girard [Gira87] και τον Curry-Howard ισομορφισμό [Grif90] μπορεί να προκύψει άμεσα ένα καθαρά γραμμικό σύστημα τύπων. Σε ένα τέτοιο σύστημα τύπων οι τιμές πρέπει να χρησιμοποιηθούν ακριβώς μία φορά, δεν μπορούν ούτε να διπλασιαστούν ούτε να αγνοηθούν. Άρα, το aliasing δεν είναι πλέον δυνατό. Συνεπώς η ρητή αποδέσμευση και η ισχυρή ανάθεση είναι πλέον ασφαλείς.

Το μειονέκτημα των καθαρά γραμμικών συστημάτων είναι ότι κάθε τιμή που διαβάζεται αυτόματα καταναλώνεται. Το γεγονός αυτό περιορίζει την γλώσσα περισσότερο από ότι είναι επιθυμητό. Για παράδειγμα, είναι ασφαλές να υπάρχουν περισσότερες από μία αναφορές σε μία θέση μνήμης αν αυτές χρησιμοποιούνται για ανάγνωση από την θέση μνήμης. Μάλιστα οι πολλαπλές αναφορές σε μία θέση μνήμης δίνουν την δυνατότητα πρόσβασης σε περιεχόμενα της μνήμης από πολλά σημεία του προγράμματος παράλληλα, ενώ η έλλειψή τους επιβάλει δεντρική πρόσβαση των δεδομένων, αφού σε κάθε έκφραση που περιέχει περισσότερες από μία υποεκφράσεις κάθε γραμμικό δεδομένο θα είναι διαθέσιμο μόνο σε μία από αυτές.

Για τον λόγο αυτόν κρίθηκε σκόπιμο να οριστούν συστήματα τύπων που υποστηρίζουν γραμμικούς αλλά και μη γραμμικούς τύπους. Τα συστήματα αυτά ονομάζονται substructural συστήματα [Walk05, Ahme05] και παρουσιάζονται στην συνέχεια.

2.2.1 Substructural Συστήματα Τύπων

Τα συστήματα τύπων μπορούν να χαρακτηριστούν από τρεις ιδιότητες ανάλογα με τον τρόπο που χειρίζονται το περιβάλλον Γ με βάση το οποίο γίνεται ο έλεγχος τύπου μίας έκφρασης e.

exchange: Αν μπορεί να αποδοθεί ένας τύπος σε έναν όρο δεδομένου του περιβάλλοντος Γ, τότε μπορεί να αποδοθεί ο ίδιος τύπος στον όρο δεδομένης κάθε μετάθεσης του περιβάλλοντος Γ.

weakening: Αν μπορεί να αποδοθεί ένας τύπος σε έναν όρο δεδομένου του περιβάλλοντος Γ, τότε μπορεί να αποδοθεί ο ίδιος τύπος στον όρο δεδομένης κάθε ασφαλούς επέκτασης του περιβάλλοντος Γ.

contraction: Αν μπορεί να αποδοθεί ένας τύπος σε έναν όρο e δεδομένου του περιβάλλοντος Γ που περιέχει δύο όμοιες υποθέσεις ($x_1 : t$ και $x_2 : t$), τότε μπορεί να αποδοθεί ο ίδιος τύπος στον όρο $e[x_1 \mapsto x][x_2 \mapsto x]$ δεδομένου του περιβάλλοντος Γ' που προκύπτει από το Γ αν οι όμοιες υποθέσεις $x_1 : t$ και $x_2 : t$ αντικατασταθούν από την $x : t$.

Ένα substructural σύστημα τύπων είναι ένα σύστημα τύπων στο οποίο μία ή περισσότερες από τις δομικές ιδιότητες δεν ισχύουν. Συγκεκριμένα μπορούν να οριστούν τα ακόλουθα συστήματα τύπων:

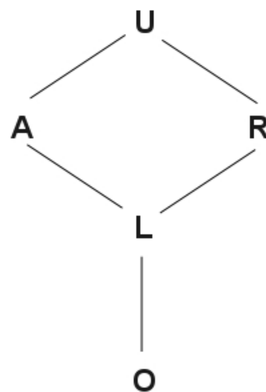
linear: Διασφαλίζουν ότι κάθε μεταβλητή χρησιμοποιείται ακριβώς μία φορά, επιτρέποντας την exchange αλλά όχι την weakening ή την contraction.

affine: Διασφαλίζουν ότι κάθε μεταβλητή χρησιμοποιείται το πολύ μία φορά, επιτρέποντας την exchange και την weakening αλλά όχι την contraction.

relevant: Διασφαλίζουν ότι κάθε μεταβλητή χρησιμοποιείται τουλάχιστον μία φορά, επιτρέποντας την exchange και την contraction αλλά όχι την weakening.

ordered: Διασφαλίζουν ότι όλες οι μεταβλητές χρησιμοποιούνται ακριβώς μία φορά και με την σειρά με την οποία εισήχθηκαν στον περιβάλλον, μη επιτρέποντας καμία δομική ιδιότητα.

Τα συστήματα αυτά μπορούν να αναπαρασταθούν σε μία ιεραρχική δομή όπως φαίνεται στο σχήμα 2.1, όπου τα U, A, R, L, O αντιστοιχούν στα Unrestricted, Affine, Relevant, Linear και Ordered συστήματα τύπων. Τα συστήματα που βρίσκονται χαμηλότερα στο σχήμα (τα ordered συστήματα) δεν έχουν δομικές ιδιότητες. Όσο ανεβαίνουμε στο διάγραμμα προστίθενται δομικές ιδιότητες.



Σχήμα 2.1: Ιεραρχία των Substructural Συστημάτων Τύπων

Το σχήμα μπορεί να θεωρηθεί σαν μία σχέση μεταξύ των συστημάτων. Λέμε ότι το σύστημα q_1 είναι πιο περιοριστικό από το σύστημα q_2 και γράφουμε $q_1 \sqsubseteq q_2$ όταν το σύστημα q_1 υποστηρίζει λιγότερες δομικές ιδιότητες από το σύστημα q_2. Ειδικότερα ισχύει $L \sqsubseteq U$.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{new}\ e :^{L} \text{Ref}\ \tau}\ \text{new} \qquad \frac{\Gamma \vdash e :^{U} \text{Ref}^{U} \tau}{\Gamma \vdash \text{deref}\ e :^{U} \tau}\ \text{deref} \qquad \frac{\Gamma \vdash e :^{L} \text{Ref}\ \tau}{\Gamma \vdash \text{free}\ e : \tau}\ \text{free}$$

$$\frac{\Gamma \vdash e_1 :^{U} \text{Ref}^{L} \tau \qquad \Gamma \vdash e_2 :^{L} \tau}{\Gamma \vdash e_1 {:=:} e_2 :^{L} \tau}\ \text{weakSwap}$$

$$\frac{\Gamma \vdash e_1 :^{L} \text{Ref}^{L} \tau_1 \qquad \Gamma \vdash e_2 :^{L} \tau_2}{\Gamma \vdash e_1 {:=:}^{s} e_2 :^{L} \tau_1}\ \text{strongSwap}$$

$$\frac{\Gamma \vdash e_1 :^{U} \text{Ref}^{U} \tau \qquad \Gamma \vdash e_2 :^{U} \tau}{\Gamma \vdash e_1 {:=} e_2 : \text{Unit}}\ \text{weakAssign}$$

$$\frac{\Gamma \vdash e_1 :^{L} \text{Ref}^{U} \tau_1 \qquad \Gamma \vdash e_2 :^{U} \tau_2}{\Gamma \vdash e_1 {:=}^{s} e_2 : \text{Unit}}\ \text{strongAssign}$$

Σχήμα 2.2: Κανόνες Τύπων για Αναφορές σε Μεικτό Σύστημα Τύπων

2.2.2 Μεικτά Συστήματα Τύπων για Χειρισμό Αναφορών

Συνηθίζεται, στις γλώσσες που υποστηρίζουν αναφορές να επιλέγεται η χρήση συστημάτων τύπων που υποστηρίζουν linear και unrestricted τιμές. Η ύπαρξη affine και relevant τιμών θα μετρίαζε κάποιους περιορισμούς που θέτει η γραμμικότητα, αλλά για απλοποίηση επιλέγουμε να τις αγνοήσουμε. Σημειώνουμε ότι σε συστήματα που συνυπάρχουν πολλά είδη τιμών η διάκριση γίνεται από έναν qualifier που επισημειώνεται στον τύπο κάθε τιμής. Στην συνέχεια θα θεωρούμε ότι ο qualifier μπορεί να πάρει τιμές L ή U για να σηματοδοτεί linear ή unrestricted τιμές αντίστοιχα.

Όπως προαναφέρθηκε η ισχυρή ανάθεση και η αποδέσμευση μνήμης επιτρέπονται μόνο όταν εξασφαλίζεται ότι δεν υπάρχουν aliases, άρα μόνο σε linear αναφορές. Για τον λόγο αυτό επιβάλλεται όταν δημιουργείται μία αναφορά να είναι linear. Αντίθετα, είναι βολικό η ανάγνωση από την μνήμη και η ασθενής ανάθεση (weak update) να επιτρέπεται σε unrestricted αναφορές (επιτρέποντας έτσι την παράλληλη χρήση των αναφορών σε πολλά μέρη του προγράμματος).

Ένα θέμα που πρέπει να εξασφαλιστεί στο σύστημα τύπων που υποστηρίζει linear και unrestricted τιμές είναι ότι αν κάποια linear τιμή είναι αποθηκευμένη σε κάποια θέση μνήμης θα χρησιμοποιηθεί ακριβώς μία φορά. Για τον λόγο αυτό επιτρέπουμε την ανάγνωση και την ανάθεση (ισχυρής ή ασθενής) σε θέσεις μνήμης με unrestricted περιεχόμενο. Αν επιτρεπόταν ανάγνωση σε θέσεις μνήμης με linear περιεχόμενο, η linear τιμή θα μπορούσε να ανακληθεί από την μνήμη πολλές φορές, ενώ αν επιτρεπόταν ανάθεση σε θέσεις μνήμης με linear περιεχόμενο, η linear τιμή που προϋπήρχε στην μνήμη δεν θα χρησιμοποιούταν καμία φορά. Για ανάγνωση και ανάθεση τιμών σε θέσεις μνήμης που περιέχουν linear τιμές χρησιμοποιούμε το νέο τελεστή της ανταλλαγής e_1 :=: e_2 [Bake92]. Ο τελεστής αυτός αναθέτει μία τιμή στη θέση μνήμης, ενώ ταυτόχρονα επιστρέφει την παλιά της τιμή. Για τον ίδιο λόγο απαιτείται ο τελεστής αποδέσμευσης μνήμης free να επιστρέφει το περιεχόμενο της θέσης μνήμης που αποδεσμεύει έτσι ώστε αν είναι γραμμικό να μην αγνοηθεί.

Οι απαιτήσεις αυτές συνοψίζονται στο σχήμα 2.2 που παρουσιάζονται οι κανόνες τύπων ως προς τις αναφορές ενός συστήματος που υποστηρίζει linear και unrestricted αναφορές. Σημειώνεται ότι στο σχήμα αυτό ο qualifier εμφανίζεται μόνο όπου είναι απαραίτητος.

2.3 Μετατροπή Linear Τιμής σε Unrestricted

Στην ενότητα 2.2.2 είδαμε την ανάγκη συνύπαρξης linear και unrestricted αναφορών. Σε αυτήν την ενότητα θα παρουσιαστούν οι τρόποι που έχουν προταθεί για την αλλαγή του qualifier μιας τιμή από linear σε unrestricted καταλήγοντας στην γλώσσα $let!$ με scopes.

2.3.1 Το αρχικό let!

Η αρχική μορφή του $let!$ προτάθηκε από τον Wadler [Wadl90] και έχει σύνταξη let! (x) $y = e_1$ in e_2. Η έκφραση αυτή αποτιμάται σαν μία συμβατική let έκφραση: αρχικά αποτιμάται η e_1 και ανατίθεται στην y το αποτέλεσμα και έπειτα αποτιμάται η e_2 στο περιβάλλον που έχει επεκταθεί με την μεταβλητή y. Η διαφορά της από το συμβατικό let είναι ότι η αρχικά linear μεταβλητή x μέσα στην έκφραση e_1 μετατρέπεται σε unrestricted και επανέρχεται σε linear στην έκφραση e_2. Για καλύτερη κατανόηση ακολουθεί ένα παράδειγμα στο οποίο η αναφορά r μετατρέπεται από linear σε unrestricted ώστε να χρησιμοποιηθεί πολλαπλές φορές:

Παράδειγμα 4
let r = new 5 in
let! (r)
  $y = r$:=(deref $r + 1$)
in
  free $r; y$

Για να είναι το σύστημα ασφαλές τίθενται τρεις περιορισμοί: Ο πρώτος περιορισμός είναι ότι η αποτίμηση της e_1 πρέπει να ολοκληρωθεί πριν αρχίσει η αποίμηση της e_2 (πρέπει να γίνει hyperstrict αποτίμηση). Για παράδειγμα, αν η e_1 επιστρέψει μία λίστα, τότε όλα τα στοιχεία της λίστας πρέπει να αποτιμηθούν. Αυτό εξασφαλίζει ότι όλες οι αναφορές στην x έχουν ελευθερωθεί πριν αρχίσει η αποτίμηση της e_2.

Ο δεύτερος περιορισμός αφορά τους τύπους της x και της e_1 και ελέγχεται αν είναι "αμοιβαία αποκλειόμενοι". Δεν επιτρέπεται ο τύπος της e_1 (ή κάποιο συστατικό του) να είναι ίδιος με τον τύπο της x (ή κάποιο συστατικό του). Ο έλεγχος γίνεται με αναδρομικό τρόπο και εγγυάται ότι κανένα linear συστατικό του x δεν θα διαφύγει μέσω της y. Συγκεκριμένα μέσω του ελέγχου εξασφαλίζεται ότι κανένα linear συστατικό του τύπου του x δεν έχει αντίστοιχο unrestricted συστατικό στον τύπο της e_1. Ο περιορισμός αυτός έχει πολύ σημαντικό κόστος αφού δεν επιτρέπει την εμφάνιση στον τύπο της e_1 κανενός unrestricted συστατικού που να αποτελεί συστατικό του τύπου της x.

Τέλος, ο τρίτος περιορισμός απαγορεύει στον τύπο της e_1 να περιέχει τύπους συναρτήσεων. Οι συναρτήσεις απαγορεύονται αφού κάθε όρος συνάρτησης μπορεί να περιέχει το x ή κάποιο συστατικό του σαν ελεύθερη μεταβλητή, όπως στο ακόλουθο παράδειγμα:

Παράδειγμα 5
let r = new 5 in
let! (r)
  $y = \lambda x$: Unit .deref r
in
  free $r; y$unit

Παρά τους περιορισμούς, όταν το σύστημα του Wadler εφαρμοστεί σε γλώσσα με αναφορές (το σύστημα κατασκευάστηκε για γλώσσα με πίνακες, όπου η εγγραφή σε ένα στοιχείο του πίνακα οδηγούσε σε έναν νέο πίνακα) μπορεί να εμφανιστεί το ακόλουθο πρόβλημα: η μεταβλητή που έχει μετατραπεί από linear σε unrestricted μπορεί να διαφύγει από την έκφραση e_1 με την αποθήκευσή της σε κάποια θέση μνήμης, όπως στο παράδειγμα που ακολουθεί:

Παράδειγμα 6
($*$έστω $p :^U$ Ref (^{U}Ref Int)$*$)
let r = new 5 in
let! (r)
  $y = p := r$
in
  free r; deref (deref p)

Στο παράδειγμα αυτό αρχικά αποδεσμεύεται η θέση μνήμης r και έπειτα γίνεται ανάγνωση από αυτή, άρα θα προκύψει σφάλμα κατά τον χρόνο εκτέλεσης.

2.3.2 Τύποι Observer

Ο Odersky θέλοντας να αναιρέσει τον περιορισμό των ”αμοιβαίως αποκλειόμενων τύπων” που περιορίζει σημαντικά το σύστημα του Walder, εισήγαγε την έννοια του observer τύπου [Oder92]. Σημειώνεται ότι το σύστημα του Odersky είναι πολυμορφικό άρα κατά τον έλεγχο συμβατότητας των τύπων των x και e_1 η έννοια της ισότητας αντικαθίσταται από την έννοια της ενοποίησης.

Συγκεκριμένα, στο σύστημά του υπάρχουν πλέον τρεις qualifiers, o linear, o unrestricted και ο observer. Ο όρος let! στο σύστημα του Odersky συντάσσεται let! $x = e_1$ in e_2 και η κύρια διαφορά του από το συμβατικό let είναι ότι όλες οι linear μεταβλητές μέσα στην έκφραση e_1 μετατρέπονται σε observer ενώ στην e_2 επανέρχονται σε linear. Η απαίτηση των ”αμοιβαίως αποκλειόμενων τύπων” αντικαθίσταται από την εξής απλούστερη: ο τύπος της x δεν μπορεί να περιέχει τον qualifier observer.

Στο σύστημα του Odersky (όπως και στου Wadler) για να γίνει εγγραφή σε κάποια θέση μνήμης απαιτείται linear αναφορά, έτσι η διαφυγή ενός observer πλέον τύπου μέσω της αποθήκευσης στην μνήμη παύει να υφίσταται. Αυτό συμβαίνει αφού το let! του Odersky μετατρέπει κατά την αποτίμηση της e_1 όλες τις linear τιμές σε observer, σε αντίθεση με αυτό του Walder που άλλαζε την κατάσταση μίας μόνο τιμής. Όμως, η αδυναμία συνύπαρξης linear και observer τιμών, περιορίζει σημαντικά τις δυνατότητες της γλώσσας αφού μέσα στην e_1 δεν μπορεί να υπάρξει καμία έκφραση εγγραφής ή αποδέσμευσης μνήμης.

2.3.3 let! με scopes

Οι Papakyriakou-Papaspyrou στο σύστημά τους [Papa10] αναιρούν τους περιορισμούς των προηγούμενων συστημάτων εισάγοντας την έννοια του scope. Συγκεκριμένα, οι τύποι πλέον επισημειώνονται όχι μόνο με τον qualifier αλλά και με ένα scope που μπορεί να θεωρηθεί σαν ένα δικαίωμα χρήσης κάθε τιμής.

Η σύνταξη του όρου let! στο σύστημα των Papakyriakou-Papaspyrou γίνεται πλέον at η let! $(x = e_1)$ then $y = e_2$ in e. Κατά τον υπολογισμό, αρχικά πρέπει να αποτιμηθεί η έκφραση e_1 σε μία γραμμική τιμή, η οποία ανατίθεται στη μεταβλητή x. Έπειτα υπολογίζεται η έκφραση e_2, χρησιμοποιώντας την μεταβλητή x σαν unrestricted, και το αποτελεσμά του υπολογισμού ανατίθεται στην μεταβλητή y. Τέλος, υπολογίζεται η έκφραση e χρησιμοποιώντας την μεταβλητή y και την μεταβλητή x στην αρχική linear μορφή.

Η μετατροπή της μεταβλητής x από linear σε unrestricted συνοδεύεται από την μετατροπή του scope της σε η, το οποίο είναι έγκυρο μόνο μέσα στην έκφραση e_1. Κάθε έκφραση let! έχει το δικό της μοναδικό scope. Οι τιμές με scope η είναι έγκυρες μόνο κατά την αποτίμηση της e_1 και δεν μπορούν να χρησιμοποιηθούν πουθενά αλλού στο πρόγραμμα. Γενικά, ένας όρος της γλώσσας είναι έγκυρος μόνο αν το scope του είναι έγκυρο, κάτι που μπορεί να ελεγχθεί στατικά. Για να διασφαλιστεί η μοναδικότητα του scope κάθε let! χωρίς να περιορίζεται η γλώσσα από ελέγχους μοναδικότητας, κατά την αποτίμηση και τον έλεγχο τύπων το scope η αντικαθίσταται με ένα καινούριο μοναδικό (fresh) scope ρ.

Για να κατανοηθεί πως εξασφαλίζεται ότι η τιμή που έχει μετατραπεί σε unrestricted δεν μπορεί να δραπετεύσει ακολουθεί ένα αφελές παράδειγμα:

Παράδειγμα 7

at η let! $(r = \text{new } 7)$ then
  $y = r$ $(*r :^{U}_{\eta} \text{ Ref Int}*)$
in
  free r; $(*r :^{L} \text{ Ref Int}*)$
  deref y $(*y :^{U}_{\eta} \text{ Ref Int}*)$

Ο έλεγχος τύπων σε αυτό το πρόγραμμα αποτυγχάνει, γιατί το scope της μεταβλητής y, που είναι η, δεν είναι έγκυρο κατά την ανάγνωσή της, ανεξάρτητα από το αν η r έχει αποδεσμευτεί ή όχι.

Στο σύστημα του Wadler μία μεταβλητή που έχει μετατραπεί σε unrestriced μπορεί να διαφύγει με την αποθήκευσή της στην μνήμη. Με την εισαγωγή των scopes, η διαφυγή αυτή είναι εφικτή, όταν

όμως η μεταβλητή αυτή που έχει αποθηκευτεί στην μνήμη χρησιμοποιηθεί ο έλεγχος θα τερματίσει με σφάλμα, αφού το scope της δεν θα είναι έγκυρο. Αυτό γίνεται περισσότερο κατανοητό στο επόμενο παράδειγμα το οποίο είναι το αντίστοιχο του παραδείγματος 6:

Παράδειγμα 8
(*έστω p: $^{U}\mathrm{Ref}\,^{U}_{\eta'}(\mathrm{Ref\ Int})$*)
at η let! $(r = \text{new } 5)$ then
  $y = p := r$ $\qquad (*r :^{U}_{\eta} \text{Ref Int}*)$
in
  free r;
  deref (deref p) $\qquad (*\text{deref } p :^{U}_{\eta} \text{Ref Int}*)$

Ο έλεγχος τύπων σε αυτό το πρόγραμμα αποτυγχάνει, γιατί το scope που εμφανίζεται κατά την ανάγνωση του περιεχομένου της p, που είναι η, δεν είναι έγκυρο.

Το πρόβλημα της διαφυγής μιας μεταβλητής που έχει μετατραπεί σε unrestricted μέσα από το σώμα μιας συνάρτησης αντιμετωπίζεται με έναν πιο σύνθετο τρόπο. Για να μην περιοριστεί η έκφραση e_2 από το να είναι συνάρτηση ή να χρησιμοποιεί συναρτήσεις και αναφορές με οποιονδήποτε τρόπο, επιλέγεται κάθε τύπος συνάρτησης να επισημειώνεται με την λίστα των scopes που χρησιμοποιεί το σώμα της. Κατά την εφαρμογή μιας συνάρτησης ελέγχεται αν όλα τα scopes της λίστας είναι έγκυρα και αν δεν είναι ο έλεγχος αποτυγχάνει. Αυτό γίνεται περισσότερο κατανοητό από το ακόλουθο παράδειγμα, που είναι το αντίστοιχο του παραδείγματος 5:

Παράδειγμα 9
at η let! $(r = \text{new } 7)$ then
  $y = \lambda u$: Unit.deref r $\qquad (*y : \text{Unit} \xrightarrow{\eta} \text{Int}*)$
in
  free $r; y$ unit

Ο έλεγχος τύπων σε αυτό το πρόγραμμα αποτυγχάνει, γιατί το scope που εμφανίζεται στην λίστα scopes του τύπου της συνάρτησης κατά την εφαρμογή, που είναι η, δεν είναι έγκυρο.

Το σύστημα των Papakyriakou-Papaspyrou δεν χρειάζεται τους περιορισμούς του Wadler και επιτρέπει την συνύπαρξη των μεταβλητών που έχουν μετατραπεί από linear σε unrestricted με τις linear μεταβλητές, σε αντίθεση με το σύστημα του Odersky. Παρ'όλα αυτά πρέπει να τεθεί ένας περιορισμός (που υπάρχει και στα δύο άλλα συστήματα). Δεν μπορεί να επιτραπεί να μετατρέπονται οι linear συναρτήσεις σε unresricted. Μια συνάρτηση είναι linear όταν στο κλείσιμο της χρησιμοποιεί linear τιμές. Η μετατροπή της σε unrestricted θα επέτρεπε unrestricted πρόσβαση στις τιμές αυτές, αφού θα επέτρεπε μέσω της εφαρμογής της συνάρτησης πολλές ή και καμία χρήσεις της linear αυτής τιμής.

Κεφάλαιο 3

Δικαιώματα Πρόσβασης

3.1 Τι είναι τα Δικαιώματα Πρόσβασης

Το δικαίωμα πρόσβασης (capability) που είναι γνωστό και ως κλειδί (key), σύμφωνα με τον ορισμό της wikipedia, είναι ένα μεταδόσιμο (communicable) και ανεπίδεκτο πλαστογράφησης (unforgeable) τεκμήριο εξουσίας. Αναφέρεται σε ένα αντικείμενο και περιέχει ένα σύνολο δικαιωμάτων πρόσβασης για το αντικείμενο αυτό.

Πιο συγκεκριμένα, όταν κάποιος u_1 έχει στην κατοχή του ένα δικαίωμα πρόσβασης c για κάποιο αντικείμενο x μπορεί να το μεταβιβάσει στον u_2 επιτρέποντάς του να έχει τα δικαιώματα πρόσβασης που δίνει το c πάνω στο x. Η πράξη αυτή θα στερούσε από τον u_1 κάθε δικαίωμα πρόσβασης στο x, για τον λόγο αυτό συνηθίζεται να επιτρέπεται αντιγραφή των δικαιωμάτων πρόσβασης. Έτσι, ο u_1 θα μπορούσε να αντιγράψει το c και να μεταβιβάσει στον u_2 το αντίγραφο αυτό. Επιπλέον, είναι δυνατόν να υπάρχουν πολλά διαφορετικά δικαιώματα πρόσβασης για το αντικείμενο x. Για παράδειγμα, το c_1 να παρέχει δικαίωμα ανάγνωσης του x και το c_2 ανάγνωσης και γραφής. Τέλος, κάθε δικαίωμα πρόσβασης πρέπει να είναι ανεπίδεκτο πλαστογράφησης, διαφορετικά τα δικαιώματα πρόσβασης δεν θα είχαν λόγο ύπαρξης.

Η έννοια των capabilities εισήχθηκε από τους Dennis & Van Horn [Denn66] για τον έλεγχο της πρόσβασης των πόρων σε υπολογιστικά συστήματα που υποστηρίζουν ”πολυπρογραμματισμό” (multiprogrammed computer systems). Το 1981 κατασκευάστηκε το πρώτο λειτουργικό σύστημα που χρησιμοποιούσε δικαιώματα πρόσβασης, το Hydra [Wulf81] και έκτοτε ακολούθησαν πολλά λειτουργικά συστήματα βασιζόμενα σε δικαιώματα πρόσβασης (capability-based).

3.2 Δικαιώματα Πρόσβασης στις Γλώσσες Προγραμματισμού

Στις γλώσσες προγραμματισμού τα δικαιώματα πρόσβασης μπορούν να χρησιμοποιηθούν ώστε να εξασφαλιστεί στατικά η ασφαλής αποδέσμευση μνήμης και οι ασφαλείς ισχυρές αναθέσεις, αφού παρέχουν δυνατότητα ελέγχου του aliasing.

Η πρώτη εφαρμογή των δικαιωμάτων πρόσβασης έγινε στην ενδιάμεση γλώσσα Capability Calculus [Walk00], που υποστηρίζει region-based διαχείριση μνήμης, έχει ασφαλές σύστημα τύπων και μπορεί άμεσα να μεταγλωττιστεί σε assemply γλώσσα με τύπους.

3.2.1 Ο λογισμός των regions

Με τον λογισμό των regions επιτρέπεται η ragion-based διαχείρηση της μνήμης, δηλαδή η μνήμη διαχωρίζεται σε περιοχές (regions) με ίδιο χρόνο ζωής. Η ομαδοποίηση των δεδομένων σε περιοχές (region) και η μαζική αποδέσμευσή τους οδηγεί σε καλύτερη χρήση του ελεύθερου χώρου της μνήμης ακόμα και σε γλώσσες με χαλαρά συστήματα τύπων όπως η C.

Ο λογισμός των regions, η απόδειξη της συνέπειας και της ασφάλειάς του και η δημιουργία αλγορίθμου που επιτρέπει τον αυτόματο συμπερασμό τους οφείλεται στους Toefte και Talpin [Toft94, Toft97, Toft98].

Σημαντικό μειονέκτημα του λογισμού των region είναι ότι η σύνταξη της γλώσσας επιβάλει την δέσμευση και την αποδέσμευση της μνήμης με μία μόνο εντολή. Συγκεκριμένα, η έκφραση:

$$\text{letregion } r \text{ in } e \text{ end}$$

καθορίζει την διάρκεια ζωής της περιοχής r. Η μνήμη για την περιοχή δεσμεύεται όταν εκτελείται η υποέκφραση e μιας έκφρασης letregion και αποδεσμεύεται όταν η υποέκφραση αυτή τελειώσει. Η σύνταξη αυτή, που επιβάλλεται για να είναι ασφαλές το σύστημα, οδηγεί σε μιας LIFO ταξινόμηση των χρόνων ζωής των περιοχών και συνεπώς σε μη αποδοτική διαχείριση της μνήμης σε πολλές περιπτώσεις.

Σαν παράδειγμα δίνεται ο ορισμός της αναδρομικής συνάρτησης count που δέχεται σαν όρισμα έναν αριθμό και μετράει από αυτόν έως το μηδέν:

Παράδειγμα 10

letregion r_1 in
letregion r_2 in
  letrec $count[r, r_{count}]$ at $r_1(n : (\text{Int}, r)) =$
    if $n = 0$
      then ()
      else $count[r, r_{count}]$ at $r_{count}(< n - 1 >$ at $r)$
  in
  $count[r_2, r_1]$ at $r_{count}(< 10 >$ at $r_2)$
end
end

Η συνάρτηση count αποθηκεύεται στην περιοχή r_1 και αποθηκεύει τα ενδιάμεσα αποτελέσματά στην r_2 που δέχεται σαν όρισμα.

3.2.2 Ο λογισμός των Capabilities

Στον λογισμό των Capabilities παρέχεται η δυνατότητα ρητής δέσμευσης και αποδέσμευσης περιοχών μνήμης με διαφορετικές εντολές αναιρώντας έτσι τον περιορισμό της αυστηρά LIFO διάταξης στον χρόνο ζωής των περιοχών. Σημειώνεται ότι στον λογισμό των Capabilities χρησιμοποιείται συνεχής μέθοδος περάσματος (continuation-passing style CPS), δηλαδή κάθε συνάρτηση δεν μπορεί να επιστρέψει τιμή αλλά μπορεί είτε να τερματίσει το πρόγραμμα είτε να καλέσει κάποια άλλη συνάρτηση, η οποία αποτελεί το υπόλοιπο πρόγραμμα. Ο περιορισμός αυτός διευκολύνει τον εντοπισμό των capabilities στο σύστημα.

Η δέσμευση μίας περιοχής μνήμης γίνεται με την έκφραση:

$$\text{newrgn } \rho, x$$

που δεσμεύει μια νέα περιοχή και αναθέτει στην ρ το όνομά της και στην x τον χειριστή της (handler). Ο χειριστής της περιοχής είναι απαραίτητος για να αναφερθούμε στην περιοχή όταν γίνεται ανάγνωση, εγγραφή ή αποδέσμευση σε αυτήν και χρειάζεται και κατά τον χρόνο εκτέλεσης. Αντίθετα, τα ονόματα χρειάζονται κατά τον έλεγχο ασφάλειας και μπορούν να διαγραφούν στον χρόνο εκτέλεσης.

Η αποδέσμευση μιας περιοχής μνήμης γίνεται με την έκφραση:

$$\text{freergn } v$$

όπου v ο χειριστής της περιοχής. Για να είναι ασφαλής η αποδέσμευση πρέπει να εξασφαλιστεί η μοναδικότητα της region, ότι δηλαδή δεν υπάρχουν aliases της. Για τον λόγο αυτό, ορίζονται τα capabilities (C) σαν σύνολα επισημειωμένων ονομάτων περιοχών. Συγκεκριμένα, το capability $\{r^+\}$ παρέχει δικαίωμα πρόσβασης στην περιοχή r, ενώ το το capability $\{r^1\}$ παρέχει εκτός από το δικαίωμα πρόσβασης την πληροφορία ότι το όνομα r είναι μοναδικό, άρα η αντίστοιχη περιοχή μπορεί να αποδεσμευτεί. Σημειώνεται ότι $\{r^+\} = \{r^+, r^+\}$ αφού το πρώτο παρέχει όλα τα δικαιώματα που παρέχει

και το δεύτερο, ενώ το σύνολο $\{r^1, r^1\}$ δεν μπορεί να υπάρξει αφού ο δείκτης 1 εγγυάται μοναδικότητα.

Για να παρακολουθούνται τα capabilities κατά τον έλεγχο τύπων διατηρείται ένα περιβάλλον C με όλα τα έγκυρα capabilities, δηλαδή κάθε φορά που δεσμεύεται μια περιοχή r στο C προστίθεται το r^1 ενώ όταν αποδεσμεύεται αφαιρείται. Για να αποφευχθεί ο συμπερασμός περιοχών (region inference), που μπορεί να γίνει αλγοριθμικά [Toft98], τα capabilities πρέπει να εμφανίζονται στον ορισμό και τον τύπο των συναρτήσεων, που παίρνουν αντίστοιχα τις μορφές:

$$f[\Delta](C, x_1 : \tau_1, \dots, x_n : \tau_n).e$$
$$\forall[\Delta].(C, x_1 : \tau_1, \dots, x_n : \tau_n) \to 0 \text{ at } r$$

όπου το Δ απαιτείται για την αντιστοίχηση μεταβλητών κατασκευαστών με kinds λόγω πολυμορφισμού σε τύπους, περιοχές και capabilities, το 0 σαν αποτέλεσμα της συνάρτησης δείχνει τον CPS τρόπο ελέγχου και το "at r" δείχνει την περιοχή που έχει αποθηκευτεί η συνάρτηση.

Με την δέσμευση μιας περιοχής παρέχεται η capability $\{r^1\}$, για να είναι δυνατή η παραγωγή της $\{r^+\}$ παρέχεται η έννοια του υποσυνόλου capability. Για παράδειγμα $\{r^1\} \leq \{r+\}$, αφού η πρώτη παρέχει όλα τα δικαιώματα που παρέχει και η δεύτερη. Έτσι κατά την κλήση μιας συνάρτησης αντί του συνόλου C μπορεί να παρέχεται κάποιο υποσύνολο αυτού. Για παράδειγμα στην κλήση κάποιας συνάρτησης με τύπο $\forall[\rho_1 : \text{Rgn}, \rho_2 : \text{Rgn}].(\{\rho_1^+, \rho_2^+\}, \dots) \to 0 \text{ at } r'$ μπορεί να παρέχεται το capability $\{r^1\}$, αφού $\{r^1\} \leq \{r^+\} = \{r^+, r^+\}$. Ο ορισμός του υποσυνόλου capability απαγορεύει το να αγνοηθούν περιοχές, για παράδειγμα το capability $\{r_1^+, r_2^+\}$ δεν είναι υποσύνολο του $\{r_1^+\}$ επιβάλλοντας ρητή αποδέσμευση μνήμης όταν μια περιοχή δεν θα ξαναχρησιμοποιηθεί και καθορίζοντας ποια συνάρτηση πρέπει να αποδεσμεύσει μνήμη. Για παράδειγμα μία συνάρτηση με τύπο

$$\forall[\rho : \text{Rgn}, \epsilon : \text{Cap}].((\rho^1, \epsilon), \rho \text{ handle}, (\epsilon) \to 0 \text{ at } r') \to 0 \text{ at } r$$

επιβάλλεται να απελευθερώσει την περιοχή ρ, αφού δεν περνάει το αντίστοιχο capability στην συνάρτηση που θα καλέσει. Με τον τρόπο αυτό και επιβάλλοντας καμία capability να μην είναι διαθέσιμη κατά τον τερματισμό, το σύστημα δεν επιτρέπει διαρροές μνήμης: κάθε πρόγραμμα πρέπει να αποδεσμεύσει όλη την μνήμη που χρησιμοποίησε πριν τερματίσει.

Για να είναι δυνατή η επαναφορά ενός capability που έχει διπλασιαστεί, σε μοναδικό, χρησιμοποιείται η δεσμευμένη ποσοτικοποίηση (bounded quantification). Για παράδειγμα όταν κληθεί η συνάρτηση με τύπο $\forall[\rho_1 : \text{Rgn}, \rho_2 : \text{Rgn}].(\{\rho_1^+, \rho_2^+\}, \dots) \to 0 \text{ at } r'$ με capability κάποιο $\{r^1\}$ τότε η μοναδικότητα του r χάνεται. Με την δεσμευμένη ποσοτικοποίηση ο τύπος της συνάρτησης γίνεται $\forall[\rho_1 : \text{Rgn}, \rho_2 : \text{Rgn}, \epsilon \leq \{\rho_1^+, \rho_2^+\}].(\epsilon, \dots) \to 0 \text{ at } r'$ και η μεταβλητή ϵ παρέχει την πληροφορία για την μοναδικότητα της r.

Για την καλύτερη κατανόηση του λογισμού των capabilities παρέχεται το αντίστοιχο του παραδείγματος 10:

Παράδειγμα 11

let newrgn ρ_1, x_{ρ_1} in
let newrgn ρ_2, x_{ρ_2} in (*capability held is $\{\rho_1^1, \rho_2^1\}$*)
let count =
  (fix count
    $[\rho : \text{Rgn}, \rho_{count} : \text{Rgn}, \epsilon \leq \{\rho_1^+, \rho^+, \rho_{count}^+\}]$
    $(\epsilon, x_\rho : \rho \text{ handle}, x :< \text{Int} > \text{ at } \rho, k : (\epsilon) \to 0 \text{ at } \rho_{count})$.
        (*capability held is $\epsilon \leq \{\rho_1^+, \rho^+, \rho_{count}^+\}$*)
    let $n = \pi_0(x)$ in (*having ρ^+*)
    if $n = 0$
      then $k()$ (*having ρ_{cont}^+*)
      else
        let $n' = n - 1$ in
        let $x' =< n' >$ at x_ρ in (*having ρ^+*)

count $[\rho, \rho_{count}, \epsilon](x_\rho, x', k)$ (*having ρ_1^{+}*)
) at x_{ρ_1} in
let ten=$< 10 >$ at x_{ρ_2} in
let cont =
($\lambda(\{\rho_1^1, \rho_2^1\})$.
let freergn x_{ρ_1} in (*having ρ_1^1*)
let freergn x_{ρ_2} in (*having ρ_2^1*)
halt 0
) at x_{ρ_2}
in
count$[\rho_2, \rho_2, \{\rho_1^1, \rho_2^1\}](x_{\rho_2}$,ten,cont)

Στο παράδειγμα αυτό δεσμεύονται δύο περιοχές, οι ρ_1 και ρ_2. Η συνάρτηση $count$ δέχεται σαν όρισμα ένα σύνολο από capabilities που είναι υπερσύνολο του $\{\rho_1^+, \rho^+, \rho_{count}^+\}$, όπου ρ η περιοχή που αποθηκεύονται τα ενδιάμεσα αποτελέσματα της συνάρτησης και ρ_{count} η περιοχή που είναι αποθηκευμένη η συνάρτηση που θα καλέσει η $count$. Τέλος, η συνάρτηση $cont$ δέχεται σαν όρισμα δύο μοναδικά capabilities και απελευθερώνει τις αντίστοιχες περιοχές. Παρατηρούμε ότι η συνάρτηση $count$ δεν απαιτεί μοναδικότητα για κάποιο capability όμως όταν καλείται με μοναδικά capabilities διατηρεί την πληροφορία της μοναδικότητας ώστε να μπορεί να την παρέχει στην συνάρτηση που θα καλέσει. Στο παράδειγμα η συνάρτηση αυτή είναι η $cont$ που απαιτεί μοναδικά capabilities.

3.3 Γραμμικά Δικαιώματα Πρόσβασης

Οι Fluet , Morrisett και Ahmed χειρίστηκαν τα capabilities σαν πρώτης τάξης εκφράσεις της γλώσσας με γραμμική συμπεριφορά. Με τον τρόπο αυτό μπόρεσαν να αποτρέψουν (- ελέγξουν) το aliasing καθιστώντας δυνατή την ασφαλή ισχυρή ανάθεση στην μνήμη και την ασφαλή αποδέσμευση μνήμης.

Η εισαγωγή των γραμμικών capalities σε σύστημα που έχει σαν βάση τον λογισμό των regions οδήγησε στην γλώσσα λ^{rgnUL}, ενώ έχοντας σαν βάση του Alias Types [Smit99] οδήγησε στην L [3]. Οι δύο αυτές γλώσσες παρουσιάζονται στη συνέχεια.

3.3.1 Linear Regions

Όπως προαναφέρθηκε, το σύστημα των Tofte-Talpin, ο λογισμός των regions διαχειρίζεται σωστά την μνήμη, αφού κάθε πρόγραμμα αποδεσμεύει όλες τις περιοχές που χρησιμοποιεί, αλλά έχει έναν σημαντικό περιορισμό, ότι οι διάρκειες ζωής των περιοχών έχουν LIFO διάταξη. Οι Fluet , Morrisett και Ahmed προτείνουν έναν διαφορετικό τρόπο αντιμετώπισης του περιορισμού αυτού χρησιμοποιώντας ένα γραμμικό σύστημα [Flue06] . Συγκεκριμένα, παρουσιάζουν την ενδιάμεση γλώσσα λ^{rgnUL} που περιέχει (όπως και στον λογισμό των capabilities) τις εντολές newrgn και freergn για ανεξάρτητη δημιουργία και αποδέσμευση περιοχών. Επίσης περιέχονται αναφορές τύπου ref $r\ \tau$, όπου r η περιοχή που δείχνει η αναφορά και τ ο τύπος του περιεχομένου της, ενώ όλες οι προσβάσεις σε μία περιοχή (για δέσμευση, ανάγνωση ή εγγραφή κάποιας αναφοράς) πρέπει να παρέχουν ένα γραμμικό capability που δημιουργείται με την newrgn και καταστρέφεται με την freergn.

Οι τύποι των τελεστών που αφορούν τις περιοχές είναι οι ακόλουθοι:

$$newrgn :^U ({}^L\text{Unit} \to^L \exists\rho.{}^L({}^L\text{Cap}\ \rho\ *^U \text{Hnd}\ \rho))$$
$$freergn :^U \forall\rho.{}^U({}^L({}^L\text{Cap}\ \rho\ *^U \text{Hnd}\ \rho) \to^L \text{Unit})$$
$$new :^U \forall\rho.{}^U({}^U\forall\tau.{}^U({}^L({}^L\text{Cap}\ \rho\ *^U \text{Hnd}\ \rho *^U \tau) \to^L ({}^L\text{Cap}\ \rho\ *^U \text{Ref}\ \rho\ {}^U\tau)))$$
$$read :^U \forall\rho.{}^U({}^U\forall\tau.{}^U({}^L({}^L\text{Cap}\ \rho\ *^U \text{Ref}\ \rho\ {}^U\tau) \to^L ({}^L\text{Cap}\ \rho\ *^U \tau)))$$
$$write :^U \forall\rho.{}^U({}^U\forall\tau.{}^U({}^L({}^L\text{Cap}\ \rho\ *^U \text{Ref}\ \rho\ {}^U\tau *^U \tau) \to^L ({}^L\text{Cap}\ \rho\ *^U \text{Unit})))$$

Όπως φαίνεται από τους τύπους ο τελεστής newrgn παράγει μια νέα περιοχή και επιστρέφει ένα γραμμικό υπαρξιακό πακέτο στο οποίο κρύβεται το όνομα της περιοχής. Το πακέτο αυτό αφού είναι

γραμμικό πρέπει να ανοιχτεί ακριβώς μία φορά δίνοντας το γραμμικό capability για την περιοχή και τον unrestricted χειριστή. Ο χειριστής μπορεί να διπλασιαστεί δίνοντας μεγαλύτερη ευελιξία στον προγραμματιστή, όμως κάθε πρόσβαση μιας περιοχής (new, read, write) απαιτεί την παρουσία του γραμμικού capabiliy το οποίο και επιστρέφεται για να επιτρέψει μελλοντική πρόσβαση στην περιοχή. Τέλος, με το freergn το γραμμικό capability καταναλώνεται.

Το σύστημα αυτό είναι απλούστερο σε σύγκριση με τον λογισμό των capabilities, αφού δεν απαιτεί την ύπαρξη επιπλέον περιβάλλοντος με capabilities κατά τον έλεγχο τύπων και ούτε περιορίζει την ροή ελέγχου σε CPS. Όπως όμως όλα τα γραμμικά συστήματα που δεν επιτρέπουν την προσωρινή μετατροπή των linear τιμών σε unrestricted, δεν επιτρέπει παράλληλη πρόσβαση σε κάποια περιοχή από πολλά σημεία του προγράμματος και οδηγεί σε δεντρική πρόσβαση των δεδομένων.

3.3.2 L^3

Η ενδιάμεση γλώσσα L ^{3} (Linear Language with Locations) [Amal05] κατασκευάστηκε από την ίδια ομάδα, των Morrisett, Ahmed και Fluet, και έχει σαν βάση της τους Alias Types των Smith, Walker και Morrisett [Smit99].

Η διαφορά της από τα συστήματα που αναφέρθηκαν μέχρι τώρα είναι ότι οι αναφορές (handlers) και τα capabilities αναφέρονται σε θέσεις μνήμης και όχι σε περιοχές. Συγκεκριμένα, στους Alias Types προτείνεται η κατασκευή mutable αντικειμένων και ο διαχωρισμός τους σε δύο συστατικά: έναν δείκτη (ή αναφορά) για ο αντικείμενο που μπορεί ελεύθερα να διπλασιάζεται και ένα capability που επιτρέπει την πρόσβαση στο αντικείμενο αυτό. Ο τύπος του capability πρέπει να καταγράφει τον τύπο του αντικειμένου και πρέπει να παραμένει linear ώστε να υποστηρίζονται ορθά οι ισχυρές αναθέσεις και η αποδέσμευση μνήμης.

Στην L ^{3} διατηρούνται οι πολλαπλές αναφορές και τα γραμμικά capabilities των Alias Types, όμως τα capabilities αποτελούν (όπως και στην λ^{rgnUL}) γραμμικά αντικείμενα πρώτης τάξης.

Μία ιδιομορφία της γλώσσας L^3 είναι η διάκριση μεταξύ των linear και των unrestricted τιμών. Οι unrestriced τιμές (και οι τύποι τους) σημειώνονται με ένα ! και αυτές, και μόνο αυτές, μπορούν να διπλασιαστούν ή να αγνοηθούν με τους τελεστές dup ή drop αντίστοιχα. Για παράδειγμα αν η τιμή $!v$ έχει τύπο $!\tau$ τότε dup $!v \rightarrow (!v, !v)$ και drop $!v \rightarrow$ unit. Με άλλα λόγια ο διπλασιασμός και η απομάκρυνση των unrstricted τιμών γίνεται ρητά.

Ακολουθεί η σημασιολογία των εκφράσεων που δίνουν πρόσβαση στην μνήμη:

$$\begin{aligned} new &: (\sigma; \text{new } v) \rightarrow (\sigma, l \mapsto v; \ulcorner l, (\text{cap } l, !\text{ptr } l)\urcorner) \\ free &: (\sigma, l \mapsto v; \text{free } \ulcorner l, (\text{cap } l, !\text{ptr } l)\urcorner) \rightarrow (\sigma; \ulcorner l, v\urcorner) \\ swap &: (\sigma, l \mapsto v_1; \text{swap ptr } l \ (\text{cap } l, v_2)) \rightarrow (\sigma, l \mapsto v_2; (\text{cap } l, v_1)) \end{aligned}$$

Παρατηρούμε ότι η έκφραση new παράγει ένα linear capability για την θέση μνήμης που δεσμεύει. Επειδή στην γλώσσα υπάρχουν γραμμικές τιμές, η ανάγνωση και η εγγραφή στην μνήμη είναι δυνατή μόνο με την έκφραση swap που επιστρέφει το περιεχόμενο της μνήμης, ώστε να μην επιτραπεί πολλαπλή ή μηδενική χρήση των γραμμικών περιεχομένων της μνήμης. Η έκφραση swap επιστρέφει το γραμμικό capability για να είναι δυνατή η μελλοντική πρόσβαση στην ίδια θέση μνήμης. Το γραμμικό capability καταναλώνεται με την έκφραση free.

Ακολουθεί ένα παράδειγμα στην L ^{3}, στο οποίο ορίζεται μία συνάρτηση που παίρνει σαν παράμετρο ένα πακέτο με το capability και τον δείκτη μίας θέσης μνήμης και μία τιμή x και αναθέτει στην θέση μνήμης την τιμή x:

Παράδειγμα 12

lrswap = $\lambda r : \exists\rho.\langle \text{Cap } \rho\ \tau {*} !\text{Ptr } \rho\rangle.\lambda x : \tau'$
  let $\ulcorner \rho, cp \urcorner = r$ in $(*cp : \langle \text{Cap } \rho\ \tau {*} !\text{Ptr } \rho\rangle *)$
  let $(c_0, p_0) = r$ in $(*c_0 : \text{Cap } \rho\ \tau, p_0 : !\text{Ptr } \rho *)$
  let $(p_1, p_2) =$ dup p_0 in $(*p_1 : !\text{Ptr } \rho, p_2 : !\text{Ptr } \rho *)$
  let $!p_2' = p_2$ in $(*p_2' : !\text{Ptr } \rho *)$

let $(c_1, y) = \text{swap } p'_2\ (c_0, x)$ in $\qquad (*c_1 : \text{Cap } \rho\ \tau', y : \tau*)$
$(\ulcorner \rho, (c_1, p_1)\urcorner, y)$

Η επέκταση της L^3 υποστηρίζει την μετατροπή των linear capabilities σε unrestricted. Η μετατροπή αυτή απαιτεί την ύπαρξη ενός νέου swap τελεστή που να εγγυάται την διατήρηση του τύπου του περιεχομένου της αντίστοιχης θέσης μνήμης, αφού θα μπορεί πλέον να προσπελαστεί από πολλά σημεία του προγράμματος.

Στην επέκταση λοιπόν της L^3 εισάγεται ένας νέος τύπος capability, τα frozen capabilities, τύπου Frzn $\rho\ \tau$, που έχουν unrestricted συμπεριφορά, αλλά δεν δίνουν δικαίωμα ισχυρής ανάθεσης ή απελευθέρωσης της θέσης μνήμης, αφού μπορεί να έχουν aliases . Τα capabilities αυτά προκύπτουν από τα γραμμικά μέσω του τελεστή freeze. Ένα frozen capability μπορεί να επανακτήσει γραμμική συμπεριφορά και να γίνει thawed (μέσω του τελεστή thaw). Για να γίνει αυτό, όμως, πρέπει να παρέχεται απόδειξη ότι κανένα άλλο frozen capability για την ίδια θέση μνήμης δεν είναι thawed.

Για την καλύτερη κατανόηση της μετατροπής ενός capability από frozen σε thwaed δίνεται ένα παράδειγμα:

Παράδειγμα 13
read = $\lambda r^! :!\exists\rho.(!Frzn\ \rho\ !\tau{*}!Ptr\ \rho).\lambda\tau^0 : Thwd\ \bullet$.
$(*$ο τύπος $Thwd\ \bullet$ του τ δείχνει το σύνολο των capabilities που έχουν γίνει thawed$*)$
let $\ulcorner\rho, (f_a^!, l^!)\urcorner = r$ in
let $(c^l, t^l) =$ thaw $(f_a, t^0)\ void_\rho$
$(*$το capability f_a επανακτά γραμμική συνπεριφορά - γίνεται thawed, προστίθεται στο σύνολο των capabilities που έχουν γίνει thawed, ενώ το $void_\rho$ παρέχει απόδειξη ότι κανένα άλλο frozen capability για την ίδια θέση μνήμης δεν είναι thawed$*)$
let $(c^2, x^!)$ =swap $l\ (c^1, ())$ in
let (c^3, y)=swap $l\ (c^2, X)$ in
let $(f_b^!, t^2)$ = refreeze (c^3, t^l) in
$(*$το capability c^3 επανακτά unrestricted συνπεριφορά - γίνεται frozen και αφαιρείται από το σύνολο των capabilities που έχουν γίνει thawed που είναι πλέον το $t^2*)$
(x, t^2)

Οι ιδέες αυτές είναι παρόμοιες με τον τελεστή restrict του CQUAL [Aike03], ενός συστήματος που επεκτείνει την C με qualifiers τύπων που ορίζονται από τον χρήστη.

Κεφάλαιο 4

Εφαρμογή σε ιδιότητες ασφάλειας μνήμης

Η γλώσσα που θα παρουσιαστεί είναι ένας συνδυασμός της L^3 και της $let!$. Όπως και στην L^3, ένα mutable δεδομένο διαχωρίζεται σε δύο μέρη: σε έναν *δείκτη* στο δεδομένο και σε ένα δικαίωμα πρόσβασης σε αυτό. Κατά την δημιουργία ενός mutable αντικειμένου παράγεται ένα γραμμικό δικαίωμα πρόσβασης. Η έκφραση $let!$, όπως και στην αντίστοιχη γλώσσα, μας παρέχει την δυνατότητα μετατροπής του δικαιώματος πρόσβασης αυτού σε unrestricted .

Με τον τρόπο αυτό, διατηρούνται τα πλεονεκτήματα των δύο γλωσσών. Συγκεκριμένα. η χρήση γραμμικών δικαιωμάτων πρόσβασης επιτρέπει την ύπαρξη πολλαπλών δεικτών (multiple pointers) για μία θέση μνήμης ενώ ταυτόχρονα οι ισχυρές ενημερώσεις και η ρητή αποδέσμευση μνήμης γίνονται με ορθό τρόπο (sound strong updates and explicit deallocation). Συγχρόνως, το βασικό μειονέκτημα των γραμμικών συστημάτων τύπων, δηλαδή ότι ένα αντικείμενο που χρησιμοποιείται καταναλώνεται αυτόματα, παρακάμπτεται με τη χρήση του $let!$.

4.1 Σύνταξη

Η γλώσσα που παρουσιάζεται στο κεφάλαιο αυτό βασίζεται στον λάμβδα λογισμό με τύπους και references τύπου ML. Αρχικά θα παρουσιαστεί ο πυρήνας, ενώ στην ενότητα 4.4 θα επεκταθεί.

Η σύνταξη της γλώσσας φαίνεται στο σχήμα 4.1.

4.1.1 References

Οι references (r) αντιστοιχούν σε θέσεις μνήμης και μπορεί να είναι είτε μεταβλητές (l), είτε σταθερές (i). Οι σταθερές χρησιμοποιούνται από τα operational semantics, αλλά δεν μπορούν να χρησιμο-

$$
\begin{array}{rcl}
\pi & ::= & \rho \,|\, \bot \\
q & ::= & \mathrm{L} \,|\, \mathrm{U} \text{ at } \pi \\
r & ::= & l \,|\, i \\
\phi & ::= & \mathrm{Cap}\ r\ \tau \,|\, \langle \tau_1 * \tau_2 \rangle \,|\, \tau_1 \xrightarrow{\overrightarrow{\pi}} \tau_2 \,|\, \exists l.\tau \\
\tau & ::= & \mathrm{Unit} \,|\, \mathit{Ref}\, r \,|\, {}^q\phi \\
e & ::= & \mathrm{unit} \,|\, x \,|\, {}^q\lambda x : \tau.e \,|\, e_1\ e_2 \,|\, e_1; e_2 \\
& | & {}^q(e_1, e_2) \,|\, \text{let } (x, y) = e_1 \text{ in } e_2 \,|\, {}^q\ulcorner r, e \urcorner \,|\, \text{let } \ulcorner l, x \urcorner = e_1 \text{ in } e_2 \\
& | & \text{loc } i \,|\, {}^q\mathit{cap}\ i \\
& | & \text{new } e \,|\, \text{free } e \,|\, \text{deref } e \,|\, e_1 := e_2 \,|\, e_1 :=^s e_2 \,|\, e_1 :=: e_2 \,|\, e_1 :=:^s e_2 \\
& | & \text{at } \rho \text{ let! } (x = e) \text{ then } y = e_1 \text{ in } e_2 \,|\, \text{at } \rho \text{ let\$ } (x = z) \text{ then } y = e_1 \text{ in } e_2 \\
v & ::= & \mathrm{unit} \,|\, \text{loc } i \,|\, {}^q u \\
u & ::= & \mathit{cap}\ i \,|\, (v_1, v_2) \,|\, \ulcorner i, v \urcorner \,|\, \lambda x : \tau.e
\end{array}
$$

Σχήμα 4.1: Σύνταξη

ποιηθούν από τον προγραμματιστή. Ο προγραμματιστής μπορεί να χρησιμοποιεί τις μεταβλητές που αποτελούν αφαίρεση των σταθερών.

4.1.2 Qualifiers - Scopes

O qualifier είναι είτε L για linear συμπεριφορά είτε U at π για unrestricted, όπου π κάποιο scope. Τα scopes μπορούν να είναι είτε μεταβλητές είτε το προκαθορισμένο scope $\perp$. Ένα scope ρ είναι έγκυρο μόνο μέσα στην υπο-έκφραση e_2 της έκφρασης at ρ let! $(x = e_1)$ then $y = e_2$ in e, ενώ το $\perp$ είναι έγκυρο σε όλο το πρόγραμμα.

4.1.3 Τύποι

Οι τύποι είναι ο Unit, ο Ref r, που δηλώνει τύπο για κάποια θέση μνήμης r ή αποτελούνται από τον qualifier και τον pretype.

Οι pretypes επισημειώνονται με τον qualifier αφού μπορεί να περιέχουν γραμμικά δεδομένα και μπορεί να είναι ο Cap $r\ \tau$, ο $\langle\tau_1 * \tau_2\rangle$, ο $\tau_1 \xrightarrow{\vec{\pi}} \tau_2$ και ο $\exists l.\tau$.

O Cap $r\ \tau$ αντιστοιχεί σε κάποιο capability για την θέση μνήμης r που το περιεχόμενό της έχει τύπο τ. O $\langle\tau_1 * \tau_2\rangle$ αντιστοιχεί σε κάποιο ζευγάρι με τύπους τ_1 και τ_2. O $\tau_1 \xrightarrow{\vec{\pi}} \tau_2$ αντιστοιχεί σε συνάρτηση από τ_1 σε τ_2, σημειωμένη με το διάνυσμα από τα scopes που χρησιμοποιούνται στο κλείσιμό της. Τέλος, ο $\exists l.\tau$ αντιστοιχεί σε ένα πακέτο και η reference l είναι δεσμευμένη στον τύπο τ.

4.1.4 Εκφράσεις

Οι εκφράσεις της γλώσσας περιέχουν το unit, την μεταβλητή, την αφαίρεση, την εφαρμογή και τον σειριακό συνδυασμό δύο υποεκφράσεων. Περιέχουν επίσης το ζεύγος ${}^{q}(e_1, e_2)$ και την έκφραση let $(x, y) = e_1$ in e_2 που αναθέτει στις μεταβλητές x και y τα δύο συστατικά του ζεύγους e_1. Είναι σημαντικό να τονίσουμε ότι σε κάθε γραμμικό σύστημα τύπων η απόσπαση των δύο συστατικών ενός ζεύγους πρέπει να γίνει ταυτόχρονα, αφού αν το ζεύγος είναι γραμμικό πρέπει να χρησιμοποιηθεί ακριβώς μία φορά. Η έκφραση ${}^{q}\ulcorner r, e\urcorner$ είναι ένα υπαρξιακό πακέτο που η θέση μνήμης r είναι δεσμευμένη στην έκφραση e και έχει τύπο $\exists l.\tau$, αν η έκφραση e έχει τύπο τ, όπου το r έχει αντικατασταθεί με l. Το πακέτο μπορεί να ανοιχθεί με την έκφραση let $\ulcorner r, x\urcorner = e_1$ in e_2 Το ζεύγος και το πακέτο πρέπει να σημειώνονται με qualifier αφού μπορεί να περιέχουν γραμμικές υποεκφράσεις.

Υπάρχουν εκφράσεις για μία θέση μνήμης (loc i) και για ένα δικαίωμα πρόσβασης σε αυτήν (^{q}cap i), αλλά δεν μπορούν να χρησιμοποιηθούν από τον προγραμματιστή. Επίσης, υπάρχουν εκφράσεις για την δημιουργία μιας θέσης μνήμης, τη ανάγνωση των περιεχομένων της και την αποδέσμευσή της. Τα περιεχόμενα μιας θέσης μνήμης μπορεί να αλλάξουν με ανάθεση είτε με ανταλλαγή. Η ανταλλαγή, που επιστρέφει το παλιό περιεχόμενο της μνήμης και είναι απαραίτητη όταν χρειάζεται να ενημερωθεί κάποια θέση μνήμης που περιέχει γραμμικά δεδομένα. Επίσης, υπάρχει η ισχυρή (strong) ανάθεση και ανταλλαγή που επιτρέπει την ανάθεση στην θέση μνήμης δεδομένου οποιουδήποτε τύπου.

Τέλος, υπάρχουν δύο $let!$ εκφράσεις αντίστοιχες της γλώσσας $let!$. Συγκεκριμένα, η έκφραση at ρ let! $(x = e_1)$ then $y = e_2$ in e εισάγει το scope ρ που είναι έγκυρο μόνο μέσα στην e_2, αποτιμά την e_1 σε ένα linear capability, το μετατρέπει σε unrestricted και το αναθέτει στην μεταβλητή x. Έπειτα αποτιμά την e_2 και αναθέτει το αποτέλεσμά της στην y. Τέλος, κατά την αποτίμηση της e είναι διαθέσιμες τόσο η μεταβλητή y όσο και η x, η τιμή της οποίας έχει επανέλθει στην αρχική linear μορφή. Ενώ η at ρ let\$ $(x = z)$ then $y = e_2$ in e είναι ενδιάμεση έκφραση της γλώσσας, δηλαδή δεν μπορεί να χρησιμοποιηθεί από τον προγραμματιστή και είναι απαραίτητη για την αποτίμηση.

4.2 Κανόνες Συστήματος Τύπων

Η σχέση για τον συμπερασμό τύπων στης γλώσσας είναι $\Gamma; \Delta; Z; M \vdash e : \tau$.

$$\boxed{\Gamma = \Gamma_1 \oplus \Gamma_2}$$

$$\frac{}{\emptyset = \emptyset \oplus \emptyset}\ \text{Empty} \qquad \frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x :^{U \text{ at } \pi} \phi = \Gamma_1, x :^{U \text{ at } \pi} \phi \oplus \Gamma_2, x :^{U \text{ at } \pi} \phi}\ \text{Un}$$

$$\frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x :^{L} \phi = \Gamma_1, x :^{L} \phi \oplus \Gamma_2}\ \text{Lin1} \qquad \frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x :^{L} \phi = \Gamma_1 \oplus \Gamma_2, x :^{L} \phi}\ \text{Lin2}$$

Σχήμα 4.2: Διάσπαση Περιβάλλοντος Γ

4.2.1 Περιβάλλοντα

- Το περιβάλλον Γ των μεταβλητών των εκφράσεων. Το Γ αντιστοιχεί τις ενεργές μεταβλητές της γλώσσας με τους τύπους τους:

  $\Gamma \ ::= \ \emptyset \,|\, \Gamma, x : \tau$

  Για να εξασφαλίσουμε ότι κάθε linear μεταβλητή χρησιμοποιείται ακριβώς μία φορά απαιτείται ο ορισμός του τελεστή $\oplus$ (Context Split) όπως φαίνεται στην εικόνα 4.2 που περιγράφει πως διασπάται το περιβάλλον Γ σε δύο περιβάλλοντα Γ_1 και Γ_2 που θα χρησιμοποιηθούν για τον συμπερασμό τύπων διαφορετικών υποεκφράσεων στις υποθέσεις του κανόνα.

- Το περιβάλλον Δ των μεταβλητών references .

  $\Delta \ ::= \ \emptyset \,|\, \Delta, l$

- Το περιβάλλον των μεταβλητών scopes.

  $Z \ ::= \ \emptyset \,|\, Z, \rho$

  Το περιβάλλον αυτό περιέχει ένα υποσύνολο των ενεργών scopes. Συγκεκριμένα, τα scopes του Z χειρίζονται με relevant τρόπο, δηλαδή πρέπει να χρησιμοποιηθούν τουλάχιστον μία φορά. Αυτός ο τρόπος χρήσης εξυπηρετεί στον καθορισμό των απαιτούμενων scopes για το σώμα μιας συνάρτησης (και μόνο αυτών) στη λίστα που σημειώνεται στον τύπο της συνάρτησης. Για τον σκοπό αυτό ορίζουμε την ελάχιστη σχέση μεταξύ του περιβάλλοντος Z και κάποιου qualifier, όπως φαίνεται στην εικόνα 4.3 . Αυτή είναι η ελάχιστη σχέση που εγγυάται ότι το scope του qualifier, αν υπάρχει, είναι ενεργό.

$$\boxed{Z \models q}$$

$$\emptyset \models L \qquad \emptyset \models U \qquad \{\rho\} \models U \text{ at } \rho$$

Σχήμα 4.3: Έλεγχος εγκυρότητας scope

- Το περιβάλλον M των σταθερών references (store typing). Το M αντιστοιχεί τις σταθερές references με τους τύπους τους:

  $M \ ::= \ \emptyset \,|\, M, i : \tau$

  Το περιβάλλον αυτό χρησιμοποιείται, σε συνδυασμό με το περιβάλλον Δ, για τον έλεγχο εγκυρότητας reference σύμφωνα με την σχέση της εικόνας 4.4.

$$\boxed{M \models i} \qquad \boxed{M; \Delta \models r}$$

$$M \models i \Leftrightarrow \exists \tau.((i, \tau) \in M) \quad M; \Delta \models i \Leftrightarrow M \models i$$
$$M; \Delta \models l \Leftrightarrow l \in \Delta$$

Σχήμα 4.4: Έλεγχος εγκυρότητας reference

4.2.2 Χρήσιμες Συναρτήσεις

- Το κατηγόρημα $q(\tau)$ που παρουσιάζεται στην εικόνα 4.5 ελέγχει αν ένας τύπος είναι linear ή unrestricted. Τύποι χωρίς qualifier έχουν unrestricted συμπεριφορά. Στην ίδια εικόνα παρουσιάζεται και η επέκτασή του κατηγορήματος στο περιβάλλον Γ, $q(\Gamma)$. Το περιβάλλον Γ είναι unrestricted αν δεν περιέχει καμία μεταβλητή με linear περιεχόμενο.

$$\boxed{\text{q(.)}}$$

$$L(\tau) \Leftrightarrow (\tau =^q \phi \wedge (q = L \vee q = \text{U at } \pi)) \vee \tau \neq^q \phi$$
$$U(\tau) \Leftrightarrow (\tau =^q \phi \wedge (q = \text{U at } \pi)) \vee \tau \neq^q \phi$$
$$q(\Gamma) \Leftrightarrow ((x : \tau) \in \Gamma \Rightarrow q(\tau))$$

Σχήμα 4.5: Κατηγόρημα ελέγχου qualifier

- Η συνάρτηση FRV (free reference variables) ορίζεται στην εικόνα 4.6 και επιστρέφει, για κάποιον τύπο όλες τις ελεύθερες reference μεταβλητές του.

$$\boxed{FRV(\tau)}$$

$$FRV(\text{Unit}) = \emptyset \qquad FRV(\text{Ref}\, l) = \{l\}$$
$$FRV(\text{Ref}\, i) = \emptyset \qquad FRV({}^q\text{Cap}\; l\; \tau) = \{l\} \cup FRV(\tau)$$
$$FRV({}^q\text{Cap}\; i\; \tau) = FRV(\tau) \qquad FRV({}^q\exists l.\tau) = FRV(\tau) \setminus \{l\}$$
$$FRV({}^q(\langle \tau_1 * \tau_2 \rangle)) = FRV(\tau_1) \cup FRV(\tau_2) \qquad FRV({}^q\tau_1 \xrightarrow{\vec{\pi}} \tau_2) = FRV(\tau_1) \cup FRV(\tau_2)$$

Σχήμα 4.6: Ελεύθερες reference μεταβλητές (FRV)

4.2.3 Οι κανόνες

Οι κανόνες τύπων της γλώσσας φαίνονται στα σχήματα εικόνα 4.7 - 4.13. Σε αυτούς, για απλοποίηση έχουμε ορίσει:

$${}^q\text{Lref}\; r\; \tau \equiv {}^q\langle {}^q\text{Cap}\; r\; \tau * \text{Ref}\; r \rangle$$
$${}^q\text{Xref}\; \tau \equiv {}^q\exists l. {}^q\text{Lref}\; l\; \tau$$

Βασικές Εκφράσεις

Από τον ορισμό της διάσπασης του περιβάλλοντος Γ προκύπτει ότι κάθε γραμμική έκφραση χρησιμοποιείται το πολύ σε μία ροή ελέγχου. Για να εξασφαλίσουμε ότι κάθε γραμμική έκφραση χρησιμοποιείται ακριβώς μία φορά αρκεί να ελέγχουμε ότι το περιβάλλον στο οποίο γίνεται έλεγχος για κάθε

$$\frac{U(\Gamma)}{\Gamma; \Delta; Z; M \vdash \text{unit} : \text{Unit}} \text{ unit} \qquad \frac{U(\Gamma) \quad FRV(\tau) \subseteq \Delta \quad Z \models \text{qual } \tau}{\Gamma, x : \tau; \Delta; Z; M \vdash x : \tau} \text{ var}$$

Σχήμα 4.7: Κανόνες τύπων: Βασικές Εκφράσεις

βασική έκφραση είναι unrestricted. Για τις μεταβλητές θα πρέπει επιπλέον να ελέγχεται ότι το scope τους είναι έγκυρο και ότι όλες οι ελεύθερες references μεταβλητές ανήκουν στο Δ.

Σειριακή Εκτέλεση

$$\frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : \tau_1 \quad U(\tau_1) \quad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau_2}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1; e_2 : \tau_2} \text{ seq}$$

Σχήμα 4.8: Κανόνες τύπων: Σειριακή Εκτέλεση

Κατά την σειριακή εκτέλεση δύο εκφράσεων $e_1; e_2$ απαιτούμε η πρώτη έκφραση (e_1) να επιστρέψει ένα unrestricted δεδομένο, αφού το δεδομένο αυτό θα αγνοηθεί.

Αφαίρεση και Εφαρμογή

$$\frac{\begin{array}{c} q(\Gamma) \qquad Z \models q \\ \Gamma, x : \tau; \Delta; Z_e; M \vdash e : \tau_1 \end{array}}{\Gamma; \Delta; Z; M \vdash {}^q\lambda x : \tau.e : {}^q(\tau \xrightarrow{Z_e} \tau_1)} \text{ abs}$$

$$\frac{\begin{array}{c} \Gamma_1; \Delta; Z_1; M \vdash e_1 : {}^q(\tau_1 \xrightarrow{Z_e} \tau_2) \\ \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau_1 \quad Z \models \text{qual } \tau_2 \end{array}}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2 \cup Z \cup Z_e; M \vdash e_1 \; e_2 : \tau_2} \text{ appl}$$

Σχήμα 4.9: Κανόνες τύπων: Αφαίρεση και Εφαρμογή

Από το σχήμα 4.3 και από τον relevant χειρισμό του Z στο σύστημα τύπων προκύπτει ότι το Z_e στον κανόνα abs είναι το ελάχιστο σύνολο από scopes που χρησιμοποιούνται στο σώμα της συνάρτησης, άρα είναι το κατάλληλο σύνολο για να σημειωθεί στον τύπο της.

Επίσης πρέπει να τονιστεί η αναγκαιότητα του ελέγχου $q(\Gamma)$ στον ίδιο κανόνα: Αν η συνάρτηση έχει δηλωθεί από τον προγραμματιστή σαν unrestricted θα πρέπει να ισχύει και $U(\Gamma)$, διαφορετικά οι linear μεταβλητές που θα υπήρχαν στο Γ θα μπορούσαν να χρησιμοποιηθούν στο σώμα της συνάρτησης όσες φορές αυτή εφαρμόζονταν.

Ζεύγη και Πακέτα

Στον κανόνα του letPack ο έλεγχος $FLV(\tau) \subseteq \Delta$ εξασφαλίζει ότι η reference μεταβλητή l αυτή δεν μπορεί να εμφανίζεται ελεύθερη στον τελικό τύπο της e_2.

Capabilities και References

Διαχείριση Μνήμης

Όσον αφορά την διαχείριση μνήμης αξίζει να σημειωθούν τα ακόλουθα: Η ανάκτηση δεδομένων και η ασθενής ανάθεση - ανταλλαγή απαιτούν unrestricted capadility. Αντίθετα, η ισχυρή ανάθεση - ανταλλαγή απαιτούν linear capability για να εξασφαλιστεί ότι δεν υπάρχουν aliases όταν επιτρέπεται η αλλαγή του τύπου του περιεχομένου της θέσης μνήμης. Όμως, το linear αυτό capability επιστρέφεται για να επιτρέπεται επιπλέον πρόσβαση στην θέση μνήμης και να επιβάλλεται η αποδέσμευσή της.

$$\frac{q(\tau_1) \quad q(\tau_2) \quad \Gamma_1;\Delta;Z_1;M \vdash e_1 : \tau_1 \qquad Z_1 \cup Z_2 \models q \quad \Gamma_2;\Delta;Z_2;M \vdash e_2 : \tau_2}{\Gamma_1 \oplus \Gamma_2;\Delta;Z_1 \cup Z_2;M \vdash {}^q(e_1,e_2) : {}^q\langle \tau_1 * \tau_2 \rangle} \text{ pair}$$

$$\frac{x \neq y \quad \Gamma_1;\Delta;Z_1;M \vdash e_1 : {}^q\langle \tau_1 * \tau_2 \rangle \qquad \Gamma_2, x:\tau_1, y:\tau_2;\Delta;Z_2;M \vdash e_2 : \tau}{\Gamma_1 \oplus \Gamma_2;\Delta;Z_1 \cup Z_2;M \vdash \text{let } (x,y) = e_1 \text{ in } e_2 : \tau} \text{ letPair}$$

$$\frac{q(\tau) \quad \Delta;M \models r' \quad Z \models q \qquad \Gamma;\Delta;Z;M \vdash e : \tau[r \mapsto r']}{\Gamma;\Delta;Z;M \vdash {}^q\ulcorner r', e \urcorner : {}^q\exists r.\tau} \text{ pack}$$

$$\frac{\Gamma_1;\Delta;Z_1;M \vdash e_1 : {}^q\exists l.\tau_1 \quad FLV(\tau) \subseteq \Delta \qquad \Gamma_2, x:\tau_1;\Delta,l;Z_2;M \vdash e_2 : \tau}{\Gamma_1 \oplus \Gamma_2;\Delta;Z_1 \cup Z_2;M \vdash \text{let } \ulcorner l, x \urcorner = e_1 \text{ in } e_2 : \tau} \text{ letPack}$$

Σχήμα 4.10: Κανόνες τύπων: Ζεύγη και Πακέτα

$$\frac{M \models i}{\Gamma;\Delta;Z;M \vdash \text{loc } i : \text{Ref } i} \text{ loc} \qquad \frac{}{\Gamma;\Delta;Z;M, i:\tau \vdash^q \text{cap } i : {}^q\text{Cap } \tau\ i} \text{ cap}$$

Σχήμα 4.11: Κανόνες Τύπων: Capabillities και References

$$\frac{\Gamma;\Delta;Z;M \vdash e : \tau}{\Gamma;\Delta;Z;M \vdash \text{new } e : {}^L\text{Xref } \tau} \text{ new} \qquad \frac{Z_T \models \text{qual } \tau \qquad \Gamma;\Delta;Z;M \vdash e : {}^L\text{Xref } \tau}{\Gamma;\Delta;Z \cup Z_T;M \vdash \text{free } e : \tau} \text{ free}$$

$$\frac{Z_\tau \models \text{qual } \tau \quad U(\tau) \qquad \Gamma;\Delta;Z;M \vdash e : {}^{U \text{ at } \pi}\text{Lref } r\ \tau}{\Gamma;\Delta;Z \cup Z_\tau;M \vdash \text{deref } e : \tau} \text{ deref}$$

$$\frac{\Gamma_1;\Delta;Z_1;M \vdash e_1 : {}^L\text{Lref } r\ \tau_2 \qquad \Gamma_2;\Delta;Z_2;M \vdash e_2 : \tau_1 \quad U(\tau_2)}{\Gamma_1 \oplus \Gamma_2;\Delta;Z_1 \cup Z_2;M \vdash e_1 :=^s e_2 : {}^L\text{Cap } r\ \tau_1} \text{ sAss}$$

$$\frac{\Gamma_1;\Delta;Z_1;M \vdash e_1 : {}^{U \text{ at } \pi}\text{Lref } r\ \tau \qquad \Gamma_2;\Delta;Z_2;M \vdash e_2 : \tau \quad U(\tau)}{\Gamma_1 \oplus \Gamma_2;\Delta;Z_1 \cup Z_2;M \vdash e_1 := e_2 : \text{Unit}} \text{ wAss}$$

$$\frac{\Gamma_1;\Delta;Z_1;M \vdash e_1 : {}^{U \text{ at } \pi}\text{Lref } r\ \tau \qquad \Gamma_2;\Delta;Z_2;M \vdash e_2 : \tau}{\Gamma_1 \oplus \Gamma_2;\Delta;Z_1 \cup Z_2;M \vdash e_1 :=: e_2 : \tau} \text{ wSw}$$

$$\frac{\Gamma_1;\Delta;Z_1;M \vdash e_1 : {}^L\text{Lref } r\ \tau_2 \qquad \Gamma_2;\Delta;Z_2;M \vdash e_2 : \tau_1}{\Gamma_1 \oplus \Gamma_2;\Delta;Z_1 \cup Z_2;M \vdash e_1 :=:^s e_2 : {}^L\langle \tau_2 * {}^L\text{Cap } r\ \tau_1 \rangle} \text{ sSw}$$

Σχήμα 4.12: Κανόνες Τύπων: Διαχείριση References

Η απλή ανάκτηση απαιτεί unrestricted περιεχόμενο μνήμης, αφού αν το περιεχόμενο ήταν linear, θα επιτρεπόταν η πολλαπλή ανάκτηση (άρα και χρήση) αυτού. Κατά την ανάθεση σε μία θέση μνήμης το παλιό περιεχόμενο αυτής δεν μπορεί να χρησιμοποιηθεί πλέον, άρα πρέπει να είναι unrestricted, ενώ κατά την ανταλλαγή επιστρέφεται άρα δεν υπάρχει κάποιος περιορισμός.

Οι παρατηρήσεις αυτές συνοψίζονται στον πίνακα 4.1.

cap	data	deref	wAss	wSwap	sAss	sSwap
L	L	-	-	-	-	+
L	U	-	-	-	+	+
U	L	-	-	+	-	-
U	U	+	+	+	-	-

Πίνακας 4.1: Διαχείριση Μνήμης

Let!

$$\frac{\text{fresh } \rho' \quad \Gamma; \Delta; Z; M \vdash e : {}^{L}\text{Cap } r\ \tau \quad \Gamma_1, x : {}^{\text{U at } \rho'}\text{Cap } r\ \tau; \Delta; Z_1'; M \vdash e_1[\rho \mapsto \rho'] : \tau_1 \quad Z_1' = Z_1 \vee Z_1, \rho' \quad \Gamma_2, x : {}^{L}\text{Cap } r\ \tau, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_2 \cup Z_3; M \vdash \text{at } \rho \text{ let! } (x = e) \text{ then } y = e_1 \text{ in } e_2 : \tau_2}$$

$$\frac{\Gamma; \Delta; Z; M \vdash z : {}^{q}\text{Cap } r\ \tau \quad \Gamma_1; \Delta; Z_1'; M \vdash e_1 : \tau_1 \quad Z_1' = Z_1 \vee Z_1, \rho \quad \Gamma_2, x : {}^{L}\text{Cap } r\ \tau, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_2 \cup Z_3; M \vdash \text{at } \rho \text{ let\$ } (x = z) \text{ then } y = e_1 \text{ in } e_2 : \tau_2}$$

Σχήμα 4.13: Κανόνες τύπων: let!

Οι κανόνες αυτοί είναι ανάλογοι της γλώσσας $let!$. Η κύρια διαφορά είναι ότι μόνο εκφράσεις με τύπο ${}^{L}\text{Cap}\ r\ \tau$ μπορούν να γίνουν unrestricted (να είναι bangable). Από την σύνταξη της γλώσσας οι εκφράσεις που μπορούν να είναι linear, εκτός από τα capabilities, είναι οι συναρτήσεις, τα ζεύγη και τα πακέτα. Μία γραμμική συνάρτηση δεν επιτρέπεται να γίνει unrestricted, όπως και στην $let!$, αφού όντας linear στο σώμα της μπορεί να χρησιμοποιεί linear τιμές. Αν μετατρέπονταν σε unrestricted οι τιμές αυτές θα χρησιμοποιούνταν τόσες φορές όσες η συνάρτηση εφαρμόζονταν. Αντίστοιχα, τα linear ζεύγη και πακέτα μπορούν να περιέχουν linear εκφράσεις, άρα η μετατροπή τους σε unresticted θα επέτρεπε στις εκφράσεις αυτές να χρησιμοποιηθούν καμία ή περισσότερες από μία φορές.

4.3 Λειτουργική Σημασιολογία

Η σχέση για τον υπολογισμό μιας έκφρασης της γλώσσας είναι: $S; \mu; e \hookrightarrow S'; \mu'; e'$

4.3.1 Περιβάλλοντα

- Το περιβάλλον S (Store). Το S αντιστοιχεί τις μεταβλητές με εκφράσεις:

  $S ::= \emptyset \,|\, S, x \mapsto v$

  Για την ανάκτηση μιας μεταβλητής από το S χρησιμοποιούμε την συνάρτηση αναζήτησης του σχήματος 4.14, κατά την οποία όταν η τιμή της μεταβλητής x είναι linear, η x αφαιρείται από το S, διαφορετικά όχι. Συνεπώς, οι qualifiers πρέπει να επιβιώνουν μέχρι και τον χρόνο εκτέλεσης.

$$\boxed{S; x \Downarrow S'; v}$$

$$\frac{v =^{L} u}{S, x \mapsto v; x \Downarrow S; v} \qquad \frac{v \neq^{L} u}{S, x \mapsto v; x \Downarrow S, x \mapsto v; v}$$

Σχήμα 4.14: Συνάρτηση αναζήτησης S

- Το περιβάλλον μ (memory). Το μ αντιστοιχεί τις references με τις τιμές τους:

  $\mu ::= \emptyset \mid \mu, i \mapsto x$

4.3.2 Οι κανόνες

Στα σχήματα 4.15 - 4.19 ορίζεται η small-step, left-to-right, call-by-value λειτουργική σημασιολογία της γλώσσας. Σημειώνουμε ότι όπως και σε άλλες γραμμικές γλώσσες το πρόγραμμα δεν τερματίζει με τιμή, αλλά με μεταβλητή, ή πιο συγκεκριμένα ”auto-variable”[Walk05]. Οι auto-variables είναι μεταβλητές που εισάγονται αυτόματα κατά τον χρόνο εκτέλεσης, πέρα από αυτό δεν μπορούν να διακριθούν από τις μεταβλητές της γλώσσας.

Τιμές

$$\frac{\text{fresh } z}{S;\mu;v \hookrightarrow S, z\mapsto v;\mu;z}\ \text{value}$$

Σχήμα 4.15: Κανόνες αποτίμησης: Τιμή

Όπως και σε άλλες γλώσσες με linear τιμές, όταν συναντάται μία τιμή, δεσμεύεται από μία ”auto-variable” και αποθηκεύεται στο S. Η ανάκτηση αυτής της τιμής μπορεί να γίνει μόνο μέσω του S. Αυτό, σε συνδυασμό με την συνάρτηση ανάκτησης μεταβλητών του σχήματος 4.14, εξασφαλίζει ότι τα γραμμικά αντικείμενα μπορούν να χρησιμοποιηθούν το πολύ μία φορά από το πρόγραμμα.

Ζεύγη και Πακέτα

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;{}^q(e_1,e_2) \hookrightarrow S';\mu';{}^q(e_1',e_2)}\ \text{pairPr1} \qquad \frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;{}^q(x,e_2) \hookrightarrow S';\mu';{}^q(x,e_2')}\ \text{pairPr2}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;\text{let }(x,y)=e_1\text{ in }e_2 \hookrightarrow S';\mu';\text{let }(x,y)=e_1'\text{ in }e_2}\ \text{letPairPr}$$

$$\frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;{}^q\ulcorner r,e\urcorner \hookrightarrow S';\mu';{}^q\ulcorner r,e'\urcorner}\ \text{packPr} \qquad \frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{\begin{array}{c}S;\mu;\text{let }\ulcorner r,x\urcorner=e_1\text{ in }e_2 \hookrightarrow\\ S';\mu';\text{let }\ulcorner r,x\urcorner=e_1'\text{ in }e_2\end{array}}\ \text{packPr}$$

$$\frac{S;z \Downarrow S';{}^q\langle z_1,z_2\rangle}{\begin{array}{c}S;\mu;\text{let }(x,y)=z\text{ in }e_1 \hookrightarrow\\ S';\mu;e_1[x\mapsto z_1][y\mapsto z_2]\end{array}}\ \text{pair} \qquad \frac{S;z \Downarrow S';{}^q\ulcorner i,z_1\urcorner}{\begin{array}{c}S;\mu;\text{let }\ulcorner l,x\urcorner=z\text{ in }e_1 \hookrightarrow\\ S';\mu;e_1[l\mapsto i][x\mapsto z_1]\end{array}}\ \text{pack}$$

Σχήμα 4.16: Κανόνες αποτίμησης: Ζεύγη και Πακέτα

Εφαρμογή - Σειριακή Αποτίμηση

Διαχείριση Μνήμης

Κατά την αποτίμηση του new x, όπου x μία ”auto-variable”, δημιουργείται μία νέα θέση μνήμης i στην οποία αποθηκεύεται η τιμή x και επιστρέφεται το ζευγάρι ${}^L({}^L\text{cap } i, \text{loc } i)$ σε ένα πακέτο που κρύβει τη θέση μνήμης i. Κατά την αποτίμηση του free w ανακαλείται (και επειδή είναι linear ταυτόχρονα διαγράφεται) από το S το w. Το w θα αποτιμηθεί σταδιακά σε ${}^L\ulcorner i, {}^L({}^L\text{cap } i, \text{loc } i)\urcorner$. Κατά την αποτίμηση αυτή, η linear capability για τη θέση μνήμης i θα καταναλωθεί και δεν θα είναι δυνατή πλέον η πρόσβαση σε αυτήν τη θέση μνήμης. Η τιμή που επιστρέφει η $free$ είναι η τιμή που είναι το περιεχόμενο της i.

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1\ e_2 \hookrightarrow S';\mu';e_1'\ e_2}\ \text{apPr1} \qquad \frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x\ e_2 \hookrightarrow S';\mu';x\ e_2'}\ \text{apPr2}$$

$$\frac{S;w \Downarrow S';{}^{q}\lambda x:\tau.e}{S;\mu;w\ y \hookrightarrow S;\mu;e[x \mapsto y]}\ \text{appl}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1;e_2 \hookrightarrow S';\mu';e_1';e_2}\ \text{seqPr} \qquad \frac{}{S;\mu;z;e_2 \hookrightarrow S;\mu;e_2}\ \text{seq}$$

Σχήμα 4.17: Κανόνες αποτίμησης: Εφαρμογή - Σειριακή Αποτίμηση

Let!

Για την αποτίμηση της έκφρασης at ρ let! $(x = z)$ then $y = e_1$ in e_2 θα πρέπει η μεταβλητή z να δεσμεύει μια linear τιμή. Η κατάσταση της τιμής αυτής αλλάζει σε U at ρ', όπου ρ' ένα νέο scope και καλείται η αποτίμηση της έκφρασης at ρ' let\$ $(x = z)$ then $y = e_1[\rho \mapsto \rho'][x \mapsto z]$ in e_2, όπου το scope ρ έχει αντικατασταθεί από το νέο scope ρ' και στην υποέκφραση e_1 το scope ρ έχει αντικατασταθεί από το νέο scope ρ' και η μεταβλητή x από την μεταβλητή z. Για την αποτίμηση της έκφρασης αυτής, αρχικά θα κληθεί ο propagator $let\$Pr$ μέχρι η έκφραση να έρθει στη μορφή at ρ let\$ $(x = z)$ then $y = w$ in e_2. Τότε η τιμή της x θα επανέλθει στην αρχική linear κατάσταση και θα επιστραφεί η έκφραση $e_2[x \mapsto z][y \mapsto w]$.

4.4 Επεκτάσεις

Στην ενότητα αυτή η γλώσσα θα επεκταθεί ώστε να περιέχει ακέραιες και λογικές εκφράσεις και τους αντίστοιχους τελεστές.

4.4.1 Σύνταξη

Στο σχήμα 4.20 φαίνονται οι επεκτάσεις που πρέπει να γίνουν στη σύνταξη της γλώσσας.

4.4.2 Κανόνες Συστήματος Τύπων

Στο σχήμα 4.21 φαίνονται οι επιπλέον κανόνες τύπων. Ιδιαίτερη προσοχή αξίζει να δοθεί στον κανόνα if: οι δύο υποεκφράσεις e_1 και e_2 πρέπει να εξετασθούν στο ίδιο περιβάλλον Γ_2 αφού μόνο μία από αυτές θα εκτελεστεί.

4.4.3 Λειτουργική Σημασιολογία

Οι κανόνες αποτίμησης φαίνονται στο σχήμα 4.22. Σε αυτούς χρησιμοποιείται ο συμβολισμός $\|v_1\ bop\ v_2\|$ για να δηλώσει την τιμή της αποτιμημένης έκφρασης $v_1\ bop\ v_2$ (αντίστοιχα για τον εναδικό τελεστή).

$$\frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\text{new } e \hookrightarrow S';\mu';\text{new } e'}\ \text{nPr} \qquad \frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\text{free } e \hookrightarrow S';\mu';\text{free } e'}\ \text{frPr}$$

$$\frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\text{deref } e \hookrightarrow S';\mu';\text{deref } e'}\ \text{dePr}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1 := e_2 \hookrightarrow S';\mu';e_1' := e_2}\ \text{wAssPr1}$$

$$\frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x := e_2 \hookrightarrow S';\mu';x := e_2'}\ \text{wAssPr2}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1 :=^s e_2 \hookrightarrow S';\mu';e_1' :=^s e_2}\ \text{sAssPr1}$$

$$\frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x :=^s e_2 \hookrightarrow S';\mu';x :=^s e_2'}\ \text{sAssPr2}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1 :=: e_2 \hookrightarrow S';\mu';e_1' :=: e_2}\ \text{wSwPr1}$$

$$\frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x :=: e_2 \hookrightarrow S';\mu';x :=: e_2'}\ \text{wSwPr2}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1 :=:^s e_2 \hookrightarrow S';\mu';e_1' :=:^s e_2}\ \text{sSwPr1} \qquad \frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x :=:^s e_2 \hookrightarrow S';\mu';x :=:^s e_2'}\ \text{sSwPr2}$$

$$\frac{\text{fresh } i}{\begin{array}{c}S;\mu;\text{new } x \hookrightarrow \\ S;\mu,i \mapsto x;{}^{L}\ulcorner i, {}^{L}({}^{L}\text{cap } i, \text{loc } i)\urcorner\end{array}}\ \text{new}$$

$$\frac{\begin{array}{cc}S;w \Downarrow S_1;{}^{L}\ulcorner i, w_0\urcorner & S_1;w_0 \Downarrow S_2;{}^{L}(w_1,w_2) \\ S_2;w_1 \Downarrow S_3;{}^{L}\text{cap } i & S_3;w_2 \Downarrow S_4;\text{loc } i\end{array}}{S;\mu,i \mapsto z;\text{free } w \hookrightarrow S_4;\mu;z}\ \text{free}$$

$$\frac{S;w \Downarrow S_1;{}^{\text{U at }\bot}(w_1,w_2) \quad S_1;w_1 \Downarrow S_2;{}^{\text{U at }\pi}\text{cap } i \quad S_2;w_2 \Downarrow S_3;\text{loc } i}{S;\mu,i \mapsto z;\text{deref } w \hookrightarrow S_3;\mu,i \mapsto z;z}\ \text{deref}$$

$$\frac{S;x \Downarrow S_1;{}^{\text{U at }\bot}(x_1,x_2) \quad S_1;x_1 \Downarrow S_2;{}^{\text{U at }\pi}\text{cap } i \quad S_2;x_2 \Downarrow S_3;\text{loc } i}{S;\mu,i \mapsto z;x := y \hookrightarrow S_3;\mu,i \mapsto y;\text{unit}}\ \text{wAss}$$

$$\frac{S;x \Downarrow S_1;{}^{L}(x_1,x_2) \quad S_1;x_1 \Downarrow S_2;{}^{L}\text{cap } i \quad S_2;x_2 \Downarrow S_3;\text{loc } i}{S;\mu,i \mapsto z_1;x :=^s y \hookrightarrow S_4;\mu,i \mapsto y;{}^{L}\text{cap } i}\ \text{sAss}$$

$$\frac{S;x \Downarrow S_1;{}^{q}(x_1,x_2) \quad S_1;x_1 \Downarrow S_2;{}^{\text{U at }\pi}\text{cap } i \quad S_2;x_2 \Downarrow S_3;\text{loc } i}{S;\mu,i \mapsto z_1;x :=: y \hookrightarrow S_4;\mu,i \mapsto y;z_1}\ \text{wSw}$$

$$\frac{S;x \Downarrow S_1;{}^{L}(x_1,x_2) \quad S_1;x_1 \Downarrow S_2;{}^{L}\text{cap } i \quad S_2;x_2 \Downarrow S_3;\text{loc } i}{S;\mu,i \mapsto z_1;x :=:^s y \hookrightarrow S_4;\mu,i \mapsto y;{}^{L}(z_1,{}^{L}\text{cap } i)}\ \text{sSw}$$

Σχήμα 4.18: Κανόνες αποτίμησης: Διαχείριση Μνήμης

4.5 Η υλοποίηση

Σύμφωνα με τους κανόνες που παρουσιάστηκαν έχουν δημιουργηθεί ένας ελεγκτής τύπων και ένας διερμηνέας.

Στην υλοποίηση που θα παρουσιαστεί έχουν χρησιμοποιηθεί τα μετα-εργαλεία της ocaml ocamllex και ocamlyacc για την παραγωγή του λεκτικού και συντακτικού αναλυτή [Lero10]. Σαν ενδιάμεση

$$\frac{S; \mu; e_1 \hookrightarrow S'; \mu'; e_1'}{\begin{array}{c} S; \mu; \text{at } \rho \text{ let! } (x = e_1) \text{ then } y = e_2 \text{ in } e \hookrightarrow \\ S'; \mu'; \text{at } \rho \text{ let! } (x = e_1') \text{ then } y = e_2 \text{ in } e \end{array}} \text{ let!Pr}$$

$$\frac{S; \mu; e_2 \hookrightarrow S'; \mu'; e_2'}{\begin{array}{c} S; \mu; \text{at } \rho \text{ let\$ } (x = z) \text{ then } y = e_2 \text{ in } e \hookrightarrow \\ S'; \mu'; \text{at } \rho \text{ let\$ } (x = z) \text{ then } y = e_2' \text{ in } e \end{array}} \text{ let\$Pr}$$

$$\frac{\text{fresh } \rho'}{\begin{array}{c} S, z \mapsto^{L} u; \mu; \text{at } \rho \text{ let! } (x = z) \text{ then } y = e_1 \text{ in } e_2 \hookrightarrow \\ S, z \mapsto^{\text{U at } \rho'} u; \mu; \text{at } \rho' \text{ let\$ } (x = z) \text{ then } y = e_1[\rho \mapsto \rho'][x \mapsto z] \text{ in } e_2 \end{array}} \text{ let!}$$

$$\frac{}{\begin{array}{c} S, z \mapsto^{q} u; \mu; \text{at } \rho \text{ let\$ } (x = z) \text{ then } y = w \text{ in } e_2 \hookrightarrow \\ S, z \mapsto^{L} u; \mu; e_2[x \mapsto z][y \mapsto w] \end{array}} \text{ let\$}$$

Σχήμα 4.19: Κανόνες αποτίμησης: Let!

$$\begin{array}{lll} \tau & ::= & \ldots \mid \text{Bool} \mid \text{Int} \\ \mathit{int} & ::= & 0 \mid 1 \mid \ldots \\ \mathit{iiop} & ::= & + \mid - \\ \mathit{ibop} & ::= & < \mid <= \mid > \mid >= \mid = \mid != \\ \mathit{bbop} & ::= & \text{or} \mid \text{and} \end{array} \qquad \begin{array}{lll} \mathit{bop} & ::= & \text{iiop} \mid \text{ibop} \mid \mathit{bbop} \\ \mathit{uop} & ::= & \text{not} \\ e & ::= & \ldots \mid \text{int} \mid \text{true} \mid \text{false} \mid e_1 \; \mathit{bop} \; e_2 \\ & \mid & \mathit{unop} \; e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\ v & ::= & \ldots \mid \text{int} \mid \text{true} \mid \text{false} \end{array}$$

Σχήμα 4.20: Επεκτάσεις: Σύνταξη

$$\boxed{\Gamma; \Delta; Z; M \vdash e : \tau}$$

$$\frac{U(\Gamma)}{\Gamma; \Delta; Z; M \vdash \text{true} : \text{Bool}} \text{ true} \qquad \frac{U(\Gamma)}{\Gamma; \Delta; Z; M \vdash \text{false} : \text{Bool}} \text{ false}$$

$$\frac{U(\Gamma)}{\Gamma; \Delta; Z; M \vdash \text{int} : \text{Int}} \text{ int} \qquad \frac{\begin{array}{c} \Gamma_1; \Delta; Z_1; M \vdash e_1 : \text{Int} \\ \Gamma_2; \Delta; Z_2; M \vdash e_2 : \text{Int} \end{array}}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1 \; \mathit{iiop} \; e_2 : \text{Int}} \text{ iiop}$$

$$\frac{\begin{array}{c} \Gamma_1; \Delta; Z_1; M \vdash e_1 : \text{Int} \\ \Gamma_2; \Delta; Z_2; M \vdash e_2 : \text{Int} \end{array}}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1 \; \mathit{ibop} \; e_2 : \text{Bool}} \text{ ibop} \qquad \frac{\Gamma; \Delta; Z; M \vdash e : \text{Bool}}{\Gamma; \Delta; Z; M \vdash \mathit{unop} \; e : \text{Bool}} \text{ unop}$$

$$\frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : \text{Bool} \Gamma_2; \Delta; Z_2; M \vdash e_2 : \text{Bool}}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1 \; \mathit{bbop} \; e_2 : \text{Bool}} \text{ bbop}$$

$$\frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : \text{Bool} \quad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau \quad \Gamma_2; \Delta; Z_3; M \vdash e_3 : \tau}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2 \cup Z_3; M \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ if}$$

Σχήμα 4.21: Επεκτάσεις: Κανόνες Συστήματος Τύπων

γλώσσα έχει χρησιμοποιηθεί το αφηρημένο συντακτικό δέντρο όπως παράγεται από τον συντακτικό αναλυτή.

Για κάθε (λεκτικά και συντακτικά ορθό) πρόγραμμα της γλώσσας, καλείται η συνάρτηση $tinf\ e$, όπου e το αφηρημένο συντακτικό δέντρο του προγράμματος. Αν αυτή επιτύχει, τότε επιστρέφεται ο τύπος του προγράμματος και καλείται $evaluate\ e$ που υπολογίζει την τιμή του προγράμματος.

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1\ bop\ e_2 \hookrightarrow S';\mu';e_1'\ bop\ e_2}\ \text{bopPr1} \qquad \frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x\ bop\ e_2 \hookrightarrow S';\mu';x\ bop\ e_2'}\ \text{bopPr2}$$

$$\frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;unop\ e \hookrightarrow S';\mu';unop\ e'}\ \text{unopPr}$$

$$\frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\text{if } e \text{ then } e_1 \text{ else } e_2 \hookrightarrow S';\mu';\text{if } e' \text{ then } e_2 \text{ else } e_3}\ \text{ifPr}$$

$$\frac{\begin{array}{c} fresh\ z \\ S;x \Downarrow S_1;v_1 \quad S_1;y \Downarrow S_2;v_2 \end{array}}{S;\mu;x\ bop\ y \hookrightarrow S_2, z \mapsto \|v_1\ bop\ v_2\|;\mu;z}\ \text{bop}$$

$$\frac{\begin{array}{c} fresh\ z \\ S;x \Downarrow S';v \end{array}}{S;\mu;unop\ x \hookrightarrow S', z \mapsto \|bop\ v\|;\mu;z}\ \text{unop}$$

$$\frac{S;z \Downarrow S';\text{true}}{S;\mu;\text{if } z \text{ then } e_1 \text{ else } e_2 \hookrightarrow S';\mu;e_1}\ \text{if1} \qquad \frac{S;z \Downarrow S';\text{false}}{S;\mu;\text{if } z \text{ then } e_1 \text{ else } e_2 \hookrightarrow S';\mu;e_2}\ \text{if2}$$

Σχήμα 4.22: Επεκτάσεις: Κανόνες Αποτίμησης

4.5.1 Ελεγκτής Τύπων

Οι κανόνες τύπων που παρουσιάστηκαν αποτελούν τον σκελετό της υλοποίησης του ελεγκτής τύπων. Οι κανόνες όμως αυτοί είναι μη-ντετερμινιστικοί και δεν μπορούν να υλοποιηθούν άμεσα. Απαιτούνται λοιπόν κάποιες διαφοροποιήσεις κυρίως όσον αφορά τον χειρισμό των περιβαλλόντων Γ και Z:

Το περιβάλλον Γ

Ο χειρισμός του περιβάλλοντος Γ στηρίχθηκε στον αλγοριθμικό ελεγκτή τύπων του Walker [Walk05].

Συγκεκριμένα, το περιβάλλον Γ υλοποιείται σαν έναν global hashtable. Για να εξασφαλιστεί η γραμμική συμπεριφορά αρκεί να εξασφαλιστεί ότι κάθε γραμμική μεταβλητή χρησιμοποιείται ακριβώς μία φορά.

Όταν στον ελεγκτή εφαρμόζεται ο κανόνας της μεταβλητής, ο τύπος της μεταβλητής ανακτάται από το hashtable και αν ο qualifier είναι linear η μεταβλητή αφαιρείται από το περιβάλλον. Άρα, κάθε γραμμική μεραβλητή χρησιμοποιείται το πολύ μία φορά.

Ο έλεγχος για το αν κάθε γραμμική μεταβλητή χρησιμοποιείται τουλάχιστον μία φορά γίνεται στους κανόνες που εισάγουν μεταβλητές. Για παράδειγμα, στην έκφραση let $(x, y) = e_1$ in e_2 αρχικά ελέγχεται ότι η έκφραση e_1 είναι τύπου ${}^q\langle\tau_1 * \tau_2\rangle$, έπειτα εισάγονται στο Γ τα ζεύγη (x, τ_1), (y, τ_2) και ελέγχεται ο τύπος της έκφρασης e_2, τέλος, οι μεταβλητές x και y ”αφαιρούνται” από το περιβάλλον με την συνάρτηση $removeGamma$. Η συνάρτηση αυτή όταν κληθεί με όρισμα ένα όνομα μεταβλητής και τον τύπο της $(x,^q \phi)$, αν η μεταβλητή είναι unrestricted, την αφαιρεί από το Γ, διαφορετικά ελέγχει την ύπαρξή της στο Γ: αν η μεταβλητή υπάρχει σημαίνει ότι μία γραμμική μεταβλητή δεν έχει χρησιμοποιηθεί και ο ελεγκτής τερματίζει με σφάλμα.

Ο ελεγκτής τύπων πρέπει να ελέγχει και ότι μια unrestricted συνάρτηση δεν χρησιμοποιεί γραμμικές μεταβλητές. Αυτό επιτυγχάνεται ελέγχοντας το μέγεθος του περιβάλλοντος Γ πριν και μετά τον έλεγχο της συνάρτησης, αφού όταν ο έλεγχος του τύπου της συνάρτησης τελειώσει το μέγεθος του πίνακα Γ θα έχει μειωθεί αν και μόνο αν στο σώμα της συνάρτησης έχει χρησιμοποιηθεί κάποια γραμμική μεταβλητή. Αυτό εξασφαλίζεται αφού οποιαδήποτε μεταβλητή οριστεί στο σώμα της συνάρτησης θα έχει εμβέλεια μόνο μέσα σε αυτό και θα διαγραφεί από το Γ πριν ολοκληρωθεί ο έλεγχος.

Το περιβάλλον Z

Το περιβάλλον Z, όπως έχει αναφερθεί, απαιτεί relevant χειρισμό, έτσι ώστε να μπορεί να καθοριστεί το ελάχιστο σύνολο scopes που χρησιμοποιούνται τουλάχιστον μια φορά από κάποια έκφραση. Για τον σκοπό αυτό κατά την υλοποίηση διατηρείται μία global λίστα $zetta$ όλων των ενεργών scopes, δηλαδή των scopes που έχουν εισαχθεί από κάποια εντολή $let!$. Για κάθε έκφραση που ελέγχεται επιστρέφεται η λίστα των scopes που χρησιμοποιήθηκαν κατά τον έλεγχο.

Όταν γίνεται έλεγχος σε κάποια έκφραση που πρέπει να ελεγχθεί η εγκυρότητα κάποιου scope (όπως στις var, deref και free) αρχικά ελέγχεται αν το scope ανήκει στο $zetta$, αν όχι ο ελεγκτής τερματίζει με σφάλμα. Αν ναι, το scope προστίθεται στην λίστα των scope που χρησιμοποιήθηκαν.

Όταν γίνεται έλεγχος σε κάποια συνάρτηση το scope της συνάρτησης ελέγχεται για την εγκυρότητά του και προστίθεται στην λίστα των απαιτούμενων scopes, ενώ τα scopes που απαιτούνται για τον έλεγχο του σώματος της συνάρτησης επισημειώνονται στον τύπο της. Τέλος, όταν ελέγχεται κάποια εφαρμογή $e_1 e_2$, απαιτείται να είναι έγκυρα και τα scopes που είναι επισημειωμένα στον τύπο της e_1.

Το περιβάλλον Δ

Το περιβάλλον Δ υλοποιείται σαν μία global λίστα και σε αυτό διατηρούνται οι μεταβλητές θέσεων μνήμης. Πριν ελεγχθεί η υποέκφραση e_2 της έκφρασης let $\ulcorner l, x \urcorner = e_1$ in e_2 εισάγεται στον περιβάλλον η μεταβλητή l και αφαιρείται όταν ο έλεγχος της e_2 ολοκληρωθεί.

4.5.2 Διερμηνέας

Οι κανόνες της λειτουργικής σημασιολογίας είναι ντετερμινιστικοί και μπορούν να υλοποιηθούν άμεσα. Αξίζει να σημειωθεί ότι τα περιβάλλοντα Σ και μ έχουν υλοποιηθεί σαν global hashtables και ότι κάθε φορά που αναζητείται μία linear μεταβλητή από το Σ διαγράφεται.

4.6 Παραδείγματα

Για την καλύτερη κατανόηση της γλώσσας παρουσιάζονται ορισμένα παραδείγματα. Τα παραδείγματα των υποενοτήτων 4.6.1 - 4.6.3 αποτυγχάνουν, αφού επιχειρείται διαφυγή του scope σε περιβάλλον που δεν είναι έγκυρο. Στις υποενότητες 4.6.4 - 4.6.6 παρουσιάζονται ορθά παραδείγματα.

4.6.1 Αφελής προσπάθεια διαφυγής scope

Στο ακόλουθο παράδειγμα δημιουργείται μία θέση μνήμης και το capability της μετατρέπεται σε unrestricted μέσω του $let!$. Το unrestricted capability διαφεύγει μέσω του y από το $let!$, όμως όταν η μεταβλητή y χρησιμοποιηθεί θα διαπιστωθεί ότι το scope της δεν είναι έγκυρο:

Παράδειγμα 14

let $n =$ new $unit$ in $\qquad (*n :^L \exists l.^L \langle^L \text{Cap } l \text{ Unit} * \text{Ref } l \rangle *)$
let $\ulcorner r, p \urcorner = n$ in $\qquad (*p :^L \langle^L \text{Cap } l \text{ Unit} * \text{Ref } l \rangle *)$
let $(c, re) = p$ in $\qquad (*c :^L \text{Cap } l \text{ Unit}, re : \text{Ref } l *)$
at h let! $(x = c)$
  then $y = x$ $\qquad (*y :^{U \text{ at } h} \text{Cap } l \text{ Unit} *)$
in
free $\ulcorner r, (x, re) \urcorner$;
deref (y, re) $\qquad (*y :^{U \text{ at } h} \text{Cap } l \text{ Unit} *)$

Ο ελεγκτή τερματίζει με μήνυμα: *scope of var y : U at sc@1 is not in zetta : []*

4.6.2 Προσπάθεια διαφυγής scope μέσω συνάρτησης

Στο ακόλουθο παράδειγμα το unrestricted capability προσπαθεί να διαφύγει μέσω του κλεισίματος μιας συνάρτησης. Και πάλι όμως αποτυγχάνει, αφού ο τύπος της συνάρτησης έχει σημειωθεί με το αντίστοιχο scope το οποίο και απαιτείται κατά την εφαρμογή της:

Παράδειγμα 15
let $n =$ new $unit$ in
let $\ulcorner r, p \urcorner = n$ in
let $(c, re) = p$ in
at h let! $(x = c)$
  then $y =^{U \text{ at } \bot} \lambda u$: Unit.deref (x, re) $\quad (*y :^{U \text{ at } \bot} \text{Unit} \xrightarrow{h} \text{Unit}*)$
 in
(free $\ulcorner r, (x, re) \urcorner$); $(y\ unit)$

Ο ελεγκτή τερματίζει με μήνυμα: *fun scopes: [sc@1] are not in zetta: []*, όπου $sc@1$ η μετονομασία του scope h.

4.6.3 Προσπάθεια διαφυγής μέσω Ανάθεσης στην Μνήμη

Στο ακόλουθο παράδειγμα το unrestricted capability προσπαθεί να διαφύγει μέσω ανάθεσης στην μνήμη. Και πάλι όμως αποτυγχάνει, αφού όταν επιχειρηθεί η χρήση του θα διαπιστωθεί ότι το scope του δεν είναι έγκυρο:

Παράδειγμα 16
let $n =$ new $unit$ in
let $\ulcorner l, p \urcorner = n$ in
let $(c, re) = p$ in
let $n1 =$ new $unit$ in
let $\ulcorner l1, p1 \urcorner = n1$ in
let $(c1, re1) = p1$ in
at $h1$ let! $(x = c1)$
  then $y = (c, re) :=^s x$
 in
let $(c11, c01) = (y, re) :=:^s unit$ in
(free $\ulcorner l, (c01, re) \urcorner$;
free $\ulcorner l1, (x, re1) \urcorner$);
deref $(c11, re1)$ $\quad (*c11 :^{U \text{ at } h} \text{Cap } l \text{ Unit}*)$

Ο ελεγκτή τερματίζει με μήνυμα: *scope of var c11 : U at sc@1 is not in zetta : []*, όπου $sc@1$ η μετονομασία του scope h.

4.6.4 Ανάθεση σε θέση μνήμης γραμμικό περιεχόμενο

Στο ακόλουθο παράδειγμα, αναθέτουμε ένα linear πακέτο r σε μία νέα θέση μνήμης και χρησιμοποιούμε το $let!$ για να αλλάξουμε τα περιεχόμενά της.

Παράδειγμα 17
let $r =$ new 7 in $\quad (*r :^L \exists l.^L \langle {}^L\text{Cap } l \text{ Int} * \text{Ref } l \rangle \equiv {}^L\text{Xref Int}*)$
let $p =$ new r in $\quad (*p :^L \exists l.^L \langle {}^L\text{Cap } l\ {}^L\text{Xref Int} * \text{Ref } l \rangle *)$
let $\ulcorner r1, p1 \urcorner = p$ in
let $(c1, re1) = p1$ in $\quad (*c1 :^L \text{Cap } l\ {}^L\text{Xref Int}*)$
at h let! $(x = c1)$
  then $y =$

let $s = (x, re1)$:=: new 8 $(*s : {}^{L}\text{Xref Int}*)$
in free s $(*\text{free } s \rhd 7*)$
in
free (free $\ulcorner r1, (x, re1)\urcorner$) $(*\text{free } \ulcorner r1, (x, re1)\urcorner : {}^{L}\text{Xref Int}*)$
$\rhd$ 8 : Int

Εναλλακτικά, θα μπορούσαμε να χρησιμοποιούσαμε την ισχυρή ανταλλαγή:

Παράδειγμα 18
let r = new 7 in $(*r :^{L} \exists l.^{L}\langle {}^{L}\text{Cap } l \text{ Int} * \text{Ref } l\rangle \equiv {}^{L}\text{Xref Int}*)$
let p = new r in $(*p :^{L} \exists l.^{L}\langle {}^{L}\text{Cap } l\ {}^{L}\text{Xref Int} * \text{Ref } l\rangle*)$
let $\ulcorner r, pack\urcorner = p$ in
let $(c1, re1) = pack$ in $(*c1 :^{L} \text{Cap } l\ {}^{L}\text{Xref Int}*)$
let $(oldr, c2) = (c1, re1)$:=:^{s} new 8 in $(*oldr : {}^{L}\text{Xref Int}, c2 :^{L} \text{Cap } l\ {}^{L}\text{Xref Int}*)$
free $oldr$; $(*\text{free } oldr \rhd 7*)$
free (free $\ulcorner r, (c2, re1)\urcorner$) $(*\text{free } \ulcorner r, (c2, re1)\urcorner : {}^{L}\text{Xref Int}*)$
$\rhd$ 8 : Int

4.6.5 Συνάρτηση για ανάγνωση περιεχομένου θέσης μνήμης

Στο ακόλουθο παράδειγμα ορίζουμε μία συνάρτηση που δέχεται ένα γραμμικό πακέτο μιας θέσης μνήμης, και χρησιμοποιώντας το $let!$ αποδεσμεύει την θέση μνήμης και επιστρέφει το περιεχόμενό της:

Παράδειγμα 19
let $test =^{U \text{ at } \bot} \lambda x : {}^{L}\text{Xref Unit}.$
let $\ulcorner r1, p\urcorner = x$ in
let $(c1, re1) = p$ in
at h let! $(x = c1)$
then y = deref $(x, re1)$
in
free $\ulcorner r1, (x, re1)\urcorner; y$
in
let z = new $unit$ in
$test\ z$
$\rhd$ $unit$: Unit

4.6.6 Fibonacci

Σαν τελευταίο παράδειγμα δίνεται ο υπολογισμός του $i-$ όρου της ακολουθίας fibonacci:

Ο υπολογισμός θα γίνει με δύο τρόπος με αναδρομή και με επανάληψη. Και στις δύο υλοποιήσεις ορίζεται μέσω των αναφορών μία αναδρομική συνάρτηση και φαίνεται η εκφραστική δυνατότητα της γλώσσας.

Σαν πρώτη υλοποίηση παρουσιάζεται ο αφελής υπολογισμός όπου η αναδρομική συνάρτηση υπολογίζει $fibn = fib(n-1) + fib(n-2)$

Παράδειγμα 20
let $fibonacci =^{U \text{ at } \bot} \lambda n$: Int.
let f = new $^{U \text{ at } \bot}\lambda x$: Int.x in
let $\ulcorner r, p\urcorner = f$ in
let $(c, re) = p$ in
at h let! $(x = c)$
then y =
let $fib =^{U \text{ at } \bot} \lambda num$: Int.

if $num < 2$ then 1 else deref (x, re) $num - 1 +$ deref (x, re) $num - 2$ in
$(x, re) := fib; fib\ n$
in
free $\ulcorner r, (x, re) \urcorner; y$ in
$fibonacci\ 7$
$\triangleright$ 21 : Int

Μια πιο αποδοτική υλοποίηση φαίνεται στο παράδειγμα 21. Σε αυτό η κύρια συνάρτηση δέχεται τέσσερα ορίσματα $(n, (k, (i, j)))$, όπου $j = fib_k$ και $i = fib_{k-1}$ και η το αρχικό όρισμα. Αν $n = k$,τότε επιστρέφεται ο j διαφορετικά υπολογίζεται ο f_{k+1} καλώντας την συνάρτηση με όρισμα $(n, ((k + 1), (j, (i + j))))$.

Παράδειγμα 21
let $fibonacci =^{U \text{ at } \bot} \lambda n$: Int.
let $f =$ new $^{U \text{ at } \bot} \lambda x : {}^{U \text{ at } \bot}\langle \text{Int} * {}^{U \text{ at } \bot}\langle \text{Int} * {}^{U \text{ at } \bot}\langle \text{Int} * \text{Int}\rangle\rangle\rangle.0$ in
let $\ulcorner r, p \urcorner = f$ in
let $(c, re) = p$ in
at h let! $(x = c)$
then $y =$
let $fib =^{U \text{ at } \bot} \lambda x1 : {}^{U \text{ at } \bot}\langle \text{Int} * {}^{U \text{ at } \bot}\langle \text{Int} * {}^{U \text{ at } \bot}\langle \text{Int} * \text{Int}\rangle\rangle\rangle.$
let $(num, n1) = x1$ in
let $(k, n2) = n1$ in
let $(i, j) = n2$ in
if $num = k$ then j else deref (x, re) $(num, (k + 1, (j, i + j)))$
in
$(x, re) := fib;$
if $n = 1$ or $n = 0$ then 1 else $fib\ (n, (1, (1, 1)))$
in
free $\ulcorner r, (x, re) \urcorner; y$ in
$fibonacci\ 7$
$\triangleright$ 21 : Int

Κεφάλαιο 5

Εφαρμογή σε γλώσσες ταυτόχρονου προγραμματισμού

Στο κεφάλαιο αυτό η γλώσσα θα επεκταθεί έτσι ώστε να υποστηρίζει τον παράλληλο υπολογισμό δύο εκφράσεων.

5.1 Εισαγωγή

Στο κεφάλαιο 4 είδαμε ότι για να γίνει μία πρόσβαση στην μνήμη πρέπει να παρέχεται το κατάλληλο capability. Αν το capability επεκταθεί έτσι ώστε να παρέχει συγκεκριμένα δικαιώματα πρόσβασης (δικαίωμα ανάγνωσης, εγγραφής, αποδέσμευσης) η γλώσσα μπορεί να επεκταθεί ώστε να υποστηρίζει ταυτόχρονο προγραμματισμό με κοινόχρηστη μνήμη χωρίς την εμφάνιση συνθηκών ανταγωνισμού δεδομένων.

Κατά τον ταυτόχρονο προγραμματισμό με κοινόχρηστη μνήμη όλες οι θέσεις μνήμης που έχουν δημιουργηθεί μπορούν να προσπελαστούν από πολλά νήματα δημιουργώντας συνθήκες ανταγωνισμού δεδομένων. Μία συνθήκη ανταγωνισμού δεδομένων εμφανίζεται όταν παράλληλα νήματα προσπελάσουν την ίδια κοινόχρηστη θέση μνήμης με μη καθορισμένη σειρά και τουλάχιστον μία προσπέλαση είναι εγγραφή [Chen06]. Η κατάσταση αυτή οδηγεί σε απροσδιόριστα αποτελέσματα:

- αν δύο παράλληλα νήματα γράφουν στην ίδια θέση μνήμης

  $$\ldots\ r := v_1; \ldots || \ldots\ r := v_2; \ldots$$

  η τιμή της θέσης μνήμης όταν ο έλεγχος επιστρέψει στο κύριο νήμα εξαρτάται από την σειρά που εκτελέστηκαν οι δύο εγγραφές και είναι απροσδιόριστη.

- αν κατά την εκτέλεση δύο παράλληλων νημάτων το ένα γράφει και το άλλο διαβάζει την ίδια θέση μνήμης

  $$\ldots\ r := v_1; \ldots || \ldots !r \ldots$$

  η τιμή που θα διαβάσει το νήμα εξαρτάται από το αν η ανάγνωση θα γίνει πριν ή μετά την εγγραφή και είναι απροσδιόριστη.

Για να αποφευχθούν οι ανταγωνισμοί δεδομένων αρκεί όταν ένα νήμα έχει δικαίωμα εγγραφής σε μία θέση μνήμης, κανένα άλλο νήμα που εκτελείται παράλληλα να μην έχει δικαίωμα προσπέλασης αυτής της θέσης μνήμης, Στη γλώσσα που θα παρουσιαστεί στο κεφάλαιο αυτό για να γίνει εγγραφή σε μία θέση μνήμης πρέπει να παρέχεται capability με κατάλληλες ιδιότητες. Επιπλέον, με τον τρόπο που διασπάται το περιβάλλον Γ εξασφαλίζεται ότι το capability που παρέχει δικαίωμα εγγραφής για κάποια θέση μνήμης δεν μπορεί να βρίσκεται σε πάνω από ένα νήματα που εκτελούνται παράλληλα και ότι δύο capabilities που το ένα παρέχει δικαίωμα εγγραφής και το άλλο δικαίωμα ανάγνωσης της ίδια θέσης μνήμης, δεν μπορούν να βρίσκονται σε νήματα που εκτελούνται παράλληλα.

5.2 Σύνταξη

Η σύνταξη της γλώσσας θα διαφοροποιηθεί όπως φαίνεται στο σχήμα 5.1.

$$
\begin{array}{rcl}
s & ::= & \mathrm{L} \,|\, \mathrm{T} \,|\, \mathrm{R} \,|\, \mathrm{U} \\
\pi & ::= & \rho \,|\, \bot \\
q & ::= & s \text{ at } \pi \\
cl & ::= & \overrightarrow{{}^{q}\text{cap } i} \\
e & ::= & \ldots \,|\, {}^{cl_1}e_1 || {}^{cl_2}e_2 \,|\, n_1 : e_1 \# n_2 : e_2 \\
& | & \text{at } \rho \text{ wlock } (x = e_1) \text{ then } y = e_2 \text{ unlock } e \,|\, \text{at } \rho \text{ rlock } (x = e_1) \text{ then } y = e_2 \text{ unlock } e \\
& | & \text{at } \rho \text{ wlock\$ } (x = z) \text{ then } y = e_2 \text{ unlock } e \,|\, \text{at } \rho \text{ rlock\$ } (x = z) \text{ then } y = e_2 \text{ unlock } e \\
& | & \text{at } \rho \; {}^{w}\text{let! } (x = e_1) \text{ then } y = e_2 \text{ in } e \,|\, \text{at } \rho \; {}^{r}\text{let! } (x = e_1) \text{ then } y = e_2 \text{ in } e
\end{array}
$$

Σχήμα 5.1: Σύνταξη

5.2.1 Οι καταστάσεις s

Η χρησιμότητα των καταστάσεων είναι ότι χαρακτηρίζουν τα capabilities, δηλαδή τα δικαιώματα πρόσβασης της μνήμης. Αν το capability cap i έχει κατάσταση U (Unrestricted), τότε δεν παρέχει κανένα δικαίωμα πρόσβασης στη θέση μνήμης i. Αν το capability cap i έχει κατάσταση R (Readable), τότε παρέχει δικαίωμα ανάγνωσης της θέσης μνήμης i. Αν το capability cap i έχει κατάσταση (Thread-Exclusive), τότε παρέχει δικαίωμα ανάγνωσης και εγγραφής της θέσης μνήμης i. Τέλος, αν το capability cap i έχει κατάσταση L (Linear), τότε παρέχει όλα τα δικαίωμα πρόσβασης στη θέση μνήμης i, δηλαδή επιτρέπει εγγραφή και αποδέσμευση. Το capability με κατάσταση L έχει linear συμπεριφορά, δηλαδή κάθε φορά που χρησιμοποιείται καταναλώνεται. Για τον λόγο αυτό δεν παρέχει δικαίωμα ανάγνωσης, αφού η με την ανάγνωση το capability δεν επιστρέφεται.

Λέμε ότι μία κατάσταση s_1 είναι πιο περιοριστική από την κατάσταση s_2 και γράφουμε $s_1 \sqsubseteq s_2$ όταν κάποιο capability με κατάσταση s_2 παρέχει περισσότερα δικαιώματα για την αντίστοιχη θέση μνήμης από κάποιο capability με κατάσταση s_1. Η σχέση $\sqsubseteq$ ορίζεται στο σχήμα 5.2.

$\boxed{s_1 \sqsubseteq s_2}$

$$L \sqsubseteq T \sqsubseteq R \sqsubseteq U$$
$$s \sqsubseteq s$$

Σχήμα 5.2: Σχέση των καταστάσεων

5.2.2 Οι qualifiers q

Στη σειριακή γλώσσα ένας qualifier μπορούσε να ήταν είτε L είτε U at π. Επεκτείνοντας την γλώσσα εισάγουμε εκφράσεις που όπως θα δούμε μπορούν να αλλάξουν την κατάσταση ενός capability από U σε T ή R ενώ η πληροφορία για το scope πρέπει να διατηρηθεί. Άρα, οι qualifiers πρέπει επιπλέον να μπορούν να είναι T at π και R at π. Για απλοποίηση θεωρούμε ότι οι qualifiers έχουν πλέον την μορφή s at π, αλλά πρέπει να τονιστεί ότι αν η κατάσταση ενός qualifier είναι linear, η τιμή του scope μπορεί να είναι μόνο botton ($\bot$) και είναι ουσιαστικά περιττή.

5.2.3 Οι εκφράσεις e

Η έκφραση ${}^{c_1}e_1||{}^{c_2}e_2$ υπολογίζει παράλληλα τις δύο υποεκφράσεις e_1 και e_2 και επιστρέφει ένα ζεύγος με τα αντίστοιχα αποτελέσματα. Κάθε έκφραση παρατηρούμε ότι είναι επισημειωμένη με μια λίστα από τα capabilities που χρησιμοποιούνται. Η λίστα αυτή απαιτείται να είναι γνωστή από τους

κανόνες αποτίμησης και καθορίζει πως ένα thread κληροδοτεί τα locks του στα νέα thread που δημιουργεί, όμως δεν είναι απαραίτητο να γραφτεί από τον προγραμματιστή αφού μπορεί να προκύψει από τον ελεγκτή τύπων (όπως εξηγείται στην ενότητα 5.5.1)

Η έκφραση $n_1 : e_1 \# n_2 : e_2$ προκύπτει κατά την αποτίμηση της ${}^{c_1}e_1||^{c_2}e_2$ και δεν μπορεί να χρησιμοποιηθεί από τον προγραμματιστή. Η ερμηνεία της είναι ότι η έκφραση e_1 θα αποτιμηθεί από ένα νήμα με κωδικό n_1 και η e_2 από ένα νήμα με κωδικό n_2.

Με τις εκφράσεις at ρ rlock $(x = e_1)$ then $y = e_2$ unlock e και at ρ wlock $(x = e_1)$ then $y = e_2$ unlock e γίνεται αλλαγή στην κατάσταση ενός capability. Η σύνταξή τους είναι αντίστοιχη της $let!$. Η έκφραση e_1 πρέπει να είναι κάποιο capability κατάλληλης κατάστασης, κατά την αποτίμηση της έκφρασης e_2 η κατάσταση του capability γίνεται R και T αντίστοιχα και τέλος, κατά την αποτίμηση της e η κατάσταση του capability έχει επιστρέψει στην αρχική, ενώ το αποτέλεσμα της αποτίμησης της έκφρασης e_2 έχει ανατεθεί στη μεταβλητή y.

Οι εκφράσεις at ρ rlock\$ $(x = z)$ then $y = e_2$ unlock e_3 και at ρ wlock\$ $(x = z)$ then $y = e_2$ unlock e_3 είναι απαραίτητες για την αποτίμηση των εκφράσεων και δεν μπορούν να χρησιμοποιηθούν από τον προγραμματιστή.

Τέλος οι εκφράσεις at ρ ^{w}let! $(x = e_1)$ then $y = e_2$ in e και at ρ ^{r}let! $(x = e_1)$ then $y = e_2$ in e είναι αντίστοιχες του let! μόνο που στο εσωτερικό της έκφρασης e_2 το capability που γίνεται banged αποκτά κατάσταση T ή R αντίστοιχα, αντί για U. Ως ενδιάμεση έκφραση για τις νέες αυτές εκφράσεις μπορεί να χρησιμοποιηθεί η έκφραση let\$, άρα δεν χρειάζεται να ορίσουμε κάποια νέα ενδιάμεση έκφραση.

5.3 Κανόνες Συστήματος Τύπων

Η σχέση για τον συμπερασμό τύπων της γλώσσας παραμένει ίδια, δηλαδή: $\Gamma; \Delta; Z; M \vdash e : \tau$. Ίδιος παραμένει και ο ορισμός των περιβαλλόντων.

5.3.1 Διάσπαση Περιβάλλοντος Γ

Η αλλαγή στην σύνταξη των qualifiers και η εισαγωγή των δύο νέων καταστάσεων επιβάλει αλλαγές στον ορισμό του τελεστή για την διάσπαση του περιβάλλοντος Γ. Δεδομένου ότι οι μεταβλητές με κατάσταση R ή T έχουν unrestricted συμπεριφορά, δηλαδή δεν μας ενδιαφέρει πόσες φορές θα χρησιμοποιηθούν, αρκεί απλώς στον παλιό ορισμό να αντικαταστήσουμε την "κατάσταση" U με $s = U \vee R \vee T$. Ο νέος ορισμός του τελεστή $\oplus$ φαίνεται στο σχήμα 5.3

$$\boxed{\Gamma = \Gamma_1 \oplus \Gamma_2}$$

$$\frac{}{\emptyset = \emptyset \oplus \emptyset}\ \text{empty} \qquad \frac{\Gamma = \Gamma_1 \oplus \Gamma_2 \quad s = T \vee R \vee U}{\Gamma, x :^{s \text{ at } \pi} \phi = \Gamma_1, x :^{s \text{ at } \pi} \phi \oplus \Gamma_2, x :^{s \text{ at } \pi} \phi}\ \text{TRU}$$

$$\frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x :^{L \text{ at } \pi} \phi = \Gamma_1, x :^{L \text{ at } \pi} \phi \oplus \Gamma_2}\ \text{L1} \qquad \frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x :^{L \text{ at } \pi} \phi = \Gamma_1 \oplus \Gamma_2, x :^{L \text{ at } \pi} \phi}\ \text{L2}$$

Σχήμα 5.3: Διάσπαση Περιβάλλοντος Γ

Επιπλέον, πρέπει να εισαχθεί ένας τελεστής που να διασπά το περιβάλλον Γ σε δύο περιβάλλοντα Γ_1 και Γ_2 που θα χρησιμοποιηθούν για τον συμπερασμό τύπων διαφορετικών υποεκφράσεων που θα υπολογιστούν παράλληλα. Για τον σκοπό αυτό αρκεί να εξεταστούν τα δικαιώματα πρόσβασης στην μνήμη που παρέχονται από κάποιο capability με κατάσταση s. Αν $s = U$, τότε δεν παρέχεται κανένα δικαίωμα, άρα το capability μπορεί να βρεθεί και στα δύο περιβάλλοντα. Αν $s = R$, τότε παρέχεται δικαίωμα ανάγνωσης, άρα το capability μπορεί να βρεθεί και στα δύο περιβάλλοντα. Αν $s = T$, τότε

παρέχεται και δικαίωμα εγγραφής, άρα το capability μπορεί να βρεθεί το πολύ σε ένα περιβάλλον. Τέλος, αν $s = L$, τότε παρέχεται και δικαίωμα αποδέσμευσης, ομοίως το capability μπορεί να βρεθεί το πολύ σε ένα περιβάλλον. Επιπλέον επιβάλλεται κάθε capability να βρεθεί σε τουλάχιστον ένα περιβάλλον έτσι ώστε να είναι δυνατόν από τα δύο περιβάλλοντα Γ_1 και Γ_2 να συνθεθεί το αρχικό περιβάλλον Γ. Η μόνη απόφαση που μένει να πάρουμε είναι αν ένα capability με κατάσταση U ή R μπορεί να βρεθεί σε ακριβώς ένα περιβάλλον: θεωρούμε ότι ένα capability με $s = R$ μπορεί να βρεθεί σε ακριβώς ένα περιβάλλον Γ_i αφού έτσι καθίσταται εφικτό η έκφραση που θα ελεγχθεί κάτω από το περιβάλλον Γ_i να αναβαθμίσει την κατάσταση του capability σε T, μέσω ενός $wlock$ χωρίς να προκύψει αδιέξοδο (deadlock), όπως φαίνεται και στο παράδειγμα της ενότητας 5.6.4. Επίσης, θεωρούμε ότι ένα capability με $s = U$ πρέπει να βρεθεί σε ακριβώς δύο περιβάλλοντα αφού δεν υπάρχει λόγος για το αντίθετο. Όλες αυτές οι απαιτήσεις οδηγούν στον ορισμό του τελεστή $\odot$ για την διάσπαση του περιβάλλοντος Γ όπως φαίνεται στο σχήμα 5.4.

$$\boxed{\Gamma = \Gamma_1 \odot \Gamma_2}$$

$$\frac{}{\emptyset = \emptyset \odot \emptyset}\ \text{empty} \qquad \frac{\Gamma = \Gamma_1 \odot \Gamma_2 \quad s = R \vee U}{\Gamma, x :^{s \text{ at } \pi} \phi = \Gamma_1, x :^{s \text{ at } \pi} \phi \odot \Gamma_2, x :^{s \text{ at } \pi} \phi}\ \text{RU}$$

$$\frac{\Gamma = \Gamma_1 \odot \Gamma_2 \quad s = L \vee T \vee R}{\Gamma, x :^{s \text{ at } \pi} \phi = \Gamma_1, x :^{s \text{ at } \pi} \phi \odot \Gamma_2}\ \text{LTR1} \qquad \frac{\Gamma = \Gamma_1 \odot \Gamma_2 \quad s = L \vee T \vee R}{\Gamma, x :^{s \text{ at } \pi} \phi = \Gamma_1 \odot \Gamma_2, x :^{s \text{ at } \pi} \phi}\ \text{LTR2}$$

Σχήμα 5.4: Διάσπαση Περιβάλλοντος Γ για παράλληλες υποεκφράσεις

Το περιβάλλον Γ χρησιμοποιείται για τον έλεγχο της εγκυρότητας των διανυσμάτων capabilities που επισημειώνουν τις εκφράσεις που πρόκειται να υπολογιστούν παράλληλα, όπως φαίνεται στο σχήμα 5.5. Η έκφραση $\Gamma \models cl$ ισχύει όταν για όλα τα capabilities που υπάρχουν στο cl υπάρχει και στο Γ η αντίστοιχη μεταβλητή αλλά και για όλες τις μεταβλητές που υπάρχουν στο Γ με τύπο $^{s \text{ at } \pi}\text{Cap } i\ \tau$ για κάποιο τ και $s = T \vee R$ υπάρχει το αντίστοιχο capability στο cl. Αυτή η απαίτηση είναι απαραίτητη για τον σωστό υπολογισμό των εκφράσεων της γλώσσας, όπως θα φανεί στην συνέχεια.

$$\boxed{\Gamma \models cl}$$

$$s = T \vee R.(^{s \text{ at } \pi}\text{cap } i \in cl \Leftrightarrow \exists x, \tau.(x :^{s \text{ at } \pi} \text{Cap } i\ \tau) \in \Gamma)$$

Σχήμα 5.5: Έλεγχος εγκυρότητας cl

5.3.2 Οι κανόνες

Στα σχήματα 5.6 - 5.12 παρουσιάζονται οι κανόνες που διαφοροποιούνται από αυτούς της ενότητας 4.2.3. Σημειώνουμε ότι η διαφοροποίηση του qualifier L σε L at $\perp$ είναι μόνο συντακτικοί και οι αντίστοιχοι κανόνες δεν επαναλαμβάνονται.

Βασικές Εκφράσεις

Από τους κανόνες για τις βασικές εκφράσεις πρέπει όπως και στην ακολουθιακή γλώσσα να εξασφαλίσουμε ότι το περιβάλλον Γ στο οποίο γίνεται ο έλεγχος δεν περιέχει γραμμικές τιμές. Το περιβάλλον όμως πλέον δεν επιβάλλεται να είναι unrestricted, αφού μπορεί να περιέχει μεταβλητές κατάστασης T ή R. Άρα οι κανόνες για τις βασικές εκφράσεις αλλάζουν όπως φαίνεται στο σχήμα 5.6.

$$\frac{s(\Gamma) \quad s \sqsupset L}{\Gamma; \Delta; Z; M \vdash \text{unit} : \text{Unit}}\ \text{unit} \qquad \frac{s(\Gamma) \quad s \sqsupset L \quad FRV(\tau) \subseteq \Delta \quad Z \models \text{qual}\ \tau}{\Gamma, x : \tau; \Delta; Z; M \vdash x : \tau}\ \text{var}$$

Σχήμα 5.6: Κανόνες τύπων: Βασικές Εκφράσεις

Παράλληλος Υπολογισμός

$$\frac{\begin{array}{c}\Gamma_1; \Delta; Z_1; M \vdash e_1 : \tau_1 \quad \Gamma_1 \models cl_1 \\ \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau_2 \quad \Gamma_2 \models cl_2 \\ s = LUB(\text{qual}\ \tau_1, \text{qual}\ \tau_2)\end{array}}{\Gamma_1 \odot \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash {}^{cl_1}e_1 || {}^{cl_2}e_2 : {}^{s\ \text{at}\ \bot}\langle \tau_1 * \tau_2 \rangle}\ \text{par}$$

$$\frac{\begin{array}{c}\Gamma_1; \Delta; Z_1; M \vdash e_1 : \tau_1 \quad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau_2 \\ s = LUB(\text{qual}\ \tau_1, \text{qual}\ \tau_2)\end{array}}{\Gamma_1 \odot \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash n_1 : e_1 \# n_2 : e_2 : {}^{s\ \text{at}\ \bot}\langle \tau_1 * \tau_2 \rangle}\ \text{par2}$$

Σχήμα 5.7: Κανόνες Τύπων: Παράλληλος Υπολογισμός

Ο τύπος της έκφρασης ${}^{cl_1}e_1 || {}^{cl2}e_2$ είναι ${}^{s\ \text{at}\ \bot}\langle \tau_1 * \tau_2 \rangle$, όπου τ_1 ο τύπος της e_1, τ_2 ο τύπος της e_2 και s το ελάχιστο άνω φράγμα των qualifier των τ_1 και τ_2, σύμφωνα με την σχέση 5.2. Κατά τον έλεγχο της έκφρασης αυτής πρέπει επιπλέον να ελεγχθεί ότι τα διανύσματα που επισημειώνουν τις δύο υποεκφράσεις είναι έγκυρα, όπως απαιτείται για την ορθή αποτίμηση των εκφράσεων της γλώσσας.

Ο τύπος της έκφρασης $n_1 : e_1 \# n_2 : e_2$ είναι ${}^{s\ \text{at}\ \bot}\langle \tau_1 * \tau_2 \rangle$, όπου τ_1 ο τύπος της e_1, τ_2 ο τύπος της e_2 και s το ελάχιστο άνω φράγμα των qualifier των τ_1 και τ_2. Σημειώνουμε ότι n_1 και n_2 είναι οι κωδικοί thread των δύο υποεκφράσεων και χρειάζονται στην λειτουργική σημασιολογία της γλώσσας.

Αφαίρεση

$$\frac{\begin{array}{c}q(\Gamma_1) \qquad Z \models q \\ \Gamma_1, x : \tau; \Delta; Z_e; M \vdash e : \tau_1\end{array}}{\Gamma_1 \odot \Gamma_2; \Delta; Z; M \vdash {}^{q}\lambda x : \tau.e : {}^{q}(\tau \xrightarrow{Z_e} \tau_1)}\ \text{abs}$$

Σχήμα 5.8: Κανόνες τύπων: Αφαίρεση

Ο κανόνας της αφαίρεσης διαφοροποιείται αφού ο έλεγχος του σώματος της αφαίρεσης δεν γίνεται σε όλο το Γ αλλά σε ένα υποσύνολό του. Αν δεν γινόταν αυτή η διαφοροποίηση τότε, αν στο αρχικό Γ υπήρχε κάποια μεταβλητή κατάστασης T ή R, τότε θα έπρεπε η κατάσταση της αφαίρεσης να είναι ή R αντίστοιχα ανεξάρτητα από το αν η μεταβλητή αυτή χρησιμοποιούταν ή όχι από το σώμα της.

Αλλαγή Κατάστασης Capability

Για να ελεγχθεί ο τύπος της έκφρασης at ρ wlock $(x = e_1)$ then $y = e_2$ unlock e_3 (αντ. για την rlock) απαιτείται η έκφραση e_1 να έχει τύπο ${}^{s\ \text{at}\ \pi}\text{Cap}\ r\ \tau_1$ όπου s μία κατάσταση που παρέχει λιγότερα δικαιώματα, αφού επιτρέπεται μόνο αναβάθμιση της κατάστασης. Έπειτα, στο περιβάλλον προστίθεται η μεταβλητή x με τύπο ${}^{s\ \text{at}\ \rho}\text{Cap}\ r\ \tau_1$, όπου $s = T$ (αντ. $s = R$), ενώ το scope ρ διασφαλίζει ότι το capability κατάστασης s δεν θα διαφύγει από την υποέκφραση e_2 και ελέγχεται η έκφραση e_2. Τέλος, στο περιβάλλον προστίθεται η μεταβλητή y με τύπο τον τύπο της e_2 και ελέγχεται η έκφραση e_3.

Η έκφραση $wlock\$$ (αντ. $rlock\$$) προκύπτει κατά την αποτίμηση της $wlock$ (αντ. $rlock$) και διαφοροποιείται από αυτήν στο ότι το $(x = e_1)$ έχει αντικατασταθεί από την auto-variable x στην οποία

$$\dfrac{\begin{array}{c}\Gamma_1; \Delta; Z_1; M \vdash e_1 : {}^{s \text{ at } \pi}\text{Cap } r\ \tau_1 \qquad \text{fresh } \rho' \qquad s = R \vee U \qquad Z_2' = Z_2 \vee Z_2, \rho' \\ \Gamma_2, x : {}^{T \text{ at } \rho'}\text{Cap } r\ \tau_1; \Delta; Z_2'; M \vdash e_2[\rho \mapsto \rho'] : \tau_2 \qquad \Gamma_3, y : \tau_2; \Delta; Z_3; M \vdash e : \tau\end{array}}{\Gamma_1 \oplus \Gamma_2 \oplus \Gamma_3; \Delta; Z_1 \cup Z_2 \cup Z_3; M \vdash \text{at } \rho \text{ wlock } (x = e_1) \text{ then } y = e_2 \text{ unlock } e : \tau} \ \text{wlock}$$

$$\dfrac{\begin{array}{c}\Gamma_1; \Delta; Z_1; M \vdash e_1 : {}^{U \text{ at } \pi}\text{Cap } r\ \tau_1 \qquad \text{fresh } \rho' \qquad Z_2' = Z_2 \vee Z_2, \rho' \\ \Gamma_2, x : {}^{R \text{ at } \rho'}\text{Cap } r\ \tau_1; \Delta; Z_2'; M \vdash e_2[\rho \mapsto \rho'] : \tau_2 \qquad \Gamma_3, y : \tau_2; \Delta; Z_3; M \vdash e : \tau\end{array}}{\Gamma_1 \oplus \Gamma_2 \oplus \Gamma_3; \Delta; Z_1 \cup Z_2 \cup Z_3; M \vdash \text{at } \rho \text{ rlock } (x = e_1) \text{ then } y = e_2 \text{ unlock } e : \tau} \ \text{rlock}$$

$$\dfrac{\begin{array}{c}\Gamma_1; \Delta; Z_1; M \vdash z : {}^{s \text{ at } \rho}\text{Cap } r\ \tau_1 \qquad s = R \vee U \qquad Z_2' = Z_2 \vee Z_2, \rho \\ \Gamma_2, x : {}^{T \text{ at } \rho}\text{Cap } r\ \tau_1; \Delta; Z_2'; M \vdash e_2 : \tau_2 \qquad \Gamma_3, y : \tau_2; \Delta; Z_3; M \vdash e : \tau\end{array}}{\Gamma_1 \oplus \Gamma_2 \oplus \Gamma_3; \Delta; Z_1 \cup Z_2 \cup Z_3; M \vdash \text{at } \rho \text{ wlock\$ } (x = z) \text{ then } y = e_2 \text{ unlock } e : \tau} \ \text{wunlock}$$

$$\dfrac{\begin{array}{c}\Gamma_1; \Delta; Z_1; M \vdash z : {}^{U \text{ at } \rho}\text{Cap } r\ \tau_1 \qquad Z_2' = Z_2 \vee Z_2, \rho \\ \Gamma_2, x : {}^{R \text{ at } \rho}\text{Cap } r\ \tau_1; \Delta; Z_2'; M \vdash e_2 : \tau_2 \qquad \Gamma_3, y : \tau_2; \Delta; Z_3; M \vdash e : \tau\end{array}}{\Gamma_1 \oplus \Gamma_2 \oplus \Gamma_3; \Delta; Z_1 \cup Z_2 \cup Z_3; M \vdash \text{at } \rho \text{ rlock\$ } (x = z) \text{ then } y = e_2 \text{ unlock } e : \tau} \ \text{runlock}$$

Σχήμα 5.9: Κανόνες Τύπων: Αλλαγή Κατάστασης Capability

έχει ανατεθεί η τιμή που προκύπτει από την αποτίμηση της έκφρασης e_1. Για τον λόγο αυτόν, οι κανόνες παραμένουν ίδιοι.

Let!

$$\dfrac{\begin{array}{c}\text{fresh } \rho' \qquad \Gamma; \Delta; Z; M \vdash e : {}^{L}\text{Cap } r\ \tau \qquad \Gamma_1, x : {}^{T \text{ at } \rho'}\text{Cap } r\ \tau; \Delta; Z_1'; M \vdash e_1[\rho \mapsto \rho'] : \tau_1 \\ Z_1' = Z_1 \vee Z_1, \rho' \qquad \Gamma_2, x : {}^{L}\text{Cap } r\ \tau, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2\end{array}}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_2 \cup Z_3; M \vdash \text{at } \rho\ {}^{w}\text{let! } (x = e) \text{ then } y = e_1 \text{ in } e_2 : \tau_2}$$

$$\dfrac{\begin{array}{c}\text{fresh } \rho' \qquad \Gamma; \Delta; Z; M \vdash e : {}^{L}\text{Cap } r\ \tau \qquad \Gamma_1, x : {}^{R \text{ at } \rho'}\text{Cap } r\ \tau; \Delta; Z_1'; M \vdash e_1[\rho \mapsto \rho'] : \tau_1 \\ Z_1' = Z_1 \vee Z_1, \rho' \qquad \Gamma_2, x : {}^{L}\text{Cap } r\ \tau, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2\end{array}}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_2 \cup Z_3; M \vdash \text{at } \rho\ {}^{r}\text{let! } (x = e) \text{ then } y = e_1 \text{ in } e_2 : \tau_2}$$

$$\dfrac{\begin{array}{c}\Gamma; \Delta; Z; M \vdash z : {}^{q}\text{Cap } r\ \tau \qquad \Gamma_1; \Delta; Z_1'; M \vdash e_1 : \tau_1 \\ Z_1' = Z_1 \vee Z_1, \rho \qquad \Gamma_2, x : {}^{L}\text{Cap } r\ \tau, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2\end{array}}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_2 \cup Z_3; M \vdash \text{at } \rho\ {}^{w}\text{let\$ } (x = z) \text{ then } y = e_1 \text{ in } e_2 : \tau_2}$$

$$\dfrac{\begin{array}{c}\Gamma; \Delta; Z; M \vdash z : {}^{q}\text{Cap } r\ \tau \qquad \Gamma_1; \Delta; Z_1'; M \vdash e_1 : \tau_1 \\ Z_1' = Z_1 \vee Z_1, \rho \qquad \Gamma_2, x : {}^{L}\text{Cap } r\ \tau, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2\end{array}}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_2 \cup Z_3; M \vdash \text{at } \rho\ {}^{r}\text{let\$ } (x = z) \text{ then } y = e_1 \text{ in } e_2 : \tau_2}$$

Σχήμα 5.10: Κανόνες τύπων: let!

Οι κανόνες αυτοί είναι ανάλογοι των κανόνων $let!$ και $let\$$ της ακολουθιακής γλώσσας. Η μόνη διαφορά τους είναι ότι κατά τον έλεγχο της έκφρασης e_1, η κατάσταση του capability γίνεται R ή T αντί για U.

Ανάγνωση Περιεχομένου Θέσης Μνήμης

Για να επιτραπεί η ανάγνωση του περιεχομένου μίας θέσης μνήμης απαιτείται πλέον να είναι διαθέσιμο το capability για αυτήν την θέση με κατάσταση που να παρέχει δικαίωμα ανάγνωσης. Άρα, στον κανόνα προστίθεται η υπόθεση ότι το ζεύγος (άρα και το capability) έχει κατάσταση $s = R \vee T$.

$$\frac{\Gamma; \Delta; Z; M \vdash e : {}^{s \text{ at } \pi}\text{Lref}\, r\ \tau \qquad s = R \vee T \qquad Z_\tau \models \text{qual}\ \tau}{\Gamma; \Delta; Z \cup Z_\tau; M \vdash \text{deref}\, e : \tau}\ \text{deref}$$

Σχήμα 5.11: Κανόνες Τύπων: Ανάγνωση Περιεχομένου Θέσης Μνήμης

Εγγραφή Περιεχομένου Θέσης Μνήμης

$$\frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : {}^{T \text{ at } \pi}\text{Lref}\, r\ \tau \qquad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau \qquad \text{qual}\ \tau \sqsupset L}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1 := e_2 : \text{Unit}}\ \text{weakAss}$$

$$\frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : {}^{T \text{ at } \pi}\text{Lref}\, r\ \tau \qquad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1 :=: e_2 : \tau}\ \text{weakSwap}$$

Σχήμα 5.12: Κανόνες Τύπων: Εγγραφή Περιεχομένου Θέσης Μνήμης

Για να επιτραπεί η εγγραφή σε μία θέσης μνήμης απαιτείται πλέον να είναι διαθέσιμο το capability για αυτήν τη θέση με κατάσταση που να παρέχει δικαίωμα εγγραφής. Άρα, στον κανόνα η κατάσταση του ζεύγους (άρα και του capability) γίνεται $s = T$.

5.4 Λειτουργική Σημασιολογία

Κατά τον υπολογισμό μιας έκφρασης είναι απαραίτητο κάθε έκφραση να είναι επισημειωμένη με έναν κωδικό που να χαρακτηρίζει το νήμα στο οποίο υπολογίζεται. Έτσι, το αρχικό πρόγραμμα υπολογίζεται από την σχέση $S; \mu; t; e \hookrightarrow S'; \mu'; t'; n : e'$ όπου n ο κωδικός του αρχικού νήματος ενώ για κάθε ενδιάμεση έκφραση η σχέση υπολογισμού είναι: $S; \mu; t; n : e \hookrightarrow S'; \mu'; t'; n : e'$.

Τα περιβάλλοντα S και μ είναι όμοια με τα περιβάλλοντα της αποτίμησης της σειριακής γλώσσας, ενώ εισάγεται το περιβάλλον t.

5.4.1 Το περιβάλλον t

Το περιβάλλον t αντιστοιχεί ζεύγη διευθύνσεων θέσης μνήμης και κωδικούς νημάτων σε καταστάσεις:

$t ::= \emptyset \,|\, t, (i \times n : s)$

Αν $(i \times n : s) \in t$, τότε το νήμα με κωδικό n έχει όλα τα δικαιώματα πρόσβασης της θέσης μνήμης i που παρέχει ένα capability με κατάσταση s, ουσιαστικά διατηρεί όλα τα read και write locks του thread. Σημειώνεται ότι το αντίστροφο δεν ισχύει, αφού κάθε linear capability πρέπει να χρησιμοποιηθεί ακριβώς μία φορά, άρα όταν ένα νήμα (με κωδικό n) διαθέτει ένα linear capability (για την θέση μνήμης i) δεν μπορεί να το διαθέτει κανένα άλλο νήμα , άρα το νήμα αυτό έχει όλα τα δικαιώματα πρόσβασης της αντίστοιχης θέσης μνήμης χωρίς να απαιτείται $(i \times n : L) \in t$. Για το λόγο αυτό και επειδή η κατάσταση U ενός capability δεν παρέχει δικαιώματα πρόσβασης της μνήμης, αρκεί στο περιβάλλον t να εισάγονται οι καταστάσεις R και T.

5.4.2 Οι κανόνες

Στα σχήματα 5.13 - 5.19 παρουσιάζονται οι κανόνες που διαφοροποιούνται από αυτούς της ενότητας 4.3.2.

Αρχικός Κανόνας

Όπως προαναφέρθηκε, κατά την αποτίμηση πρέπει κάθε έκφραση να επισημειώνεται με τον κωδικό του νήματος στο οποίο υπολογίζεται. Ο σκοπός του αρχικού κανόνα είναι να επισημειώσει το πρό-

$$\frac{\text{start } n}{S;\mu;t;e \hookrightarrow S;\mu;t;n:e}$$

Σχήμα 5.13: Κανόνες αποτίμησης: Αρχικός Κανόνας

γραμμα (την αρχική έκφραση) με κάποιον κωδικό ώστε να μπορεί μετά να εφαρμοστεί η σχέση υπολογισμού για την επισημειωμένη πλέον έκφραση. Αν θεωρήσουμε ότι οι κωδικοί είναι ακέραιοι αριθμοί ο αρχικός κωδικός ($start\ n$) θα είναι 0.

Παράλληλος Υπολογισμός

$$\frac{\begin{array}{c}\text{fresh } n_1 \qquad\qquad \text{fresh } n_2 \\ t' = t \setminus \{(i' \times n : s') | \exists i's'.(i' \times n : s') \in t\} \\ \cup\{(i \times n_1 : s)|(^{s \text{ at } \pi}\text{cap } i \in cl_1)\} \cup \{(i \times n_2 : s)|(^{s \text{ at } \pi}\text{cap } i \in cl_2)\}\end{array}}{S;\mu;t;n: {}^{cl_1}e_1||^{cl_2}e_2 \hookrightarrow S;\mu;t';n:n_1:e_1\#n_2:e_2} \text{ par1}$$

$$\frac{S;\mu;t;n_1:e_1 \hookrightarrow S';\mu';t';n_1:e_1'}{S;\mu;t;n:n_1:e_1\#n_2:e_2 \hookrightarrow S';\mu';t';n:n_1:e_1'\#n_2:e_2} \text{ parPr1}$$

$$\frac{S;\mu;t;n_2:e_2 \hookrightarrow S';\mu';t';n_2:e_2'}{S;\mu;t;n:n_1:e_1\#n_2:e_2 \hookrightarrow S';\mu';t';n:n_1:e_1\#n_2:e_2'} \text{ parPr2}$$

$$\frac{\begin{array}{c} t' = t \setminus \{(i' \times n' : s') | (i' \times n' : s') \in t \wedge (n' = n_1 \vee n' = n_2)\} \\ \cup\{(i' \times n : s') | (n' = n_1 \vee n' = n_2) \wedge \exists i's'.(i' \times n' : s') \in t\} \\ S;z_1 \Downarrow S';^{q_1} u_1 \qquad S';z_2 \Downarrow S'';^{q_2} u_2 \qquad s = LUB(\text{qual } q_1, \text{qual } q_2)\end{array}}{S;\mu;t;n:n_1:z_1\#n_2:z_2 \hookrightarrow S'';\mu;t';n:^{s \text{ at } \bot}(z_1,z_2)} \text{ par2}$$

Σχήμα 5.14: Κανόνες αποτίμησης: Παράλληλος Υπολογισμός

Για να υπολογιστεί η έκφραση ${}^{cl_1}e_1||^{cl_2}e_2$ αρχικά πρέπει να υπολογιστεί το νέο t: από το t αφαιρούνται τα στοιχεία που αναφέρονται στο υπάρχον νήμα (με κωδικό n) και προστίθενται στοιχεία όπως προκύπτουν από τα επισημειωμένα διανύσματα $cl_{1,2}$, έπειτα επιστρέφεται η έκφραση $n_1 : e_1\#n_2 : e_2$, όπου n_1 και n_2 είναι νέοι κωδικοί νημάτων θα επισημειώσουν τις εκφράσεις που πρόκειται να υπολογιστούν παράλληλα.

Για τον υπολογισμό της νέας αυτής έκφρασης ορίζονται οι κανόνες $parPr_{1,2}$. Η σειρά με την οποία θα κληθούν οι δύο αυτοί κανόνες είναι μη καθορισμένη όπως επιβάλλεται για τον παράλληλο υπολογισμό των δύο εκφράσεων. Όταν οι δύο υποεκφράσεις έχουν υπολογιστεί, ο έλεγχος θα επιστρέψει στο αρχικό νήμα (με κωδικό n) και θα επιστραφεί το ζεύγος που αποτελείται από τις τιμές των δύο υποεκφράσεων.

Αφού ο έλεγχος επιστρέφει στο νήμα με κωδικό n, θα πρέπει το περιβάλλον t της σχέσης par1 να ταυτίζεται με το περιβάλλον t' της σχέσης par2 . Από την σχέση 5.5 προκύπτει ότι για κάθε στοιχείο $(i \times n : s)$ που αφαιρείται από το t της σχέσης par1 θα προστεθεί το $(i \times n_i : s), i = 1, 2$. Από το σύστημα τύπων και συγκεκριμένα την σχέση par2 προκύπτει ότι για κάθε στοιχείο $(i \times n_i : s)$ προστίθεται στο t' το στοιχείο $(i \times n : s)$, άρα όλα τα στοιχεία που αφαιρούνται από το t της par1 προστίθενται στο t' της par2. Επιπλέον, κάθε στοιχείο που προστέθηκε στο t' της σχέσης par1 έχει κωδικό νήμα n_i και θα αφαιρεθεί από το t' της par2, άρα τα δύο περιβάλλοντα ταυτίζονται.

Αλλαγή Κατάστασης

Στο σχήμα 5.15 φαίνονται οι κανόνες που αφορούν την αλλαγή της κατάστασης ενός capability σε T. Συγκεκριμένα, ο κανόνας wlockPr εκτελείται μέχρι η έκφραση e_1 να αποτιμηθεί σε τιμή (autovariable z). Έπειτα εκτελείται ένας από τους κανόνες wlock ή wlock2: Αν υπάρχουν $n' \neq n$ και s' τέτοια ώστε $(i \times n', s') \in t$, δηλαδή κάποιο νήμα με κωδικό n' έχει δικαιώματα πρόσβασης στην θέση

$$\frac{S; \mu; t; n : e_1 \hookrightarrow S'; \mu'; t'; n : e_1'}{\begin{array}{c} S; \mu; t; n : \text{at } \rho \text{ wlock } (x = e_1) \text{ then } y = e_2 \text{ unlock } e \hookrightarrow \\ S'; \mu'; t'; n : \text{at } \rho \text{ wlock } (x = e_1') \text{ then } y = e_2 \text{ unlock } e \end{array}} \text{ wlockPr}$$

$$\frac{S; z \Downarrow S'; {}^{s \text{ at } \rho}\text{cap } i \qquad \nexists n's'.(n \neq n' \land (i \times n', s') \in t)}{\begin{array}{c} S; \mu; t; n : \text{at } \rho \text{ wlock } (x = z) \text{ then } y = e_2 \text{ unlock } e \hookrightarrow \\ S', x : {}^{T \text{ at } \rho}\text{cap } i; \mu; t, (i \times n, T); n : \text{at } \rho \text{ wlock\$ } (x = z) \text{ then } y = e_2 \text{ unlock } e \end{array}} \text{ wlock}$$

$$\frac{S; z \Downarrow S'; {}^{q}\text{cap } i \qquad \exists n's'.(n \neq n' \land (i \times n', s') \in t)}{\begin{array}{c} S; \mu; t; n : \text{at } \rho \text{ wlock } (x = z) \text{ then } y = e_2 \text{ unlock } e \hookrightarrow \\ S'; \mu; t; n : \text{at } \rho \text{ wlock } (x = z) \text{ then } y = e_2 \text{ unlock } e \end{array}} \text{ wlock2}$$

$$\frac{S; \mu; t; n : e_2 \hookrightarrow S'; \mu'; t'; n : e_2'}{\begin{array}{c} S; \mu; t; n : \text{at } \rho \text{ wlock\$ } (x = z) \text{ then } y = e_2 \text{ unlock } e \hookrightarrow \\ S'; \mu'; t'; n : \text{at } \rho \text{ wlock\$ } (x = z) \text{ then } y = e_2' \text{ unlock } e \end{array}} \text{ wunlockPr}$$

$$\frac{S; z \Downarrow S'; {}^{q} \text{ cap } i}{\begin{array}{c} S, x : v; \mu; t, (i \times n, T); n : \text{at } \rho \text{ wlock\$ } (x = z) \text{ then } y = w \text{ unlock } e \hookrightarrow \\ S'; \mu; t; n : e[y \mapsto w] \end{array}} \text{ wunlock}$$

Σχήμα 5.15: Κανόνες αποτίμησης: Αλλαγή Κατάστασης σε T

μνήμης i, τότε εκτελείται ο κανόνας wlock2 που επιστρέφει την ίδια έκφραση για να ξαναελεγχθεί η συνθήκη (spinlock). Αν δεν υπάρχουν $n' \neq n$ και s' τέτοια ώστε $(i \times n', s') \in t$, δηλαδή κανένα άλλο νήμα δεν έχει δικαιώματα πρόσβασης στην θέση μνήμης i, τότε εισάγεται στο t το $(i \times n, T)$ για να δηλωθεί ότι το νήμα με κωδικό n απέκτησε δικαιώματα γραφής και ανάγνωσης στην θέση μνήμης i και επιστρέφεται η ενδιάμεση έκφραση $wkock\$$ όπου στην έκφραση e_2 έχει αντικατασταθεί η μεταβλητή x με την auto-variable z. Στην συνέχεια εκτελείται ο κανόνας wunlockPr μέχρι η έκφραση e_2 να αποτιμηθεί σε τιμή (auto-variable w). Τέλος, στον κανόνα wunlock, το στοιχείο $(i \times n, T)$ αφαιρείται από το t, αφαιρώντας από το νήμα με κωδικό n τα δικαιώματα γραφής και ανάγνωσης στην θέση μνήμης i και επιστρέφεται η έκφραση e, όπου η μεταβλητή y έχει αντικατασταθεί με την auto-variable w

Οι κανόνες για την αλλαγή κατάστασης ενός capability σε R είναι αντίστοιχοι και φαίνονται στο σχήμα 5.16. Οι κανόνες αυτοί είναι αντίστοιχοι των κανόνων του σχήματος 5.15 με την διαφορά ότι στους κανόνες rlock και rlock2 ελέγχεται πλέον αν υπάρχει νήμα που να κατέχει δικαίωμα γραφής στην θέση μνήμης i (ελέγχεται η συνθήκη: $\exists n'.((i \times n', T) \in t)$) αφού το να υπάρχει άλλο νήμα με δικαίωμα ανάγνωσης στην θέση μνήμης i δεν απαγορεύει στο τρέχον νήμα να αποκτήσει δικαίωμα ανάγνωσης για την ίδια θέση μνήμης. Σημειώνεται ότι το τρέχον νήμα δεν μπορεί να κατέχει δικαίωμα γραφής στην θέση μνήμης i αφού από τον ελεγκτή τύπων επιβάλλεται η αλλαγή κατάστασης να γίνεται μόνο για αναβάθμιση.

Let!

Οι κανόνες αυτοί είναι ανάλογοι του κανόνα $let!$ της ακολουθιακής γλώσσας. Η μόνη διαφορά τους είναι ότι κατά την αποτίμηση της έκφρασης e_1, η κατάσταση του capability γίνεται R ή T αντί για U. Αυτό προφανώς συνεπάγεται ότι κατά τον υπολογισμό της e_2 προστίθενται τα κατάλληλα locks και ότι αφαιρούνται μετά από αυτόν.

Ανάγνωση Περιεχομένου Θέσης Μνήμης

Στον κανόνα deref προστίθεται πλέον η υπόθεση ότι τρέχον νήμα έχει δικαίωμα ανάγνωσης της συγκεκριμένης θέσης μνήμης, ότι δηλαδή ισχύει $(i \times n : s) \in t$, όπου $s = T \lor R$.

$$\frac{S;\mu;t;n:e_1 \hookrightarrow S';\mu';t';n:e_1'}{\begin{array}{c}S;\mu;t;n:\text{at }\rho\text{ rlock }(x=e_1)\text{ then }y=e_2\text{ unlock }e \hookrightarrow\\ S';\mu';t';n:\text{at }\rho\text{ rlock }(x=e_1')\text{ then }y=e_2\text{ unlock }e\end{array}}\ \text{rlockPr}$$

$$\frac{S;z \Downarrow S';{}^{s\text{ at }\rho}\text{cap } i \qquad \nexists n'.((i\times n',T)\in t)}{\begin{array}{c}S;\mu;t;n:\text{at }\rho\text{ rlock }(x=z)\text{ then }y=e_2\text{ unlock }e \hookrightarrow\\ S',x:{}^{R\text{ at }\rho}\text{cap } i;\mu;t,(i\times n,R);n:\text{at }\rho\text{ rlock\$ }(x=z)\text{ then }y=e_2\text{ unlock }e\end{array}}\ \text{rlock}$$

$$\frac{S;z \Downarrow S';{}^{s\text{ at }\rho}\text{cap } i \qquad \exists n'.((i\times n',T)\in t)}{\begin{array}{c}S,z\mapsto^{q}\text{ cap } i;\mu;t;n:\text{at }\rho\text{ rlock }(x=z)\text{ then }y=e_2\text{ unlock }e \hookrightarrow\\ S';\mu;t;n:\text{at }\rho\text{ rlock }(x=z)\text{ then }y=e_2\text{ unlock }e\end{array}}\ \text{rlock2}$$

$$\frac{S;\mu;t;n:e_2 \hookrightarrow S';\mu';t';n:e_2'}{\begin{array}{c}S;\mu;t;n:\text{at }\rho\text{ rlock\$ }(x=z)\text{ then }y=e_2\text{ unlock }e \hookrightarrow\\ S';\mu';t';n:\text{at }\rho\text{ rlock\$ }(x=z)\text{ then }y=e_2'\text{ unlock }e\end{array}}\ \text{runlockPr}$$

$$\frac{S;z \Downarrow S';{}^{q}\text{ cap } i}{\begin{array}{c}S,x:v;\mu;t,(i\times n,R);n:\text{at }\rho\text{ rlock\$ }(x=z)\text{ then }y=w\text{ unlock }e \hookrightarrow\\ S';\mu;t;n:e[y\mapsto w]\end{array}}\ \text{runlock}$$

Σχήμα 5.16: Κανόνες αποτίμησης: Αλλαγή Κατάστασης σε R

Εγγραφή Περιεχομένου Θέσης Μνήμης

Οι κανόνες για την εγγραφή σε μία θέση μνήμης φαίνονται στο σχήμα 5.19. Στους κανόνες weakAss και weakSwap προστίθεται πλέον η υπόθεση ότι το τρέχον νήμα έχει δικαίωμα εγγραφής της συγκεκριμένης θέσης μνήμης, ότι δηλαδή ισχύει $(i \times n : T) \in t$.

5.5 Η Υλοποίηση

Ο ελεγκτής τύπων και ο διερμηνέας της σειριακής γλώσσας επεκτάθηκαν ώστε να χρησιμοποιηθούν για την παράλληλη γλώσσα. Στην ενότητα αυτή παρουσιάζονται οι βασικές επεκτάσεις που έγιναν.

5.5.1 Ελεγκτής Τύπων

Οι κυριότερες επεκτάσεις που έγιναν στον ελεγκτή τύπο αφορούν την σωστή ”διάσπαση” του περιβάλλοντος Γ, την επισημείωση των δύο υποεκφράσεων της έκφρασης για παράλληλη εκτέλεση με τα κατάλληλα διανύσματα και τον έλεγχο ύπαρξης κατάλληλων δικαιωμάτων κάθε φορά που προσπελάζεται η μνήμη.

”Διάσπαση” του περιβάλλοντος Γ

Το περιβάλλον Γ υλοποιείται με έναν global hashtable, άρα δεν διασπάται στην πραγματικότητα, απλώς πρέπει να εξασφαλιστεί ότι οι περιορισμοί που επιβάλλει η διάσπαση ισχύουν. Ο περιορισμός που εισάγει η σειριακή διάσπαση $\oplus$, δηλαδή ότι κάθε μεταβλητή με κατάσταση L θα χρησιμοποιηθεί ακριβώς μία φορά, εξασφαλίζεται όπως και στην σειριακή υλοποίηση. Ο περιορισμός που επιβάλλει η παράλληλη διάσπαση $\odot$, είναι ουσιαστικά ότι κάθε μεταβλητή με κατάσταση T είναι ορατή από το πολύ ένα νήμα . Για να εφαρμοστεί πρέπει η χρήση των μεταβλητών με κατάσταση T να παρακολουθείται. Αυτό μπορεί να επιτευχθεί εύκολα αν για κάθε έκφραση που ελέγχεται επιστρέφεται μια λίστα (έστω $tlist$) των μεταβλητών με κατάσταση T που ελέγχθηκαν. Συγκεκριμένα, όταν ελέγχεται:

- μια μεταβλητή x αν αυτή έχει κατάσταση T τότε $tlist = [x]$, διαφορετικά $tlist = []$

$$\frac{S;\mu;t;n:e_1 \hookrightarrow S';\mu';t';n:e_1'}{\begin{array}{c}S;\mu;t;n:\text{at }\rho\ {}^{w}\text{let! }(x=e_1)\text{ then }y=e_2\text{ in }e \hookrightarrow\\ S';\mu';t';n:\text{at }\rho\ {}^{w}\text{let! }(x=e_1')\text{ then }y=e_2\text{ in }e\end{array}}\ \text{wlet!Pr}$$

$$\frac{S;\mu;t;n:e_2 \hookrightarrow S';\mu';t';n:e_2'}{\begin{array}{c}S;\mu;t;n:\text{at }\rho\ {}^{w}\text{let\$ }(x=z)\text{ then }y=e_2\text{ in }e \hookrightarrow\\ S';\mu';t';n:\text{at }\rho\ {}^{w}\text{let\$ }(x=z)\text{ then }y=e_2'\text{ in }e\end{array}}\ \text{wlet\$Pr}$$

$$\frac{\text{fresh }\rho'}{\begin{array}{c}S,z\mapsto^{L}\text{cap }i;\mu;t;n:\text{at }\rho\ {}^{w}\text{let! }(x=z)\text{ then }y=e_1\text{ in }e_2 \hookrightarrow\\ S,z\mapsto^{T\text{ at }\rho'}\text{cap }i;\mu;t,(i\times n,T);n:\text{at }\rho'\ {}^{w}\text{let\$ }(x=z)\text{ then }y=e_1[\rho\mapsto\rho'][x\mapsto z]\text{ in }e_2\end{array}}\ \text{wlet!}$$

$$\frac{}{\begin{array}{c}S,z\mapsto^{q}\text{cap }i;\mu;t,(i\times n,T);n:\text{at }\rho\ {}^{w}\text{let\$ }(x=z)\text{ then }y=w\text{ in }e_2 \hookrightarrow\\ S,z\mapsto^{L}\text{cap }i;\mu;t;n:e_2[x\mapsto z][y\mapsto w]\end{array}}\ \text{wlet\$}$$

$$\frac{S;\mu;t;n:e_1 \hookrightarrow S';\mu';t';n:e_1'}{\begin{array}{c}S;\mu;t;n:\text{at }\rho\ {}^{r}\text{let! }(x=e_1)\text{ then }y=e_2\text{ in }e \hookrightarrow\\ S';\mu';t';n:\text{at }\rho\ {}^{r}\text{let! }(x=e_1')\text{ then }y=e_2\text{ in }e\end{array}}\ \text{rlet!Pr}$$

$$\frac{S;\mu;t;n:e_2 \hookrightarrow S';\mu';t';n:e_2'}{\begin{array}{c}S;\mu;t;n:\text{at }\rho\ {}^{r}\text{let\$ }(x=z)\text{ then }y=e_2\text{ in }e \hookrightarrow\\ S';\mu';t';n:\text{at }\rho\ {}^{r}\text{let\$ }(x=z)\text{ then }y=e_2'\text{ in }e\end{array}}\ \text{rlet\$Pr}$$

$$\frac{\text{fresh }\rho'}{\begin{array}{c}S,z\mapsto^{L}\text{cap }i;\mu;t;n:\text{at }\rho\ {}^{r}\text{let! }(x=z)\text{ then }y=e_1\text{ in }e_2 \hookrightarrow\\ S,z\mapsto^{R\text{ at }\rho'}u;\mu;t,(n\times i,R);n:\text{at }\rho'\text{ let\$ }(x=z)\text{ then }y=e_1[\rho\mapsto\rho'][x\mapsto z]\text{ in }e_2\end{array}}\ \text{rlet!}$$

$$\frac{}{\begin{array}{c}S,z\mapsto^{q}\text{cap }i;\mu;t,(i\times n,R);n:\text{at }\rho\ {}^{r}\text{let\$ }(x=z)\text{ then }y=w\text{ in }e_2 \hookrightarrow\\ S,z\mapsto^{L}\text{cap }i;\mu;t;n:e_2[x\mapsto z][y\mapsto w]\end{array}}\ \text{rlet\$}$$

Σχήμα 5.17: Κανόνες αποτίμησης: Let!

$$\frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\text{deref }e \hookrightarrow S';\mu';\text{deref }e'}\ \text{derefPr}$$

$$\frac{\begin{array}{ccc}(i\times n,s)\in t & & T\sqsubseteq s\sqsubseteq R\\ S;w\Downarrow S_1;{}^{q}(w_1,w_2) & S_1;w_1\Downarrow S_2;{}^{q}\text{cap }i & S_2;w_2\Downarrow S_3;\text{loc }i\end{array}}{S;\mu,i\mapsto z;t;n:\text{deref }w \hookrightarrow S_3;\mu,i\mapsto z;t;n:z}\ \text{deref}$$

Σχήμα 5.18: Κανόνες αποτίμησης: Ανάγνωση Περιεχομένου Θέσης Μνήμης

- η έκφραση at ρ wlock $(x = e_1)$ then $y = e_2$ unlock e η $tlist$ είναι το αποτέλεσμα της συγχώνευσης των $tlist$ των υποεκφράσεων e_1, e_2, e, όμως από την $tlist$ της e_2 αφαιρείται η x αν υπάρχει.

- η συνάρτηση ${}^{q}\lambda x : \tau.e$ η $tlist$ είναι κενή, αφού οι μεταβλητές που εμφανίζονται στο σώμα μιας συνάρτησης δεν χρησιμοποιούνται κατά τον ορισμό αλλά κατά την εφαρμογή της. Θα πρέπει όμως οι μεταβλητές με κατάσταση T να επισημειωθούν στον τύπο της έτσι ώστε να είναι διαθέσιμες όταν η συνάρτηση εφαρμοστεί. Οι μεταβλητές αυτές αποτελούνται από την τομή της $tlist$ της έκφρασης e και του περιβάλλοντος Γ μετά τον έλεγχο της έκφρασης e (αφού από την $tlist$ της έκφρασης e πρέπει να αφαιρεθούν οι μεταβλητές που ορίστηκαν στην e). Σημειώνουμε επίσης ότι η κατάσταση της συνάρτησης μπορεί να είναι R (αντ. U) αν στο σώμα της δεν χρησιμοποιούνται μεταβλητές κατάστασης $s \sqsubseteq T$ (αντ. $s \sqsubseteq R$) δηλαδή αν η τομή

$$\frac{S;\mu;t;n:e_1 \hookrightarrow S';\mu';t;n:e_1'}{S;\mu;t;n:e_1 := e_2 \hookrightarrow S';\mu';t;n:e_1' := e_2}\ \text{weakAssPr1}$$

$$\frac{S;\mu;t;n:e_2 \hookrightarrow S';\mu';t;n:e_2'}{S;\mu;t;n:x := e_2 \hookrightarrow S';\mu';t;n:x := e_2'}\ \text{weakAssPr2}$$

$$\frac{S;\mu;t;n:e_1 \hookrightarrow S';\mu';t;n:e_1'}{S;\mu;t;n:e_1 :=: e_2 \hookrightarrow S';\mu';t;n:e_1' :=: e_2}\ \text{weakSwapPr1}$$

$$\frac{S;\mu;t;n:e_2 \hookrightarrow S';\mu';t;n:e_2'}{S;\mu;t;n:x :=: e_2 \hookrightarrow S';\mu';t;n:x :=: e_2'}\ \text{weakSwapPr2}$$

$$\frac{\begin{array}{c}(i \times n, T) \in t \\ S;x \Downarrow S_1;{}^{q}(x_1,x_2) \quad S_1;x_1 \Downarrow S_2;{}^{q'}\text{cap } i \quad S_2;x_2 \Downarrow S_3;\text{loc } i\end{array}}{S;\mu,i \mapsto z;t;n:x := y \hookrightarrow S_3;\mu,i \mapsto y;t;n:\text{unit}}\ \text{weakAss}$$

$$\frac{\begin{array}{c}(i \times n, T) \in t \\ S;x \Downarrow S_1;{}^{q}(x_1,x_2) \quad S_1;x_1 \Downarrow S_2;{}^{q'}\text{cap } i \quad S_2;x_2 \Downarrow S_3;\text{loc } i\end{array}}{S;\mu,i \mapsto z_1;t;n:x :=: y \hookrightarrow S_4;\mu,i \mapsto y;t;n:z_1}\ \text{weakSwap}$$

Σχήμα 5.19: Κανόνες αποτίμησης: Εγγραφή Περιεχομένου Θέσης Μνήμης

της $tlist$ της έκφρασης e και του περιβάλλοντος Γ είναι κενή (ο αντίστοιχος έλεγχος δεν έχει σημασιολογικό ενδιαφέρον και δεν υλοποιείται).

- η εφαρμογή $e_1\ e_2$ η $tlist$ είναι το αποτέλεσμα της συγχώνευσης της $tlist$ της e_2 και της λίστας που είναι επισημειωμένη στον τύπο της e_1.

- η έκφραση $e_1||e_2$ η $tlist$ είναι το αποτέλεσμα της συγχώνευσης των $tlist$ των υποεκφράσεων e_1, e_2. Όμως, κατά την συγχώνευση των $tlist$ των δύο υποεκφράσεων όταν βρεθεί κοινή μεταβλητή ο ελεγκτής τερματίζει, αφού μία μεταβλητή με κατάσταση T χρησιμοποιείται από δύο νήματα που τρέχουν παράλληλα.

- κάποια άλλη έκφραση η $tlist$ είναι το αποτέλεσμα της συγχώνευσης των $tlist$ όλων των υποεκφράσεών της. Στην υλοποίηση η συγχώνευση γίνεται με την $mymerge$ που διατηρεί τις λίστες ταξινομημένες και διαγράφει τα διπλότυπα.

Εύρεση κατάλληλων διανυσμάτων cl

Όπως αναφέρθηκε, για κάθε έκφραση διατηρείται μία λίστα, η $tlist$, των μεταβλητών με κατάσταση T που η έκφραση χρησιμοποιεί τουλάχιστον μία φορά. Με αντίστοιχο τρόπο για κάθε έκφραση διατηρείται μία λίστα $rlist$ για τις μεταβλητές τύπου ${}^{R\text{ at }\pi}\text{Cap } r\ \tau$ που χρησιμοποιεί. Από τις δύο αυτές λίστες μπορούν άμεσα να βρεθούν για κάθε έκφραση όλα τα capabilities με κατάσταση T ή R που χρησιμοποιεί, έστω cl' τα capabilities αυτά.

Όπως προαναφέραμε, οι δύο υποεκφράσεις της έκφρασης παράλληλης εκτέλεσης πρέπει να επισημειωθούν με διανύσματα που να περιέχουν όλα τα capabilities με κατάσταση T ή R που χρησιμοποιεί κάθε έκφραση και που η ένωσή τους να περιέχει όλα τα capabilities με κατάσταση T ή R που υπάρχουν στο περιβάλλον Γ έτσι ώστε να καθίσταται δυνατή η ανάκτηση των locks του parent thread. Στην υλοποίηση το περιβάλλον t που διατηρεί τα locks του parent thread μπορεί να αποθηκευτεί προσωρινά σε κάποια μεταβλητή και η ανάκτησή του να γίνει από αυτήν. Άρα, αρκεί οι δύο υποεκφράσεις να επισημειωθούν με διανύσματα που να δείχνουν πώς θα μοιραστούν τα locks, δηλαδή με τα διανύσματα cl'.

Έλεγχος ύπαρξης κατάλληλων δικαιωμάτων

Η υλοποίηση του ελέγχου ύπαρξης κατάλληλων δικαιωμάτων κατά την προσπέλαση της μνήμης είναι απλός. Σε έκφραση που κάνει ανάγνωση της μνήμης ελέγχεται ότι το capability που παρέχεται έχει κατάσταση R ή T ενώ σε εκφράσεις που κάνουν ανάθεση στη μνήμης ελέγχεται ότι το capability που παρέχεται έχει κατάσταση T.

5.5.2 Διερμηνέας

Οι κυριότερες επεκτάσεις που έγιναν στον διερμηνέα αφορούν την υλοποίηση του περιβάλλοντος t, και τον υπολογισμό των εκφράσεων ${}^{cll_1}e_1||^{cl_2}e_2$, at ρ wlock $(x = e_1)$ then $y = e_2$ unlock e και at ρ rlock $(x = e_1)$ then $y = e_2$ unlock e.

Η υλοποίηση του περιβάλλοντος t

Το περιβάλλον t αντιστοιχίζει ζεύγη διευθύνσεων θέσης μνήμης και κωδικούς νημάτων σε καταστάσεις. Για τον λόγο αυτόν, υλοποιήθηκε σαν ένας global hashtable κάθε στοιχείο του οποίου έχει κλειδί την διευθύνση θέσης μνήμης και τιμή ένα hashtable κάθε στοιχείο του οποίου έχει κλειδί ένα κωδικό νήματος και τιμή μία κατάσταση. Με αυτήν την υλοποίηση μπορεί εύκολα να ελεγχθεί αν υπάρχει κάποιο νήμα που να έχει δικαίωμα εγγραφής σε μία θέση μνήμης.

Ο υπολογισμός της παράλληλης έκφρασης

Για την υλοποίηση του παραλληλισμού χρησιμοποιήθηκε το module Thread της ocaml. Η συνάρτηση υπολογισμού δέχεται ένα επιπλέον όρισμα, τον κωδικό νήματος στο οποίο ανήκει η έκφραση που υπολογίζεται και χρησιμοποιεί μια νέα βοηθητική συνάρτηση, την $interpThread(tid, e)$ που υπολογίζει την έκφραση e στην τιμή v και εισάγει το ζεύγος (tid, v) σε έναν hashtable $threads$.

Όταν καλείται να υπολογιστεί η έκφραση ${}^{cll_1}e_1||^{cl_2}e_2$ δημιουργούνται δύο νέοι κωδικοί νημάτων $n1$ και $n2$, αποθηκεύεται σε μία τοπική μεταβλητή ο πίνακας t ώστε να τροποποιηθεί κατάλληλα. Έπειτα, δημιουργούνται δύο νήματα της ocaml που καλούν την $interpThread(n_i, e_i)$, $i = 1, 2$ και γίνεται παύση μέχρι να τελειώσουν. Τέλος, επαναφέρεται ο πίνακας t και επιστρέφεται το ζευγάρι $Epair(v_1, v_2)$, όπου v_i οι τιμές στις οποίες υπολογίστηκαν οι εκφράσεις e_i και ανακτήθηκαν από τον πίνακα $threads$. Η τροποποίηση του t γίνεται ως εξής: αν n ο κωδικός νήματος της αρχικής έκφρασης και ^{s} cap $r \in cl_i$, τότε αφαιρείται από τον t το στοιχείο που αντιστοιχεί στο $(n \times r, s)$ και εισάγεται το $(n_i \times r, s)$.

Ο υπολογισμός των εκφράσεων αλλαγής κατάστασης

Για την υλοποίηση της αλλαγής κατάστασης χρησιμοποιήθηκε το module Mutex της ocaml. Για να αλλάξει η κατάσταση ενός capability για την θέση μνήμης i σε T (αντ. R) πρέπει να εξασφαλιστεί ότι κανένα άλλο thread δεν έχει δικαίωμα πρόσβασης (αντ, εγγραφής) της θέσης μνήμης i. Για να εξασφαλιστεί αυτό πρέπει να ελεγχθεί ο πίνακας t. Για να αποφευχθεί η περίπτωση κατά την οποία ο πίνακας t τροποποιείται κατά τον ελεγχό του τα νήματα της ocaml συγχρονίστηκαν ως προς την πρόσβαση του πίνακα t με ένα mutex.

5.6 Παραδείγματα

Για την καλύτερη κατανόηση της γλώσσας παρουσιάζονται ορισμένα παραδείγματα. Αρχικά θα παρουσιαστούν κάποια παραδείγματα που αποτυγχάνουν στον ελεγκτή, έπειτα κάποια ορθά προγράμματα και τέλος ένα πιο σύνθετο παράδειγμα.

5.6.1 Συνθήκη Ανταγωνισμού: Εγγραφή με Εγγραφή

Στο ακόλουθο παράδειγμα εμφανίζεται μία συνθήκη ανταγωνισμού εγγραφή - εγγραφή: Δημιουργείται μία νέα θέση μνήμης (n), το capability γίνεται αρχικά unrestricted και έπειτα κατάστασης T και δύο παράλληλα νήματα γράφουν σε αυτήν την θέση μνήμης. Το πρόγραμμα αυτό δεν είναι σωστό, αφού μετά την εκτέλεση των παράλληλων νήματα το περιεχόμενο της θέσης μνήμης θα ήταν απροσδιόριστο. Ο ελεγκτής τύπων τερματίζει με σφάλμα, αφού δύο νήματα ζητούν μία μεταβλητή κατάστασης T. Δύο παράλληλα νήματα μπορούν να γράψουν σε μία θέση μνήμης μόνο αν η αλλαγή της κατάστασης του αντίστοιχου capability γίνει τοπικά, όπως φαίνεται στο παράδειγμα 27

Παράδειγμα 22
let $n =$ new 5 in
let $\ulcorner r, p \urcorner = n$ in
let $(cp, re) = p$ in $\quad (*cp :^{L}$ Cap l Int$*)$
at h let! $(x = cp)$ $\quad (*x :^{U \text{ at } h}$ Cap l Int$*)$
  then $y =$
    at ρ wlock $(s = x)$ $\quad (*s :^{T \text{ at } h}$ Cap l Int$*)$
      then $q = (s, re) := 10 || (s, re) := 10$
     unlock $unit$
in
free $\ulcorner r, (x, re) \urcorner; y$

Ο ελεγκτής τερματίζει με μήνυμα: *both threads ask the thread exclusive var s*

5.6.2 Συνθήκη Ανταγωνισμού: Εγγραφή με Ανάγνωση

Στο ακόλουθο παράδειγμα εμφανίζεται μία συνθήκη ανταγωνισμού εγγραφή - ανάγνωση: Δημιουργείται μία νέα θέση μνήμης (n), το capability γίνεται αρχικά unrestricted και έπειτα κατάστασης T και τρέχουν δύο παράλληλα νήματα που το ένα γράφει και το άλλο διαβάζει αυτήν την θέση μνήμης. Το πρόγραμμα αυτό δεν είναι σωστό, αφού το αποτέλεσμα της ανάγνωσης της θέσης μνήμης θα ήταν απροσδιόριστο. Ο ελεγκτής τύπων τερματίζει με σφάλμα, αφού δύο νήματα ζητούν μία μεταβλητή κατάστασης T. Δύο παράλληλα νήματα μπορούν να γράψουν και να διαβάσουν μία θέση μνήμης μόνο αν η αλλαγή της κατάστασης του αντίστοιχου capability γίνει τοπικά.

Παράδειγμα 23
let $n =$ new 5 in
let $\ulcorner r, p \urcorner = n$ in
let $(cp, re) = p$ in $\quad (*cp :^{L}$ Cap l Int$*)$
at h let! $(x = cp)$ $\quad (*x :^{U \text{ at } h}$ Cap l Int$*)$
  then $y =$
    at ρ wlock $(s = x)$ $\quad (*s :^{T \text{ at } h}$ Cap l Int$*)$
      then $q = (s, re) := 10 || (s, re) := 10$
     unlock $unit$
in
free $\ulcorner r, (x, re) \urcorner; y$

Ο ελεγκτής τερματίζει με μήνυμα: *both threads ask the thread exclusive var s*

5.6.3 Συνθήκες Ανταγωνισμού και Συναρτήσεις

Στο ακόλουθο παράδειγμα η συνθήκη ανταγωνισμού εγγραφή - εγγραφή κρύβεται μέσα στην κλήση συναρτήσεων, αφού κάθε νήμα δεν γράφει άμεσα στην νέα θέση μνήμης, αλλά καλεί μία συνάρτηση που το κάνει. Το αποτέλεσμα όμως κατά τον έλεγχο τύπων είναι το ίδιο.

Παράδειγμα 24
let $n =$ new 5 in
let $\ulcorner r, p \urcorner = n$ in
let $(cp, re) = p$ in
at h let! $(x = cp)$
then $y =$
at ρ wlock $(s = x)$
then $q =$
let $writeS =^{T \text{ at } \perp} \lambda n$: Int.$(s, re) := n$
in
$writeS$ 10$||writeS$ 5
unlock $unit$
in
free $\ulcorner r, (x, re) \urcorner; y$

Ο ελεγκτής τερματίζει με μήνυμα: *both threads ask the thread exclusive var s*

Αντίστοιχο αποτέλεσμα θα είχε και η ανάγνωση θέσης μνήμης μέσω συνάρτησης. Επίσης πρέπει να σημειωθεί ότι αφού η συνάρτηση χρησιμοποιεί μία μεταβλητή κατάστασης T πρέπει να έχει κατάσταση τουλάχιστον T, διαφορετικά ο ελεγκτής θα έβγαζε σφάλμα: *function state should be T*, όπως στο παράδειγμα που ακολουθεί

Παράδειγμα 25
let $n =$ new 5 in
let $\ulcorner r, p \urcorner = n$ in
let $(cp, re) = p$ in
at h let! $(x = cp)$
then $y =$
at ρ wlock $(s = x)$
then $q =$
let $writeS =^{U \text{ at } \perp} \lambda n$: Int.$(s, re) := n$
in
$writeS$ 10$||writeS$ 5
unlock $unit$
in
free $\ulcorner r, (x, re) \urcorner; y$

5.6.4 Αποφυγή "ψεύτικου" αδιεξόδου

Στο ακόλουθο παράδειγμα το κύριο thread αποκτά ένα rlock και εκτελεί δύο παράλληλες εκφράσεις του. Η μία έκφραση δεν κάνει τίποτα (επιστρέφει $unit$), ενώ η άλλη αναβαθμίζει το lock σε wlock και γράφει στην θέση μνήμης. Στο παράδειγμα αυτό φαίνεται γιατί πρέπει να επιτρέπουμε κατά την παράλληλη διάσπαση του περιβάλλοντος Γ σε μεταβλητές με κατάσταση R να επιτρέπουμε να βρεθούν σε ακριβός ένα περιβάλλον, αφού αν δεν επιτρέπονταν, και τα δύο παράλληλα thread θα είχαν rlock και κανένα thread δεν θα μπορούσε να το αναβαθμίσει σε T.

Παράδειγμα 26
let $n =$ new 5 in
let $\ulcorner r, p \urcorner = n$ in
let $(cp, re) = p$ in
at h let! $(x = cp)$
then $y =$
at ρ rlock $(s = x)$
then $q =$

at ρ wlock $(s1 = s)$
then $q1 = (s1, re) := 10$
unlock $unit$
$||$ $unit$
unlock $unit$
in
free $\ulcorner r, (x, re)\urcorner$

$\triangleright$ 10 : Int

5.6.5 Συγχρονισμένη Εγγραφή στην Ίδια Θέση Μνήμης

Στο ακόλουθο παράδειγμα γίνεται συγχρονισμένη εγγραφή στην ίδια θέση μνήμης n από δύο παράλληλα νήματα . Για τον συντονισμό απαιτείται μία επιπλέον θέση μνήμης $flag$, η τιμή της οποίας θα καθορίζει ποιο νήμα έχει την δυνατότητα να γράψει στην θέση μνήμης και μία αναδρομική συνάρτηση $check$ που θα ελέγχει την βοηθητική θέση μνήμης μέχρι να πάρει την κατάλληλη τιμή.

Συγκεκριμένα στο παράδειγμα, ορίζονται οι δύο θέσεις μνήμης και γίνονται unrestricted ώστε να είναι δυνατή η προσπέλασή τους και από τα δύο νήματα . Έπειτα ορίζεται η αναδρομική συνάρτηση $check$ που δέχεται σαν όρισμα έναν ακέραιο και ελέγχει αν είναι ίσος με την τιμή που έχει η βοηθητική θέση μνήμης. Σημειώνουμε ότι για τον ορισμό αναδρομικής συνάρτησης απαιτείται η δέσμευση μιας επιπλέον θέσης μνήμης.Έπειτα τρέχουν παράλληλα τα δύο νήματα που γράφουν στην θέση μνήμης n: Αν η βοηθητική θέση μνήμης έχει τιμή 1 (αντ. 2) τότε το νήμα γράφει στην n την τιμή 20 (αντ. 30) και στην βοηθητική θέση μνήμης την τιμή 2 (αντ. 1). Και τελικά αποδεσμεύονται οι θέσεις μνήμης δίνοντας σαν τιμή του προγράμματος το περιεχόμενο της θέσης μνήμης n.

Αφού η βοηθητική θέση μνήμης αρχικοποιείται στην τιμή 1 το αποτέλεσμα του προγράμματος θα είναι 30, ενώ αν αρχικοποιούνταν σε 2 το αποτέλεσμα θα ήταν 20.

Παράδειγμα 27
let $n =$ new 5 in
let $\ulcorner r, p\urcorner = n$ in
let $(cp, n) = p$ in
let $flag =$ new 2 in
let $\ulcorner fr, fp\urcorner = flag$ in
let $(fcp, flag) = fp$ in
at h let! $(x = cp)$
then $y =$
at fh let! $(fx = fcp)$
then $fy =$
($*$ορισμός της αναδρομικής συνάρτησης check που δέχεται έναν ακέραιο και
επιστρέφει true όταν ο ακέραιος αυτός γίνει ίδιος με το περιεχόμενο της
θέσης μνήμης με αναφορά flag διαφορετικά επαναλαμβάνει την σύγκριση$*$)
let $check =^{U \text{ at } \perp} \lambda number$: Int.
let $foo1 =$ new $^{U \text{ at } \perp} \lambda x1$: Int.$false$ in
let $\ulcorner funr, funp\urcorner = foo1$ in
let $(func, funre) = funp$ in
at $hfun$ let! $(xfun = func)$
then $yfun =$
at ρ wlock $(x3 = xfun)$
then $qfun =$
let $check_flag =^{U \text{ at } \perp} \lambda x2$: Int.
if
at ρ rlock $(s1 = fx)$

then $q1 = \text{deref}\ (s1, flag) = x2$
unlock $q1$
then $true$
else deref $(x3, funre)\ x2$
in
$(x3, funre) := check_flag; check_flag\ number$
unlock $qfun$
in
free $\ulcorner funr, (xfun, funre) \urcorner; yfun$
$(*$τέλος check$*)$
in
$(*$κώδικας πρώτου νήματος$*)$
if $check\ 1$
then
at ρ wlock $(s11 = x)$
then $q1 = (s11, n) := 20;$
at ρ wlock $(s12 = fx)$
then $q12 = (s12, flag) := 2$
unlock $unit$
unlock $unit$
else $unit$
$||$
$(*$κώδικας δεύτερου νήματος$*)$
if $check\ 2$
then
at ρ wlock $(s11 = x)$
then $q1 = (s11, n) := 30;$
at ρ wlock $(s12 = fx)$
then $q12 = (s12, flag) := 1$
unlock $unit$
unlock $unit$
else $unit$
in
free $\ulcorner fr, (fx, flag) \urcorner$
in
free $\ulcorner r, (x, n) \urcorner$
$\triangleright$ 20 : Int

5.6.6 Fibonacci

Σαν τελευταίο παράδειγμα δίνεται ο παράλληλος υπολογισμός του $i-$ όρου της ακολουθίας fibonacci. Ο υπολογισμός είναι αντίστοιχος του αφελούς υπολογισμού προηγούμενου κεφαλαίου (παράδειγμα 20), όμως η συνάρτηση fibonacci υπολογίζει το ζεύγος (f_{i-1}, f_{i-2}) παράλληλα και επιστρέφει το άθροισμα των δύο όρων:

Παράδειγμα 28
let $fibonacci =^{U \text{ at } \bot} \lambda n$: Int.
let $f = \text{new}^{U \text{ at } \bot} \lambda x$: Int.x in
let $\ulcorner r, p \urcorner = f$ in
let $(c, re) = p$ in
at h let! $(x = c)$
then $y =$

```
            let fib =^{U at ⊥} λnum : Int.
              at ρ rlock (s = x)
                then q =
                  if num < 2
                    then
                      1
                    else
                      let (f1, f2) = deref (s, re) num − 1||deref (s, re) num − 2 in
                      f1 + f2
              unlock q
            in
            at ρ wlock (s1 = x)
              then q1 = (s1, re) := fib
            unlock fib n
        in
        free ⌜r, (x, re)⌝; y
      in
      fibonacci 7
      ▷ 21 : Int
```

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

Η συνεισφορά της παρούσας εργασίας συνοψίζεται στα ακόλουθα:

- Σχεδιάστηκε πλήρως μία συναρτησιακή γλώσσα προγραμματισμού με ισχυρό σύστημα τύπων που στηρίζει αναφορές, ισχυρές αναθέσεις και ρητή αποδέσμευση μνήμης. Η γλώσσα αυτή χρησιμοποιεί γραμμικά δικαιώματα πρόσβασης για την σωστή διαχείριση της μνήμης ενώ υποστηρίζει την ελεγχόμενη μετατροπή τους σε unrestricted για την διευκόλυνση του προγραμματιστή.
- Υλοποιήθηκαν ο εξαγωγέας τύπων και ο διερμηνέας της ακολουθιακής αυτής γλώσσας ώστε να μπορεί να ελεγχθεί η ασφάλειά της μέσω παραδειγμάτων.
- Σχεδιάστηκε πλήρως μια συναρτησιακή γλώσσα που υποστηρίζει ταυτόχρονο προγραμματισμό με κοινόχρηστη μνήμη και εντοπίζει τις συνθήκες ανταγωνισμού. Αυτό επιτεύχθηκε επεκτείνοντας τους qualifier των capabilities ώστε να υποστηρίζουν και τις καταστάσεις T (Thread Exclusive) και R (Read Only).
- Τέλος, υλοποιήθηκαν ο εξαγωγέας τύπων και ο διερμηνέας της ταυτόχρονης αυτής γλώσσας ώστε να μπορεί να ελεγχθεί η ασφάλειά της μέσω παραδειγμάτων.

6.2 Μελλοντική Έρευνα

Στο πλαίσιο μιας μελλοντικής έρευνας που καθιστούμε την γλώσσα λειτουργική είναι απαραίτητο:

- Να επεκταθεί η γλώσσα ώστε να υποστηρίζει πολυμορφισμό. Η επέκταση αυτή δεν έγινε για να διατηρηθεί η γλώσσα απλούστερη.
- Να αποδειχθεί η μεταθεωρία των δύο γλωσσών. Η απόδειξη της ακολουθιακής γλώσσας μοιάζει απλούστερη, δεδομένου ότι παρέχονται αποδείξεις και για τις δύο γλώσσες (let! και L^3) που αποτελούν θεμέλιο για την κατασκευή της.
- Να συσχετιστεί η γλώσσα με κάποια γλώσσα προγραμματισμού υψηλού επιπέδου. Αυτό μπορεί να γίνει είτε μεταφράζοντας κάποιο υποσύνολο μιας γλώσσας της ML-family στην γλώσσα αυτή είτε εισάγοντας το σύστημα τύπων της σε κάποια υπάρχουσα γλώσσα που να μπορεί να το υποστηρίξει, όπως η Cyclone.
- Τέλος, στα ταυτόχρονα προγράμματα θα μπορούσε μετά τον στατικό έλεγχο να εφαρμόζεται ένας αλγόριθμος εντοπισμού αδιεξόδων. Αν γινόταν αυτό, η γλώσσα θα παρείχε πλήρη ασφάλεια από τους κινδύνους του ταυτόχρονου προγραμματισμού.

Βιβλιογραφία

[Ahme05] Amal Ahmed, "A step-indexed model of substructural state", in *Proceedings of the International Conference on Functional Programming*, pp. 78–91, ACM Press, 2005.

[Aike03] Alex Aiken, John Kodumal, Jeffrey S. Foster and Tachio Terauchi, "Checking and Inferring Local Non-Aliasing", in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming language design and implementation*, pp. 129–140, 2003.

[Amal05] Greg Morrisett Amal, Greg Morrisett, Amal Ahmed and Matthew Fluet, "A Linear Language with Locations", in *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, pp. 293–307, 2005.

[Bake92] Henry G. Baker, "Lively Linear Lisp: Look Ma, No Garbage!", *ACM SIGPLAN Notices*, vol. 27, no. 8, pp. 89–98, August 1992.

[Chen06] Liang T. Chen, "The Challenge of Race Conditions in Parallel Programming", July 2006.

[Denn66] Jack B. Dennis and Earl C. Van Horn, "Programming Semantics For Multiprogrammed Computations", *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[Flue06] Matthew Fluet, Greg Morrisett and Amal Ahmed, "Linear Regions Are All You Need", in *Proceedings of the European Symposium on Programming*, pp. 7–21, 2006.

[Gira87] Jean-Yves Girard, "Linear Logic", *Theoretical Computer Science*, vol. 50, pp. 1–102, 1987.

[Grif90] Timothy G. Griffin, "A Formulae-as-Types Notion of Control", in *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 47–58, ACM Press, 1990.

[Haya91] Susumu Hayashi, "Singleton, Union and Intersection Types for Program Extraction", in *TACS*, pp. 701–730, 1991.

[Hick03] Michael Hicks, Greg Morrisett, Dan Grossman and Trevor Jim, "Safe and flexible memory management in Cyclone", Technical Report CS-TR-4514, University of Maryland, Department of Computer Science, July 2003.

[Lero10] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy and Jérôme Vouillon, *The Objective Caml system release 3.12, Documentation and user's manual*, chapter 12: Lexer and parser generators (ocamllex, ocamlyacc), Institut National de Recherche en Informatique et en Automatique, June 2010.

[Netz90] Robert H. B. Netzer and Barton P. Miller, "On the Complexity of Event Ordering for Shared-Memory Parallel Program Execution", in *International Conference on Parallel Processing*, pp. II93–II97, August 1990.

[Oder92] Martin Odersky, "Observers for Linear Types", in *Proceedings of the 4th European Symposium on Programming*, pp. 390–407, Springer-Verlag, 1992.

[Papa10] Michalis A. Papakyriakou and Nikolaos S. Papaspyrou, "From Linear to Unrestricted and Back: Type Safety and the Let-bang Construct". School of Electrical and Computer Engineering, National Technical University of Athens, Greece, 2010.

[Pier02] Benjamin C. Pierce, *Types and programming languages*, MIT Press, Cambridge, MA, USA, 2002.

[Smit99] Frederick Smith, David Walker and Greg Morrisett, "Alias Types", in *Proceedings of the European Symposium on Programming*, pp. 366–381, Springer-Verlag, 1999.

[Swam06] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman and Trevor Jim, "Safe manual memory management in Cyclone", *Science of Computer Programming*, vol. 62, no. 2, pp. 122–144, 2006.

[Toft94] Mads Tofte and Jean-Pierre Talpin, "Implementation of the typed call-by-value λ-calculus using a stack of regions", in *Conference Record of the 21st ACM Symposium on Principles of Programming Languages*, pp. 188–201, ACM Press, 1994.

[Toft97] Mads Tofte and Jean-Pierre Talpin, "Region-Based Memory Management", *Information and Computation*, vol. 132, no. 2, pp. 109–176, 1997.

[Toft98] Mads Tofte and Lars Birkedal, "A Region Inference Algorithm", *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 4, pp. 724–767, 1998.

[Wadl90] Philip Wadler, "Linear Types Can Change the World!", in M. Broy and C. Jones, editors, *Programming Concepts and Methods*, pp. 347–359, North Holland, Amsterdam, 1990.

[Walk00] David Walker, Karl Crary and Greg Morrisett, "Typed Memory Management in a Calculus of Capabilities", in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 262–275, ACM Press, 2000.

[Walk05] David Walker, "Substructural Type Systems", in Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, The MIT Press, 2005.

[Wulf81] W. A. Wulf, R. Levin and P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.

[Xi98] Hongwei Xi and Frank Pfenning, "Eliminating Array Bound Checking through Dependent Types", in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 249–257, Montreal, June 1998.