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

Γραμμικοί Τύποι και Μετασχηματισμός Περάσματος Συνεχειών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΘΕΟΦΙΛΟΠΟΥΛΟΣ

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

Αθήνα, Νοέμβριος 2011

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

Γραμμικοί Τύποι και Μετασχηματισμός Περάσματος Συνεχειών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΘΕΟΦΙΛΟΠΟΥΛΟΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Νοεμβρίου 2011.

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

Αθήνα, Νοέμβριος 2011

Παναγιώτης Θεοφιλόπουλος

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

Περίληψη

Σε αυτήν τη διπλωματική εργασία διερευνάται η αλληλεπίδραση ενός γραμμικού συστήματος τύπων που υποστηρίζει την προσωρινή μετατροπή γραμμικών σε μη-περιορισμένους τύπους και του στυλ περάσματος συνεχειών (continuation-passing style, CPS). Για το σκοπό αυτό ορίζεται η σύνταξη και η σημασιολογία της γλώσσας LetbangCPS, μίας γλώσσας προγραμματισμού χαμηλού επιπέδου η οποία χρησιμοποιεί ένα τέτοιο σύστημα τύπων, υποστηρίζει αναφορές στο στυλ της ML και έχει μορφή συμβατή με το στυλ περάσματος συνεχειών. Επιπλέον, ορίζεται ένας μετασχηματισμός περάσματος συνεχειών που μεταφράζει κάθε έγκυρο πρόγραμμα της συγγενικής γλώσσας Letbang σε ένα έγκυρο πρόγραμμα LetbangCPS.

Στο πλαίσιο της εργασίας αυτής υλοποιήθηκαν ένας πειραματικός ελεγκτής τύπων και ένας διερμηνέας για τη γλώσσα LetbangCPS, καθώς και ο μετασχηματισμός περάσματος συνεχειών από τη Letbang στη LetbangCPS.

Λέξεις κλειδιά

Γραμμικό σύστημα τύπων, μετασχηματισμός περάσματος συνεχειών, αναφορές, ML.

Abstract

This diploma dissertation investigates the interaction between a linear type system supporting the temporary conversion of linear to unrestricted types and the continuation-passing style (CPS). To this goal, we define the syntax and semantics of LetbangCPS, a low-level programming language which uses such a type system, supports ML-style references and its form is compatible with the continuation-passing style. Moreover, we define a continuation-passing style transformation, which transforms every valid program written in the related language Letbang to a valid LetbangCPS program.

Within the scope of this work, an experimental type checker and an interpreter for the language LetbangCPS have been implemented, alongside with the continuation-passing style transformation from Letbang to LetbangCPS.

Key words

Linear type system, continuation-passing style transformation, references, ML.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Νικόλαο Παπασπύρου για την καθοδήγηση που μου παρείχε για την πραγμάτωση αυτής της εργασίας, καθώς και για την υποστήριξη του σε όλες της φάσεις της υλοποίησής της. Θα ήθελα επίσης να ευχαριστήσω το διδακτορικό φοιτητή Μιχάλη Παπακυριάκου για τη συμμετοχή του και τις εύστοχες παρατηρήσεις του που συνέβαλαν θετικά σε διάφορα στάδια της εργασίας.

Τέλος, ευχαριστώ την οικογένειά μου για την ηθική και υλική υποστήριξή τους σε όλη τη διάρκεια των μαθητικών και φοιτητικών μου χρόνων.

Παναγιώτης Θεοφιλόπουλος,

Αθήνα, 8η Νοεμβρίου 2011

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-11, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Νοέμβριος 2011.

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

Περιεχόμενα

Σχήματα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Η διπλωματική αυτή εργασία αποσκοπεί στον ορισμό μιας γλώσσας προγραμματισμού, που χρησιμοποιεί γραμμικό σύστημα τύπων για αναφορές στο στυλ της ML και που ακολουθεί το στυλ του μετασχηματισμού περάσματος συνεχειών (continuation-passing style, CPS). Τη γλώσσα που προέκυψε (LetbangCPS) μπορεί κανείς να τη δει απλώς ως μία εκδοχή σε μορφή CPS της υπάρχουσας γλώσσας Letbang, που περιγράφεται στο Κεφάλαιο 3. Παρόλα αυτά η LetbangCPS μπορεί και να θεωρηθεί ως μία γλώσσα χαμηλού επιπέδου με γραμμικό σύστημα τύπων, τελείως ανεξάρτητη από τη Letbang.

Δεν υπάρχει ακόμα απόδειξη για τη συνέπεια της LetbangCPS. Το γεγονός όμως ότι αντλεί αρκετές ιδέες από το σχεδιασμό της Letbang (για την οποία υπάρχει απόδειξη συνέπειας) καθώς και τα πρακτικά αποτελέσματα από τον ελεγκτή τύπων που υλοποιήθηκε για την LetbangCPS ενισχύουν την εμπιστοσύνη στο σχεδιασμό μας και την υπόθεση ότι η απόδειξη της συνέπειάς της είναι εφικτή.

1.2 Κίνητρο

Τα γραμμικά συστήματα τύπων έχουν προ πολλού προταθεί για τη σωστή διαχείρηση των πόρων και πιο συγκεκριμένα για το συνδυασμό της ασφάλειας και της αποδοτικής διαχείρισης της μνήμης. Οι περιορισμοί που θέτουν τα γραμμικά συστήματα τύπων οδήγησαν στο σχεδιασμό μεικτών συστημάτων τύπων, συστημάτων δηλαδή που επιβάλλουν τη γραμμικότητα σε τιμές με συγκεκριμένους τύπους (όπως αναφορές στο στυλ της ML) ή/και επιτρέπουν την προσωρινή αλλαγή του χαρακτηρισμού ενός τύπου από γραμμικό σε μη-περιορισμένο και αντίστροφα.

Το κυρίως κίνητρο για τη χρήση του μετασχηματισμού περάσματος συνεχειών είναι ότι στη μορφή περάσματος συνεχειών μπορούν να αναπαρασταθούν πληρέστερα η ροή ελέγχου και η ροή δεδομένων. Η μορφή περάσματος συνεχειών ενός προγράμματος μπορεί να αποτελέσει μια ενδιάμεση γλώσσα που διευκολύνει την εξαγωγή συμπερασμάτων ενός μεταγλωττιστή ή άλλου εργαλείου σχετικά με τη ροή ελέγχου και τη ροή δεδομένων του αρχικού προγράμματος.

Ο σχεδιασμός μιας γλώσσας που να χρησιμοποιεί γραμμικό σύστημα τύπων για αναφορές στο στυλ της ML και που η σύνταξή της ακολουθεί μορφή περάσματος συνεχειών προέκυψε ως υποπρόβλημα ενός σχετικού αλλά διαφορετικού αρχικού προβλήματος. Σε αυτό το πρόβλημα, το επιθυμητό ήταν η προσθήκη του τελεστή `free` σε μία γλώσσα με αναφορές στο στυλ της ML με τέτοιο τρόπο ώστε να διατηρείται η ασφάλεια μνήμης. Η προτεινόμενη λύση περιελάμβανε τη χρήση ενός γραμμικού συστήματος τύπων και της γλώσσας Letbang για τη διαχείριση γραμμικών και μη-περιορισμένων αναφορών. Σε αυτή τη γλώσσα θα μεταφράζονταν αυτόματα τα προγράμματα της αρχικής γλώσσας, διατηρώντας τη σημασιολογία. Ο τελεστής `free` θα εφαρμοζόταν σε γραμμικές αναφορές ενώ οι υπόλοιποι τελεστές σε μη-περιορισμένες αναφορές. Αν ένα μεταφρασμένο πρόγραμμα περνούσε

επιτυχώς τον έλεγχο τύπων της Letbang, τότε αυτό θα σήμαινε ότι το ισοδύναμο αρχικό πρόγραμμα είναι ασφαλές ως προς τη χρήση της μνήμης.

Σκοπός ήταν η μετάφραση όσο το δυνατόν περισσότερων ασφαλών αρχικών προγραμμάτων σε προγράμματα Letbang που να περνούν τον έλεγχο τύπων και αυτό προϋπέθετε τη σωστή εναλλαγή μεταξύ γραμμικών και μη-περιορισμένων αναφορών. Σύντομα διαπιστώθηκε ότι η μετατροπή του αρχικού προγράμματος σε μορφή CPS διευκολύνει το έργο της μετάφρασης, καθώς κάνει ρητές τη ροή ελέγχου και τη ροή δεδομένων. Όμως, η ύπαρξη της δομής `let!` δε συμβαδίζει με το μετασχηματισμό CPS. Η μετάφραση από τη μορφή CPS του αρχικού προγράματος σε Letbang δεν ήταν εύκολο να γίνει, διατηρώντας το τελικό πρόγραμμα Letbang σε μορφή CPS.

Ας σημειωθεί ότι δεν θα ήταν αρκετό να επεκτείνουμε απλώς το μετασχηματισμό CPS ώστε να μεταφράζει και το στοιχείο `let!`: η ταυτόχρονη διατήρηση της ασφάλειας και της μορφής CPS του τελικού προγράμματος Letbang φάνηκε να είναι ένα δύσκολο πρόβλημα, ενώ ο μετασχηματισμός του αφηρημένου συντακτικού δέντρου (AST) του προγράμματος CPS στο AST του ισοδύναμου προγράμματος Letbang δεν μπορούσε να είναι αποτελεσματικός ελλείψει μιας μορφής CPS για τη δομή `let!`. Επιπλέον, η εφαρμογή του προϋπέθετε ότι το προς μετασχηματισμό AST είχε προκύψει από την εφαρμογή του μετασχηματισμού CPS σε ένα αρχικό πρόγραμμα ML. Μία ανεξάρτητη γλώσσα όμως, η οποία έχει υποχρεωτικά μορφή CPS και χρησιμοποιεί γραμμικό σύστημα τύπων για τις αναφορές είναι κατάλληλος στόχος για το μετασχηματισμό CPS της Letbang και επιπλέον μπορεί να αντικαταστήσει την ίδια τη Letbang ως το στόχο της μετάφρασης που προαναφέρθηκε.

Παρόλο που υπάρχουν ήδη τεχνικές για την ανάλυση προγραμμάτων ως προς την ασφάλεια μνήμης και έχουν υλοποιηθεί πολλά εργαλεία που τις χρησιμοποιούν, η χρήση ενός συστήματος τύπων για τέτοιους ελέγχους έχει το πλεονέκτημα ότι μπορεί να δώσει ένα πρώτο (πιθανόν λίγο συντηρητικό) συμπέρασμα σε σύντομο χρόνο ακόμα και στην περίπτωση μεγάλων προγραμμάτων. Επιπλέον, με την προσέγγιση που περιγράφηκε παραπάνω, δε θα υπήρχε ανάγκη για τη χρήση οποιασδήποτε μορφής επισημάνσεων (annotations) μέσα στον κώδικα του προγράμματος, γεγονός που αφενός μεν διευκολύνει και επιταχύνει την ανάπτυξη προγραμμάτων, αφετέρου αποκλείει την περίπτωση προγραμματιστικών σφαλμάτων συγκεκριμένα στις επισημάνσεις: όταν οι απαιτούμενες επισημάνσεις γίνονται αρκετά περίπλοκες τότε αρχίζουν να αποτελούν και το περισσότερο επιρρεπές σε προγραμματιστικά λάθη μέρος του κώδικα. Τέλος, μια τέτοια προσέγγιση πιθανότατα να είναι εύκολα επεκτάσιμη και σε άλλους τύπους δεδομένων πέρα των αναφορών στο στυλ της ML, ακόμα και σε τύπους δεδομένων ορισμένων από το χρήστη. Λόγω του τελευταίου η προσέγγισή μας θα μπορούσε να συμβάλει και στην απόδοση (κατά την εκτέλεση) των προγραμμάτων αφαιρώντας ένας μέρος του φόρτου του συλλέκτη σκουπιδιών.

Ανεξάρτητα από όσα αναφέραμε παραπάνω, μια μορφή CPS της γλώσσας Letbang θα μπορούσε να χρησιμεύσει ως μια ενδιάμεση γλώσσα στη διαδικασία μεταγλώττισης της Letbang που ευνοεί την εφαρμογή βελτιστοποιήσεων από το μεταγλωττιστή. Επιπλέον, θα μπορούσε πιθανόν να φανεί χρήσιμη και για την επέκταση της Letbang με νέα στοιχεία ελέγχου, όπως για παράδειγμα οι εξαιρέσεις (exceptions).

1.3 Σύνοψη

Η διπλωματική εργασία είναι οργανωμένη ως εξής:

– Το Κεφάλαιο 2 παρουσιάζει τις αναφορές στο στυλ της ML, περιγράφει το πρόβλημα της απελευθέρωσης μνήμης μέσω του τελεστή `free`, περιγράφει τον τρόπο με τον οποίο μπορεί να χρησιμοποιηθεί ένα γραμμικό σύστημα τύπων για την αντιμετώπιση του προβλήματος αυτού και δίνει μερικά παραδείγματα δουλειάς που έχει γίνει στην κατεύθυνση αυτή.

– Το Κεφάλαιο 3 παρουσιάζει τη γλώσσα Letbang, η οποία σχεδιάστηκε και υλοποιήθηκε από τους Νικόλαο Παπασπύρου και Μιχάλη Παπακυριάκου.

– Το Κεφάλαιο 4 παρουσιάζει την ιδέα του μετασχηματισμού CPS καθώς και τη μορφή που έδωσαν στο μετασχηματισμό CPS οι Olivier Danvy και Andrzej Filinski.

– Το Κεφάλαιο 5 παρουσιαζει τη γλώσσα LetbangCPS.

– Το Κεφάλαιο 6 παρουσιάζει το μετασχηματισμό CPS για τη γλώσσα Letbang στην ειδική μορφή που παράγει προγράμματα της γλώσσας LetbangCPS.

– Το Κεφάλαιο 7 παρουσιάζει μερικά στοιχεία της υλοποίησης του ελέγχου τύπων για την LetbangCPS.

– Τέλος, το Κεφάλαιο 8 συνοψίζει τη συνεισφορά και τα συμπεράσματα της εργασίας αυτής και παραθέτει μερικές πιθανές χρήσεις της γλώσσας LetbangCPS καθώς και μερικές προτάσεις για μελλοντική έρευνα.

Κεφάλαιο 2

Αναφορές στο στυλ της ML

2.1 Επισκόπηση

2.1.1 Ορισμός

Μία αναφορά στο στυλ της ML (ML-style reference) είναι ένα είδος αναφοράς που στις διάφορες διαλέκτους της γλώσσας ML υλοποιεί την ιδέα του αποθηκευτικού χώρου στον οποίο μπορεί να ανατεθεί περιεχόμενο. Ας σημειωθεί ότι στην ML και στις περισσότερες συναρτησιακές γλώσσες τα περισσότερα αντικείμενα είναι μη τροποποιήσιμα (immutable). Ο τροποποιήσιμος αποθηκευτικός χώρος όμως αυξάνει σημαντικά την εκφραστικότητα μιας γλώσσας και διευκολύνει την ανάπτυξη προγραμμάτων, συνεπώς είναι μια χρήσιμη ιδέα την οποία οι περισσότερες γλώσσες επιδιώκουν να υλοποιούν με κάποιον τρόπο. Μια ML-style αναφορά είναι λειτουργικά ισοδύναμη με έναν πίνακα (array) μήκους 1. Το περιεχόμενο μιας ML-style αναφοράς μπορεί να είναι οποιουδήποτε τύπου. Μπορούμε να αναθέσουμε πολλές φορές περιεχόμενο στην αναφορά αλλά στις διαλέκτους της ML το περιεχόμενο που αναθέτουμε κάθε φορά πρέπει να είναι του ίδιου τύπου, δηλαδή η ML επιτρέπει μόνο weak update της αναφοράς.

Παρακάτω συνοψίζονται οι λειτουργίες που υποστηρίζουν οι ML-style αναφορές

- Δημιουργία Νέας Αναφοράς (Reference Allocation)
  `let` r $=$ `new` 2 `in` $\ldots$ r : `Ref int`
- Ανάθεση (Assignment)
  $\ldots r := 42 \ldots$ destructive update
- Αποδεικτοδότηση (Dereference)
  $\ldots$`print (deref` r`)` prints 42

Αυτό που δεν υποστηρίζουν οι ML-style αναφορές είναι

- Αποδέσμευση μνήμης (Memory Deallocation)
  $\ldots$`free` r

Η οικογένεια της ML, καθώς και οι περισσότερες συναρτησιακές γλώσσες, χρησιμοποιούν συνήθως garbage collector για την απελευθέρωση μνήμης. Η λύση αυτή απαλλάσσει τον προγραμματιστή από το δύσκολο έργο της διαχείρισης της μνήμης και τα προγράμματα από προγραμματιστικά σφάλματα που θα μπορούσαν να οδηγήσουν σε μη ασφαλή χρήση της μνήμης (όπως για παράδειγμα dangling pointers) αλλά παράλληλα επιβαρύνει κατά το χρόνο εκτέλεσης του προγράμματος.

2.2 Το Πρόβλημα της Αποδέσμευσης Μνήμης

Αν, σε αντίθεση με όσο γράφονται παραπάνω, και επιδιώκοντας να αποφύγουμε την επιβάρυνση του garbage collector κατά την εκτέλεση του προγράμματος, θελήσουμε να υποστηρίξουμε τη ρητή αποδέσμευση μνήμης μέσω του τελεστή `free`, που επιδρά πάνω σε αναφορές, τότε προκύπτουν δύο βασικά προβλήματα σχετικά με τη διαχείριση της μνήμης, τα οποία και παρουσιάζονται στη συνέχεια.

Η σημασία του τελεστή `free` είναι η προφανής: αποδεσμεύει άμεσα τη μνήμη που αποτελεί τον αποθηκευτικό χώρο της δοσμένης αναφοράς.

2.2.1 Ξεκρέμαστες Αναφορές

Το πρόβλημα της ξεκρέμαστης (dangling) αναφοράς συνίσταται στη χρήση μιας αναφοράς μετά από την απελευθέρωση της μνήμης που αποτελεί τον αποθηκευτικό της χώρο. Προγράμματα που δημιουργούν ξεκρέμαστες αναφορές μπορούν να προκαλέσουν σφάλματα κατά το χρόνο εκτέλεσης:

Παράδειγμα 2.2.1
```
let r = new 2 in
   free r; deref r
```

Σε συνδυασμό με το aliasing, το οποίο υποστηρίζουν όλες σχεδόν οι γλώσσες προγραμματισμού που έχουν πρακτική εφαρμογή, μπορούν να δημιουργηθούν προγράμματα όπου οι ξεκρέμαστες αναφορές είναι πιο δύσκολο να εντοπιστούν από τον προγραμματιστή:

Παράδειγμα 2.2.2
```
let r = new 2 in
let a = r in
   free r; deref a
```

2.2.2 Διαρροή Μνήμης

Επιπλέον των ξεκρέμαστων αναφορών, στην περίπτωση που χρησιμοποιούμε τον τελεστή `free` για την αποδέσμευση μνήμης και όχι garbage collector, δημιουργείται επίσης το πρόβλημα της μη αποδέσμευσης μνήμης: ο προγραμματιστής αμελεί να απελευθερώσει μνήμη που δεν χρησιμοποιείται από ένα σημείο της εκτέλεσης και έπειτα από το πρόγραμμα, με αποτέλεσμα διαρροή μνήμης που σπαταλά πόρους του συστήματος.

Παράδειγμα 2.2.3
```
let r = new 2 in
   deref r
```

2.3 Γραμμικές Αναφορές

2.3.1 Γραμμικά Συστήματα Τύπων

Για την επίλυση των προβλημάτων που περιγράφονται παραπάνω (αλλά ταυτόχρονα και άλλων προβλημάτων που δεν έχουν άμεσα να κάνουν με τη διαχείριση της μνήμης) έχουν συχνά προταθεί τεχνικές που περιλαμβάνουν τη χρήση ενός συστήματος τύπων. Συχνά δε, το σύστημα τύπων που χρησιμοποιείται είναι γραμμικό.

Ένα καθαρά γραμμικό σύστημα τύπων αντιστοιχίζεται με τη Γραμμική Λογική (Linear Logic) μέσω το ισομορφισμού Curry-Howard. Όταν χρησιμοποιείται ένα τέτοιο σύστημα τύπων κάθε αντικείμενο πρέπει να χρησιμοποιηθεί *ακριβώς μία* φορά. Αυτό σημαίνει ότι δεν είναι δυνατή η αντιγραφή (duplication) ούτε και η απόρριψη (discard) ενός αντικειμένου. Η απαγόρευση της αντιγραφής καθιστά το aliasing αδύνατο και συνεπώς καθιστά την άμεση απελευθέρωση μνήμης μέσω του τελεστή `free` ασφαλή. Παράλληλα όμως αποτελεί και το σημαντικότερο μειονέκτημα των γραμμικών συστημάτων τύπων: Απλώς και μόνο η ανάγνωση μιας τιμής προκαλεί την "κατανάλωση" της τιμής αυτής και συνεπώς η περαιτέρω χρήση της μέσα στο πρόγραμμα καθίσταται αδύνατη.

2.3.2 Μικτά Συστήματα Τύπων

Για να αποφευχθεί ο τελευταίος σοβαρός περιορισμός μπορούν να χρησιμοποιηθούν μεικτά συστήματα τύπων τα οποία υποστηρίζουν γραμμικούς τύπους καθώς και τύπους χωρίς περιορισμούς (unrestricted). Ας σημειωθεί ότι υπάρχουν και μικτά συστήματα τύπων που υποστηρίζουν επιπλέον και άλλων ειδών τύπους όπως:

relevant: αντικείμενα με relevant τύπο δεν μπορούν να απορριφθούν αλλά μπορούν να αντιγραφούν

affine: αντικείμενα με affine τύπο μπορούν να απορριφθούν αλλά όχι να αντιγραφούν

Παρ όλη τη χρησιμότητα που μπορούν να έχουν οι παραπάνω κατηγορίες τύπων (και αντίστοιχα ένα σύστημα τύπων που να τις υποστηρίζει), εδώ θα εστιάσουμε στη χρήση ενός μικτού συστήματος τύπων που υποστηρίζει γραμμικούς τύπους και τύπους χωρίς περιορισμούς για το typing των ML-style αναφορών. Σε ένα τέτοιο σύστημα είναι λογική επιλογή να επιτρέψουμε ανάθεση και αποδεικτοδότηση αναφορών με τύπο χωρίς περιορισμούς, αφού και στις δύο αυτές περιπτώσεις συνήθως δεν επιθυμούμε την κατανάλωση της δοσμένης αναφοράς. Αποδέσμευση πρέπει να επιτρέπεται μόνο για αναφορές με γραμμικό τύπο έτσι ώστε να διασφαλίζεται η απουσία ετερώνυμων (aliased) και να αποκλείεται η επαναχρησιμοποίηση της δοσμένης αναφοράς. Ας σημειωθεί ότι ο μόνος τρόπος να περιορίσουμε το aliasing για μία αναφορά είναι να δίνουμε γραμμικό τύπο στην αναφορά αυτή κατά τη δημιουργία της. Συνεπώς ο τελεστής `new` δημιουργίας νέας αναφοράς πρέπει να επιστρέφει μια γραμμική αναφορά. Ως προς το τι μπορεί να αποθηκευτεί σε μία αναφορά υπάρχουν διάφορες επιλογές. Οι περιορισμοί που χρειάζεται να μπουν εδώ αποσκοπούν στο να αποτρέψουν την αντιγραφή και την απόρριψη αντικειμένων με γραμμικό τύπο. Αντί να επιβάλουμε οποιονδήποτε περιορισμό στο επίπεδο των τύπων μπορούμε να πετύχουμε τα παραπάνω ζητούμενα περιορίζοντας τους τελεστές ανάθεσης και αποδεικτοδότησης έτσι ώστε να μπορούν να εφαρμοστούν μόνο σε αναφορές με περιεχόμενο μη περιορισμένου τύπου. Δε χρειάζεται να περιορίσουμε τον τελεστή αποδέσμευσης `free` εφόσον η εφαρμογή του πάνω σε μια γραμμική αναφορά επιστρέφει ως αποτελέσματα το περιεχόμενο της αναφοράς (και έχει φυσικά ως side effect την αποδέσμευση της αναφοράς). Επιπλέον, μπορούμε να δημιουργήσουμε ένα νέο τελεστή ανταλλαγής `swap`, ο οποίος θα επιτελεί (ταυτόχρονα) τις λειτουργίες αποδεικτοδότησης και ανάθεσης στην περίπτωση των αναφορών με μη περιορισμένο τύπο που έχουν περιεχόμενο με γραμμικό τύπο. Πιο συγκεκριμένα, ο τελεστής `swap` αναθέτει στην αναφορά τη νέα τιμή και επιστρέφει σαν αποτέλεσμα την προηγούμενη τιμή της.

Το τελευταίο απαραίτητο στοιχείο για την αποτελεσματική χρήση ενός μικτού συστήματος τύπων είναι ένας μηχανισμός που να επιτρέπει την ασφαλή προσωρινή μετατροπή του χαρακτηρισμού ενός τύπου από γραμμικό σε μη περιορισμένο και αντίστροφα.

2.4 Σχετική Έρευνα

2.4.1 Το κλασσικό `let!`

Στην κλασσική του μορφή το στοιχείο `let!`, το οποίο αρχικά προτάθηκε από τον Wadler [Wadl90], γράφεται ως

$$\texttt{let!}\,(x)\;y{=}e_1\;\texttt{in}\;e_2$$

Η μεταβλητή x, που έχει αρχικά γραμμικό τύπο, χρησιμοποιείται με τύπο χωρίς περιορισμούς μέσα στο scope του όρου e_1, ο οποίος αποτιμάται πρώτος, και με γραμμικό τύπο μέσα στο scope του όρου e_2. Επιπλέον το αποτέλεσμα της αποτίμησης του όρου e_1 δένεται με τη μεταβλητή y, η οποία μπορεί να εμφανίζεται μέσα στον όρο e_2. Στο παρακάτω παράδειγμα η αναφορά r αποδεικτοδοτείται πολλές φορές μέσα στο σώμα του `let!` για τον υπολογισμό του y.

Παράδειγμα 2.4.1

```
let r = new 6 in
let! (r)
   y = (let a = deref r in
        r := deref r + 1;
        a * deref r)
in
   free r;
   print y
```

Είναι απαραίτητο να μην επιτρέπεται στην τιμή η οποία έχει προσωρινά τύπο χωρίς περιορισμό (εντός του scope του όρου e_1) να διαφύγει με οποιονδήποτε τρόπο από τα όρια της αποτίμησης του όρου e_1. Διαφορετικά, η συνύπαρξη της τιμής με γραμμικό τύπο με τον εαυτό της με τύπο χωρίς περιορισμούς κατά τη διάρκεια της αποτίμησης του όρου e_2 καταστρέφει την ασφάλεια του στοιχείου `let!` και είναι πιθανόν να οδηγήσει σε σφάλματα εκτέλεσης που σχετίζονται με τη λανθασμένη διαχείριση της μνήμης.

Προκειμένου να επιτευχθεί αυτό ο Wadler επιβάλλει τρεις περιορισμούς στο στοιχείο `let!` [Wadl90]. Ο πρώτος περιορισμός είναι ότι ο όρος e_1 πρέπει να έχει αποτιμηθεί πλήρως πριν την αποτίμηση του όρου e_2. "Πλήρης αποτίμηση" σημαίνει ότι η αποτίμηση όλων των υποόρων του e_1, αναδρομικά, πρέπει να έχει τελειώσει πριν ξεκινήσει η αποτίμηση του όρου e_2. Ο δεύτερος περιορισμός είναι ότι οι τύποι των x και e_1 δεν πρέπει να έχουν κοινά τμήματα τα οποία να είναι γραμμικά στην πρώτη περίπτωση και μη-γραμμικά στη δεύτερη. Αυτό απαγορεύει τη χρήση οποιασδήποτε τιμής (σε περίπτωση που αυτή αντικατοπτρίζεται στον τύπο του e_1) με μη-γραμμικό τύπο, όμοιο κατά τα άλλα με τον τύπο του x, στον όρο e_1. Ο τρίτος περιορισμός είναι ότι κανένα τμήμα του τύπου του όρου e_1 δεν πρέπει να είναι τύπος συνάρτησης. Χωρίς αυτόν τον περιορισμό η τιμή με τύπο χωρίς περιορισμού θα μπορούσε να διαφύγει από τα όρια της αποτίμησης του όρου e_1 μέσα στο closure μίας συνάρτησης. Λόγω του τελευταίου αυτού περιορισμού το ακόλουθο παράδειγμα απορρίπτεται.

Παράδειγμα 2.4.2

```
let r = new 7 in
let! (r)
   y =(λ u : Unit. deref r)
in
   free r;
   print (y ())                                  (* ↯ r is dereferenced *)
```

Στη δημοσίευση του Wadler δεν υπάρχουν ML-style αναφορές. Υπάρχουν μόνο συναρτησιακοί πίνακες (functional arrays). Η ενημέρωση (update) ενός τέτοιου πίνακα επιστρέφει ένα νέο πίνακα και συ-

νεπώς ο τύπος της τιμής που αποθηκεύεται στον πίνακα αντικατοπτρίζεται στον τύπο του αποτελέσματος. Στην ML όμως η ενημέρωση είναι καταστροφική για το προηγούμενο περιεχόμενο (destructive update) και ο τελεστής ανάθεσης επιστρέφει τιμή με τύπο unit. Συνεπώς η προσέγγιση του Wadler δεν μπορεί να χρησιμοποιηθεί ως έχει στην περίπτωση που θέλουμε να υποστηρίξουμε ML-style αναφορές ή μη-συναρτησιακούς πίνακες. Αυτό επιβεβαιώνεται και από το ακόλουθο παράδειγμα, όπου κατά την αποτίμηση του e_1 η τιμή r με μη-γραμμικό τύπο αποθηκεύεται μέσα σε μία αναφορά, η οποία μετά χρησιμοποιείται κατά την αποτίμηση του e_2.

Παράδειγμα 2.4.3

```
(* assume p : ᵁRef (ᵁRef Int) *)
let r = new 7 in
let! (r)
    y =(p := r)
in
    free r;
    print (deref (deref p))                    (* ↯ r is dereferenced *)
```

Κανένας από τους προαναφερθέντες περιορισμούς δε φαίνεται να παραβιάζεται στο Παράδειγμα 2.4.3. Δεν είναι προφανές το πώς θα μπορούσε να επεκταθεί η προσέγγιση του Wadler για να υποστηρίξει ML-style αναφορές ή μη-συναρτησιακούς πίνακες.

2.4.2 Τύποι Παρατηρητή

Προκειμένου να ξεπεράσει τους περιορισμούς που θέτει ο Wadler, Ο Odersky ορίζει την έννοια του *παρατηρητή* (observer) τύπου. Στη δουλειά του Odersky υπάρχουν τριών ειδών qualifiers τύπων: *γραμμικός*, *μη-γραμμικός* και *παρατηρητής*. Ο τελευταίος αντιστοιχεί στις γραμμικές τιμές του Wadler οι οποίες έχουν μετατραπεί σε μη-γραμμικές μέσα σε κάποιο `let!`. Επιπλέον προσθέτει στη γλώσσα του Wadler πολυμορφισμό πάνω σε τύπους και περιορισμένο (bounded) πολυμορφισμό πάνω σε qualifiers.

Το στοιχείο `let!` του Odersky γράφεται `let!` $x{=}e_1$ `in` e_2. Οι παρατηρητές χρησιμοποιούνται για να διακρίνουν μεταξύ των αυθεντικών μη-γραμμικών τιμών και των γραμμικών τιμών που έχουν μετατραπεί σε μη-γραμμικές. Το στοιχείο `let!` μετατρέπει τους τύπους από γραμμικούς σε παρατηρητές και αποτελεί το μόνο τρόπο να κατασκευαστεί τιμή με παρατηρητή τύπο.

Δύο χαρακτηριστικά της γλώσσας του Odersky θα μπορούσαν να θεωρηθούν περιορισμοί. Το πρώτο είναι ότι το στοιχείο `let!` μετατρέπει *όλες* τις γραμμικές τιμές που βρίσκονται στο περιβάλλον σε παρατηρούμενες. Αυτό σε συνδυασμό με το ότι η ανάθεση επιτρέπεται μόνο για γραμμικές τιμές εξασφαλίζει ότι το πρόγραμμα του Παραδείγματος 2.4.3 θα απορριφθεί. Ταυτόχρονα όμως απαγορεύει τη χρήση *οποιασδήποτε* γραμμικής τιμής μέσα στον όρο e_1. Το δεύτερο χαρακτηριστικό είναι ότι η μετατροπή των qualifiers γίνεται αναδρομικά μέσα σε ένα τύπο. Για παράδειγμα μια γραμμική αναφορά που περιέχει μια γραμμική τιμή μετατρέπεται σε μια παρατηρούμενη αναφορά που περιέχει μια παρατηρούμενη τιμή. Αυτός είναι ένας ισχυρός περιορισμός. Για παράδειγμα, απαγορεύει την απελευθέρωση (deallocation) μιας γραμμικής αναφοράς που έχει αποθηκευτεί μέσα σε μια άλλη αναφορά.

2.4.3 Γραμμικές Περιοχές

Ένα άλλο σύστημα που χρησιμοποιεί γραμμικούς τύπους για να εξάγει συμπεράσματα σχετικά με τη χρήση αναφορών και του aliasing είναι το $\mathbf{L}^3$ και οι βαριάντες του [Morr05, Flue06, Ahme07]. Στο $\mathbf{L}^3$ κάθε αναφορά συνδέεται σε επίπεδο τύπων με ένα capability και τα capabilities υπόκεινται σε

linear typing. Μία επέκταση του συστήματος αυτού, η οποία υποστηρίζει την προσωρινή μετατροπή των γραμμικών capabilities σε άνευ περιορισμών (unrestricted) capabilities παρέχει τα εξής στοιχεία: `freeze`, `thaw` και `refreeze`. Το πρώτο μετατρέπει ένα γραμμικό capability σε *frozen* (unrestricted). Προκειμένου να επιτραπεί strong update για frozen capabilities παρέχεται το δεύτερο στοιχείο (`thaw`) το οποίο μετατρέπει ένα frozen capability σε γραμμικό. Ο συνδυασμός των `freeze` και `thaw` μοιάζει με με το στοιχείο `let!` του Wadler ή του Odersky με την έννοια ότι μετατρέπει ένα γραμμικό capability σε unrestricted capability για ένα συγκεκριμένο τμήμα του προγράμματος. Έχει το σημαντικό πλεονέκτημα ότι η μετατροπή αυτή δεν περιορίζεται μέσα σε ένα λεκτικό scope. Από την άλλη μεριά, το πρόγραμμα πρέπει να παρέχει ρητώς αποδεικτικά στοιχεία ότι κανένα άλλο frozen capability για μία συγκεκριμένη αναφορά δεν έχει μετατραπεί μέσω του `thaw`. Το στοιχείο `refreeze` χρησιμοποιείται για να μετατρέψει σε frozen ένα capability που έχει προηγουμένως μετατραπεί σε γραμμικό μέσω του `thaw`.

2.4.4 Η γλώσσα Letbang

Η βασική ιδέα που εισάγει η Letbang σε σχέση με τη δουλειά του Wadler είναι ότι ένας τύπος επισημειώνεται επιπλέον με ένα *scope* πέραν του qualifier [Papa08]. Το scope αυτό δείχνει το κατά πόσο μπορεί να χρησιμοποιηθεί η αντίστοιχη τιμή που το έχει ως επισημείωση στον τύπο της. Η γλώσσα Letbang παρουσιάζεται αναλυτικά στο Κεφάλαιο 3 και αποτελεί τη δουλειά στην οποία στηρίζεται κατά κύριο λόγω η παρούσα διπλωματική.

Κεφάλαιο 3

Η γλώσσα Letbang

Η γλώσσα είναι ουσιαστικά ένας λ-calculus με τύπους και ML-style αναφορές. Είναι πολυμορφική πάνω σε pretypes, τα οποία είναι τύποι χωρίς επισημειώσεις (qualifier και scope). Ο συνδυασμός ML-style αναφορών και πολυμορφισμού οδηγεί στην υιοθέτηση του *value restriction*.

3.1 Περιγραφή

Η γλώσσα Letbang [Papa08], σε σχέση με τη δουλειά του Wadler, προσθέτει ένα scope στις επισημειώσεις των τύπων (διατηρεί και τον qualifier) και επεκτείνει το στοιχείο `let!` ώστε να επιτελεί επιπλέον και ένα binding ενός νέου scope:

$$\texttt{at}\ \eta\ \texttt{let!}\ (x{=}e)\ y{=}e_1\ \texttt{in}\ e_2$$

Ο όρος e αποτιμάται και το αποτέλεσμα της αποτίμησης δένεται στη μεταβλητή x. Κατά τα άλλα, παρόμοια με το `let!` του Wadler, η μεταβλητή x χρησιμοποιείται με μη-περιορισμένο τύπο στον όρο e_1 και με γραμμικό τύπο στον όρο e_2. Το αποτέλεσμα της αποτίμησης του όρου e_1 δένεται στη μεταβλητή y η οποία μπορεί να χρησιμοποιηθεί στον όρο e_2. Η μετατροπή του qualifier του τύπου του x από γραμμικό σε μη-περιορισμένο συνοδεύεται και από τη μετατροπή του scope του τύπου του x σε η. Η μεταβλητή scope η είναι δεσμευμένη (bound) μόνο μέσα στον όρο e_1. Τιμές με scope η επιτρέπεται να χρησιμοποιηθούν μόνο μέσα στον όρο e_1 και δεν μπορούν να ξεφύγουν από τα όρια της αποτίμησης αυτού. Γενικότερα, μία τιμή είναι έγκυρη μόνο αν το scope της είναι έγκυρο. Αυτό μπορεί να διασφαλιστεί με στατικό έλεγχο.

Κάθε δομή `let!` του προγράμματος έχει το δικό της μοναδικό scope. Για να διασφαλιστεί αυτό χωρίς να επιβαρυνθεί η γλώσσα με ελέγχους μοναδικότητας, οι κανόνες τύπων και η λειτουργική σημασιολογία της γλώσσας αντικαθιστούν τη μεταβλητή scope η (η οποία παρέχεται από τον προγραμματιστή) με ένα νέο συγκεκριμένο scope (σταθερά) ρ.

Ακολουθούν μερικά παραδείγματα που επιδεικνύουν τον τρόπο με τον οποίο η Letbang αποτρέπει τη διαφυγή τιμών με μη-περιορισμένο τύπο από τα όρια της αποτίμησης του όρου e_1.

Παράδειγμα 3.1.1

```
at η let! (r=new 7)
    y =r                                   (* r : U_η Ref Int *)
in
    free r;                                (* r : L Ref Int *)
    print (deref y)                        (* ↯ y : U_η Ref Int *)
```

Το πρόγραμμα του Παραδείγματος 3.1.1 δεν περνά τον έλεγχο τύπο της γλώσσας και απορρίπτεται επειδή το scope η του τύπου της μεταβλητής y δεν είναι έγκυρο στο σημείο της αποδεικτοδότησης της y, ανεξάρτητα με τον αν η αναφορά r έχει απελευθερωθεί ή όχι.

$$
\begin{array}{rcl}
q & ::= & \mathsf{U} \mid \mathsf{L} \\
\pi & ::= & \eta \mid \rho \mid \bot \\
\phi & ::= & \alpha \mid \mathsf{Unit} \mid \mathsf{Base} \mid \tau \xrightarrow{\vec{\pi}} \tau \mid \forall\, \alpha.\, \tau \mid \mathsf{Ref}\, \tau \\
\tau & ::= & {}^{q}_{\pi}\phi \\
e & ::= & v \mid x \mid e_1\, e_2 \mid e\,[\tau] \mid \texttt{new}\, e \mid \texttt{deref}\, e \mid e_1 := e_2 \mid e_1 :=: e_2 \mid \texttt{free}\, e \\
 & \mid & \texttt{at}\, \eta\, \texttt{let!}\, (x{=}e)\, y{=}e_1\, \texttt{in}\, e_2 \mid \texttt{at}\, \rho\, \texttt{let\$}\, (z)\, y{=}e_1\, \texttt{in}\, e_2 \mid {}^{q}_{\pi}\texttt{loc}\, l \\
v & ::= & {}^{q}_{\pi}\texttt{unit} \mid {}^{q}_{\pi}\lambda\, x : \tau.\, e \mid {}^{q}_{\pi}\Lambda\, \alpha.\, v \mid {}^{q}_{\pi}\texttt{loc}\, l
\end{array}
$$

Σχήμα 3.1: Σύνταξη

Για την περίπτωση που μία τιμή με μη-περιορισμένο τύπο προσπαθεί να διαφύγει μέσα στο closure μιας συνάρτησης, η οποία είναι διαθέσιμη μέσα στον όρο e_2 μέσω της μεταβλητής y, μπορούμε να επανεξετάσουμε το Παράδειγμα 2.4.2. Για να απορρίψουμε προγράμματα σαν αυτό δεν απαγορεύουμε στον όρο e_1 να είναι συνάρτηση ή να χρησιμοποιεί συναρτήσεις και αναφορές με οποιονδήποτε τρόπο. Αντί αυτών, επισημειώνουμε κάθε τύπο συνάρτησης με τη λίστα των scopes τα οποία το σώμα της συνάρτησης χρησιμοποιεί. Προκειμένου να εφαρμοστεί επιτυχώς ο κανόνας τύπων της εφαρμογής συνάρτησης πρέπει όλα τα scopes της λίστας αυτής να είναι έγκυρα στο σημείο της εφαρμογής. Με αυτόν τον χειρισμό το αντίστοιχο του Παραδείγματος 2.4.2 αποτυγχάνει να περάσει τον έλεγχο τύπων της γλώσσας:

Παράδειγμα 3.1.2

```
at η let! (r=new 7)
    y = (λ u : Unit. deref r)                    (* y : Unit -η-> Int *)
in
    free r;
    print (y ())                                 (* ↯ application fails: η is not live *)
```

Παρατηρούμε ότι το y έχει τύπο συνάρτησης που απαιτεί να είναι έγκυρο το scope η προκειμένου η συνάρτηση να μπορεί να εφαρμοστεί.

Οι τελεστές `deref` και `assign` είναι περιορισμένοι ώστε η εφαρμογή τους να επιτρέπεται μόνο σε μη-περιορισμένες αναφορές με μη-περιορισμένο περιεχόμενο. Υπάρχει επιπλέον και ο τελεστής `swap`, οποίος επιτρέπεται να εφαρμοστεί σε μη-περιορισμένες αναφορές με γραμμικό περιεχόμενο. Λειτουργικά ο τελεστής `swap` αναθέτει στη δοσμένη αναφορά νέο γραμμικό περιεχόμενο και επιστρέφει το προηγούμενο περιεχόμενο της αναφοράς.

Τέλος, ένας απαραίτητος περιορισμός (ο οποίος υπάρχει και στα συστήματα του Wadler και του Odersky) είναι ότι δεν μπορεί να επιτραπεί η μετατροπή γραμμικών συναρτήσεων σε μη-περιορισμένες. Με την ορολογία που χρησιμοποιείται και στους κανόνες τύπων: τιμές με τύπο συνάρτησης δεν είναι "bangable". Οι γραμμικές συναρτήσεις υπάρχουν για ένα μόνο λόγο: χρησιμοποιούν στο σώμα τους κάποια γραμμική τιμή και συνεπώς πρέπει να εφαρμοστούν ακριβώς μία φορά. Η μετατροπή μιας γραμμικής συνάρτησης σε μη-περιορισμένη θα έδινε ανεξέλεγκτη πρόσβαση στις γραμμικές τιμές που αυτή χρησιμοποιεί στο σώμα της καθώς θα έδινε τη δυνατότητα η συνάρτηση να εφαρμοστεί πολλές φορές.

3.2 Σύνταξη

Η σύνταξη της Letbang φαίνεται στο σχήμα 3.1. Οι εκφράσεις περιλαμβάνουν το unit, μεταβλητές όρων, λ-αφαίρεση όρου και λ-αφαίρεση pretype, εφαρμογή όρου πάνω σε όρο, εφαρμογή όρου πάνω σε pretype, καθώς και εφαρμογή διαφόρων τελεστών πάνω σε όρους. Υπάρχουν δύο μορφές του

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

$$\frac{\mathsf{qual}((\Gamma_1 \cap \Gamma_2)) \ = \ \mathsf{U}}{\Gamma_1 \oplus \Gamma_2 = \Gamma_1 \cup \Gamma_2}$$

$\boxed{Z \models \pi}$

$$\{\} \models \bot \quad \{\rho\} \models \rho$$

$\boxed{\mathsf{qual}(\Gamma)}$

$$\frac{\mathsf{qual}(\Gamma) \ = \ \mathsf{U} \quad \mathsf{qual}(\tau) \ = \ q}{\mathsf{qual}(\Gamma, (x : \tau)) \ = \ q} \qquad \frac{\mathsf{qual}(\Gamma) \ = \ \mathsf{L}}{\mathsf{qual}(\Gamma, (x : \tau)) \ = \ \mathsf{L}}$$

Σχήμα 3.2: Βοηθητικοί ορισμοί

στοιχείου `let!`. Η πρώτη περιγράφηκε στην Ενότητα 3.1. Η δεύτερη (`let$`) δεν είναι διαθέσιμη στον προγραμματιστή και χρησιμοποιείται εσωτερικά στη λειτουργική σημασιολογία της γλώσσας. Το ίδιο ισχύει και για τα memory locations.

Οι τύποι της γλώσσας αποτελούνται από ένα pretype (ϕ) επισημειωμένο με ένα qualifier (q) και με ένα scope (π). Τα scopes μπορούν να είναι είτε μεταβλητές scope (η) είτε σταθερές scopes (ρ). Οι σταθερές scopes χρησιμοποιούνται μόνο εσωτερικά στη λειτουργική σημασιολογία της γλώσσας. Παρέχεται επίσης ένα προκαθορισμένο scope ($\bot$) το οποίο είναι έγκυρο σε κάθε σημείο του προγράμματος.

3.3 Κανόνες Τύπων

Η σχέση τυποποίησης (typing relation) $\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \ \vdash \ e \ : \ \tau$ της γλώσσας ορίζεται στο Σχήμα 3.3. Το Γ είναι ένα περιβάλλον που δένει μεταβλητές με τύπους, το Δ είναι ένα σύνολο από μεταβλητές pretype, το M δένει locations με τύπους, και το Z είναι ένα σύνολο από έγκυρα scopes.

Οι κανόνες τύπων γλωσσών που χρησιμοποιούν γραμμικά συστήματα τύπων διαφέρουν από αυτούς των συνηθισμένων γλωσσών κυρίως ως προς τον χειρισμό των περιβαλλόντων (typing environments). Στην περίπτωση της Letbang μπορούμε να έχουμε τιμές που να είναι γραμμικές (τίποτε άλλο δεν μπορεί να είναι γραμμικό), οπότε πρέπει να έχουμε ειδικό χειρισμό μόνο για το περιβάλλον Γ. Για την τυποποίηση μιας σύνθετης έκφρασης το περιβάλλον αυτό σπάει σε κατάλληλο αριθμό τμημάτων για την τυποποίηση των συστατικών στοιχείων της σύνθετης έκφρασης και διασφαλίζεται ότι κάθε μεταβλητή με γραμμικό τύπο εμφανίζεται σε ακριβώς ένα τμήμα του αρχικού περιβάλλοντος. Για το σκοπό αυτό ορίζεται ένας τελεστής ένωσης $\Gamma_1 \oplus \Gamma_2$ για περιβάλλοντα, οποίος εφαρμόζεται έγκυρα μόνο εάν η τομή των Γ_1 και Γ_2 δεν περιέχει τιμές με γραμμικό τύπο. Ο τελεστής ορίζεται στο Σχήμα 3.2. Ακόμη, προκειμένου να αποτραπεί η απόρριψη γραμμικών τιμών, οι κανόνες τύπων των βασικών περιπτώσεων (`unit`, μεταβλητές και locations) στο Σχήμα 3.3 περιορίζουν το περιβάλλον Γ ώστε μην περιέχει καμία μεταβλητή με γραμμικό τύπο πέραν αυτής που χρησιμοποιείται (αν υπάρχει).

Μία άλλη απαίτηση είναι ότι ο χειρισμός των scopes πρέπει να είναι *σχετικός*. Μπορούν να χρησιμοποιηθούν πολλές φορές, δεν μπορούν όμως να απορριφθούν. Αυτό εξυπηρετεί στο να περιλαμβάνουμε μόνο τα scopes που πρέπει να είναι έγκυρα στο σημείο εφαρμογής μίας συνάρτησης (και μόνο αυτά) στη λίστα των scopes με την οποία είναι επισημειωμένος ο τύπος της συνάρτησης και έτσι να μην αποδυναμώνεται το typing των συναρτήσεων προσθέτοντας scopes που δε χρησιμοποιούνται στην επισημείωση του τύπου της συνάρτησης. Για το σκοπό αυτό ορίζεται μία σχέση *minimal* scope μεταξύ των scopes π και των περιβαλλόντων scope Z (Σχήμα 3.2). Η σχέση αυτή χρησιμοποιείται στους κανόνες τύπων ώστε να διασφαλίζεται ότι το top-level scope του τύπου που έχει ως αποτέλεσμα ο κανόνας να είναι πάντα έγκυρο.

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

$$\frac{\mathsf{qual}(\Gamma) = \mathsf{U} \qquad Z \models \pi}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash \frac{q}{\pi}\mathtt{unit} : \frac{q}{\pi}\mathsf{Unit}} \qquad \frac{\mathsf{qual}(\Gamma) = \mathsf{U} \qquad Z \models \mathsf{scope}\,\tau}{\Gamma, x : \tau; \Delta; \mathrm{M}; \mathrm{Z} \vdash x : \tau}$$

$$\frac{\mathsf{qual}(\Gamma) = q \qquad \mathrm{Z} \models \pi \qquad \Delta \models \tau \qquad \Gamma, x : \tau; \Delta; \mathrm{M}; \mathrm{Z}_e \vdash e : \tau_r}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash \frac{q}{\pi}\lambda\, x : \tau.\, e : \frac{q}{\pi}(\tau \xrightarrow{\mathrm{Z}_e} \tau_r)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \mathrm{M}; \mathrm{Z}_1 \vdash e_1 : \frac{q}{\pi}(\tau_1 \xrightarrow{Z_e} \tau_2) \qquad \Gamma; \Delta; \mathrm{M}; \mathrm{Z}_2 \vdash e_2 : \tau_1 \\ \mathrm{Z} \models \mathsf{scope}\,\tau_2 \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_1) \Rightarrow \rho \notin \mathrm{Z}_2 \cup \mathrm{Z} \cup \mathrm{Z}_e \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_2) \Rightarrow \rho \notin \mathrm{Z}_1 \cup \mathrm{Z} \cup \mathrm{Z}_e\end{array}}{\Gamma_1 \oplus \Gamma_2; \Delta; \mathrm{M}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \cup \mathrm{Z} \cup \mathrm{Z}_e \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\mathsf{qual}(\Gamma) = q \qquad \mathrm{Z} \models \pi \qquad \Gamma; \Delta, \alpha; \mathrm{M}; \mathrm{Z} \vdash e : \tau}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash \frac{q}{\pi}\Lambda\, \alpha.\, v : \frac{q}{\pi}\forall\, \alpha.\, \tau}$$

$$\frac{\begin{array}{c}\mathrm{Z}_\tau \models \mathsf{scope}\,\tau \qquad \Delta \models \phi \\ \Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash e : \frac{q}{\pi}\forall\, \alpha.\, \tau \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e) \Rightarrow \rho \notin \mathrm{Z}_\tau \cup \mathrm{Z}_e\end{array}}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \cup \mathrm{Z}_e \cup \mathrm{Z}_\tau \vdash e\,[\phi] : \tau\{\alpha \mapsto \phi\}}$$

$$\frac{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash e : \tau}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash \mathtt{new}\, e : \frac{\mathsf{L}}{\perp}\mathsf{Ref}\,\tau}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash e : \frac{\mathsf{U}}{\pi}\mathsf{Ref}\,\tau \\ \mathsf{qual}(\tau) = \mathsf{U} \qquad \mathrm{Z}_\tau \models \mathsf{scope}\,\tau \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e) \Rightarrow \rho \notin \mathrm{Z}_\tau\end{array}}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \cup \mathrm{Z}_\tau \vdash \mathtt{deref}\, e : \tau}$$

$$\frac{\begin{array}{c}\Gamma_1; \Delta; \mathrm{M}; \mathrm{Z}_1 \vdash e_1 : \frac{\mathsf{U}}{\pi}\mathsf{Ref}\ \tau \\ \Gamma_2; \Delta; \mathrm{M}; \mathrm{Z}_2 \vdash e_2 : \tau \\ \mathsf{qual}(\tau) = \mathsf{U} \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_1) \Rightarrow \rho \notin \mathrm{Z}_2 \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_2) \Rightarrow \rho \notin \mathrm{Z}_1\end{array}}{\Gamma_1 \oplus \Gamma_2; \Delta; \mathrm{M}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \vdash e_1 := e_2 : \frac{\mathsf{U}}{\perp}\mathsf{Unit}} \qquad \frac{\begin{array}{c}\Gamma_1; \Delta; \mathrm{M}; \mathrm{Z}_1 \vdash e_1 : \frac{\mathsf{U}}{\pi}\mathsf{Ref}\ \tau \qquad \Gamma_2; \Delta; \mathrm{M}; \mathrm{Z}_2 \vdash e_2 : \tau \\ \mathsf{qual}(\tau) = \mathsf{L} \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_1) \Rightarrow \rho \notin \mathrm{Z}_2 \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_2) \Rightarrow \rho \notin \mathrm{Z}_1\end{array}}{\Gamma_1 \oplus \Gamma_2; \Delta; \mathrm{M}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \vdash e_1 :=: e_2 : \tau}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash e : \frac{\mathsf{L}}{\perp}\mathsf{Ref}\,\tau \\ \mathrm{Z}_\tau \models \mathsf{scope}\,\tau \qquad \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e) \Rightarrow \rho \notin \mathrm{Z}_\tau\end{array}}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \cup \mathrm{Z}_\tau \vdash \mathtt{free}\, e : \tau} \qquad \frac{\mathsf{qual}(\Gamma) = \mathsf{U} \qquad Z \models \pi}{\Gamma; \Delta; \mathrm{M}, l : \tau; \mathrm{Z} \vdash \frac{q}{\pi}\mathtt{loc}\ l : \frac{q}{\pi}\mathsf{Ref}\,\tau}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash e : \frac{\mathsf{L}}{\perp}\phi \\ \Gamma_1, x : \frac{\mathsf{U}}{\rho}\, \phi; \Delta; \mathrm{M}; \mathrm{Z}_1, \rho \vdash e_1\{\eta \mapsto \rho\} : \tau_1 \qquad \mathsf{fresh}\,\rho \\ \Gamma_2, x : \frac{\mathsf{L}}{\perp}\, \phi, y : \tau_1; \Delta; \mathrm{M}; \mathrm{Z}_2 \vdash e_2 : \tau_2 \\ \mathsf{bangable}\,\phi \qquad \mathsf{scope}\,\tau_1 \neq \rho \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e) \Rightarrow \rho \notin \mathrm{Z}_1 \cup \mathrm{Z}_2 \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_1) \Rightarrow \rho \notin \mathrm{Z} \cup \mathrm{Z}_2 \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_2) \Rightarrow \rho \notin \mathrm{Z} \cup \mathrm{Z}_1\end{array}}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; \mathrm{M}; \mathrm{Z} \cup \mathrm{Z}_1 \cup \mathrm{Z}_2 \vdash \mathtt{at}\, \eta\, \mathtt{let!}\, (x{=}e)\, y{=}e_1\, \mathtt{in}\, e_2 : \tau_2}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash x : \frac{\mathsf{U}}{\rho}\phi \\ \Gamma_1, x : \frac{\mathsf{U}}{\rho}\, \phi; \Delta; \mathrm{M}; \mathrm{Z}_1 \vdash e_1 : \tau_1 \\ \Gamma_2, x : \frac{\mathsf{L}}{\perp}\, \phi, y : \tau_1; \Delta; \mathrm{M}; \mathrm{Z}_2 \vdash e_2 : \tau_2 \\ \mathsf{bangable}\,\phi \qquad \mathsf{scope}\,\tau_1 \neq \rho \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_1) \Rightarrow \rho \notin \mathrm{Z} \cup \mathrm{Z}_2 \\ \forall \rho.\rho \in \mathsf{FREE}_{\mathsf{raw}}(e_2) \Rightarrow \rho \notin \mathrm{Z} \cup \mathrm{Z}_1\end{array}}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; \mathrm{M}; \mathrm{Z} \cup \mathrm{Z}_1 \cup \mathrm{Z}_2 \vdash \mathtt{at}\, \rho\, \mathtt{let\$}\, (x)\, y{=}e_1\, \mathtt{in}\, e_2 : \tau_2}$$

Σχήμα 3.3: Κανόνες τύπων

Η σχέση minimal scope επίσης διασφαλίζει ότι ο έλεγχος τύπων επιτρέπει στους τύπους των όρων να είναι επισημειωμένοι μόνο με σταθερές scope (ρ) ή με το προκαθορισμένο scope ($\perp$) αλλά όχι με μεταβλητές scopes. Νέες σταθερές scope εισάγονται από τον κανόνα του `let!`, όπου ο έλεγχος τύπων της έκφρασης e_1 γίνεται αφού η μεταβλητή scope έχει αντικατασταθεί από μία νέα σταθερά scope ρ.

Οι σταθερές scope ρ πρέπει να είναι μοναδικές, να εισάγονται δηλαδή μόνο μία φορά, και να μην μπορούν να διαφύγουν από το σώμα του αντίστοιχου `let$`. Αυτό εξασφαλίζεται από την προϋπόθεση:

$$\forall \rho . \rho \in \mathsf{FREE}_{\mathsf{raw}}(e) \Rightarrow \rho \notin \mathrm{Z}$$

η οποία εμφανίζεται σε κανόνες τύπων ώστε να απαγορεύει σε όλες τις σταθερές scope που εισάγονται από υποόρους του όρου e να εμφανίζονται σε άλλα περιβάλλοντα scope (Z) που χρησιμοποιούνται στον κανόνα –πέραν φυσικά αυτού που χρησιμοποιείται για το typing του ίδιου του όρου e. Για παράδειγμα, στον κανόνα για την εφαρμογή πάνω σε όρο οι σταθερές scope που εισάγονται στον όρο e_1 δεν μπορούν να εμφανίζονται στο περιβάλλον scope το οποίο χρησιμοποιείται για το typing του όρου e_2.

Το typing του όρου `let$` διαφέρει από αυτό του όρου `let!` μόνο στο ότι η μεταβλητή scope η έχει αντικατασταθεί από τη σταθερά scope ρ και η γραμμική έκφραση e του `let!` έχει μετατραπεί σε μη-περιορισμένη.

3.4 Λειτουργική Σημασιολογία

Για τη Letbang ορίζεται small-step, call-by-value λειτουργική σημασιολογία (Σχήμα 3.4). Η λειτουργική σημασιολογία της γλώσσας είναι μια σχέση ανάμεσα σε διαρρυθμίσεις (configurations) που αποτελούνται από ένα store S, που είναι μία αντιστοιχία μεταξύ μεταβλητών και τιμών, μία μνήμη μ, που έιναι μια αντιστοιχία μεταξύ locations και μεταβλητών, και ένα όρο της γλώσσας e.

Για τον ορισμό της λειτουργικής σημασιολογίας θεωρούμε ότι η αποτίμηση τερματίζει σε μία μεταβλητή και όχι σε μία τιμή (τέτοιες μεταβλητές θα μπορούσαν να οριστούν ξεχωριστά ως "auto-variables", παρόλα αυτά εδώ αντιμετωπίζονται με τον ίδιο τρόπο που αντιμετωπίζονται και οι μεταβλητές του προγράμματος). Ο κανόνας αποτίμησης για τις τιμές ορίζει ότι μία τιμή δένεται σε μια νέα μεταβλητή z και αποθηκεύεται στο store S. Πρόσβαση σε αυτή την τιμή μπορεί να δοθεί μόνο μέσω της μεταβλητής z.

Στο Σχήμα 3.5 ορίζεται μία συνάρτηση $S; x \Downarrow S'; v$, η οποία αναζητά τη μεταβλητή με όνομα x στο store S. Εάν ο qualifier της τιμής v είναι `L` τότε η μεταβλητή αφαιρείται από το store S' που παίρνουμε στο αποτέλεσμα, διαφορετικά το S' είναι ίδιο με το S. Αυτό διασφαλίζει ότι οι γραμμικές τιμές μπορούν να χρησιμοποιηθούν από το πρόγραμμα μόνο μία φορά, ενώ μη-περιορισμένες τιμές μπορούν να χρησιμοποιηθούν οσεσδήποτε φορές.

Η μνήμη μ της σημασιολογίας της γλώσσας χρησιμοποιείται για το χειρισμό των περιεχομένων των αναφορών. Καθώς η αποτίμηση τερματίζει σε μεταβλητές αντί για τιμές, η μνήμη αντιστοιχίζει locations σε μεταβλητές και όχι σε τιμές. Η συγκεκριμένη τιμή που είναι αποθηκευμένη μπορεί να προσπελαστεί μέσω της αντιστοίχισης της μεταβλητής μέσα στο store. Με τον τρόπο αυτό έχουμε και γραμμικό χειρισμό των περιεχομένων των αναφορών.

Το στοιχείο `let!` χρησιμοποιείται στη σημασιολογία της Letbang μόνο για να αποτιμήσει τον όρο e, το αποτέλεσμα της αποτίμησης του οποίου πρόκειται να μετατραπεί σε μη-περιορισμένο. Όταν ο όρος e έχει αποτιμηθεί σε μία μεταβλητή (z) αντιστοιχισμένη με την τιμή v μέσω του store S, ο αντίστοιχος σημασιολογικός κανόνας μετατρέπει την τιμή v μέσα στο store S σε μη-περιορισμένη και της δίνει ένα νέο scope ρ. Η δέσμευση της μεταβλητής x και οι εμφανίσεις της μέσα στον όρο e_1 αλλάζουν

$$\boxed{S;\mu;e \hookrightarrow S';\mu';e'}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1\ e_2 \hookrightarrow S';\mu';e_1'\ e_2} \qquad \frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x\ e_2 \hookrightarrow S';\mu';x\ e_2'} \qquad \frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;e\ [\phi] \hookrightarrow S';\mu';e'\ [\phi]}$$

$$\frac{S\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\texttt{new}\ e \hookrightarrow S';\mu';\texttt{new}\ e'} \qquad \frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\texttt{deref}\ e \hookrightarrow S';\mu';\texttt{deref}\ e'} \qquad \frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\texttt{free}\ e \hookrightarrow S';\mu';\texttt{free}\ e'}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1 := e_2 \hookrightarrow S';\mu';e_1' := e_2} \qquad \frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x := e_2 \hookrightarrow S';\mu';x := e_2'}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;e_1 :=: e_2 \hookrightarrow S';\mu';e_1' :=: e_2} \qquad \frac{S;\mu;e_2 \hookrightarrow S';\mu';e_2'}{S;\mu;x :=: e_2 \hookrightarrow S';\mu';x :=: e_2'}$$

$$\frac{S;\mu;e \hookrightarrow S';\mu';e'}{S;\mu;\texttt{at}\ \eta\ \texttt{let!}\ (x{=}e)\ y{=}e_1\ \texttt{in}\ e_2 \hookrightarrow S';\mu';\texttt{at}\ \eta\ \texttt{let!}\ (x{=}e')\ y{=}e_1\ \texttt{in}\ e_2}$$

$$\frac{S;\mu;e_1 \hookrightarrow S';\mu';e_1'}{S;\mu;\texttt{at}\ \rho\ \texttt{let\$}\ (x)\ y{=}e_1\ \texttt{in}\ e_2 \hookrightarrow S';\mu';\texttt{at}\ \rho\ \texttt{let\$}\ (x)\ y{=}e_1'\ \texttt{in}\ e_2}$$

$$\frac{\text{fresh } z}{S;\mu;v \hookrightarrow S, z \mapsto v;\mu;z} \qquad \frac{S;z \Downarrow S';^{q}_{\pi}\ \lambda\, x : \tau.\, e}{S;\mu;z\ y \hookrightarrow S';\mu;e\{x \mapsto y\}} \qquad \frac{S;z \Downarrow S';^{q}_{\pi}\ \Lambda\, \alpha.\, e}{S;\mu;z\ [\phi] \hookrightarrow S';\mu;e\{\alpha \mapsto \phi\}}$$

$$\frac{\text{fresh } z \qquad \text{fresh } l}{S;\mu;\texttt{new}\ x \hookrightarrow S, z \mapsto^{\mathsf{L}}_{\perp}\ \texttt{loc}\ l;\mu, l \mapsto x;z} \qquad \frac{S;x \Downarrow S';^{q}_{\pi}\ \texttt{loc}\ l}{S;\mu, l \mapsto z;\texttt{deref}\ x \hookrightarrow S';\mu, l \mapsto z;z}$$

$$\frac{S;x \Downarrow S';^{q}_{\pi}\ \texttt{loc}\ l}{S;\mu, l \mapsto z;\texttt{free}\ x \hookrightarrow S';\mu;z} \qquad \frac{S;x \Downarrow S';^{q}_{\pi}\ \texttt{loc}\ l}{S;\mu, l \mapsto z;x := y \hookrightarrow S';\mu, l \mapsto y;^{\mathsf{U}}_{\perp}\ \texttt{unit}}$$

$$\frac{S;x \Downarrow S';^{q}_{\pi}\ \texttt{loc}\ l}{S;\mu, l \mapsto z;x := y \hookrightarrow S';\mu, l \mapsto y;z}$$

$$\frac{\begin{array}{c}\text{fresh } \rho \\ v' = v, \text{ with scope } \rho \text{ and qual } \mathsf{U}\end{array}}{S, z \mapsto v;\mu;\texttt{at}\ \eta\ \texttt{let!}\ (x{=}z)\ y{=}e_1\ \texttt{in}\ e_2 \hookrightarrow S, z \mapsto v';\mu;\texttt{at}\ \rho\ \texttt{let\$}\ (z)\ y{=}e_1\{\eta \mapsto \rho\}\{x \mapsto z\}\ \texttt{in}\ e_2}$$

$$\frac{\text{fresh } w \qquad v' = v, \text{ with scope } \perp \text{ and qual } \mathsf{L}}{S, x \mapsto v;\mu;\texttt{at}\ \rho\ \texttt{let\$}\ (x)\ y{=}z\ \texttt{in}\ e_2 \hookrightarrow S, w \mapsto v';\mu;e2\{x \mapsto w\}\{y \mapsto z\}}$$

Σχήμα 3.4: Operation semantics rules

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

$$\frac{\text{qual}(v) = \mathsf{U}}{S, x \mapsto v;x \Downarrow S, x \mapsto v;v} \qquad \frac{\text{qual}(v) = \mathsf{L}}{S, x \mapsto v;x \Downarrow S;v}$$

Σχήμα 3.5: Store lookup

σε z και η μεταβλητή scope η αντικαθίσταται από το scope ρ. Ο κανόνας έχει σαν αποτέλεσμα τον αντίστοιχο όρο `let$`.

Η αποτίμηση του όρου `let$` όταν ο όρος e_1 έχει αποτιμηθεί σε μια μεταβλητή z καταλήγει στον όρο e_2 όπου οι μεταβλητές x και y έχουν αντικατασταθεί κατάλληλα: η y αντιστοιχεί στο αποτέλεσμα της αποτίμησης του e_1 και έχει αντικατασταθεί από την z ενώ η x έχει αντικατασταθεί από μία νέα μεταβλητή w. Η w, η οποία είναι αντιστοιχισμένη στο store S με την τιμή v' (η οποία είναι η αρχική τιμή, που έχει αποκτίσει ξανά γραμμικό qualifier), χρησιμοποιείται για να αντικαταστάσει τη μεταβλητή x, που είναι αντιστοιχισμένη με την τιμή που έχει μετατραπεί σε μη περιορισμένη νωρίτερα (v).

Κεφάλαιο 4

Μετασχηματισμός Περάσματος Συνεχειών

4.1 Περιγραφή του Στυλ Περάσματος Συνεχειών

Το στυλ περάσματος συνεχειών (continuation-passing style, CPS) είναι μία μορφή συναρτησιακού προγραμματισμού όπου σε κάθε σημείο του προγράμματος όλοι οι μελλοντικοί υπολογισμοί αντιπροσωπεύονται από ένα continuation και ο έλεγχος περνιέται ρητά σε continuations. Συνεπώς, στην περίπτωση του προγραμματισμού σε CPS, καμία συνάρτηση δεν επιστρέφει ποτέ αλλά αντί αυτού περνά τον έλεγχο σε ένα continuation, το οποίο έχει αυτή δεχτεί ως παράμετρο και στο οποίο πιθανόν δίνει ως παράμετρο το αποτέλεσμα που κανονικά θα επέστρεφε.

Ένα continuation είναι μία αφηρημένη αναπαράσταση της κατάστασης ελέγχου (control state) ενός προγράμματος. Ο τρόπος υλοποίησης των continuations σε μία γλώσσα προγραμματισμού ποικίλει και γενικά εξαρτάται από την προτιθέμενη χρήση τους, από το αν είναι εμφανή στον προγραμματιστή ή αν χρησιμοποιούνται μόνο εσωτερικά και από άλλους παράγοντες. Σε επίπεδο λ-calculus, ένα continuation μπορεί να θεωρηθεί ως μία συνάρτηση που δέχεται ως όρισμα το αποτέλεσμα ενός υπολογισμού και περιέχει τους υπολογισμούς που θα ακολουθήσουν. Κατά τα άλλα, δε διαφέρει από τις κοινές συναρτήσεις. Στο Παράδειγμα 4.1.1 παρουσιάζεται η συνάρηση παραγοντικό στην αναδρομική της μορφή γραμμένη στη γλώσσα OCaml καθώς και η εκδοχή αυτής σε CPS.

Παράδειγμα 4.1.1

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)
```

```
(* converted to CPS primitives *)
let addcps x y k = k (x + y)
let multcps x y k = k (x * y)
let eqcps x y k = k (x = y)

let rec factcps n k =
  eqcps n 0 (fun c ->
    if c then k 1
    else addcps n (-1) (fun a0 -> factcps a0 (fun a1 -> multcps n a1 k)))
```

Ας σημειωθεί ότι ο άμεσος προγραμματισμός σε CPS σε μία γλώσσα που δε διαθέτει πρώτης τάξης continuations αλλά διαθέτει πρώτης τάξης συναρτήσεις απαιτεί βελτιστοποίηση tail call (tail call optimization, TCO) προκειμένου τα προγράμματα να συμπεριφέρονται ικανοποιητικά ως προς τη χρήση των πόρων του συστήματος. Διαφορετικά η στοίβα ενεργοποίησης (call stack) μεγαλώνει συνεχώς για κάθε κλήση συνάρτησης.

4.1.1 Περιγραφή Ιδιοτήτων και Χρησιμότητα του CPS

Σημαντική ιδιότητα του CPS είναι ότι κάνει εμφανέστερη τη σημασιολογία του προγράμματος και έτσι κάνει ευκολότερη την ανάλυσή του. Για παράδειγμα, όλες οι ενδιάμεσες τιμές αποκτούν ονόματα, οι tail calls γίνονται εμφανείς, όπως επίσης, ίσως το κυριότερο, εμφανής γίνεται και η σειρά αποτίμησης (evaluation order): Η αποτίμηση ενός όρου σε μορφή CPS είτε γίνει με στρατηγική call-by-name είτε γίνει με στρατηγική call-by-value οδηγεί στα ίδια βήματα αποτίμησης [Plot75, Reyn72].

Τα continuations μπορούν να αναπαραστήσουν άλλα στοιχεία ελέγχου και έτσι να κάνουν εμφανή και τη σημασιολογία αυτών –οι εξαιρέσεις (exceptions), για παράδειγμα, μπορούν να αναπαρασταθούν με τη χρήση continuations. Μπορούν επίσης να χρησιμοποιηθούν για την αναστολή ενός υπολογισμού και την επιστροφή σε αυτόν στο μέλλον. Τέλος, μπορούν να χρησιμοποιηθούν αυτούσια, σε μια γλώσσα που διαθέτει πρώτης τάξης (first class) continuations για να υλοποιηθεί οποιοσδήποτε έλεγχος ροής (control flow).

4.1.2 Χρήσεις του CPS

Το CPS έχει βρει πολλές διαφορετικές πρακτικές εφαρμογές μερικές από τις οποίες αναφέρονται παρακάτω:

- Ως ενδιάμεση γλώσσα σε μεταγλωττιστές συναρτησιακών γλωσσών. Η χρήση του CPS σαν ενδιάμεση γλώσσα μπορεί να έχει διάφορους σκοπούς όπως η διευκόλυνση των βελτιστοποιήσεων (optimizations) –Ο μεταγλωττιστής SML/NJ για τη γλώσσα ML είναι ένα παράδειγμα αυτής της χρήσης– ή η υλοποίηση άλλων στοιχείων, όπως για παράδειγμα οι coroutines
- Ως μέσο για τη σύλληψη, τον ορισμό και την κατανόηση (και πιθανόν και την υλοποίηση) ισχυρών στοιχείων ελέγχου διαθέσιμων στον προγραμματιστή, όπως για παράδειγμα ο τελεστής call-with-current-continuation (call/cc), ο οποίος έχει τις καταβολές του στη γλώσσα Scheme και σήμερα έχει υιοθετηθεί και από άλλες γλώσσες και με τη χρήση του οποίου μπορεί κανείς να υλοποιήσει σύνθετα στοιχεία όπως backtracking, coroutines, generators και άλλα
- Ως μέσο για την εξάλειψη της ανάγκης χρήσης στοίβας ενεργοποίησης σε γλώσσες που υποστηρίζουν tail call optimization, αφού στο CPS όλες οι κλήσεις είναι tail calls. Αυτό φαίνεται και στο Παράδειγμα 4.1.1 όπου η μετατροπή του αρχικού προγράμματος (το οποίο δεν είναι tail recursive) σε CPS μετατρέπει αυτόματα όλες τις κλήσεις σε tail calls

4.2 Αυτοματοποιημένη Μετατροπή σε CPS

Η μετατροπή ενός προγράμματος γραμμένου σε μια συναρτησιακή γλώσσα σε συμβατική μορφή σε ένα πρόγραμμα σε μορφή CPS είναι δυνατόν να γίνει με αυτοματοποιημένο τρόπο. Η μορφή CPS ενός προγράμματος δεν είναι βέβαια μοναδική. Στις περιπτώσεις που απαιτήθηκε κατά καιρούς μια τέτοια μετατροπή χρησιμοποιήθηκαν διάφορες μηχανιστικές μέθοδοι που έδιναν ως αποτέλεσμα ένα πρόγραμμα σε κάποια μορφή CPS.

Δεν είναι βέβαια όλες οι δυνατές μορφές CPS ενός προγράμματος κατάλληλες για όλες τις χρήσεις. Για παράδειγμα, στις πρακτικές εφαρμογές συνήθως ενδιαφέρει το μέγεθος του προγράμματος CPS που προκύπτει σε σχέση με το αρχικό πρόγραμμα. Σχετιζόμενο με το μέγεθος είναι και το κατά πόσο μπορούν να πραγματοποιηθούν εύκολα κάποια post-reductions στο πρόγραμμα CPS (μερική αποτίμηση κάποιων redexes δηλαδή) ενώ συχνά είναι επιθυμητό αυτά να πραγματοποιούνται σε ένα πέρασμα για λόγους αποδοτικότητας. Στην περίπτωση θεωρητικής δουλειάς συνήθως ενδιαφέρει να μην γίνονται ποτέ τέτοια post-reductions σε redexes του τελικού προγράμματος που αντιστοιχούν σε

$$
\begin{aligned}
[\![x]\!] &= \lambda\,\kappa.\,@\,\kappa\,x \\
[\![\lambda\,x.\,M]\!] &= \lambda\,\kappa.\,@\,\kappa\,(\lambda\,x.\,[\![M]\!]) \\
[\![@\,M\,N]\!] &= \lambda\,\kappa.\,@\,[\![M]\!]\,(\lambda\,m.\,@\,[\![N]\!]\,(\lambda\,n.\,@\,(@\,m\,n)\,\kappa))
\end{aligned}
$$

Σχήμα 4.1: Μετασχηματισμός CPS των Fischer και Plotkin

redexes του αρχικού προγράμματος αλλά μόνο σε redexes που εισάγονται από τη μηχανιστική μετατροπή σε CPS. Από την άλλη σε πρακτικές εφαρμογές υπάρχουν περιπτώσεις που αυτό δεν είναι σημαντικό και είναι αποδεκτό ή και επιθυμητό ένας post-reducer να εξαντλεί όλα τα περιθώρια μέσα στα οποία διατηρείται με κάποια έννοια η σημασιολογία του αρχικού προγράμματος.

4.2.1 Ο μετασχηματισμός CPS του Plotkin

Ο Gordon Plotkin στη μελέτη του για τη σχέση μεταξύ της γλώσσας ISWIM και του λ-calculus προτείνει ένα μετασχηματισμό όρων μιας call-by-value γλώσσας σε όρους μιας call-by-name γλώσσας [Plot75]. Πρόκειται για ένα μετασχηματισμό CPS (που είχε προταθεί και από τον Michael Fischer [Fisc72]) ο οποίος παρουσιάζεται στο Σχήμα 4.1 και για τον οποίο αποδεικνύονται, μεταξύ άλλων, και τα εξής:

Θεώρημα 4.2.1 (Indifference) $Eval_N(@\,[\![M]\!]\,(\lambda\,x.\,x)) = Eval_V(@\,[\![M]\!]\,(\lambda\,x.\,x))$, *για κάθε πρόγραμμα* M.

Θεώρημα 4.2.2 (Simulation) $\Psi(Eval_V(M)) = Eval_N(@\,[\![M]\!]\,(\lambda\,x.\,x))$, *για κάθε πρόγραμμα* M.

Η συνάρτηση Ψ (από τιμές σε τιμές) ορίζεται ως:

$$
\begin{aligned}
\Psi(x) &= x \\
\Psi(\lambda\,x.\,M) &= \lambda\,x.\,[\![M]\!]
\end{aligned}
$$

Οι Albert Meyer και Mitchell Wand στη μελέτη τους σχετικά με τη σημασιολογία του μετασχηματισμού CPS δουλεύουν με λ-calculus με απλούς τύπους (Σχήμα 4.2) και επεκτείνουν το μετασχηματισμό CPS του Plotkin ώστε να μετασχηματίζονται κατάλληλα και οι επισημειώσεις τύπων των υποόροων του προς μετάφραση όρου [Meye85]. Ας σημειωθεί ότι μεταξύ των τύπων υπάρχει και ο διακεκριμένος τύπος o, ο οποίος είναι ο τύπος των *απαντήσεων* (του αποτελέσματος δηλαδή στο οποίο καταλήγει ένας υπολογισμός σε CPS). Με το σκεπτικό ότι στο CPS χειριζόμαστε αντικείμενα που είναι αναπαραστάσεις αντικειμένων που υπάρχουν στο συμβατικής μορφής αρχικό πρόγραμμα, αντιστοιχίζουν σε κάθε τύπο a έναν τύπο a', που είναι ο τύπος των αναπαραστάσεων στο CPS των αντικειμένων τύπου a. Κάθε τύπος ground αναπαρίσταται από τον εαυτό του. Μία συνάρτηση τύπου $a \to b$ στον αρχικό όρο αντιστοιχίζεται στο μετασχηματισμένο σε όρο CPS με μία συνάρτηση που παίρνει δύο ορίσματα: μια αναπαράσταση ενός αντικειμένου τύπου a και ένα continuation που δέχεται μια αναπαράσταση ενός αντικειμένου τύπου b. Με τις πληροφορίες αυτές κατασκευάζουμε την ακόλουθη συνάρτηση:

$$
\begin{aligned}
\sigma' &= \sigma \\
(\tau_1 \to \tau_2)' &= {\tau_1}' \to ({\tau_2}' \to o) \to o
\end{aligned}
$$

Ο επεκτεταμένος μετασχηματισμός CPS, ο οποίος φαίνεται στο Σχήμα 4.3, για κάθε όρο M τύπου τ κατασκευάζει έναν όρο $[\![M]\!]$ τύπου $(\tau' \to o) \to o$ (αποδεικνύεται με επαγωγή στο μήκος των όρων).

$$
\begin{aligned}
\tau &::= \sigma_1, \sigma_2, \ldots \mid \tau \to \tau \mid o \\
M &::= x \mid \lambda\, x : \tau.\, M \mid @\, M_1\, M_2
\end{aligned}
$$

Σχήμα 4.2: λ-calculus με απλούς τύπους: Σύνταξη

$$
\begin{aligned}
[\![x : \tau]\!] &= \lambda\, \kappa : \tau' \to o.\, @\, \kappa\, x \\
[\![(\lambda\, x : \tau_1.\, M) : \tau_1 \to \tau_2]\!] &= \lambda\, \kappa : ({\tau_1}' \to ({\tau_2}' \to o) \to o) \to o.\, @\, \kappa\, (\lambda\, x : {\tau_1}'.\, [\![M]\!]) \\
[\![@\, (M : \tau_1 \to \tau_2)\, (N : \tau_1)]\!] &= \lambda\, \kappa : {\tau_2}' \to o.\, @\, [\![M]\!]\, (\lambda\, m : {\tau_1}' \to ({\tau_2}' \to o) \to o.\, \\
&\qquad @\, [\![N]\!]\, (\lambda\, n : {\tau_1}'.\, @\, (@\, m\, n)\, \kappa))
\end{aligned}
$$

Σχήμα 4.3: Μετασχηματισμός CPS των Meyer και Wand για λ-calculus με απλούς τύπους

$$
\begin{aligned}
[\![x]\!] &= \overline{\lambda}\, \kappa.\, \overline{@}\, \kappa\, x \\
[\![\lambda\, x.\, M]\!] &= \overline{\lambda}\, \kappa.\, \overline{@}\, \kappa\, (\underline{\lambda}\, x.\, \underline{\lambda}\, k.\, \overline{@}\, [\![M]\!]\, (\overline{\lambda}\, m.\, \underline{@}\, k\, m)) \\
[\![@\, M\, N]\!] &= \overline{\lambda}\, \kappa.\, \overline{@}\, [\![M]\!]\, (\overline{\lambda}\, m.\, \overline{@}\, [\![N]\!]\, (\overline{\lambda}\, n.\, \underline{@}\, (\underline{@}\, m\, n)\, (\underline{\lambda}\, \alpha.\, \overline{@}\, \kappa\, \alpha)))
\end{aligned}
$$

Σχήμα 4.4: Μετασχηματισμός CPS ενός περάσματος των Danvy και Filinski

4.2.2 Ο μετασχηματισμός CPS ενός περάσματος των Danvy και Filinski

Ένας μετασχηματισμός CPS που βρίσκεται σε ένα καλό σημείο ισορροπίας και καλύπτει ταυτόχρονα πολλές διαφορετικές απαιτήσεις είναι αυτός που προτείνουν οι Olivier Danvy και Andrzej Filinski [Danv92]. Πιο συγκεκριμένα, ο συγκεκριμένος μετασχηματισμός CPS πραγματοποιείται σε ένα πέρασμα και δεν παράγει περιττά β-redexes. Τα redexes που εισάγονται από τη μετάφραση ανάγονται καθοδόν, παράλληλα με την κατασκευή του όρου CPS, με αποτέλεσμα να μη χρειάζεται να κατασκευαστεί ποτέ όρος που να χρειάζεται post-reductions. Από την άλλη, δεν πραγματοποιείται καμία αναγωγή που αντιστοιχεί σε αναγωγή του αρχικού προγράμματος.

Ο μετασχηματισμός βασίζεται στο μετασχηματισμό CPS των Fischer και Plotkin (Σχήμα 4.1). Ο μετασχηματισμός των Fischer και Plotkin εισάγει "administrative" redexes τα οποία πρέπει να εξαλειφθούν σε ένα δεύτερο πέρασμα. Οι Danvy και Filinski τον τροποποιούν προσθέτοντας η-redexes με σκοπό να χωρίσουν το μετασχηματισμό σε δύο διακριτά τμήματα: το "translation-time" τμήμα και το "run-time" τμήμα. Όσα redexes ανήκουν στο "translation-time" τμήμα μπορούν να απλοποιηθούν απευθείας. Ο μετασχηματισμός που προκύπτει φαίνεται στο Σχήμα 4.4. Οι αφαιρέσεις και οι εφαρμογές που είναι μέρος του μετασχηματισμένου όρου είναι υπογραμμισμένες (χαρακτηρίζονται ως "dynamic"). Η γραμμή πάνω από σύβλολο υποδηλώνει αφαίρεση ή εφαρμογή που συνθέτει κάποιο translation-time redex και επομένως θα εξαλειφθεί κατά τη μετάφραση (χαρακτηρίζονται ως "static")

Ορισμός 4.2.1 *Ένα static continuation κ λέγεται* schematic *αν για οποιουσδήποτε όρους M και N και μεταβλητή x που δεν εμφανίζεται ελεύθερη στο κ,*

$$(@\, \kappa\, M)[x \leftarrow N] = @\, \kappa\, (M[x \leftarrow N])$$

Ο παραπάνω ορισμός ουσιαστικά διασφαλίζει ότι ένα schematic continuation διατηρεί τη συντακτική δομή του ορίσματός του και δε δεσμεύει ελεύθερες μεταβλητές που τυχόν εμφανίζονται στο όρισμα αυτό.

Ένα σημαντικό λήμμα, μεταξύ άλλων, που αποδεικνύεται και είναι σχετικό με τις ιδιότητες της αναγωγής των όρων CPS είναι το ακόλουθο: [Danv92]:

$$
\begin{aligned}
[\![x]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,\kappa\,x \\
[\![\lambda\,x.\,M]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,\kappa\,(\underline{\lambda}\,x.\,\underline{\lambda}\,k.\,\overline{@}\,[\![M]\!]\,(\overline{\lambda}\,m.\,\underline{@}\,k\,m)) \\
[\![@\,M\,N]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,[\![M]\!]\,(\overline{\lambda}\,m.\,\overline{@}\,[\![N]\!]\,(\overline{\lambda}\,n.\,\underline{@}\,(\underline{@}\,m\,n)\,(\underline{\lambda}\,\alpha.\,\overline{@}\,\kappa\,\alpha))) \\
[\![P \to M,\ N]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,[\![P]\!]\,(\overline{\lambda}\,p.\,p\underline{\to}\overline{@}\,[\![M]\!]\,\kappa\underline{,}\,\overline{@}\,[\![N]\!]\,\kappa) \\
[\![q]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,\kappa\,q \\
[\![q(M)]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,[\![M]\!]\,(\overline{\lambda}\,m.\,\overline{@}\,\kappa\,(\underline{q}(m))) \\
[\![q(M,N)]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,[\![M]\!]\,(\overline{\lambda}\,m.\,\overline{@}\,[\![N]\!]\,(\overline{\lambda}\,n.\,\overline{@}\,\kappa\,(\underline{q}(m,n)))) \\
[\![\texttt{let}\,x\,=\,N\,\texttt{in}\,M]\!] &= \overline{\lambda}\,\kappa.\,\overline{@}\,[\![N]\!]\,(\overline{\lambda}\,n.\,\underline{\texttt{let}}\,x'\,=\,n\,\underline{\texttt{in}}\,\overline{@}\,[\![M[x \leftarrow x']]\!]\,\kappa) \\
[\![\texttt{letrec}\,f\,=\,\lambda\,x.\,N\,\texttt{in}\,M]\!] &= \overline{\lambda}\,\kappa.\,\underline{\texttt{letrec}}\,f'\,=\,\underline{\lambda}\,x.\,\underline{\lambda}\,k.\,\overline{@}\,[\![N[f \leftarrow f']]\!]\,(\overline{\lambda}\,n.\,\underline{@}\,k\,n)\,\underline{\texttt{in}}\,\overline{@}\,[\![M[f \leftarrow f']]\!]\,\kappa
\end{aligned}
$$

Σχήμα 4.5: Επέκταση μετασχηματισμού CPS των Danvy και Filinski

$$
\begin{aligned}
[\![x : \tau]\!] &= \overline{\lambda}\,\kappa : \tau' \to o.\,\overline{@}\,\kappa\,x \\
[\![(\lambda\,x : \tau_1.\,M) : \tau_1 \to \tau_2]\!] &= \overline{\lambda}\,\kappa : (\tau_1' \to (\tau_2' \to o) \to o) \to o.\,\overline{@}\,\kappa\,(\underline{\lambda}\,x : \tau_1'.\,\underline{\lambda}\,k : \tau_2' \to o. \\
&\qquad \overline{@}\,[\![M]\!]\,(\overline{\lambda}\,m : \tau_2'.\,\underline{@}\,k\,m)) \\
[\![@\,(M : \tau_1 \to \tau_2)\,(N : \tau_1)]\!] &= \overline{\lambda}\,\kappa : \tau_2' \to o.\,\overline{@}\,[\![M]\!]\,(\overline{\lambda}\,m : \tau_1' \to (\tau_2' \to o) \to o. \\
&\qquad \overline{@}\,[\![N]\!]\,(\overline{\lambda}\,n : \tau_1'.\,\underline{@}\,(\underline{@}\,m\,n)\,(\underline{\lambda}\,\alpha : \tau_2'.\,\overline{@}\,\kappa\,\alpha)))
\end{aligned}
$$

Σχήμα 4.6: Μετασχηματισμός CPS των Danvy and Filinski για λ-calculus με απλούς τύπους

Λήμμα 4.2.1 *Αν τα M και είναι όροι τέτοιοι ώστε $M \to_v N$ και το κ είναι ένα schematic continuation, τότε*

$$\overline{@}\,[\![M]\!]\,\kappa \to_\alpha^+ \overline{@}\,[\![N]\!]\,\kappa$$

Μέσω αυτού αποδεικνύεται το επόμενο θεώρημα, το οποίο είναι ανάλογο των θεωρημάτων *Indifference* και *Simulation* του Plotkin:

Θεώρημα 4.2.3 *Έστω M ένας λ-όρος (όχι απαραίτητα κλειστός) και V μία τιμή. Αν $M \to_v^* V$, τότε*

$$\overline{@}\,[\![M]\!]\,(\overline{\lambda}\,x.\,x) \to_\alpha^* \overline{@}\,[\![V]\!]\,(\overline{\lambda}\,x.\,x) = \Psi(V)$$

Οι ίδιοι οι Danvy και Filinski προτείνουν διάφορες τροποποιήσεις και επεκτάσεις του μετασχηματισμού CPS κατασκεύασαν. Για παράδειγμα, μία τροποποίηση αφορά στο χειρισμό των tail-calls του αρχικού όρου ώστε να μην παράγονται περιττά η-redexes στον μεταφρασμένο όρο. Μία αναμενόμενη επέκταση του μετασχηματισμού CPS είναι η προσθήκη σταθερών, στοιχειωδών τελεστών, conditional εκφράσεων, καθώς και `let` και `letrec` εκφράσεων στην αρχική γλώσσα, τα οποία συναντώνται σε συναρτησιακές γλώσσες όπως η ML και η Scheme. Το θεώρημα 4.2.3 εξακολουθεί να ισχύει για την επέκταση αυτή του μετασχηματισμού CPS, η οποία παρουσιάζεται στο Σχήμα 4.5. Ας σημειωθεί ότι τελεστές που έχουν side-effects πρέπει αντικαθίστανται από τις παραλλαγές τους σε CPS.

Η επέκταση του μετασχηματισμού CPS των Danvy και Filinski για λ-calculus με απλούς τύπους μπορεί να γίνει με τον τρόπο που οι Albert Meyer και Mitchell Wand εφάρμοσαν στην περίπτωση του μετασχηματισμού CPS του Plotkin. Η επέκταση αυτή του (βασικού) μετασχηματισμού CPS των Danvy και Filinski, η οποία φαίνεται στο Σχήμα 4.6, για κάθε όρο M τύπου τ και πάλι κατασκευάζει έναν όρο $[\![M]\!]$ τύπου $(\tau' \to o) \to o$ (αποδεικνύεται με επαγωγή στο μήκος των όρων). Οι β-αναγωγές που πραγματοποιούνται κατά τη διάρκεια της μετάφρασης δεν επηρεάζουν αυτό το αποτέλεσμα.

Κεφάλαιο 5

Η γλώσσα LetbangCPS

Η γλώσσα LetbangCPS είναι ουσιαστικά ένα λ-calculus σε μορφή CPS με τύπους και ML-style αναφορές. Όπως και η Letbang, είναι πολυμορφική πάνω σε pretypes, υιοθετεί το *value restriction* και χρησιμοποιεί γραμμικό typing προσφέροντας τη δυνατότητα προσωρινής μετατροπής γραμμικών τιμών σε μη-περιορισμένες. Όπως έχει ήδη αναφερθεί, μπορεί κάποιος να δει την LetbangCPS σαν μια εκδοχή σε CPS της Letbang, αφού άλλωστε δανείζεται πολλά στοιχεία από αυτή, αλλά παρόλα αυτά πρόκειται για μια ανεξάρτητη γλώσσα.

Ο κατασκευή μιας γλώσσας της οποίας η σύνταξη της προσδίδει υποχρεωτικά μια μορφή CPS έχει πλεονεκτήματα σε σύγκριση με μια απλή επέκταση ενός μετασχηματισμού CPS ώστε αυτός να μεταφράζει επιπλέον και το στοιχείο `let!` της Letbang:

- Γίνεται διάκριση μεταξύ συναρτήσεων και continuations, οπότε η αναγνώριση ενός continuation δε βασίζεται στη θέση που αυτό εμφανίζεται. Αυτό υποστηρίζει ειδικό χειρισμό των continuations τόσο στο typing όσο και στη λειτουργική σημασιολογία. Αυτό συμβάλει στην εξαγωγή συμπερασμάτων για το πρόγραμμα και πιθανόν και στην προσθήκη νέων στοιχείων στη γλώσσα.
- Διευκολύνεται η εφαρμογή μετασχηματισμών πάνω στο αφηρημένο συντακτικό δένδρο ενός προγράμματος, ιδίως αν αυτοί σχετίζονται με το στοιχείο `let!`, το οποίο μπορεί να τοποθετηθεί σε συγκεκριμένες θέσεις με έναν μόνο τρόπο.
- Διευκολύνεται ο ορισμός του στοιχείου `let!` σε μορφή CPS.

5.1 Περιγραφή

Η γλώσσα LetbangCPS διατηρεί τις επισημειώσεις τύπων με scope και qualifier της Letbang ενώ η CPS μορφή του στοιχείου `let!` αποτελείται πλέον από δύο τμήματα:

1. Την έκφραση `at` η `let!` (x) `in` e , η οποία επιτελεί και το binding ενός νέου scope.
2. Το continuation $\lambda\, x : \tau .$ `unlet!` (x) `in` e.

Ο σωστός συνδυασμός τους διασφαλίζεται από τους κανόνες τύπων της γλώσσας (Ενότητα 5.3).

Στον όρο e του πρώτου τμήματος η μεταβλητή x χρησιμοποιείται ως μη-περιορισμένη, εκτός από τον υποόρο που αποτελεί το δεύτερο τμήμα του στοιχείου `let!`, όπου χρησιμοποείται ως γραμμική. Κατά τα άλλα, ο ρόλος των scopes είναι αυτός που έχουν και στην Letbang: Η μετατροπή του qualifier του τύπου του x συνοδεύεται από τη μετατροπή του scope του σε η. Η μεταβλητή scope η είναι δεσμευμένη μέσα στον όρο e εκτός από τον υποόρο που αποτελεί το δεύτερο τμήμα του στοιχείου `let!`, όπου και δεν μπορούν να χρησιμοποιηθούν τιμές με scope η αφού το scope αυτό δεν είναι έγκυρο.

Κάθε δομή let!, που αποτελείται εδώ από ένα ζεύγος στοιχείων της γλώσσας όπως αναφέρθηκε, έχει ένα δικό της μοναδικό scope. Για να διασφαλιστεί αυτό χωρίς να επιβαρυνθεί η γλώσσα με ελέγχους μοναδικότητας, οι κανόνες τύπων και η λειτουργική σημασιολογία της γλώσσας αντικαθιστούν τη μεταβλητή scope η με μία νέα σταθερά scope ρ.

Και πάλι κατά τα πρότυπα της Letbang, κάθε τύπος συνάρτησης επισημειώνεται με τη λίστα των scopes τα οποία το σώμα της συνάρτησης χρησιμοποιεί και προκειμένου να εφαρμοστεί επιτυχώς ο κανόνας τύπων της κλήσης συνάρτησης πρέπει όλα τα scopes της λίστας αυτής να είναι έγκυρα. Εδώ τα scopes που η συνάρτηση χρησιμοποιεί θεωρούνται αυτά που χρησιμοποιεί μέχρι την κλήση του continuation που της έχει δοθεί. Ακόμη, οι τελεστές deref, assign και swap είναι περιορισμένοι με τον ίδιο τρόπο όπως και στην Letbang. Το ίδιο ισχύει και για τους "bangable" όρους, που και εδώ δεν περιλαμβάνουν τιμές με τύπο συνάρτησης.

Τα continuations δεν επισημειώνονται με scope και qualifier. Οι περιορισμοί που έχουν να κάνουν με τα continuations επιβάλλονται από τη σύνταξη της γλώσσας και είναι οι εξής:

1. Ένα continuation καλείται ακριβώς μία φορά μέσα στο πρόγραμμα
2. Δεν μπορούν να δημιουργηθούν aliases ενός continuation
3. Μία συνάρτηση έχει πρόσβαση μόνο στο continuation που της έχει δοθεί κατά την κλήση της.
4. Στην κλήση μίας συνάρτησης απαιτείται πάντα να δίνεται πρώτο ένα όρισμα που δεν μπορεί να είναι continuation (συγκεκριμένα είναι μια μεταβλητή) και δεύτερο ένα continuation
5. Ένα continuation που δημιουργείται μέσα σε μία συνάρτηση περιέχει υποχρεωτικά το μοναδικό υπάρχον continuation στο οποίο η συνάρτηση έχει πρόσβαση, αυτό δηλαδή που της έχει δοθεί κατά την κλήση της

Ακολουθεί για σύγκριση το αντίστοιχο του Παραδείγματος 3.1.2 της Ενότητας 3.1 γραμμένο σε LetbangCPS.

Παράδειγμα 5.1.1

```
let r = new 7 in
  at η let! (r) in
    let x4 = ⊎⊥ λ u : Unit. cont(let x3 = deref r in ret x3) in  (* x4 :⊎⊥ (Unit, Int → ⊥) -η→ ⊥*)
      (λ y :⊎⊥ (Unit, Int → ⊥) -η→ ⊥. unlet! (r) in
        let x1 = free r in
        let x2 = unit in
          y x2 (λ a : Int. print a))                    (* ↯ application fails: η is not live *)
      x4
```

5.2 Σύνταξη

Η σύνταξη της LetbangCPS φαίνεται στο Σχήμα 5.1. Η σύνταξη αυτή επιβάλει ένα είδος μορφής CPS στη γλώσσα. Επιπλέον, όπως αναφέρθηκε στην Ενότητα 5.1, θέτει και περιορισμούς στη χρήση των continuations. Τα στοιχεία της γλώσσας, πέρα από τα continuations, είναι δανεισμένα από την Letbang και τροποποιημένα ώστε να έχουν μορφή CPS. Εδώ η συναρτήσεις δέχονται ένα "κρυμμένο" όρισμα, το οποίο αντιπροσωπεύται από το σύμβολο cont, που είναι υποχρεωτικά continuation. Όπως και στη Letbang, τα memory locations δεν είναι διαθέσιμα στον προγραμματιστή και χρησιμοποιούνται εσωτερικά από τη λειτουργική σημασιολογία της γλώσσας. Το στοιχείο unlet! δεν πραγματοποιεί binding νέου scope, στην πραγματικότητα πραγματοποιεί unbinding ενός υπάρχοντος scope.

$$
\begin{array}{rcl}
q & ::= & \mathsf{U} \mid \mathsf{L} \\
\pi & ::= & \eta \mid \rho \mid \bot \\
\phi & ::= & \alpha \mid \mathsf{Unit} \mid \mathsf{Base} \mid (\tau, \sigma) \xrightarrow{\vec{\pi}} \bot \mid \forall\, \alpha.\, \tau \mid \mathsf{Ref}\, \tau \\
\tau & ::= & \frac{q}{\pi}\phi \\
\sigma & ::= & \tau \to \bot \\
e & ::= & \mathtt{let}\, x = v\, \mathtt{in}\, e \mid \mathtt{let}\, x = y\, \mathtt{in}\, e \mid \mathtt{let}\, x = y\, [\phi]\, \mathtt{in} \mid \mathtt{let}\, x = \mathtt{new}\, y\, \mathtt{in}\, e \\
 & \mid & \mathtt{let}\, x = \mathtt{deref}\, y\, \mathtt{in}\, e \mid \mathtt{let}\, x = \mathtt{free}\, y\, \mathtt{in}\, e \mid \mathtt{let}\, x = y := z\, \mathtt{in}\, e \mid \mathtt{let}\, x = y :=: z\, \mathtt{in}\, e \\
 & \mid & c\, x \mid x\, y\, c \mid \mathtt{at}\, \eta\, \mathtt{let!}\, (x)\, \mathtt{in}\, e \\
v & ::= & \frac{q}{\pi}\lambda\, x : \tau.\, \underline{\mathtt{cont}}(e) \mid \frac{q}{\pi}\Lambda\, \alpha.\, v \mid \frac{q}{\pi}\mathtt{unit} \mid \frac{q}{\pi}\mathtt{loc}\, l \\
c & ::= & \underline{\mathtt{ret}} \mid \lambda\, x : \tau.\, e \mid \lambda\, x : \tau.\, \mathtt{unlet!}\, (y)\, \mathtt{in}\, e
\end{array}
$$

Σχήμα 5.1: Σύνταξη

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

$$\frac{\mathsf{qual}(\Gamma_1 \cap \Gamma_2) \;=\; \mathsf{U}}{\Gamma_1 \oplus \Gamma_2 = \Gamma_1 \cup \Gamma_2}$$

$$\boxed{Z \models \pi}$$

$$\{\} \models \bot \quad \{\rho\} \models \rho$$

$$\boxed{\mathsf{qual}(\Gamma)}$$

$$\frac{\mathsf{qual}(\Gamma) \;=\; \mathsf{U} \qquad \mathsf{qual}(\tau) \;=\; q}{\mathsf{qual}(\Gamma, (x : \tau)) \;=\; q} \qquad \frac{\mathsf{qual}(\Gamma) \;=\; \mathsf{L}}{\mathsf{qual}(\Gamma, (x : \tau)) \;=\; \mathsf{L}}$$

Σχήμα 5.2: Βοηθητικοί ορισμοί

Δεν απαιτείται να παρέχουμε κάποια σταθερά scope. Το ζητούμενο scope βρίσκεται πάντα από τον τύπο της μεταβλητής x την οποία επιχειρούμε να μετατρέψουμε από μη-περιορισμένη σε γραμμική.

Οι όροι της γλώσσας χωρίζονται σε τιμές (v), εκφράσεις (e) και continuations (c). Οι τύποι χωρίζονται σε τύπους τιμών και τύπους continuations. Οι εκφράσεις δεν έχουν τύπο (καταχρηστικά, θα μπορούσαμε να θεωρήσουμε ότι έχουν τύπο $\bot$). Οι τύποι των τιμών, κατά τα πρότυπα της Letbang, αποτελούνται από ένα pretype επισημειωμένο με ένα qualifer (q) και με ένα scope (π).

5.3 Κανόνες Τύπων

Η γλώσσα χρησιμοποιεί τρεις σχέσεις τυποποίησης για τιμές, εκφράσεις και continuations, αντίστοιχα,

$$
\begin{array}{rcl}
\Gamma; \Delta; \mathrm{M}; \mathrm{Z} & \vdash & v : \tau \\
\Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 & \vdash & e \\
\Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 & \vdash & c : \sigma
\end{array}
$$

οι οποίες ορίζονται στο Σχήμα 5.3. Το Γ είναι ένα περιβάλλον που δένει μεταβλητές με τύπους τιμών (τ), το Δ είναι ένα σύνολο από μεταβλητές pretype, το M δένει locations με τύπους τιμών (τ), τα Z και H είναι σύνολα από έγκυρα scopes, και το σ_0 είναι ο τύπος του continuation $\underline{\mathtt{ret}}$.

Όπως και στην Letbang, μπορούμε να έχουμε μόνο τιμές που να είναι γραμμικές (τίποτε άλλο δεν μπορεί να είναι γραμμικό). Συνεπώς, το περιβάλλον Γ υπόκειται στον ίδιο ειδικό χειρισμό όπως και στην

Letbang. Στο Σχήμα 5.2 ορίζεται ο τελεστής ένωσης $\Gamma_1 \oplus \Gamma_2$ για περιβάλλοντα, ο οποίος εφαρμόζεται έγκυρα μόνο εάν η τομή των Γ_1 και Γ_2 δεν περιέχει τιμές με γραμμικό τύπο. Η απόρριψη γραμμικών τιμών αποτρέπεται με τον περιορισμό του Γ στους κανόνες τύπων των βασικών περιπτώσεων ώστε να μην περιέχει καμία μεταβλητή με γραμμικό τύπο.

Το περιβάλλον Z έχει τον ίδιο ρόλο με αυτόν που έχει και στη Letbang. Και εδώ απαιτείται σχετικός χειρισμός των scopes έτσι ώστε να περιλαμβάνουμε μόνο τα scopes που πρέπει να έγκυρα στο σημείο εφαρμογής μιας συνάρτησης στη λίστα των scopes με την οποία επισημειώνεται ο τύπος της συνάρτησης. Για το σκοπό αυτό χρησιμοποιούμε και πάλι τη σχέση *minimal* scope μεταξύ των scopes π και των περιβαλλόντων scope Z (Σχήμα 5.2), η οποία επιπλέον διασφαλίζει ότι οι τύποι μπορούν να είναι επισημειωμένοι μόνο με σταθερές scope (ρ) ή με το προκαθορισμένο scope ($\bot$) αλλά όχι με μεταβλητές scopes.

Όπως αναφέρθηκε, η ερμηνεία που έχει μία λίστα από scopes ως επισημείωση του τύπου μίας συνάρτησης είναι διαφορετική από αυτή που έχει στη Letbang: Εδώ μία συνάρτηση επισημειώνεται με την λίστα των scopes που αυτή χρησιμοποιεί μέχρι την κλήση του continuation που της έχει δοθεί κατά την κλήση της. Αυτό αντικατοπτρίζεται και στον κανόνα τύπων του `ret`, το οποίο τυποποιείται με κενό περιβάλλον scope Z.

Το περιβάλλον H έχει το ρόλο να διασφαλίζει ότι κάθε νέο scope που εισάγεται από ένα `let!` θα κλείσει σε κάποιον υποόρο του με κάποιο `unlet!`. Η σχέση τυποποίησης των τιμών δεν περιλαμβάνει το περιβάλλον H, οπότε όλα τα scopes που εισάγονται "έξω" από κάποια συνάρτηση δεν μπορούν να κλείσουν στο σώμα αυτής. Αυτό συμφωνεί και με τη διαίσθηση που έχουμε για την ασφάλεια της μνήμης: Ένα scope που εισάγεται με την έκφραση

$$\texttt{at}\ \eta\ \texttt{let!}\ (x)\ \texttt{in}\ e$$

μπορεί να κλείσει μόνο στο μοναδικό continuation που υποχρεωτικά εμφανίζεται σε κάποια υποέκφραση της έκφρασης e. Επιπλέον, ο κανόνας τύπων για το continuation `ret` απαιτεί κενό περιβάλλον H, το οποίο διασφαλίζει ότι το scope θα κλείσει υποχρεωτικά.

Όπως και στη Letbang, οι σταθερές scope ρ πρέπει να είναι μοναδικές, να εισάγονται δηλαδή μόνο μία φορά, και να μην μπορούν να διαφύγουν από το σώμα του αντίστοιχου `let!`. Στους κανόνες τύπων του Σχήματος 5.3 για λόγους ευκρίνειας δεν αναγράφεται η προϋπόθεση

$$\forall \rho . \rho \in \mathsf{FREE}_{\mathsf{raw}}(e) \Rightarrow \rho \notin \mathrm{Z}$$

που υποδηλώνει την ισχύ της παραπάνω πρότασης.

Ας σημειωθεί ότι τα continuations δεν είναι πολυμορφικά. Καθώς η σύνταξη της γλώσσας επιβάλλει ότι κάθε continuation χρησιμοποιείται ακριβώς μία φορά, η προσθήκη πολυμορφισμού στα continuations δε θα ενίσχυε τη γλώσσα, τουλάχιστον στη μορφή της που παρουσιάζεται σε αυτό το κεφάλαιο, ενώ θα προσέθετε πολυπλοκότητα στο typing και στη μεταθεωρία της.

5.4 Λειτουργική Σημασιολογία

Στο Σχήμα 5.4 ορίζεται call-by-value λειτουργική σημασιολογία για τη γλώσσα LetbangCPS. Η λειτουργική σημασιολογία είναι μια σχέση ανάμεσα σε διαρρυθμίσεις (configurations) που αποτελούνται από ένα store S, που είναι μια αντιστοιχία μεταξύ μεταβλητών και τιμών, μια μνήμη μ, που είναι μια αντιστοιχία μεταξύ locations και μεταβλητών, και μία έφραση e της γλώσσας.

Όπως και στην Letbang, κι εδώ χρησιμοποιούνται auto-variables (για τις οποίες δεν υπάρχει διάκριση σε σχέση με τις μεταβλητές του προγράμματος). Ο κανόνας αποτίμησης για την έκφραση

$\texttt{let}\, x \;=\; v\, \texttt{in}\, e$ διασφαλίζει ότι κάθε τιμή δένεται σε μια νέα μεταβλητή z και αποθηκεύεται στο store S. Πρόσβαση σε τιμές μπορεί να δοθεί μόνο μέσω των μεταβλητών στις οποίες αυτές είναι δεμένες. Τα continuations, αντίθετα, δε δένονται ποτέ σε μεταβλητές και συνεπώς δεν εισάγονται στο store S, αφού άλλωστε δεν τα χειριζόμαστε με γραμμικό τρόπο.

Δεν ορίζεται κανόνας λειτουργικής σημασιολογίας για την εφαρμογή του continuation $\underline{\texttt{ret}}$ πάνω σε μεταβλητή. Κατά την εφαρμογή των κανόνων του Σχήματος 5.4 μια τέτοια έκφραση μπορεί να συναντηθεί μόνο αν το $\underline{\texttt{ret}}$ είναι το top-level continuation. Στην περίπτωση αυτή θεωρούμε ότι τερματίζεται και η εκτέλεση του προγράμματος.

Η συνάρτηση $S; x \Downarrow S'; v$, η οποία αναζητά τη μεταβλητή με όνομα x στο store S, είναι εκείνη που χρησιμοποιείται και στη λειτουργική σημασιολογία της Letbang. Ο ορισμός της έχει ήδη δοθεί (Σχήμα 3.5) και παρουσιάζεται ξανά στο Σχήμα 5.5 για διευκόλυνση και πληρότητα. Μια μεταβλητή που είναι δεμένη με γραμμική τιμή αφαιρείται από το store S' που επιστρέφεται διασφαλίζοντας ότι οι γραμμικές τιμές μπορούν να χρησιμοποιηθούν από το πρόγραμμα μόνο μία φορά. Μεταβλητές δεμένες με μη-περιορισμένες τιμές δεν αφαιρούνται από το store και συνεπώς μη-περιορισμένες τιμές μπορούν να χρησιμοποιηθούν οσεσδήποτε φορές.

Η μνήμη μ της σημασιολογίας της γλώσσας είναι η ίδια με εκείνη που χρησιμοποιείται στη λειτουργική σημασιολογία της Letbang τόσο ως προς το ρόλο όσο και ως προς τη λειτουργία της. Χρησιμοποιείται δηλαδή για τον (γραμμικό) χειρισμό των περιεχομένων των αναφορών και αντιστοιχίζει locations σε μεταβλητές.

$\boxed{\Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 \vdash c : \sigma}$

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

$\boxed{\Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 \vdash e}$

$$\frac{\mathsf{qual}(\Gamma) = q \quad \mathrm{Z} \models \pi \quad \tau =^q_\pi \phi \quad \Delta \models \phi \quad \Gamma, x : \tau; \Delta; \mathrm{M}; \emptyset; \mathrm{Z}_e; \sigma \vdash e}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash {}^q_\pi \lambda\, x : \tau.\, \underline{\mathtt{cont}}(e) : {}^q_\pi(\tau, \sigma) \xrightarrow{\mathrm{Z}_e} \bot} \text{(T-abs)} \qquad \frac{\mathsf{qual}(\Gamma) = q \quad \mathrm{Z} \models \pi \quad \Gamma; \Delta, \alpha; \mathrm{M}; \mathrm{Z} \vdash v : \tau}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash {}^q_\pi \Lambda\, \alpha.\, v : {}^q_\pi \forall\, \alpha.\, \tau} \text{(T-abs-type)}$$

$$\frac{\mathsf{qual}(\Gamma) = \mathsf{U} \quad \mathrm{Z} \models \pi}{\Gamma; \Delta; \mathrm{M}; \mathrm{Z} \vdash {}^q_\pi \mathtt{unit} : {}^q_\pi \mathsf{Unit}} \text{(T-unit)} \qquad \frac{\mathsf{qual}(\Gamma) = \mathsf{U} \quad \mathrm{Z} \models \pi}{\Gamma; \Delta; \mathrm{M}, l : \tau; \mathrm{Z} \vdash {}^q_\pi \mathtt{loc}\ l : {}^q_\pi \mathsf{Ref}\, \tau} \text{(T-loc)}$$

$$\frac{\Gamma, x : \tau; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 \vdash e}{\Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 \vdash \lambda\, x : \tau.\, e : \tau \to \bot} \text{(T-cont-lam)} \qquad \frac{\mathsf{qual}(\Gamma) = \mathsf{U} \quad \mathrm{Z} \models \bot}{\Gamma; \Delta; \mathrm{M}; \emptyset; \mathrm{Z}; \sigma \vdash \underline{\mathtt{ret}} : \sigma} \text{(T-cont-ret)}$$

$$\frac{\Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1; \sigma_0 \vdash c : \tau \to \bot \quad \mathrm{Z}_2 \models \mathsf{scope}\, \tau}{\Gamma \oplus \{x : \tau\}; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2; \sigma_0 \vdash c\, x} \text{(T-return)}$$

$$\frac{\mathrm{Z}_1 \models \pi \quad \mathrm{Z}_2 \models \mathsf{scope}\, \tau_1 \quad \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_3; \sigma_0 \vdash c : \sigma}{\{x : {}^q_\pi(\tau_1, \sigma) \xrightarrow{\mathrm{Z}_e} \bot\} \oplus \{y : \tau_1\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \cup \mathrm{Z}_3 \cup \mathrm{Z}_e; \sigma_0 \vdash x\, y\, c} \text{(T-call)}$$

$$\frac{\Gamma, x : {}^{\mathsf{U}}_\rho \phi; \Delta; \mathrm{M}; \mathrm{H}, \rho; \mathrm{Z}, \rho; \sigma_0 \vdash e\{\eta \mapsto \rho\} \quad \mathsf{bangable}\, \phi \quad \mathsf{fresh}\, \rho}{\Gamma, x : {}^{\mathsf{L}}_\bot \phi; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 \vdash \mathtt{at}\, \eta\, \mathtt{let!}\, (x)\, \mathtt{in}\, e} \text{(T-let!)}$$

$$\frac{\Gamma, x : {}^{\mathsf{L}}_\bot \phi, x_1 : \tau_1; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 \vdash e \quad \rho \notin \mathrm{Z} \quad \mathsf{bangable}\, \phi}{\Gamma, x : {}^{\mathsf{U}}_\rho \phi; \Delta; \mathrm{M}; \mathrm{H}, \rho; \mathrm{Z}, \rho; \sigma_0 \vdash \lambda\, x_1 : \tau_1.\, \mathtt{unlet!}\, (x)\, \mathtt{in}\, e : \tau_1 \to \bot} \text{(T-cont-unlet!)}$$

$$\frac{\Gamma_1; \Delta; \mathrm{M}; \mathrm{Z}_1 \vdash v : \tau \quad \Gamma_2, x : \tau; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_2; \sigma_0 \vdash e}{\Gamma_1 \oplus \Gamma_2; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2; \sigma_0 \vdash \mathtt{let}\, x = v\, \mathtt{in}\, e} \text{(T-val)}$$

$$\frac{\mathrm{Z}_1 \models \mathsf{scope}\, \tau \quad \Gamma, x : \tau; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_2; \sigma_0 \vdash e}{\{y : \tau\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2; \sigma_0 \vdash \mathtt{let}\, x = y\, \mathtt{in}\, e} \text{(T-var)}$$

$$\frac{\mathrm{Z}_1 \models \pi \quad \mathrm{Z}_\tau \models \mathsf{scope}\, \tau \quad \Delta \models \phi \quad \Gamma, x : \tau\{\alpha \mapsto \phi\}; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_2; \sigma_0 \vdash e}{\{y : {}^q_\pi \forall\, \alpha.\, \tau\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \cup \mathrm{Z}_\tau; \sigma_0 \vdash \mathtt{let}\, x = y\, [\phi]\, \mathtt{in}\, e} \text{(T-app-type)}$$

$$\frac{\mathrm{Z}_1 \models \mathsf{scope}\, \tau \quad \Gamma, x : {}^{\mathsf{L}}_\bot \mathsf{Ref}\, \tau; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_2; \sigma_0 \vdash e}{\{y : \tau\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2; \sigma_0 \vdash \mathtt{let}\, x = \mathtt{new}\, y\, \mathtt{in}\, e} \text{(T-new)}$$

$$\frac{\mathrm{Z}_\tau \models \mathsf{scope}\, \tau \quad \Gamma, x : \tau; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}; \sigma_0 \vdash e}{\{y : {}^{\mathsf{L}}_\bot \mathsf{Ref}\, \tau\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z} \cup \mathrm{Z}_\tau; \sigma_0 \vdash \mathtt{let}\, x = \mathtt{free}\, y\, \mathtt{in}\, e} \text{(T-free)}$$

$$\frac{\mathrm{Z}_1 \models \rho \quad \mathsf{qual}(\tau) = \mathsf{U} \quad \mathrm{Z}_\tau \models \mathsf{scope}\, \tau \quad \Gamma; x : \tau; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_2; \sigma_0 \vdash e}{\{y : {}^{\mathsf{U}}_\rho \mathsf{Ref}\, \tau\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \cup \mathrm{Z}_\tau; \sigma_0 \vdash \mathtt{let}\, x = \mathtt{deref}\, y\, \mathtt{in}\, e} \text{(T-deref)}$$

$$\frac{\mathrm{Z}_1 \models \rho \quad \mathrm{Z}_2 \models \mathsf{scope}\, \tau \quad \mathsf{qual}(\tau) = \mathsf{U} \quad \Gamma; x : {}^{\mathsf{U}}_\bot \mathsf{Unit}; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_3; \sigma \vdash e}{\{y : {}^{\mathsf{U}}_\rho \mathsf{Ref}\, \tau\} \oplus \{z : \tau\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \cup \mathrm{Z}_3; \sigma \vdash \mathtt{let}\, x = y := z\, \mathtt{in}\, e} \text{(T-assign)}$$

$$\frac{\mathrm{Z}_1 \models \rho \quad \mathrm{Z}_2 \models \mathsf{scope}\, \tau \quad \mathsf{qual}(\tau) = \mathsf{L} \quad \Gamma; x : \tau; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_3; \sigma \vdash e}{\{y : {}^{\mathsf{U}}_\rho \mathsf{Ref}\, \tau\} \oplus \{z : \tau\} \oplus \Gamma; \Delta; \mathrm{M}; \mathrm{H}; \mathrm{Z}_1 \cup \mathrm{Z}_2 \cup \mathrm{Z}_3; \sigma \vdash \mathtt{let}\, x = y :=: z\, \mathtt{in}\, e} \text{(T-swap)}$$

Σχήμα 5.3: Κανόνες Τύπων

$\boxed{S; \mu; e \hookrightarrow S'; \mu'; e'}$

$$\frac{\mathsf{fresh}\ z}{S; \mu; \mathtt{let}\ x\ =\ v\ \mathtt{in}\ e \hookrightarrow S, z \mapsto v; \mu; e\{x \mapsto z\}} \qquad \frac{}{S; \mu; \mathtt{let}\ x\ =\ y\ \mathtt{in}\ e \hookrightarrow S; \mu; e\{x \mapsto y\}}$$

$$\frac{S; y \Downarrow S';^{q}_{\pi}\ \Lambda\,\alpha.\,v \qquad \mathsf{fresh}\ z}{S; \mu; \mathtt{let}\ x\ =\ y\,[\phi]\ \mathtt{in}\ e \hookrightarrow S', z \mapsto v\{\alpha \mapsto \phi\}; \mu; e\{x \mapsto z\}}$$

$$\frac{\mathsf{fresh}\ z \qquad \mathsf{fresh}\ l}{S; \mu; \mathtt{let}\ x\ =\ \mathtt{new}\ y\ \mathtt{in}\ e \hookrightarrow S, z \mapsto^{\mathsf{L}}_{\bot}\ \mathtt{loc}\ l; \mu, l \mapsto y; e\{x \mapsto z\}}$$

$$\frac{S; y \Downarrow S';^{q}_{\pi}\ \mathtt{loc}\ l}{S; \mu, l \mapsto z; \mathtt{let}\ x\ =\ \mathtt{deref}\ y\ \mathtt{in}\ e \hookrightarrow S'; \mu, l \mapsto z; e\{x \mapsto z\}}$$

$$\frac{S; y \Downarrow S';^{q}_{\pi}\ \mathtt{loc}\ l}{S; \mu, l \mapsto z; \mathtt{let}\ x\ =\ \mathtt{free}\ y\ \mathtt{in}\ e \hookrightarrow S'; \mu; e\{x \mapsto z\}}$$

$$\frac{S; y \Downarrow S';^{q}_{\pi}\ \mathtt{loc}\ l}{S; \mu, l \mapsto z; \mathtt{let}\ x\ =\ y := w\ \mathtt{in}\ e \hookrightarrow S'; \mu, l \mapsto w; e\{x \mapsto^{\mathsf{U}}_{\bot}\ \mathtt{unit}\}}$$

$$\frac{S; y \Downarrow S';^{q}_{\pi}\ \mathtt{loc}\ l}{S; \mu, l \mapsto z; \mathtt{let}\ x\ =\ y :=: w\ \mathtt{in}\ e \hookrightarrow S'; \mu, l \mapsto w; e\{x \mapsto z\}}$$

$$\frac{\mathsf{fresh}\ x' \qquad \mathsf{fresh}\ \rho \qquad \mathsf{qual}(v)\ =\ \mathsf{L} \qquad \mathsf{scope}\ v = \bot \qquad v' = v, \text{ with scope } \rho \text{ and qual } \mathsf{U}}{S, x \mapsto v; \mu; \mathtt{at}\ \eta\ \mathtt{let!}\ (x)\ \mathtt{in}\ e \hookrightarrow S, x' \mapsto v'; \mu; e\{\eta \mapsto \rho\}\{x \mapsto x'\}}$$

$$\frac{S; w \Downarrow S';^{q}_{\pi}\ \lambda\,x : \tau.\,\underline{\mathtt{cont}}(e)}{S; \mu; w\ y\ c \hookrightarrow S'; \mu; e\{x \mapsto y\}\{\underline{\mathtt{ret}} \mapsto c\}} \qquad \frac{}{S; \mu; (\lambda\,x : \tau.\,e)\ y \hookrightarrow S; \mu; e\{x \mapsto y\}}$$

$$\frac{\mathsf{fresh}\ y' \qquad \mathsf{qual}(v)\ =\ \mathsf{U} \qquad \mathsf{scope}\ v = \rho \qquad v' = v, \text{ with scope } \bot \text{ and qual } \mathsf{L}}{S, y \mapsto v; \mu; (\lambda\,x : \tau.\,\mathtt{unlet!}\ (y)\ \mathtt{in}\ e)\ w \hookrightarrow S, y' \mapsto v'; \mu; e\{y \mapsto y'\}\{x \mapsto w\}}$$

Σχήμα 5.4: Κανόνες λειτουργικής σημασιολογίας

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

$$\frac{\mathsf{qual}(v)\ =\ \mathsf{U}}{S, x \mapsto v; x \Downarrow S, x \mapsto v; v} \qquad \frac{\mathsf{qual}(v)\ =\ \mathsf{L}}{S, x \mapsto v; x \Downarrow S; v}$$

Σχήμα 5.5: Store lookup

Κεφάλαιο 6

Μετασχηματισμός CPS για τη γλώσσα Letbang

Ο μετασχηματισμός CPS που ορίζεται σε αυτό το κεφάλαιο είναι ένας μετασχηματισμός προγραμμάτων Letbang σε προγράμματα LetbangCPS. Πρόκειται για ένα συμπληρωματικό εργαλείο που αναπτύχθηκε παράλληλα με τη γλώσσα LetbangCPS, η οποία ορίστηκε στο Κεφάλαιο 5. Συνέβαλε αφενώς στον έλεγχο των σχεδιαστικών επιλογών της γλώσσας, αφετέρου αναδεικνύει τη χρησιμότητά της ως μια ενδιάμεση γλώσσα στη μεταγλώττιση της Letbang.

Ο μετασχηματισμός CPS βασίζεται στον επεκτεταμένο μετασχηματισμό CPS των Danvy και Filinski που παρουσιάστηκε στην Υποενότητα 4.2.2 (Σχήμα 4.5) και δεν παράγει περιττά β-redexes. Ως προς το μετασχηματισμό των τύπων βασίζεται στην επέκταση που δίνουν οι Robert Harper και Mark Lillibridge [Harp93] στο μετασχηματισμό τύπων για λ-calculus με απλούς τύπους των Meyer και Wand που παρουσιάστηκε στην Υποενότητα 4.2.1. Ο μετασχηματισμός CPS των τύπων της Letbang ορίζεται στο Σχήμα 6.1 και ο μετασχηματισμός CPS των όρων της Letbang ορίζεται στο Σχήμα 6.2. Κάποιες επισημειώσεις τύπων παραλείπονται για λόγους ευκρίνειας, μπορούν πάντως εύκολα να βρεθούν από το type derivation της Letbang και το μετασχηματισμό τύπων του Σχήματος 6.1. Ο μετασχηματισμός των όρων χωρίζεται σε τρία μέρη: το μετασχηματισμό των τιμών της αρχικής γλώσσας (K_{val}), το μετασχηματισμό των εκφράσεων (K_{exp}) και ένα βοηθητικό μετασχηματισμό K_{aux}, ο οποίος χρησιμοποιείται για την εισαγωγή νέων ονομάτων μεταβλητών. Ο K_{val} δέχεται μία τιμή της αρχικής γλώσσας και επιστρέφει μία τιμή της τελικής γλώσσας. Ο K_{exp} δέχεται μία έκφραση της αρχικής γλώσσας, ένα όνομα μεταβλητής και μία έκφραση της τελικής γλώσσας και επιστρέφει μια έκφραση της τελικής γλώσσας. Ο K_{aux} δέχεται μία έκφραση της αρχικής γλώσσας και μία μετα-συνάρτηση από όνομα μεταβλητής σε έκφραση της τελικής γλώσσας και επιστρέφει μια έκφραση της τελικής γλώσσας.

$$
\begin{aligned}
K_{val} &: val_A \to val_T \\
K_{exp} &: exp_A \to var \to exp_T \to exp_T \\
K_{aux} &: exp_A \to (var \to exp_T) \to exp_T
\end{aligned}
$$

Οι μετα-συναρτήσεις συμβολίζονται με $\overline{\lambda}$ και η εφαρμογή τους, η οποία πραγματοποιείται κατά την κατασκευή του όρου CPS, συμβολίζεται με $\overline{@}$. Ο συμβολισμός αυτός κάνει εμφανέστερη και την αναλογία με το μετασχηματισμό CPS των Danvy και Filinski. Αντί του τύπου των "απαντήσεων" o των Meyer και Wand εδώ χρησιμοποιείται ο τύπος $\bot$.

Ο μετασχηματισμός CPS που ορίζεται στα Σχήματα 6.1 και 6.2 διατηρεί το typing και τη σημασιολογία. Ισχύουν δηλαδή τα δύο ακόλουθα θεωρήματα.

Θεώρημα 6.0.1 (Preserves typing) *Αν* $e : \tau$ *τότε*

$$K_{exp}[\![e]\!]\, a\, (\underline{\texttt{ret}}\, a) : K[\![\tau]\!]$$

όπου οι τυποποιήσεις γίνονται σε κενά περιβάλλοντα.

$$
\begin{aligned}
K[\![{}^{q}_{\pi}\tau]\!] &= {}^{q}_{\pi}(K[\![\tau]\!]) \\
K[\![\mathsf{Base}]\!] &= \mathsf{Base} \\
K[\![\mathsf{Unit}]\!] &= \mathsf{Unit} \\
K[\![\alpha]\!] &= \alpha \\
K[\![\mathsf{Ref}\,\tau]\!] &= \mathsf{Ref}\,(K[\![\tau]\!]) \\
K[\![\forall\, t.\, \tau]\!] &= \forall\, t.\, K[\![\tau]\!] \\
K[\![\tau_1 \xrightarrow{\vec{\pi}} \tau_2]\!] &= (K[\![\tau_1]\!], K[\![\tau_2]\!] \to \bot) \xrightarrow{\vec{\pi}} \bot
\end{aligned}
$$

Σχήμα 6.1: Μετασχηματισμός τύπων CPS για τη Letbang

$$
\begin{aligned}
K_{aux}[\![x]\!]\, k &= \overline{@}\, k\, x \\
K_{aux}[\![M]\!]\, k &= K_{exp}[\![M]\!]\, z\, (\overline{@}\, k\, z) \qquad \text{fresh } z
\end{aligned}
$$

$$
\begin{aligned}
K_{exp}[\![x]\!]\, z\, e_t &= \mathtt{let}\ z\ =\ x\ \mathtt{in}\ e_t \\
K_{exp}[\![v]\!]\, z\, e_t &= \mathtt{let}\ z\ =\ K_{val}[\![v]\!]\ \mathtt{in}\ e_t \\
K_{exp}[\![e_1\ e_2]\!]\, z\, e_t &= K_{aux}[\![e_1]\!]\, (\overline{\lambda}\, m.\, K_{aux}[\![e_2]\!]\, (\overline{\lambda}\, n.\, m\, n\, (\lambda\, z.\, e_t))) \\
K_{exp}[\![e\,[\phi]]\!]\, z\, e_t &= K_{aux}[\![e]\!]\, (\overline{\lambda}\, m.\, \mathtt{let}\ z\ =\ m\,[K[\![\phi]\!]]\ \mathtt{in}\ e_t) \\
K_{exp}[\![\mathtt{new}\ e]\!]\, z\, e_t &= K_{aux}[\![e]\!]\, (\overline{\lambda}\, m.\, \mathtt{let}\ z\ =\ \mathtt{new}\ m\ \mathtt{in}\ e_t) \\
K_{exp}[\![\mathtt{deref}\ e]\!]\, z\, e_t &= K_{aux}[\![e]\!]\, (\overline{\lambda}\, m.\, \mathtt{let}\ z\ =\ \mathtt{deref}\ m\ \mathtt{in}\ e_t) \\
K_{exp}[\![\mathtt{free}\ e]\!]\, z\, e_t &= K_{aux}[\![e]\!]\, (\overline{\lambda}\, m.\, \mathtt{let}\ z\ =\ \mathtt{free}\ m\ \mathtt{in}\ e_t) \\
K_{exp}[\![e_1 := e_2]\!]\, z\, e_t &= K_{aux}[\![e_1]\!]\, (\overline{\lambda}\, m.\, K_{aux}[\![e_2]\!]\, (\overline{\lambda}\, n.\, \mathtt{let}\ z\ =\ m := n\ \mathtt{in}\ e_t)) \\
K_{exp}[\![e_1 :=: e_2]\!]\, z\, e_t &= K_{aux}[\![e_1]\!]\, (\overline{\lambda}\, m.\, K_{aux}[\![e_2]\!]\, (\overline{\lambda}\, n.\, \mathtt{let}\ z\ =\ m :=: n\ \mathtt{in}\ e_t)) \\
K_{exp}[\![\mathtt{at}\ \eta\ \mathtt{let!}\,(x{=}e_1)\ y{=}e_2\ \mathtt{in}\ e_3]\!]\, z\, e_t &= \\
K_{exp}[\![e_1]\!]\, x'\, (\mathtt{at}\ \eta\ \mathtt{let!}\,(x')\ \mathtt{in}\ K_{aux}[\![e_2\{x \mapsto x'\}]\!]\, (\overline{\lambda}\, m.\, \lambda\, y'.\, \mathtt{unlet!}\,(x')\ \mathtt{in}\ K_{exp}[\![e_3\{y \mapsto y'\}\{x \mapsto x'\}]\!]\, z\, e_t))
\end{aligned}
$$

$$
\begin{aligned}
K_{val}[\![{}^{q}_{\pi}\mathtt{unit}]\!] &= {}^{q}_{\pi}\mathtt{unit} \\
K_{val}[\![{}^{q}_{\pi}\lambda\, x : \tau.\, M]\!] &= {}^{q}_{\pi}\lambda\, x : K[\![\tau]\!].\, \underline{\mathtt{cont}}(K_{aux}[\![M]\!]\, (\overline{\lambda}\, m.\, \underline{\mathtt{ret}}\, m)) \\
K_{val}[\![{}^{q}_{\pi}\Lambda\, \alpha.\, v]\!] &= {}^{q}_{\pi}\Lambda\, \alpha.\, K_{val}[\![v]\!]
\end{aligned}
$$

Σχήμα 6.2: Μετασχηματισμός CPS για τη Letbang

Θεώρημα 6.0.2 (Preserves semantics) *Αν* $S; \mu; e \hookrightarrow S'; \mu'; e'$ *τότε*

$$K_{store}[\![S]\!]; \mu; K_{exp}[\![e]\!]\, a\, (\underline{\mathtt{ret}}\, a) \hookrightarrow S''; \mu'; K_{exp}[\![e']\!]\, a\, (\underline{\mathtt{ret}}\, a)$$

όπου $K_{store}[\![S]\!]$ *είναι η εφαρμογή του* K_{val} *σε όλες τις τιμές ενός store* S *και* $S'' \supseteq K_{store}[\![S']\!]$.

Κεφάλαιο 7

Υλοποίηση

Παράλληλα με το σχεδιασμό της LetbangCPS και του μετασχηματισμού CPS που παρουσιάστηκε στο Κεφάλαιο 6 υλοποιήθηκε επίσης ένας typechecker για τη γλώσσα καθώς και ο μετασχηματισμός CPS. Ως front-end χρησιμοποιείται ένας parser για τη γλώσσα Letbang, ο οποίος τροφοδοτεί το μετασχηματιστή CPS με το AST του προγράμματος Letbang. Αυτός παράγει το AST του αντίστοιχου προγράμματος LetbangCPS το οποίο τελικά ελέγχει ο typechecker.

Ας σημειωθεί ότι η υλοποίηση του μετασχηματισμού CPS προκύπτει άμεσα από τον ορισμό του (Σχήμα 6.2). Οι μετα-συναρτήσεις ($\overline{\lambda}$) μπορούν να υλοποιηθούν ως συναρτήσεις της (συναρτησιακής) γλώσσας που χρησιμοποιείται για την υλοποίηση. Το ίδιο ισχύει και για τις μετα-εφαρμογές ($\overline{@}$), οι οποίες μπορούν να υλοποιηθούν ως εφαρμογές στο επίπεδο της γλώσσας υλοποίησης.

Η αντιπαραβολή των αποτελεσμάτων που πήραμε από το typechecking διαφόρων προγραμμάτων Letbang και από το typechecking των αντίστοιχων προγραμμάτων LetbangCPS, που προέκυψαν με την εφαρμογή του μετασχηματισμού CPS, ήταν ενθαρρυντική. Όπως ήταν το αναμενόμενο ένα πρόγραμμα LetbangCPS φάνηκε να περνά επιτυχώς τον έλεγχο τύπων μόνο όταν το αντίστοιχο πρόγραμμα Letbang επίσης περνά επιτυχώς τον έλεγχο τύπων και αντίστροφα. Αυτό είναι μια ένδειξη που ενισχύει την πεποίθηση ότι η LetbangCPS είναι συνεπής, αν και αυτό μένει ακόμα να αποδειχθεί τυπικά.

Κεφάλαιο 8

Συμπεράσματα

8.1 Συνεισφορά

Στην παρούσα διπλωματική εργασία ορίστηκε η γλώσσα LetbangCPS, μία γλώσσα που συνδυάζει αφενός ένα γραμμικό σύστημα τύπων με μεικτούς τύπους (γραμμικούς και μη-περιορισμένους) για αναφορές στο στυλ της ML, και αφετέρου μία μορφή κατάλληλη για το μετασχηματισμό περάσματος συνεχειών (CPS). Για τη γλώσσα αυτή υλοποιήθηκε ένας πειραματικός ελεγκτής τύπων και ένας διερμηνέας. Στη συνέχεια, ορίστηκε και υλοποιήθηκε ένας μετασχηματισμός CPS για τη γλώσσα Letbang, τέτοιος ώστε κάθε έγκυρο πρόγραμμα Letbang να μεταφράζεται σε ένα έγκυρο πρόγραμμα LetbangCPS.

Τα συστήματα τύπων των γλωσσών Letbang και LetbangCPS διαφέρουν σε μεγάλο βαθμό στην ερμηνεία που δίνουν στο περιβάλλον των scope (Z). Η ερμηνεία που δίνεται από τη γλώσσα LetbangCPS υποστηρίχθηκε και από την επιλογή του διαχωρισμού των συνεχειών από τις συναρτήσεις και του ξεχωριστού χειρισμού τους. Χωρίς αυτό το διαχωρισμό, οι επισημειώσεις των τύπων των συναρτήσεων (και άρα των συνεχειών) με περιβάλλοντα scope θα απέτρεπαν τα προγράμματα της LetbangCPS που προκύπτουν από έγκυρα προγράμματα Letbang από το να περάσουν τον έλεγχο τύπων, τουλάχιστον χωρίς την προσθήκη ενός νέου τύπου πολυμορφισμού πάνω σε περιβάλλοντα scope για τη LetbangCPS. Ακόμη, έγινε φανερή η δυσκολία που παρουσιάζει ο ορισμός της μορφής CPS της δομής `let!`, η οποία ορίζει δύο ξεχωριστές συντακτικές εμβέλειες και της οποίας η σημασιολογία δεν περιορίζεται μόνο σε αντικαταστάσεις, καθώς και η συμβολή του διαχωρισμού των συνεχειών από τις συναρτήσεις για την επίλυση και αυτού του προβλήματος.

Πέρα από τις παραπάνω παρατηρήσεις, το κύριο αποτέλεσμα της προσπάθειας αυτής, που είναι η γλώσσα LetbangCPS, μπορεί πιθανόν να βρει πρακτικές εφαρμογές (είτε ως έχει είτε κατόπιν επεκτάσεων) όπως αυτές που περιγράφονται στη συνέχεια.

8.1.1 Πιθανές χρήσεις της LetbangCPS

Η LetbangCPS είναι μια ανεξάρτητη γλώσσα παρόλη τη στενή της συγγένεια με τη Letbang. Συνεπώς μπορεί να βρει εφαρμογή είτε σε συνδυασμό με αυτήν είτε ανεξάρτητα. Είναι σημαντικό ότι πρόκειται για μια γλώσσα αρκετά χαμηλού επιπέδου σε μορφή CPS, με περιορισμένη σύνταξη και σχετικά απλή σημασιολογία. Τα χαρακτηριστικά της αυτά καταδεικνύουν και τις πιθανότερες χρήσεις της, μερικές από τις οποίες αναφέρονται παρακάτω:

– Μπορεί να χρησιμοποιηθεί για να κάνουμε μια συντηρητική και αυτοματοποιημένη διαχείριση μνήμης χωρίς τον garbage collector. Αν δηλαδή έχουμε ένα πρόγραμμα LetbangCPS όπου λοίπουν τα `free`, πιθανόν να μπορούμε εύκολα να τα προσθέσουμε. Ένα κανονικό πρόγραμμα ML (χωρίς `free`) θα μπορούσε να μετατραπεί απευθείας σε LetbangCPS στο οποίο όμως θα προστεθούν τα `free` και θα τρέξει ουσιαστικά με μη αυτόματη διαχείριση μνήμης.

– Μπορεί να χρησιμοποιηθεί ως έχει για ένα συντηρητικό έλεγχο για την ασφάλεια ενός προγράμματος ως προς τη χρήση μνήμης. Ένα πρόγραμμα ML δηλαδή με τελεστές `free` και μη αυτόματη διαχείριση μνήμης μπορεί να μετατραπεί απευθείας σε LetbangCPS και έπειτα να προστεθούν τα `let!` και `unlet!`. Ίσως δεν είναι γενικά δύσκολο να κάνουμε typecheck ένα πρόγραμμα LetbangCPS όπου λείπουν τα `let!` και `unlet!` ή έχουν φύγει γενικά από τη σύνταξη της γλώσσας.

– Μπορεί ίσως να χρησιμοποιηθεί και για τον αρχικό σκοπό: έναν όχι και τόσο συντηρητικό έλεγχο για την ασφάλεια ενός προγράμματος ως προς τη χρήση μνήμης. Εδώ πιθανόν να χρειαστούν και μετασχηματισμοί του AST του προγράμματος LetbangCPS.

– Μπορεί να χρησιμοποιηθεί (ίσως με κάποιες τροποποιήσεις) ως ενδιάμεση γλώσσα στη μεταγλώττιση της Letbang. Η απλή της σημασιολογία την κάνει κατάλληλη για διερμηνεία και η μορφή CPS συμβάλλει στην πραγματοποίηση βελτιστοποιήσεων από το μεταγλωττιστή.

8.1.2 Πιθανές επεκτάσεις της LetbangCPS

Πέρα από τις πιθανές άμεσες εφαρμογές που μπορεί να βρει η LetbangCPS ως έχει, μπορεί επιπλέον να επεκταθεί ή να τροποποιηθεί ώστε να επεκτείνει το εύρος των εφαρμογών της. Σε κάθε περίπτωση ο πιθανότερος ρόλος της παραμένει αυτός μιας ενδιάμεσης γλώσσας που στοχεύεται από μεταγλωττιστές και άλλα εργαλεία. Παρακάτω αναφέρονται μερικές εφαρμογές που μπορεί να βρει η γλώσσα μετά από τροποποίηση ή επέκταση:

– Μπορεί να επεκταθεί ώστε να υποστηρίξει νέες δομές, όπως για παράδειγμα εξαιρέσεις. Αυτό μπορεί να γίνει και σε συνδυασμό της χρήσεις της ως ενδιάμεση γλώσσα στη μεταγλώττιση της Letbang, έτσι ώστε να προστεθούν νέες δομές σε αυτήν, στη διαδεδομένη μορφή που οι δομές αυτές έχουν σε συναρτησιακές γλώσσες όπως η ML.

– Μπορεί να τροποποιηθεί ώστε να χρησιμοποιηθεί ως ενδιάμεση γλώσσα άλλων γλωσσών που χρησιμοποιούν γραμμικά συστήματα τύπων και προορίζονται για πεδία εφαρμογών που εστιάζονται σε διαφορετικά θέματα από τη διαχείριση μνήμης. Σε αυτή την περίπτωση πιθανόν να πρέπει να σχεδιαστούν και νέοι μετασχηματισμοί CPS από τις εκάστοτε αρχικές γλώσσες σε LetbangCPS.

8.2 Μελλοντική Έρευνα

Κατά τη διάρκεια της εργασίας αυτής ήρθαν στην επιφάνεια διάφορα ενδιαφέροντα θέματα τα οποία χρήζουν περαιτέρω διερεύνησης. Αρχικά, δεν είναι ίσως τελείως ξεκάθαρο το πώς συνδυάζεται ο μετασχηματισμός CPS με ένα γραμμικό (ή, γενικότερα, ένα substructural) σύστημα τύπων. Σε περίπτωση που μια γλώσσα παρέχει τη δυνατότητα προσωρινής μετατροπής του χαρακτηρισμού των τύπων δεν είναι απαραίτητα εμφανές το πώς αυτό μεταφράζεται από ένα μετασχηματισμό CPS και αναπαρίσταται στο τελικό πρόγραμμα. Ένα μεγαλύτερο ίσως πρόβλημα αποτελεί ο χειρισμός των συνεχειών. Αυτό μπορεί να γίνει φανερό αν υποθέσουμε για παράδειγμα ότι θέλουμε να έχουμε συνέχειες πρώτης τάξης (first-class continuations) σε μια γλώσσα που χρησιμοποιεί γραμμικό σύστημα τύπων. Επιπλέον, δεν είναι βέβαιο μέχρι ποιου σημείου είναι επεκτάσιμη η δομή `let!`, σε συνδυασμό με τις επισημειώσεις των τύπων με scopes και με το χειρισμό των συνεχειών όπως αυτός πραγματοποιείται στη γλώσσα LetbangCPS. Ένα αντίστοιχο πρακτικό ερώτημα θα μπορούσε να είναι πόσα από τα στοιχεία των σύγχρονων συναρτησιακών γλωσσών προγραμματισμού μπορεί να υποστηρίξει μια γλώσσα όπως η Letbang.

Τέλος, παρατίθενται ορισμένα ανοιχτά θέματα που σχετίζονται με τη γλώσσα LetbangCPS:

– Τυπική απόδειξη συνέπειας για τη LetbangCPS. Όπως αναφέρθηκε, υπάρχει ήδη η απόδειξη συνέπειας για τη γλώσσα Letbang. Καθώς η LetbangCPS δανείζεται πολλά στοιχεία και ιδέες από τη Letbang, το αναμενόμενο είναι η απόδειξη για τη συνέπειά της να μοιάζει με εκείνη της Letbang, η οποία θα μπορούσε πιθανόν να χρησιμοποιηθεί και ως πρότυπο.

– Τυπική απόδειξη του μετασχηματισμού CPS μεταξύ Letbang και LetbangCPS που ορίστηκε στο Κεφάλαιο 6.

– Προσθήκη διαφόρων ειδών πολυμορφισμού στη LetbangCPS. Ο πολυμορφισμός πάνω σε μεμονωμένα scopes αναμένεται να είναι εύκολος, καθώς έχει ήδη προστεθεί στη Letbang και έχει αποδειχθεί η συνέπειά του. Ο πολυμορφισμός πάνω σε αυθαίρετα περιβάλλοντα scope, που ίσως θα χρειαστεί για το χειρισμό των συνεχειών πρώτης τάξης, αναμένεται μάλλον δυσκολότερος.

Βιβλιογραφία

[Ahme07] Amal Ahmed, Matthew Fluet and Greg Morrisett, "L3: A Linear Language with Locations", *Fundamenta Informaticae*, vol. 77, no. 4, pp. 397–449, 2007.

[Danv92] Olivier Danvy and Andrzej Filinski, "Representing control: a study of the CPS transformation", 1992.

[Fisc72] M. J. Fischer, "Lambda calculus schemata", in *Proceedings of the ACM Conference on Proving Assertions about Programs*, pp. 104–109, 1972.

[Flue06] Matthew Fluet, Greg Morrisett and Amal Ahmed, "Linear Regions Are All You Need", in *ESOP'06: Proceedings of the Fifteenth European Symposium on Programming*, pp. 7–21, Springer-Verlag, March 2006.

[Harp93] Robert Harper and Mark Lillibridge, "Polymorphic Type Assignment and CPS Conversion", *Lisp and Symbolic Computation*, 1993.

[Meye85] Albert R. Meyer and Mitchell Wand, "Continuation semantics in typed lambda calculi", in *Logics of Programs*, vol. 193 of *Lecture Notes in Computer Science*, pp. 219–224, Springer-Verlag, 1985.

[Morr05] G. Morrisett, A. J. Ahmed and M. Fluet, "L^3: A Linear Language with Locations", in *TLCA*, pp. 293–307, 2005.

[Papa08] Nikolaos S. Papaspyrou and Michalis A. Papakyriakou, "From Linear to Unrestricted and Back: Type Safety and the Let-bang Construct". Unpublished manuscript, 2008.

[Plot75] G. D. Plotkin, "Call-by-name, call-by-value and λ-calculus", *Theoretical Computer Science*, 1975.

[Reyn72] J. C. Reynolds, "Definitional interpreters for higher-order programming languages", in *Proceedings of* 25^{th} *ACM National Conference*, pp. 717–740, 1972.

[Wadl90] P. Wadler, "Linear Types can Change the World!", in M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pp. 347–359, North Holland, 1990.