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

Destructive Update με γραμμικούς τύπους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΕΞΑΝΔΡΟΣ-ΠΕΤΡΟΣ ΜΟΣΧΟΣ

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

Αθήνα, Αύγουστος 2018

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

Destructive Update με γραμμικούς τύπους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΕΞΑΝΔΡΟΣ-ΠΕΤΡΟΣ ΜΟΣΧΟΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Αυγούστου 2018.

....	....	....
Νικόλαος Σ. Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Δημήτρης Φωτάκης Επίκ. Καθηγητής Ε.Μ.Π.	Γεώργιος Στάμου Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Αύγουστος 2018

....

Αλέξανδρος-Πέτρος Μόσχος

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

Copyright © Αλέξανδρος-Πέτρος Μόσχος, 2018.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

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

Περίληψη

Σκοπός της παρούσας εργασίας είναι η χρήση των γραμμικών τύπων για τον ορισμό και την υλοποίηση μιας γλώσσας που υλοποιεί τις εγγραφές σε πίνακα καταστροφικά με ασφαλή τρόπο. Θέλουμε ο προγραμματιστής να γράφει κώδικα χωρίς επισημειώσεις τύπων με πίνακες χρησιμοποιώντας ένα αγνά συναρτησιακό interface, και ο μεταγλωττιστής, εφόσον είναι ασφαλές, να υλοποιεί τις εγγραφές in-place.

Στο πλάισιο της εργασίας ορίστηκε τυπικά, η σύνταξη, το σύστημα τύπων και η λειτουργική σημασιολογία της ενδιάμεσης γλώσσας letdo η οποία βασίζεται σε ένα γραμμικό σύστημα τύπων. Οι γραμμικοί τύποι προσφέρουν την ιδιότητα ότι κάθε πίνακας θα χρησιμοποιηθεί ακριβώς μία φορά. Για να προσαρμόσουμε την γλώσσα στο ζητούμενο συναρτησιακό interface, προσθέτουμε στην γλώσσα το letdo construct που υποστηρίζει πρόσβαση μόνο για ανάγνωση στον πίνακα χωρίς όμως να προσθέτει περιορισμούς στην γλώσσα. Ορίζουμε έναν semantics-preserving μετασχηματισμό που λαμβάνει κώδικα στην αρχική γλώσσα και τον μετασχηματίζει σε κώδικα letdo. Έπειτα, ορίσαμε και υλοποιήσαμε έναν αλγόριθμο συμπερασμού τύπων για την γλώσσα letdo που λειτουργεί παράγωντας και επιλύοντας περιορισμούς σε δύο στάδια. Έχουμε κατασκευάσει έναν μεταγλωττιστή που λαμβάνει κώδικα στην αρχική γλώσσα τον μετασχηματίζει στην γλώσσα letdo και έπειτα γίνεται συμπερασμός των τύπων. Εάν το πρόγραμμα περάσει τον έλεγχο τύπων είναι ασφαλές οι εγγραφές σε πίνακες να γίνουν destructively.

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, Γραμμικοί Τύποι, Destructive Update, Συναρτησιακός προγραμματισμός, let!, let! with scopes, letdo, Συμπερασμός τύπων, Semantics-preserving transformation.

Abstract

The purpose of this diploma dissertation is to implement safe destructive update for a functional programming language. The programmer will write purely functional code without type annotations and the compiler will implement the updates in-place when it is safe to do so.

The need for efficient functional code is really important. In most applications, a programmer relies on referential transparency and immutability to prove mathematical properties about a functional program. However, referential transparency states that an update operation to an array should create a completely new array and leave the original array intact. In our language, called letdo, linear types will ensure that the old array is not used after being updated, allowing use to perform the update in place. A type system of linear and non linear types is presented, alongside a readonly construct that doesn't have the constraints previous read-only constructs, presented in the bibliography, have.

However, since we want the programmer to use a purely functional array interface, we implemented a semantics-preserving transformation from a functional language like OCaml to letdo. To avoid having linear type annotations in the original language, we present a type inference algorithm. The algorithm is based on constraint generation and solving in two discrete steps. The type inference algorithm does not support qualifier polymorphism, but it does support polymorphism on Non Linear types.

Finally, we have implemented a fully working compiler for our language. It parses, the original language, transforms it and infers the types. Finally, it emits machine code that implements array updates in-place.

Key words

Programming Languages, Linear Types, Linear Logic, Destructive update, Functional Arrays, Functional Programming, let!, let! with scopes, letdo, Type Inference, Semantics-preserving transformation.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Νικόλαο Παπασπύρου, για τη συνεχή καθοδήγηση και εμπιστοσύνη του και τον κ. Δημήτρη Φωτάκη που απο το πρώτο έτος στο ΕΜΠ με καθοδηγεί και με εμπνέει. Ευχαριστώ επίσης την οικογενειά μου και την Δήμητρα για όλη την βοήθεια και στην στήριξη που μου παρείχαν. Τέλος, ευχαριστώ τους φίλους μου, για όλες τις ωραίες στιγμές που περάσαμε αυτά τα χρόνια.

Αλέξανδρος-Πέτρος Μόσχος,

Αθήνα, 29η Αυγούστου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-8-18, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Αύγουστος 2018.

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

Περιεχόμενα

Περίληψη . . . 5

Abstract . . . 7

Ευχαριστίες . . . 9

Περιεχόμενα . . . 11

Κατάλογος σχημάτων . . . 13

1. Εισαγωγή . . . 15
- 1.1 Σκοπός . . . 15
- 1.2 Κίνητρο . . . 15
- 1.3 Εφαρμογές γραμμικών συστημάτων τύπων . . . 16

2. Γραμμικά συστήματα τύπων. Η γλώσσας Letbang . . . 17
- 2.1 Αγνά Γραμμικά συστήματα τύπων . . . 17
- 2.2 Αμφισημία Lollipop . . . 20
- 2.3 Σύστημα τύπων με γραμμικές και μη τιμές . . . 21
- 2.4 Πίνακες σε αγνές συναρτησιακές γλώσσες . . . 21
- 2.5 Η γλώσσα Letbang . . . 22
- 2.6 Observer Types . . . 23
- 2.7 Letbang with scopes . . . 24

3. Η γλώσσα Letdo . . . 27
- 3.1 Read only access χωρίς lexical scoping του πίνακα . . . 27
- 3.2 Τύποι συναρτήσεων . . . 29
- 3.3 Συστημα τύπων . . . 30
- 3.4 Λειτουργική σημασιολογία . . . 30

4. Μετασχηματισμός μιας αγνής συναρτησιακής γλώσσας σε Letdo . . . 35
- 4.1 Αναγνώσεις απο πίνακα . . . 35
- 4.2 Σύνταξη εκφράσεων της αρχικής γλώσσας . . . 35
- 4.3 Σύνταξη εκφράσεων της ενδιάμεσης γλώσσας . . . 36
- 4.4 Απλοποίηση εκφράσεων ενδιάμεσης γλώσσας . . . 37

5. Συμπερασμός τύπων στην ενδιάμεση γλώσσα . . . 39
- 5.1 Αλγοριθμικός έλεγχος τύπων για την ενδιάμεση γλώσσα . . . 39
- 5.2 Συμπερασμός τύπων . . . 41
- 5.3 Περιορισμοί . . . 42
- 5.4 Επίλυση των περιορισμών . . . 43
- 5.5 Παραδείγματα συμπερασμού τύπων . . . 47

6. Συμπεράσματα και προτάσεις μελλοντικής έρευνας . . . 53
6.1 Συνεισφορά . . . 53
6.2 Προτάσεις μελλοντικής έρευνας . . . 53

Βιβλιογραφία . . . 57

Κατάλογος σχημάτων

2.1 Γραμματική λαμδα λογισμού με απλούς τύπους 17
2.2 Γραμματική γλώσσας και τύπων 19
2.3 Σύστημα τύπων λάμδα λογισμού με γραμμικούς τύπους 19
2.4 Σύστημα τύπων με γραμμικούς και μη τύπους 22

3.1 Σύνταξη Τύπων 30
3.2 Σύστημα τύπων letdo 31
3.3 Λειτουργική σημασιολογία letdo 32
3.4 Λειτουργική σημασιολογία letdo(Συνέχεια) 33

4.1 Γραμματική αρχικής γλώσσας 36

5.1 Σύνταξη Τύπων 42

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Η εργασία αυτή αποσκοπεί στον ορισμό μιας γλώσσας προγραμματισμού που επιτρέπει destructive update σε πίνακες όταν αυτό είναι ασφαλές, με την χρήση ενός γραμμικού συστήματος τύπων. Ο προγραμματιστής θα γράφει προγράμματα σε μια συναρτησιακή γλώσσα, όπως η OCaml, ο κώδικας του θα μετασχηματίζεται σε μία ενδιάμεση γλώσσα η οποία θα υποστηρίζει γραμμικό σύστημα τύπων και εφόσον το πρόγραμμα ειναι well typed θα είναι ασφαλές να υλοποιηθούν καταστροφικά τα updates σε πίνακες. Ουσιαστικά λοιπόν η εργασία χωρίζεται σε 2 μέρη, το πρώτο είναι ο ορισμός του μετασχηματισμού απο την αρχική γλώσσα στην ενδιάμεση και το δεύτερο είναι ο ορισμός του συστήματος τύπων και της σημασιολογίας της ενδιάμεσης γλώσσας καθώς και ένας αλγόριθμος συμπερασμού τύπων.

1.2 Κίνητρο

Στις προστακτικές γλώσσες προγραμματισμού, ο προγραμματιστής αλλάζει την κατάσταση(state) με εντολές στο πρόγραμμα του, εκ των οποίων η πιο απλή είναι η εντολή ανάθεσης. Ένα βασικό χαρακτηριστικό που έχουν σχεδόν όλες οι προστακτικές γλώσσες είναι πίνακες με σταθερό χρόνο ανάγνωσης και εγγραφής. Οι πίνακες είναι ενα απο τα βασικά δομικά στοιχεία για την δημιουργία σύνθετων δομών δεδομένων, όπως ο πίνακας κατακερματισμού και ο δυαδικός σωρός.

Στις αγνές γλώσσες συναρτησιακού προγραμματισμού οι πίνακες μπορούν να αντικατασταθούν απο άλλες δομές δεδομένων που προσφέρουν λογαριθμική ανάγνωση και εγγραφή. Συνεπώς, η χρήση αγνών συναρτησιακών δομών δεδομένων είναι ένα μειονέκτημα των συναρτησιακών γλωσσών προγραμματισμού. Πιο συγκεκριμένα, ο Pippenger[Pipp96] έδειξε ότι για ορισμένους online αλγορίθμους ότι για να κάνει ένα αγνό πρόγραμμα Lisp ότι κάνει ένα μη αγνό πρόγραμμα Lisp σε n βήματα απαιτούνται τουλάχιστον $\Omega(nlogn)$ βήματα. Ωστόσο, το φράγμα αυτό έχει αποδειχτεί μόνο για eager συναρτησιακές γλώσσες και όπως έδειξαν οι Bird, Jones και Moor[BIRD97] δεν ισχύει όταν η γλώσσα έχει lazy evaluation.

Η ίδεα της διπλωματικής είναι να επιτρέψουμε στους πίνακες να έχουνε ενα αγνό συναρτησιακό interface το οποίο να μπορεί να χρησιμοποιεί ο προγραμματιστής, ενώ παράλληλα το σύστημα τύπων να εξασφαλίζει ότι για κάθε πίνακα υπάρχει μόνο μια αναφορά πάνω σε αυτόν για να είναι ασφαλές το destructive update, επιτρέποντας ανάγνωση και εγγραφή σε σταθερό χρόνο. Επίσης, ο προγραμματιστής θέλει η διεπαφή του με τους πίνακες να έχει συναρτησιακή μορφή οπότε πρέπει η συνάρτηση ανάγνωσης να παίρνει τον πίνακα και την θέση του στοιχείου και να επιστρέφει την τιμή και η συνάρτηση εγγραφής να παιρνεί τον πίνακα την θέση του στοιχείου που πρέπει να ανανεωθεί και να επιστρέφει έναν νέο πίνακα. Το γραμμικό σύστημα τύπων θα απαγορεύσει στον προγραμματιστή να κάνει χρήση του παλιού πίνακα μετά την ανανέωση.

Σε συναρτησιακές γλώσσες που υποστηρίζουν μη αγνά χαρακτηριστικά υπάρχουν πίνακες και references που έχουν destructive update, ωστόσο δεν υπάρχει εγγύηση ασφάλειας απο το σύστημα τύπων, εγγύηση δηλαδή ότι ο προγραμματιστής δεν θα προσπαθήσει να αναφερθεί σε πίνακα που έχει πια απελευθερωθεί.

1.3 Εφαρμογές γραμμικών συστημάτων τύπων

Η γλώσσα προγραμματισμού Rust υποστηρίζει ένα Affine σύστημα τύπων. Ουσιαστικά, είναι σαν γραμμικό σύστημα τύπων με τον περιορισμό ότι τα resources μπορούν να χρησιμοποιηθούν το πολύ μία φορά. Η Rust με την βοήθεια του affine type system, υποστηρίζει memory safety χωρίς Garbage collector, δηλαδή τα (well typed) προγράμματα Rust έχουν μια ιδιότητα ότι οι δείκτες δείχνουν πάντοτε σε έγκυρη μνήμη, σωστού τύπου και μεγέθους.

Μια γλώσσα με αμιγώς γραμμικούς τύπους λειτουργεί(θεωρητικά) χωρίς garbage collector στο runtime system της. Αυτό έχει υλοποιηθεί στην γλώσσα LinearML όπου όλες οι τιμές είναι γραμμικού τύπου και δεν υπάρχει garbage collection. Επίσης, η LinearML υποστηρίζει thread safe επικοινωνία χωρίς αντιγραφές μεταξύ των threads και thread safe IO.

Στην βιβλιογραφία οι γραμμικοί τύποι έχουν χρησιμοποιηθεί για να λυθεί το πρόβλημα strong update σε reference. Με τον όρο strong update εννοούμε μία ανανέωση σε μία θέση μνήμης που αλλάζει τον τύπο του αντικειμένου που αποθηκεύεται.

```
let val r = ref () in
r := true;
if (!r) then r := 42 else r := 15;
!r + 12
end
```

Οι Morrisett,Ahmed και Fluet[Morr05] χρησιμοποιήσαν γραμμικά capabilities που συνδέονται με κάθε reference για να επιτρέψουν το strong update σε references.

Κεφάλαιο 2

Γραμμικά συστήματα τύπων. Η γλώσσας Letbang

2.1 Αγνά Γραμμικά συστήματα τύπων

Ο λάμδα λογισμός με απλούς τύπους αποτελεί το πιο απλό παράδειγμα λαμδα λογισμού με τύπους. Οι εκφράσεις είναι απλές και οι τύποι αποτελούνται απο τους βασικούς τύπους και τύπους που κατασκευάζονται απο έναν μοναδικό κατασκευαστή, που ονομάζεται function arrow. Ο λαμδα λογισμός

$$
\begin{aligned}
e &::= x \mid \ \lambda x : \tau . v \mid \ (e\ e) \mid \ c \\
\tau &::= \tau \to \tau \mid \ T \ o\pi o\upsilon \ T \in B
\end{aligned}
$$

Σχήμα 2.1: Γραμματική λαμδα λογισμού με απλούς τύπους

με απλούς τύπους επιτρέπει ελεύθερη χρήση των μεταβλητών στο πλαίσιο του έλεγχου τύπων. Για παράδειγμα κάθε μεταβλητή μπορει να χρησιμοποιηθεί καμία, μία ή πολλές φορές. Παρατηρώντας την γλώσσα του λάμβδα λογισμού με απλούς τύπους παρατηρούμε ότι υπάρχουν 3 βασικές δομικές ιδιότητες.

Λήμμα 1 (**Ανταλλαγή**) : Αν

$$\Gamma_1, x_1 : t_1, x_2 : t_2, \Gamma_2 \vdash x : t \tag{2.1}$$

τότε

$$\Gamma_1, x_2 : t_2, x_1 : t_1, \Gamma_2 \vdash x : t \tag{2.2}$$

Λήμμα 2 (**Αποδυνάμωση**): Αν

$$\Gamma_1, \Gamma_2 \vdash x : t \tag{2.3}$$

τότε

$$\Gamma_1, \Gamma_2, x_1 : t_1 \vdash x : t \tag{2.4}$$

Λήμμα 3 (**Συστολή**):Αν

$$\Gamma_1, x_2 : t_1, x_3 : t_1, \Gamma_2 \vdash x : t \tag{2.5}$$

τότε

$$\Gamma_1, \Gamma_2, x_1 : t_1 \vdash [x_2 \mapsto x_1][x_3 \mapsto x_1]x : t \tag{2.6}$$

Το λήμμα της ανταλλαγής ουσιαστικά σημαίνει ότι η σειρά των assertions στο typing context δεν έχει σημασία. Συνεπώς, εαν Γ αποδεικνύει ότι η μεταβλητή χ έχει τύπο τ, τότε κάθε μετάθεση του Γ μπορει να αποδείξει ότι η μεταβλητή χ έχει τύπο τ. Η αποδυνάμωση ουσιαστικά σημαίνει ότι μπορούμε να προσθέσουμε αχρείαστες υποθέσεις στο context και κάθε όρος συνεχίζει να έχει τον ίδιο τύπο. Τέλος το λήμμα της συστολής αναφέρει ότι αν μπορούμε να αποδείξουμε τον τύπο ενός όρου με 2 ίδιες υποθέσεις τότε μπορούμε να αποδείξουμε τον τύπο του ίδιου όρου με μονάχα μια υπόθεση. Οι 3 αυτές ιδιότητες αποδεικνύονται εύκολα επαγωγικά πάνω στο typing judgement $\Gamma \vdash e : T$.

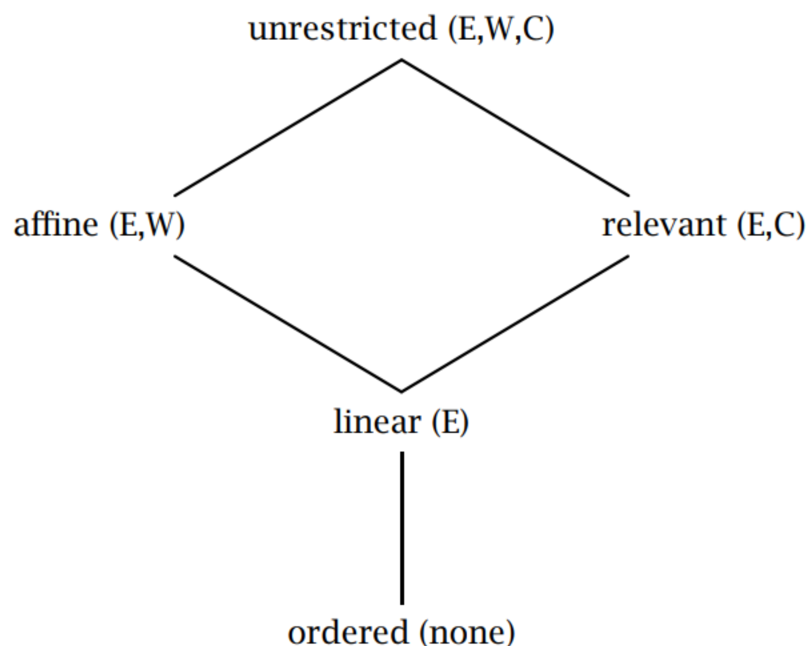
Ενα σύστημα τύπων θα λέμε ότι είναι υποδομικό εάν είναι σχεδιασμένο με τέτοιο τρόπο έτσι ώστε μια ή περισσότερες απο τις παραπάνω ιδιότητες δεν ισχύει.

- Γραμμικό(Linear) συστημα τύπων. Κάθε μεταβλητή **πρέπει** να χρησιμοποιηθεί **ακριβώς** μια φορά. Επιτρέπεται ανταλλαγή αλλα όχι αποδυνάμωση και συστολή

- Affine συστημα τύπων. Κάθε μεταβλητή **πρέπει** να χρησιμοποιηθεί **το πολύ** μία φορά. Επιτρέπεται ανταλλαγή και αποδυνάμωση αλλα όχι συστολή.

- Relevant συστημα τύπων. Κάθε μεταβλητής **πρέπει** να χρησιμοποιηθεί **τουλάχιστον** μία φορά. Επιτρέπεται ανταλλαγή και συστολή αλλα όχι αποδυνάμωση.

- Ordered συστημα τύπων. Όλες οι μεταβλητές **πρέπει** να χρησιμοποιηθεί **ακριβώς μία φορά με την σειρά που ορίζονται**. Δεν επιτρέπεται ούτε ανταλλαγή, ούτε αποδυνάμωση, ούτε συστολή.



Στην παρούσα διπλωματική εργασία θα κάνουμε χρήση γραμμικών συστημάτων τύπων. Η ιδιότητα ότι κάθε μεταβλητή πρέπει να χρησιμοποιηθεί ακριβώς μία φορά είναι αρκετά χρήσιμη καθώς εφόσον χρησιμοποιηθεί κάποια μεταβλητή γνωρίζουμε στατικά ότι είναι ασφαλές να αποδεσμευτεί η μνήμη. Κάθε μεταβλητή γραμμικού τύπου διατηρεί μια ιδιότητα singlethreadedness δηλαδή πάνω της υπάρχει ακριβώς μία αναφορά. Άμεση συνέπεια αυτού είναι ότι δεν υπάρχει aliasing των μεταβλητών, δηλαδή δεν υπάρχει η κατάσταση στην οποία δύο ή περισσότερα συμβολικά ονόματα σε ένα πρόγραμμα αναφέρονται στα ίδια δεδομένα.

Στην συνέχεια θα παρουσιάσουμε ένα αγνό γραμμικό σύστημα τύπων στο οποίο όλες οι τιμές είναι γραμμικές. Το βασικό χαρακτηριστικό του συστήματος είναι ότι κάθε εμφάνιση μιας υπόθεσης x : t στην λιστα των υποθέσεων πρεπει να χρησιμοποιηθεί ακριβώς μία φορά.

Η γραμματική των τύπων και των όρων της γλώσσας φαίνεται παρακάτω στο σχήμα 2.2.

Η συγκεκριμένη γλώσσα δεν έχει τελεστή σταθερού σημείου καθώς μια τιμή γραμμικού τύπου πρέπει να χρησιμοποιηθεί ακριβώς μία φορά αλλά ο τελεστής σταθερού σημείου fix f ισοδυναμεί με f (f (f (f ...))). Ως αποτέλεσμα δεν έχουμε αναδρομή άρα η γλώσσα δεν είναι Turing complete. Το περιβάλλον Γ θα αποτελεί μια λίστα απο assumptions της μορφής x : τ.

$$
\begin{aligned}
\tau &::= \kappa \\
&\mid \ \tau \multimap \tau \\
&\mid \ \tau \otimes \tau
\end{aligned}
$$

$$
\begin{aligned}
e &::= x \\
&\mid \ \lambda x : \tau . e \\
&\mid \ (e\ e) \\
&\mid \ (e, e)
\end{aligned}
$$

Σχήμα 2.2: Γραμματική γλώσσας και τύπων

$$\frac{}{x : T \vdash x : T}\ Var \tag{2.7}$$

$$\frac{A, x : U \vdash v : V}{\mathrm{A} \vdash \lambda x : U.\, v : U \multimap V} \multimap I \tag{2.8}$$

$$\frac{A \vdash f : U \multimap V \qquad B \vdash u : U}{A, B \vdash f\ u : V} \multimap E \tag{2.9}$$

$$\frac{A \vdash t_1 : T_1 \qquad B \vdash t_2 : T_2}{A, B \vdash (t_1, t_2) : T_1 \otimes T_2}\ PairI \tag{2.10}$$

$$\frac{\Gamma_1 \vdash t : T_1 \otimes T_2 \qquad \Gamma_2, t_1 : T_1, t_2 : T_2 \vdash u : U}{\Gamma_1, \Gamma_2 \vdash \mathit{let\ split}\ t\ \mathit{as}\ t_1, t_2\ \mathit{in}\ u : U}\ PairE \tag{2.11}$$

Σχήμα 2.3: Σύστημα τύπων λάμδα λογισμού με γραμμικούς τύπους

Αξίζει να παρατηρήσουμε ότι όλοι οι κανόνες τύπων εκτός απο τον κανόνα της νέας μεταβλητής εμφανίζονται σε ζεύγη. Υπάρχει πάντα ο κανόνας που δημιουργεί ένα νέο resource και ο κανόνας που καταναλώνει το resource. Οι συναρτήσεις είναι και αυτές γραμμικές τιμές και πρέπει να χρησιμοποιηθούν ακριβώς μία φορά.

Οι S και K combinators που υπάρχουν στον λάμδα λογισμό με απλούς τύπους στην γλώσσα με γραμμικούς τύπους είναι ill typed. Ο πρώτος καναλώνει μια μεταβλητή 2 φορές και ο δεύτερος δεν καταναλώνει μία απο τις παραμέτρους του.

$$s = \lambda x.\lambda y.\lambda z.(x\ z\ (y\ z)) \tag{2.12}$$

$$k = \lambda x.\lambda y.x \tag{2.13}$$

Στο γραμμικό σύστημα τύπων κάθε εντολή που δημιουργεί μια τιμή αντιστοιχίζεται με μια εντολή που καταναλώνει την τιμή. Οι εντολές που δημιουργούν μεταβλητές, δημιουργούν μεταβλητές για τις οποίες υπάρχει μόνο μία αναφορά. Οι αναφορές αυτές δεν μπορούν αντιγραφούν ούτε να χαθούν. Οι εντολές που καταστρέφουν(ή χρησιμοποιούν) μεταβλητές δρούν πάνω σε μεταβλητές με μονάχα μία αναφορά. Συνεπώς, αφού χρησιμοποιηθεί μια μεταβλητή η θέση μνήμης την οποία καταλαμβάνει μπορεί να απελευθερωθεί. Αυτό σημαίνει, οτι η γλώσσα αυτή επιτρέπει στατική διαχείριση μνήμης χωρίς μέτρηση αναφορών ή συλλογή σκουπιδιών. Ωστόσο, η γλώσσα το επιτυγχάνει αυτό προσθέτοντας έναν μεγάλο περιορισμό, κάθε μεταβλητή πρέπει να χρησιμοποιηθεί ακριβώς μία φορά. Ο

περιορισμός αυτός είναι λογικός στην περίπτωση που αναφερόμαστε σε μια σύνθετη δομή δεδομένων που αποθηκεύει πολλά δεδομένα αλλά δεν είναι τόσο λογικός στην περίπτωση που η μεταβλητή είναι απλά ένας ακέραιος που θέλουμε να χρησιμοποιήσουμε σε διάφορα σημεία του προγράμματος μας. Για να λυθεί, αυτός ο βασικός περιορισμός πρέπει μια γλώσσα με γραμμικούς τύπους να έχει και μεταβλητές μή γραμμικού τύπου. Η ύπαρξη τιμών μη γραμμικού τύπου θα επιτρέψει στην γλώσσα να γίνει Turing complete με την προσθήκη τελεστή σταθερού σημείου. Παρακάτω φαίνονται ορισμένα παραδείγματα ελέγχου τύπων:

$$\frac{\frac{}{x : int \vdash x : int}}{. \vdash \lambda x : int.x : int \multimap int}$$

Εδώ παρατηρούμε ότι η ταυτοτική συνάρτηση έχει τύπου $\tau \multimap \tau$. Βλέπουμε και τον περιορισμό του συστήματος καθώς ακόμη και η ταυτοτική συνάρτηση πρέπει να χρησιμοποιηθεί ακριβώς μία φορά.

$$\frac{\frac{\frac{}{t : A \otimes B \vdash t : A \otimes B} \qquad \frac{\frac{}{a : A \vdash a : A} \qquad \frac{}{b : B \vdash b : B}}{a : A, b : B \vdash (b, a) : B \otimes A}}{t : A \otimes B \vdash let\ split\ t\ as\ a, b\ in\ (b, a) : A \otimes B \multimap B \otimes A}}{. \vdash \lambda t : A \otimes B.let\ split\ t\ as\ a, b\ in\ (b, a) : A \otimes B \multimap B \otimes A}$$

Η συνάρτηση η οποία παίρνει ένα ζεύγος απο ένα A και ένα B και επιστρέφει ένα ζεύγος απο ενα B και ένα A είναι well typed καθώς κάθε μεταβλητή χρησιμοποιείται ακριβώς μία φορά.

$$\frac{\frac{\frac{}{t : A \vdash t : A} \qquad \frac{XXX}{. \vdash t : A}}{t : A \vdash (t, t) A \otimes A}}{. \vdash \lambda t : A.(t, t) : A \otimes A}$$

Στο παραπάνω παράδειγμα αποτυγχάνει η χρήση της ίδιας μεταβλητής t δύο φορές, αφού στο context split μόνο το ένα σκέλος επιτυγχάνει ενώ το άλλο αποτυγχάνει.

2.2 Αμφισημία Lollipop

Η θεωρία πίσω απο τους γραμμικούς τύπους προέρχεται απο την γραμμική λογική που ανακάλυψε ο Girard[Gira87] το 1987. Μια σημαντική πρόταση για την μαθηματικη λογική και την θεωρία των γλωσσών προγραμματισμού είναι η αντιστοίχιση Curry Howard. Ουσιαστικά, η αντιστοίχιση Curry Howard είναι η παρατήρηση ότι μια μαθηματική απόδειξη είναι ένα πρόγραμμα και το θεώρημα που αποδεικνύει είναι ο τύπος αυτού του προγράμματος. Η γραμμική λογική, δεν επιτρέπει την συστολή και την αποδυνάμωση, δηλαδή είναι ένα σύστημα στο οποίο η διαδικασία απόδειξης είναι μια διαδικασία στην οποία εκτός απο την παραγωγή αληθών προτάσεων τα ”resources” δεν μπορούνε να χρησιμοποιηθούν πολλές φορές ή να μείνουν αχρησιμοποίητα. Οι γραμμικοί τύποι είναι η αντιστοίχιση της γραμμικής λογικής στον κόσμο των γλωσσών προγραμματισμού. Ωστόσο όταν προσθέτουμε μη γραμμικούς τύπους στην γλώσσα πρέπει να κάνουμε ορισμένες σχεδιαστικές επιλογές επειδή ουσιαστικά θέλουμε να επεκτείνουμε την γραμμική λογική με ένα κομμάτι intuitionistic logic στο οποίο θα αντιστοιχίζονται οι μη γραμμικοί τύποι. Για αυτό τον λόγο οι γλώσσες προγραμματισμού με γραμμικούς τύπους καλύπτουν ένα ευρύ design space.

Έστω μια συνάρτηση $f :: a \multimap b$ σε μια γλώσσα προγραμματισμού η συνάρτηση αυτή μπορεί να σημαίνει ένα απο τα παρακάτω:

- Η συνάρτηση f θα κληθεί ακριβώς μια φορά
- Η συνάρτηση f θα λάβει το μοναδικό reference σε μια μεταβλητή τύπου a

- Η συνάρτηση f εγγυάται ότι θα χρησιμοποιήσει μια μεταβλητή τύπου a ακριβώς μια φορά

Ο Bernardy[Bern17] μαζί με την ομάδα του, προτείνουν μια επέκταση της γλώσσας προγραμματισμού Haskell που ονομάζεται Linear Haskell. Η επέκταση αυτή προσφέρει δυο πλεονεκτήματα: Το πρώτο είναι ότι ο κώδικας Linear Haskell είναι backwards compatible. Το δεύτερο είναι ότι ο κώδικας μιας βιβλιοθήκης μπορεί να χρησιμοποιηθεί σε linear και non linear context. Για να επιτύχουν αυτά τα αποτελέσματα ορίζουν τον type constructor $\alpha \multimap \beta$ έτσι ώστε να εγγυάται ότι θα χρησιμοποιήσει μια μεταβλητή τύπου α ακριβώς μια φορά. Ωστόσο, δεν είναι ξεκάθαρο τι σημαίνει να χρησιμοποιηθεί μια μεταβλητή τύπου α. Ο ορισμός της χρήσης είναι αναδρομικός: Μια μεταβλητή χ βασικού τύπου χρησιμοποιείται όταν υπολογιστεί. Μια συνάρτηση f χρησιμοποιείται ακριβώς μία φορά όταν κληθεί ακριβώς μια φορά και χρησιμοποιηθεί το αποτέλεσμα ακριβώς μία φορά. Μια τιμή x ενός σύνθετου τύπου δεδομένων χρησιμοποιείται ακριβώς μία φορά όταν γίνεται pattern match πάνω σε αυτόν και κάθε συνιστώσα χρησιμοποιείται ακριβώς μία φορά. Η προσέγγιση αυτή είναι καλή για μια lazy γλώσσα προγραμματισμού όπως η Haskell αλλά έχει αρκετά μειονεκτήματα. Για παράδειγμα, εάν έχουμε δύο ισοδύναμες εκφράσεις όπου η μία είναι η η-αναγωγή της άλλης μπορεί να παρατηρήσουμε ότι η μία προκαλεί σφάλμα τύπων ενώ η άλλη όχι. Επίσης, δεν έχει παρουσιαστεί αλγόριθμος συμπερασμού τύπων για την γλώσσα Linear Haskell που να υποστηρίζει και τον πολυμορφισμό στην γραμμικότητα. Για αυτούς τους λόγους στην συνέχεια θα θεωρήσουμε το $a \multimap b$ ώς μια συνάρτηση που πρέπει να κληθεί ακριβώς μία φορά. Η προσέγγιση αυτή μας επιτρέπει να διαχωρίσουμε τα σύνολα των γραμμικών και μη γραμμικών τύπων, και ως αποτέλεσμα υπάρχει ένας αλγόριθμος συμπερασμού τύπων δύο βημάτων, τον οποίο θα δούμε παρακάτω.

2.3 Σύστημα τύπων με γραμμικές και μη τιμές

Για να μπορέσουμε να έχουμε μεταβλητές γραμμικού και μη γραμμικού τύπου θα κάνουμε μια επέκταση της γλώσσας της προηγούμενης ενότητας. Στην γλώσσα αυτή υπάρχουν τύποι που είναι γραμμικοί (όπως για παράδειγμα ο τύπος πίνακα) και τύπου που είναι μη γραμμικοί (όπως οι ακέραιοι). Στο συγκεκριμένο σύστημα καθε τύπος μπορει να είναι είτε γραμμικός είτε μη γραμμικός και συνεπώς η τομή του συνόλου των γραμμικών και των μη γραμμικών τύπων είναι το κενό σύνολο.

Παρακάτω στο σχήμα 2.4, φαίνεται το σύστημα τύπων με γραμμικούς και μή τύπους

2.4 Πίνακες σε αγνές συναρτησιακές γλώσσες

Οι συμβατικοί πίνακες στις αγνές συναρτηιακές γλώσσες συνήθως προσφέρουν ένα interface που μοαζει με το παρακάτω

$$\begin{aligned} new &:: Nat \to Nat \to Arr, \\ read &:: Nat \to Arr \to Val, \\ update &:: Nat \to Var \to Arr \to Arr, \\ free &:: Arr \to \mathrm{I} \end{aligned}$$

Ωστόσο το πρόβλημα δημιουργείτε όταν κάποιος γράφει κώδικα ως εξής

```
let a = new 10 10 in
let b = update 5 10 a in
read 0 a + read 0 b
```

Δηλαδή όταν χρησιμοποιεί έναν πίνακα πάνω στον οποίο έχει εφαρμοστεί μια συνάρτηση update. Συνεπώς, πρέπει στην μνήμη του υπολογιστή να υπάρχουν και οι 2 πίνακες α και β. Ο στόχος λοιπόν είναι να έχουμε πίνακες με το παραπάνω interface αλλα η συνάρτηση update να είναι καταστροφική.

$$\frac{}{x : T \vdash x : T} LVar \tag{2.14}$$

$$\frac{}{\Gamma, x : \alpha; \emptyset \vdash x : \alpha} UVar \tag{2.15}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma .\, u : \sigma \multimap \psi} AbsLL \tag{2.16}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma .\, u : \sigma \multimap \psi} AbsLU \tag{2.17}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma .\, u : \sigma \multimap \psi} AbsUL \tag{2.18}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma .\, u : \sigma \multimap \psi} AbsUU \tag{2.19}$$

$$\frac{\Gamma; \Delta 1 \vdash f : \varphi \multimap \psi \qquad \Gamma; \Delta 2 \vdash u : \varphi}{\Gamma; \Delta 1, \Delta 2 \vdash f\ u : \psi} AppL \tag{2.20}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma .\, u : \sigma \multimap \psi} AppU \tag{2.21}$$

$$\frac{\Gamma; \Delta \vdash e : a \qquad \Gamma, x : a; \Delta' \vdash e' : \psi}{\Gamma; \Delta, \Delta' \vdash let\ x = e\ in\ e' : \psi} LetUnrestricted \tag{2.22}$$

$$\frac{\Gamma; \Delta \vdash e : \sigma \qquad \Gamma; \Delta', x : \sigma \vdash e' : \psi}{\Gamma; \Delta, \Delta' \vdash let\ x = e\ in\ e' : \psi} LetLinear \tag{2.23}$$

Σχήμα 2.4: Σύστημα τύπων με γραμμικούς και μη τύπους

Οι γραμμικοί τύποι μας προσφέρουν τις κατάλληλες ιδιότητες για να έχουμε το παρακάτω interface

$$\begin{aligned} new &:: Nat \multimap Nat \multimap Arr, \\ read &:: Nat \multimap Arr \multimap (Val, Arr), \\ update &:: Nat \multimap Var \multimap Arr \multimap Arr, \\ free &:: Arr \multimap \mathrm{I} \end{aligned}$$

Για να χρησιμοποιήσει ο προγραμματιστής αυτό το interface απαιτείται μια ανιαρή δουλεία αντικατάστασης σε κάθε βήμα του παλιού πίνακα με τον νέο. Επίσης, βλέπουμε ότι δυο διαδοχικές αναγνώσεις πρέπει να σειριοποιηθούν πράγμα το οποίο δεν φαίνεται να είναι απαραίτητο. Για να μπορέσουμε να βελτιώσουμε το παραπάνω interface απαιτείται ένα read only construct για την χρήση των γραμμικών τιμών σε λειτουργία μόνο για ανάγνωση(ως μη γραμμικές δηλαδή).

2.5 Η γλώσσα Letbang

Ο Wadler στο paper του Linear Types can change the world [Wadl90] διακρίνει δύο είδη προσβάσεων σε μια μεταβλητή γραμμικού τύπου. Υπάρχει η πρόσβαση μόνο για ανάγνωση και η πρόσβαση που ανανεώνει την μεταβλητή. Εφαρμόζοντας αυτή την σκέψη στους πίνακες η συνάρτηση read είναι

πρόσβαση που κάνει απλά ανάγνωση και θα έπρεπε να μην καταναλώσει το resource του πίνακα ενώ η συνάρτηση update πρέπει να καταστρέφει τον πίνακα και να επιστρέφει έναν καινούριο πίνακα. Ο Wadler παρουσίασε μια γλώσσα στην οποία οι πίνακες έχουν γραμμικό τύπο οπότε πρέπει να χρησιμοποιηθούν ακριβώς μία φορά. Αυτό είναι ακριβώς αυτό που θέλουμε στην περίπτωση του update καθώς θέλουμε η λειτουργία εγγραφής στον πίνακα να καταστρέφει τον παλιό πίνακα. Ωστόσο στην περίπτωση του read δεν θα θέλαμε να καταστρέφεται ο πίνακας και κάθε φορά να μας επιστρέφεται καινούρια μεταβλητή για τον πίνακα. Ο Wadler για να ξεπεράσει αυτό το πρόβλημα όρισε ένα νέο συντακτικό construct για να επιτρέψει την ασφαλή ανάγνωση από τον πίνακα. Το construct αυτό ονομάζεται let! και συντάσσεται ως εξής

$$let!\ (x)\, y\ =\ u\, in\, v \tag{2.24}$$

Ο παραπάνω όρος αποτιμάται όπως ένα παραδοσιακό let δηλαδή πρώτα υπολογίζεται το u και το αποτέλεσμα του δένεται με το όνομα y και έπειτα υπολογίζεται το v στο extended περιβάλλον.Ωστόσο, η διαφορά με το let είναι ότι μέσα στον όρο u η μεταβλητή x έχει μη γραμμικό τύπο, ενώ μέσα στον όρο v έχει γραμμικό τύπο. Δηλαδή, μέσα στον όρο u δίνεται δυνατότητα πρόσβασης μόνο για ανάγνωση στην μεταβλητή x. Το παραπάνω construct έχει, όμως, δύο σημαντικούς περιορισμούς: Ο πρώτος είναι ότι πρέπει η αποτίμηση του όρου u πρέπει να τελειώσει πλήρως πριν την αποτίμηση του όρου v, ο περιορισμός αυτός ονομάζεται hyperstrict evaluation. Ο περιορισμός αυτός είναι απαραίτητος για να είμαστε σίγουροι ότι δεν υπάρχουν references στην μεταβλητή x κατα την αποτίμηση του v. Ο δεύτερος περιορισμός είναι περιορισμός στους τύπους των x και u. Προφανώς δεν μπορεί η έκφραση u να είναι ίση με το x ή με κάποιο component του x. Τα components ενός τύπου ορίζονται ως εξής: Εαν ο τύπος είναι βασικός τύπος αποτελεί component του εαυτόυ του. Εαν ο τύπος είναι σύνθετος τότε τα components του είναι όλα τα components των συστατικών του.

Παρακάτω φαίνεται ο κανόνας τύπων

$$\frac{\Gamma_1, x : !\, T \vdash u : U \qquad \Gamma_2, x : T, y : U \vdash v : V}{\Gamma_1, \Gamma_2, x : \mathrm{T} \vdash let!\ (x)\, y\ =\ u\, in\, v : V}\, Let! \tag{2.25}$$

Παρατηρούμε ότι σε ένα typing judgement για να είναι well typed μια έκφραση let! πρέπει να υπάρχει στο περιβάλλον το x με γραμμικό τύπο αλλα και στην έκφραση v ξαναδίνεται το x με γραμμικό τύπο. Συνεπώς, όταν γίνεται ένα context split το let! καταναλώνει το x και το δίνει πίσω στον όρο v.

Με την χρήση του let! πάνω σε πίνακες ο Wadler μπορει να προσφέρει το παρακάτω interface

$$\begin{aligned} new &:: Arr, \\ read &:: Nat \to !\, Arr \to Val, \\ update &:: Nat \to Var \to Arr \to Arr, \\ free &:: Arr \to \mathrm{I} \end{aligned}$$

Παρατηρούμε ότι η συνάρτηση read παίρνει ως παράμετρο εναν πίνακα μη γραμμικού τύπου οπότε πρέπει να χρησιμοποιηθεί μέσα στο letbang construct έτσι ώστε να είναι ασφαλής η ανάγνωση.

2.6 Observer Types

Ο Odersky προσπάθησε να βελτιώσει το let! δημιουργώντας μια τρίτη κατηγορία τύπων τους τύπους παρατηρητή.Ενώ στα συστήματα που είδαμε μέχρι στιγμής έχουμε 2 qualifiers για τους τύπους, στο σύστημα τύπων του Odersky προστίθεται ένας τρίτος, ο observer. Απαιτείται και πάλι ένα let! construct αλλα σε αυτή την γλώσσα είναι πιο απλοποιημένο.

$$let!\ y\ =\ u\, in\, v \tag{2.26}$$

Στην έκφραση u οι γραμμικοί τύποι του περιβάλλοντος μετατρέπονται σε τύπους observer ενώ στο v είναι ξανά γραμμικοί. Το interface για πίνακες θα ήταν ως εξής

```
f :: Int
f = let a = newArray 10 10 in
    let! b = read a 5 in
    let () = freeArray a in
    b

update :: Array a -> !Nat -> !a -> Array a
read :: Observer (Array a) -> Nat -> a
```

Η χρήση του let! με observers φαίνεται ότι είναι πιο χρήσιμη απο το παραδοσιακό let! του wadler καθώς ο περιορισμός για τους τύπους τώρα είναι απλα ότι η μεταβλητή y δεν μπορεί να έχει qualifier Observer. Ωστόσο στο συγκεκριμένο construct έχουμε μετατροπή **όλων** των μεταβλητών γραμμικού τύπου σε τύπο παρατηρητή(σε αντίθεση με το παραδοσιακό let! που επιλέγαμε την τιμή). Συνεπώς, σε κάθε σημείο του προγραμμάτος μπορει να έχουμε περιβάλλον μόνο με τύπους παρατηρητή ή μόνο με γραμμικούς τύπους.

2.7 Letbang with scopes

Οι Παπασπύρου-Παπακυριάκου[Papa10] στο σύστημα τύπων τους αναιρούν τους περιορισμόυς του Wadler και του Odersky. Η βασική ιδέα είναι οτι θα υπάρχει ένας qualifier που θα είναι είτε γραμμικός είτε μη γραμμικός για κάθε τύπο και επίσης κάθε τύπος θα επισημειώνεται με ενα scope. Το scope ουσιαστικά αποτελεί μια επισημείωση της τοποθεσίας στην οποία μπορει να χρησιμοποιηθεί ένα resource. Οπότε, για να είναι well typed ένα πρόγραμμα πρέπει να έχει τους σωστούς τύπους καθώς και τα σωστά scopes. Το let! επεκτείνεται έτσι ώστε να γράφεται ως εξής:

$$at\ \eta\ let!\ (x = e)\ y = e_1\ in\ e_2 \tag{2.27}$$

Όπου η έκφραση e υπολογίζεται και το αποτέλεσμα δένεται στην μεταβλητή x. Η μεταβλητή x χρησιμοποιείται ως unrestricted μέσα στην έκφραση e1 ενώ χρησιμοποιείται ως linear στην έκφραση e2. Το αποτέλεσμα του e1 δένεται με την μεταβλητή y που μπορεί να χρησιμοποιηθεί μέσα στο e2. Ουσιαστικά η ιδέα με τα scopes δεν απαγορεύει την διαφυγή αλλα απαγορεύει ο προγραμματιστής να κάνει κάτι με μια τιμή που έχει διαφύγει. Ένα χαρακτηριστικό παράδειγμα φαίνεται παρακάτω

```
at η let! (r = newArray 10 10) y = r in
let () = freeArray r in
read y 5
```

Το παραπάνω παράδειγμα προκαλεί σφάλμα στον έλεγχο τύπων και ειδικότερα προκαλεί σφάλμα scope καθώς το scope της μεταβλητής y που είναι η δεν είναι έγκυρο όταν γίνεται η αναγνωσή της, δηλαδή στο in του let! construct. Γενικότερα οι τρόποι που μπορεί να διαφύγει ένας πίνακας που έχει γίνει unrestricted είναι είτε μέσω εγγραφής σε άλλον πίνακα είτε μέσω ξεκρέμαστης αναφοράς σε κάποιο closure. Η προσέγγιση του let! with scopes λύνει και τα 2 αυτά προβλήματα.

```
at η let! (r = newArray 10 10)
y = newArray 1 r in
let () = freeArray r in
read (read y 0) 0
```

Εδώ αποτυγχάνει ο έλεγχος τύπων καθώς στην έκφραση read y 0 εμφανίζεται το scope η το οποίο δεν είναι πια έγκυρο. Για τις συναρτήσεις τα πράγματα είναι λίγο πιο περίπλοκα. Κάθε τύπος συνάρτησης επισημειώνεται με όλα τα scopes των μεταβλητών που χρησιμοποιεί στο σώμα της. Στην εφαρμογή της συνάρτησης πρέπει να ελεγχθεί ότι όλα τα scopes που έχουν επισημειωθεί είναι έγκυρα.

```
at η let! (r = newArray 10 10)
y = λ u : (). read r 2 in
let () = freeArray r in
y ()
```

Επίσης η γλώσσα letbang-with-scopes δεν έχει συμπερασμό τύπων οπότε είναι ευθύνη του προγραμματιστή να γράψει τους τύπους των συναρτήσεων μαζί με τα επισημειωμένα scopes. Ωστόσο, ενα πλεονέκτημα της προσέγγισης letbang with scopes είναι ότι όπως και στις σύγχρονες συναρτησιακές γλώσσες προγραμματισμού, υπάρχει πλούσια πληροφορία για τους τύπους στην διάρκεια της μεταγλώττισης αλλα δεν υπάρχει καμία πληροφοριών τύπων στην εκτέλεση. Έτσι δεν υπάρχει κατα την διάρκεια της εκτέλεσης πληροφορία για τύπους και scopes οπότε δεν υπάρχει performance penalty.

Αξίζει να σημειωθεί ότι οι παραπάνω προσεγγίσεις (let!,observers,let! with scopes) επιτρέπουν την ελεύθερη χρήση μιας γραμμικής τιμής. Ωστόσο, παρατηρούμε ότι αφενός δεν μπορούν να μετατραπούν συναρτήσεις που είναι γραμμικές σε unrestricted καθώς μπορεί να έχουν references σε γραμμικές τιμές στα closures τους, και αφετέρου σε αυτή την εργασία μας ενδιαφέρει η χρήση πινάκων που έχουν γραμμικό τύπο σε λειτουργία μόνο ανάγνωσης.

Κεφάλαιο 3

Η γλώσσα Letdo

Για να λυθούν τα προβλήματα που έχουν οι προηγούμενες υλοποιήσεις, μπορούμε να δούμε το πρόβλημα ως εξής. Έχουμε μια γλώσσα προγραμματισμού με γραμμικές και μη τιμές και θέλουμε να δώσουμε ένα readonly construct που λειτουργεί μόνο για τους πίνακες. Τα βασικά προβλήματα των προηγούμενων υλοποιήσεων έχουν να κάνουν με τον συμπερασμό τύπων και την διαφυγή references. Ο λόγος για τον οποίο συμβαίνει αυτό είναι επειδή μια τιμή γραμμικού τύπου μετατρέπεται σε unrestricted και πρέπει η γλώσσα να έχει περιορισμούς για να μην ξεφύγει κάποιο reference. Η γλώσσα Letdo προτείνει μια εναλλακτική μορφή του let! στο οποίο δεν υπάρχει πλέον τιμή μη γραμμικού τύπου αλλα υπάρχουν operations που μπορούν να εφαρμοστούν έτσι ώστε να δοθεί read only access

Για να λύσουμε το πρόβλημα μπορούμε να μελετήσουμε τις κυριότερες συναρτησιακές γλώσσες προγραμματισμού. Η γλώσσα OCaml υποστηρίζει references με το παρακάτω προστακτικό interface, το οποίο όμως επιτρέπει ασφαλές update.

```
let r = ref 42 in
r := v
```

Μπορούμε να θεωρήσουμε το reference ως έναν πίνακα μιας μοναδικής θέσης. Η ανάθεση σε μια μεταβλητή έχει τύπο επιστροφής unit καθώς την κάνουμε μόνο για το side effect της που είναι η ανανέωση του reference. Εαν σκεφτούμε πως μας δίνεται ενα functional interface για references και πρέπει να υλοποιήσουμε τον παραπάνω κώδικα, μάλλον θα το κάναμε ως εξής

```
let r = ref 42 in
let r = update r v in
()
```

Το ”κλειδί” εδώ είναι ότι το νέο r επισκιάζει το παλιό r. Εάν δεν επισκιάζαμε το παλιο r στον κώδικα τότε ο κώδικας δεν είναι ασφαλής καθώς το update είναι destructive. Αυτή η λεπτομέρεια ”απόκρυψης” του παλιου r αποτελεί την έμπνευση για το letdo construct το οποίο θα επιτρέψει να γίνει ανάγνωση αλλα δεν θα επιτρέψει διαφυγή γιατι θα ”κρύψει” το r.

3.1 Read only access χωρίς lexical scoping του πίνακα

Το αρχικό let! ουσιαστικά είναι ενα συντακτικό construct που λειτουργεί ως let αλλά ορίζει 2 περιοχές στον κώδικα. Η πρώτη δίνει read only access στην μεταβλητή και η δεύτερη ειναι η περιοχή που υπολογίζουμε αποτελέσματα με μεταβλητές που ορίστηκαν στην πρώτη.

```
let!(r)
y = lookup r 0 + lookup r 1 in
let () = freeArray r in
y
```

Αυτό που παρατηρούμε είναι οτι στο παραπάνω code snippet ενώ στην πρώτη γραμμή ορίζεται ότι ο πίνακας r θα γίνει μη γραμμικός, για κάθε λειτουργία ανάγνωσης απαιτείται να γράψουμε ξανά απο ποιόν πίνακα θέλουμε να κάνουμε ανάγνωση.

Ένα ίδιο πρόγραμμα σε letdo θα έμοιαζε κάπως έτσι

```
letdo(r)
  a <- read 0
  b <- read 1
in
    let () = freeArray r in
    a + b
```

Πλέον στο πρώτο block του letdo μπορούμε να κάνουμε ανάγνωση χωρίς να χρειάζεται να γράψουμε ότι αναφερόμαστε στο r καθώς όλες οι αναγνώσεις θα είναι πάνω στον πίνακα r. Συνεπώς, στο πρώτο block του letdo το περιβάλλον δεν περιέχει **καθόλου** την μεταβλητή r οπότε εξ ορισμού δεν μπορεί να διαφύγει κάποιο reference σε αυτήν. Αυτό φαίνεται καλύτερα στο παρακάτω παράδειγμα που προκαλεί σφάλμα τύπων καθώς δεν υπάρχει το r στο context μέσα στο letdo block.

```
letdo(r)
  a <- read (f r)
in
    a
```

Επίσης η σημασιολογία του letdo construct ουσιαστικά συνδέει το τελευταίο read statement με το πιο κοντινό πίνακα, και συνεπώς επιτρέπονται εμφωλευμένα letdo statements.

```
let r' = new 40 44 in
let r = new 41 43 in
letdo (r)
    a <- read (
        letdo(r')
            b <- read 0
        in
            let x = free r' in b)
in
    free r
```

Ένα σημαντικό πλεονέκτημα του συστήματος τύπων είναι ότι δεν έχουμε πια qualifiers στους τύπους. Μια τιμή τύπου πίνακα δηλαδή είναι πάντα γραμμική, εφόσον δεν υπάρχουν μηχανισμοί για να μετατραπεί μια γραμμική τιμή σε μη γραμμική. Διατηρείται δηλαδή η ιδιότητα ότι η τομή των συνόλων των γραμμικών και μη γραμμικών τύπων είναι το κενό σύνολο. Ως αποτέλεσμα, ο κανόνας τύπων για το letdo δεν είναι απαραίτητο να περιέχει κάποια ειδική συνθήκη για τους τύπους όπως το αρχικό letbang.

$$\frac{\Gamma, \forall i \in [0,m]\ x_i : \alpha; \Delta_1, r : Array\ \alpha \vdash u : \psi \qquad \Gamma; \Delta_2 \vdash \forall i \in [0,m]\ e_i : nat}{\begin{array}{l}\Gamma; \Delta_1, \Delta_2 \vdash \\ \quad letdo(r) \\ \qquad x_0 \leftarrow read\ e_0 \\ \qquad \ldots \\ \qquad x_m \leftarrow read\ e_m \\ \quad in \\ \qquad u : \psi\end{array}} LetDo \tag{3.1}$$

Παρατηρούμε ότι στον έλεγχο τύπων για τα e_i δεν υπάρχει στο περιβάλλον ο πίνακας, ενώ στον έλεγχο τύπων του u έχουμε ξάνα τον πίνακα.

3.2 Τύποι συναρτήσεων

Το letdo επιτρέπει την άρση μερικών περιορισμών που θέτει το letbang και οι τύποι παρατηρητή. Ωστόσο, εφόσον η γλώσσα έχει ένα μικτό σύστημα τύπων πρέπει να αντιμετωπιστεί το πρόβλημα χρήσης γραμμικής τιμής σε closure συνάρτησης. Μια συνάρτηση που χρησιμοποιεί στο closure της γραμμικής τιμή πρέπει να εφαρμοστεί το πολύ μία φορά. Αντίθετα μια συνάρτηση που δεν χρησιμοποιεί γραμμικές τιμές στο closure της μπορεί να εφαρμοστεί όσες φορές θέλει ο προγραμματιστής.

```
let r = newArray in
let f = \x : (). free r in
f ()
```

Ο παραπάνω κώδικας δίνει στην συνάρτηση f τύπο () -o (). Οπότε η συνάρτηση f μπορεί να χρησιμοποιηθεί μονάχα μία φορά. Ως αποτέλεσμα ο πίνακας r δεν μπορεί να απελευθερωθεί 2 φορές.

```
let r = newArray in
let f = \x : (). free r in
let a = f () in
f ()
```

Ο παραπάνω κώδικας δημιουργεί σφάλμα τύπων καθώς η συνάρτηση f χρησιμοποιείται 2 φορές, και έχει γραμμικό τύπο καθώς χρησιμοποιεί το r στο closure της.

Είναι σημαντικό επίσης να κατανοήσουμε ότι μια συνάρτηση μπορεί να έχει τύπο a -> b ή τύπο a -o b ανεξάρτητα από το αν είναι γραμμικοί τύποι τα a και b. Οπότε επιτρέπεται μια συνάρτηση που αθροίζει τα στοιχεία ενός πίνακα, τον απελευθερώνει και επιστρέφει το άθροισμα, να χρησιμοποιηθεί πολλές φορές εφόσον θα έχει τύπο Array Int -> Int.

```
let r = newArray in
let r' = newArray in
let f = \x : Array Int. free x in
let a = f r in
f r'
```

Ο βασικός περιορισμός όμως της χρήσης μια συνάρτησης μη γραμμικού τύπου(->) είναι οτι αν παίρνει ως παράμετρο μια γραμμική τιμή οφείλει να χρησιμοποιήσει στο σώμα της την γραμμική τιμή ακριβώς μία φορά. Αρχικά φαίνεται ότι ο περιορισμός αυτός περιορίζει το modularity του κώδικα. Σαν απλή λύση όμως η συνάρτηση sum να αλλάξει τύπο προσθέτοντας ενα continuation. Δηλαδή ο προγραμματιστής να περάσει ενα continuation το οποίο ειναι μια συνάρτηση στην οποία θα ορίσει τι πρέπει να γίνει μετά με το αποτέλεσμα της άθροισης και με τον πίνακα

```
f :: Array Int -> (Array Int -> Int -> Int) -> Int
f = \r : Array Int. \cont : (Array Int -> Int -> Int).
    letdo(r)
        a <- read 0
    in
        cont r a

let arr = newArray in
let firstElement = f arr ( \r : Array Int. \a :Int. let () = freeArray r in a) in
firstElement
```

Ωστόσο, το continuation passing style δεν είναι κάτι καινούριο για τις συναρτησιακές γλώσσες προγραμματισμού. Στην γλώσσα Haskell ο υπολογισμός οργανώνεται με κάποια constructs που ονομάζονται Monads. Τα monads ουσιαστικά αναπαριστούν τον διαχωρισμό του χρόνου σύνθεσης του υπολογισμού με τον χρόνο εκτέλεσης. Για να οριστεί ένα Monad απαιτούνται δύο συναρτήσεις. Η μια εκ των δύο ονομάζεται bind και έχει το παρακάτω type signature

```
(>>=) :: m a -> (a -> m b) -> m b
```

Η συνάρτηση bind παίρνει εναν υπολογισμό που όταν εκτελεστεί θα παράξει ενα a και μια συνάρτηση που αποτελεί το continuation δηλαδή τι θα κάνει με αυτό το a και επιστρέφει έναν σύνθετο υπολογισμό που θα παράξει ενα b. Συνεπώς έτσι δομούνται και οι συναρτήσεις που κάνουνε πράξεις με πίνακες. Έχουμε εναν υπολογισμο που παράγει ενα αποτέλεσμα και έπειτα περνάμε τον πίνακα στο continuation που θα κάνει τον επόμενο υπολογισμό.

3.3 Συστημα τύπων

Η σύνταξη των τύπων φαίνεται παρακάτω Παρατηρούμε ότι δεν υπάρχει συντακτικά τρόπος να

$$
\begin{aligned}
\varphi, \psi &::= \sigma \mid \alpha \\
\sigma, \tau &::= Array\ \alpha \mid \varphi \multimap \psi \\
\alpha, \beta &::= () \mid nat \mid \varphi \to \psi
\end{aligned}
$$

Σχήμα 3.1: Σύνταξη Τύπων

εκφραστεί πίνακας με μη γραμμικό τύπο και γενικότερα δεν υπάρχει τελεστης όπως το ! που ορίζεται στην γλώσσα του letbang.

3.4 Λειτουργική σημασιολογία

Παρακάτω φαίνεται η λειτουργική σημασιολογία της γλώσσας letdo

$$\frac{}{x : T \vdash x : T} LVar \tag{3.2}$$

$$\frac{}{\Gamma, x : \alpha; \emptyset \vdash x : \alpha} UVar \tag{3.3}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma.\, u : \sigma \multimap \psi} AbsLL \tag{3.4}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma.\, u : \sigma \multimap \psi} AbsLU \tag{3.5}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma.\, u : \sigma \multimap \psi} AbsUL \tag{3.6}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma.\, u : \sigma \multimap \psi} AbsUU \tag{3.7}$$

$$\frac{\Gamma; \Delta 1 \vdash f : \varphi \multimap \psi \qquad \Gamma; \Delta 2 \vdash u : \varphi}{\Gamma; \Delta 1, \Delta 2 \vdash f\ u : \psi} AppL \tag{3.8}$$

$$\frac{\Gamma; \Delta, x : \sigma \vdash u : \psi}{\Gamma; \Delta \vdash \lambda x : \sigma.\, u : \sigma \multimap \psi} AppU \tag{3.9}$$

$$\frac{\Gamma; \Delta \vdash e : a \qquad \Gamma, x : a; \Delta' \vdash e' : \psi}{\Gamma; \Delta, \Delta' \vdash let\ x = e\ in\ e' : \psi} LetUnrestricted \tag{3.10}$$

$$\frac{\Gamma; \Delta \vdash e : \sigma \qquad \Gamma; \Delta', x : \sigma \vdash e' : \psi}{\Gamma; \Delta, \Delta' \vdash let\ x = e\ in\ e' : \psi} LetLinear \tag{3.11}$$

$$\frac{\Gamma; \Delta_1 \vdash e : \alpha \qquad \Gamma; \Delta_2 \vdash n : Nat}{\Gamma; \Delta_1, \Delta_2 \vdash new\ n\ e : Array \alpha} ArrayI \tag{3.12}$$

$$\frac{\Gamma; \Delta \vdash r : Array\ \alpha}{\Gamma; \Delta \vdash free\ r : ()} ArrayE \tag{3.13}$$

$$\frac{\Gamma; \Delta_1 \vdash r : Array\ \alpha \qquad \Gamma; \Delta_2 \vdash e : a \qquad \Gamma; \Delta_3 \vdash i : Nat}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash update\ arr\ i\ e : Array\ \alpha} ArrayUpdate \tag{3.14}$$

$$\frac{\Gamma, \forall i \in [0, m]\ x_i : \alpha; \Delta_1, r : Array\ \alpha \vdash u : \psi \qquad \Gamma; \Delta_2 \vdash \forall i \in [0, m]\ e_i : nat}{\begin{array}{l} \Gamma; \Delta_1, \Delta_2 \vdash \\ \quad letdo(r) \\ \qquad x_0 \leftarrow read\ e_0 \\ \qquad ... \\ \qquad x_m \leftarrow read\ e_m \\ \quad in \\ \qquad u : \psi \end{array}} LetDo \tag{3.15}$$

Σχήμα 3.2: Σύστημα τύπων letdo

$$\frac{(f,m) \to (f',m')}{(fe,m) \to (f'e,m')} \tag{3.16}$$

$$\frac{(e,m) \to (e',m')}{((\lambda x.\zeta)e,m) \to ((\lambda x.\zeta)e',m')} \tag{3.17}$$

$$\frac{}{(\lambda x:\tau.\zeta)v \to [v/x]\zeta} \tag{3.18}$$

$$\frac{(n,m) \to (n',m')}{(new\, n\, e,m) \to (new\, n'\, e,m')} \tag{3.19}$$

$$\frac{(e,m) \to (e',m')}{(new\, v\, e,m) \to (new\, v\, e',m')} \tag{3.20}$$

$$\frac{j : loc_j\ fresh\ location, v\ : constant\ natural\ number}{(new\, v\, u,m) \to (loc_j, m\ \cup \forall x \in [0,n)\{(j,x) \mapsto u\})} \tag{3.21}$$

$$\frac{(a,m) \to (a',m')}{(update\ a\ i\ e,m) \to (update\ a'\ i\ e,m')} \tag{3.22}$$

$$\frac{(j,m) \to (j',m')}{(update\ loc_i\ j\ e,m) \to (update\ loc_i\ j'\ e,m')} \tag{3.23}$$

$$\frac{(e,m) \to (e',m')}{(update\ loc_i\ v\ e,m) \to (update\ loc_i\ v\ e',m')} \tag{3.24}$$

$$\frac{0 \Leftarrow v < length(loc_i)}{(update\ loc_i\ v\ u,m) \to (loc_i, m\{(i,v) \mapsto u\})} \tag{3.25}$$

$$\frac{\neg(0 \Leftarrow v < length(loc_i))}{(update\ loc_i\ v\ u,m) \to (loc_i,m)} \tag{3.26}$$

$$\frac{(a,m) \to (a',m')}{(free\ a,m) \to (free\ a',m')} \tag{3.27}$$

$$\frac{}{(free\ loc_i,m) \to ((), m\quad \forall j \in [0, length(loc_i))(i,j))} \tag{3.28}$$

$$\frac{(e,m) \to (e',m')}{(let\ x\ =\ e\ in\ e_2,m) \to (let\ x\ =\ e'\ in\ e_2,m')} \tag{3.29}$$

$$\frac{}{let\ x\ =\ v\ in\ e \to e[x := v]} \tag{3.30}$$

Σχήμα 3.3: Λειτουργική σημασιολογία letdo

$$\frac{(e,m) \rightarrow (e',m')}{(letdo(e)\ ...\ in\ u,m) \rightarrow (letdo(e')\ ...\ in\ u,m')} \tag{3.31}$$

$$\begin{aligned}
&\frac{(e_z,m) \rightarrow (e'_z,m')}{(letdo(r)}\\
&\qquad x_0 \leftarrow read\ e_0\\
&\qquad ...\\
&\qquad x_z \leftarrow read\ e_z\\
&in\\
&\qquad u,m) \rightarrow (\\
&letdo(r)\\
&\qquad x_0 \leftarrow read\ e_0\\
&\qquad ...\\
&\qquad x_z \leftarrow read\ e'_z\\
&in\\
&\qquad u,m)
\end{aligned} \tag{3.32}$$

$$\begin{aligned}
&\frac{\forall j \in [0,k]\ (m(i,v_j) = u_j\ for\ v_j \in [0,n)\ and\ u_j = m(i,0)\ otherwise\ (v_j\ out\ of\ bounds))}{(letdo(loc_i)}\\
&\qquad x_0 \leftarrow read\ v_0\\
&\qquad ...\\
&\qquad x_z \leftarrow read\ v_z\\
&in\\
&\qquad u,m) \rightarrow (\forall j \in [0,k][u_j/x_j]u,m)
\end{aligned} \tag{3.33}$$

Σχήμα 3.4: Λειτουργική σημασιολογία letdo(Συνέχεια)

Κεφάλαιο 4

Μετασχηματισμός μιας αγνής συναρτησιακής γλώσσας σε Letdo

Όπως είδαμε η γλώσσα letdo επιτρέπει την υλοποίηση destructive update με ασφαλή τρόπο. Όμως ο αρχικός μας στόχος είναι να γράψουμε αγνά συναρτησιακό κώδικα στον οποίο θα υλοποιηθεί το destructive update σαν optimisation. Οπότε η διαδικασία που θα θέλαμε να κάνει ο compiler είναι να μετατρέψει τον αγνό συναρτησιακό κώδικα σε letdo να γίνει ο έλεγχος τύπων και εάν το πρόγραμμα έχει σωστούς τύπους να παραχθεί κώδικας που υλοποιεί destructive update για τους πίνακες

4.1 Αναγνώσεις απο πίνακα

Γνωρίζουμε απο την δομή της γλωσσας letdo ότι σε ένα πρόγραμμα που δεν έχει αναγνώσεις απο τον πίνακα το αρχικό πρόγραμμα είναι σωστό πρόγραμμα για την γλώσσα letdo. Το σημείο που διαφοροποιείται η αρχική και η ενδιάμεση γλώσσα είναι η ανάγνωση απο τον πίνακα. Στην αρχική γλώσσα ο προγραμματιστής χρησιμοποιεί την συνάρτηση read χωρίς κάποιο sequencing. Στην ενδιάμεση γλώσσα, εφόσον ο πίνακας έχει γραμμικό τύπο, πρέπει να υπάρχει κάποιο sequencing. Η παραπάνω παρατήρηση φαίνεται καλύτερα με το παρακάτω παράδειγμα. Αρχικά βλέπουμε τον κώδικα της αρχικής γλώσσας:

```
let arr = new 10 42 in
read r 0 + read r 1
```

Και ο κώδικας της ενδιάμεσης γλώσσας:

```
let arr = new 10 42 in
letdo(r)
    a <- read 0
    b <- read 1
in
    a + b
```

Για να γίνει λοιπόν αλγοριθμικά η μετατροπή απο την αρχική στην ενδιάμεση γλώσσα απαιτείται να δώσουμε ονόματα στα 2 read. Δηλαδή κάθε read αρχικά να μετατραπεί σε μια έκφραση let. Προφανώς, βλέπουμε ότι η μετατροπή αυτή σειριοποιεί τον κώδικα. Αν όμως παρατηρήσουμε πιο προσεκτικά βλέπουμε ότι ένα πρόγραμμα το οποίο κάνει υπολογισμούς με πίνακες έχει μια εγγενη σειρά εκτέλεσης καθώς ένα read πρέπει να γίνει υποχρεωτικά πριν απο κάποιο update. Επίσης, η σημασιολογία της έκφρασης letdo, επιτρέπει να εκτελεστούν οι αναγνώσεις με οποιαδήποτε σειρά μεσα στο read block.

4.2 Σύνταξη εκφράσεων της αρχικής γλώσσας

Η αρχική γλώσσα θέλουμε να είναι σαν μια απλή αγνή συναρτησιακή γλώσσα, έχουμε σταθερές, μεταβλητές, συναρτήσεις και εφαρμογή συναρτήσεων. Σε ότι αφορά τους πίνακες, θα υπάρχει κατάλληλος τύπος για τον πίνακα και ένα σύνολο απο 4 συναρτήσεις για την δημιουργία νέου πίνακα, την

ανάγνωση την εγγραφή και την απελευθέρωση πίνακα. Τυπικά η γραμματική της αρχικής γλώσσας φαίνεται παρακάτω Παρατηρούμε ότι το interface για τους πίνακες δεν είναι built-in στην γλώσσα

$$\begin{aligned} e ::= & \; x \\ | & \; \lambda x.v \\ | & \; (f\ u) \\ | & \; (e, e) \end{aligned}$$

Σχήμα 4.1: Γραμματική αρχικής γλώσσας

αφού θα θέλαμε η γλώσσα αυτή να έχει τους πίνακες ως μια βιβλιοθήκη την οποία θα κάνει import ο προγραμματιστής.

4.3 Σύνταξη εκφράσεων της ενδιάμεσης γλώσσας

Για την διευκόλυνση της μετατροπής της αρχικής γλώσσας στην ενδιάμεση αρκεί να παρατηρήσουμε το εξής: Έστω e μια σύνθετη έκφραση της αρχικής γλώσσας που μπορεί να περιέχει τελεστές μεταβλητές συναρτήσεις και εφαρμογή συναρτήσεων, η ίδια έκφραση όταν γραφεί στην νέα γλώσσα, θα αντιστοιχεί σε μια λίστα απο let statements και ένα τελικό in στο οποίο θα περιέχεται η αρχική έκφραση με όποιες πιθανές αντικαταστάσεις είναι απαραίτητες. Παρακάτω μπορούμε να δούμε την δομή των εκφράσεων της ενδιάμεσης γλώσσας σε μορφή τύπου δεδομένων Haskell.

```
data IAtom =
    INum Int
    | IVar Var
    | IPlus IAtom IAtom
    | INew IAtom IAtom
    | IAbs Var IExpr
    | IApp IAtom IAtom

data ILet =
    ILetRead Var IAtom IAtom
    | ILetWrite Var IAtom IAtom IAtom
    | ILet Var IAtom

type IExpr = ([ILet], IAtom)
```

Ο τύπος IExpr είναι ουσιαστικά ενα ζευγος απο μια λίστα απο ILet και ένα τελικό IAtom. Τα ILet είναι let statements είτε για ανάγνωση απο πίνακα, είτε για εγγραφή είτε συμβατικά let. Τα IAtom είναι απλές εκφράσεις όπως ορισμός συνάρτησης, εφαρμογη συνάρτησης. Το IAtom παρατηρούμε ότι δεν μπορεί να είναι let statement. Παρακάτω μπορούμε να δούμε σε (απλοποιημένη) Haskell τον κώδικα που μετατρέπει την αρχική γλώσσα στην ενδιάμεση

```
trans :: Expr -> IExpr
trans (Num n) = ([], INum n)
trans (Var x) = ([], IVar x)
trans (Plus e1 e2) = (l1 ++ l2, IPlus a1 a2)
  where
    (l1, a1) = trans e1
    (l2, a2) = trans e2
trans (New e1 e2) = (l1 ++ l2, INew a1 a2)
```

```
  where
    (l1, a1) = trans e1
    (l2, a2) = trans e2
trans (Read e1 e2) = (l1 ++ l2 ++ [ILetRead freshVar a1 a2], IVar freshVar)
  where
    (l1, a1) = trans e1
    (l2, a2) = trans e2
trans (Write e1 e2 e3) = (l1 ++ l2 ++ l3 ++ [ILetWrite freshVar a1 a2 a3], IVar freshVar)
  where
    (l1, a1) = trans e1
    (l2, a2) = trans e2
    (l3, a3) = trans e3
trans (Let x e1 e2) = return (l1 ++ [ILet x a1] ++ l2, a2)
  where
    (l1, a1) = trans e1
    (l2, a2) = trans e2
trans (Abs x b) = ([], IAbs x b')
  where
    b' = trans b
trans (App f x) = (l1 ++ l2, IApp a1 a2)
  where
    (l1,a1) = trans f
    (l2,a2) = trans x
```

4.4 Απλοποίηση εκφράσεων ενδιάμεσης γλώσσας

Για να μπορέσουμε να μετατρέψουμε κατάλληλα τις αναγνώσεις σε letdo statements δημιουργήσαμε τις εκφράσεις τύπου IExpr. Αν σκεφτούμε όμως την γλώσσα με letdo δεν είναι μονο οι IExpr οι μόνες εκφράσεις που μπορούν να γραφούν. Συνεπώς ο τύπος IExpr αναπαριστά τις εκφράσεις που μπορει να παράξει η συνάρτηση transform και όχι όλες τις εκφράσεις της γλώσσας letdo. Οπότε για την απλότητα των επόμενων σταδίων μεταγλώττισης θα μετατρέψουμε τα IExpr σε κανονικές εκφράσεις της γλώσσας Letdo. Παρακάτω μπορούμε να δούμε τον τύπο δεδομένων που αναπαριστά την σύνταξη της τελικής γλώσσας.

```
data FExpr = FNum Int
    | FVar Var
    | FPlus FExpr FExpr
    | FNew FExpr FExpr
    | FLetDo FExpr Var FExpr FExpr
    | FWrite FExpr FExpr FExpr
    | FLet Var FExpr FExpr
    | FAbs Var FExpr
    | FApp FExpr FExpr
```

Το τελευταίο βήμα του μετασχηματισμού είναι το AST που είναι στην μορφή της ενδιάμεσης γλώσσας να φτάσει στην μορφή της τελικής γλώσσας για να γίνει πάνω σε αυτήν ο συμπερασμός τύπων. Ο κώδικας μετατροπής απο την ενδιάμεση γλώσσα είναι εύκολος καθώς η τελική γλώσσα είναι μια πιο ”γενική” μορφή της ενδιάμεσης.

```
final :: IExpr -> FExpr
final ([],a) = fAtom a
final (l:xs,a) = case l of
```

```
        ILetRead x r i       -> FLetDo (fAtom r) x (fAtom i) (final (xs,a))
        ILetWrite x e1 e2 e3 -> FLet x
            (FWrite (fAtom e1) (fAtom e2) (fAtom e3)) (final (xs,a))
        ILet x e             -> FLet x (fAtom e) (final (xs,a))

fAtom :: IAtom -> FExpr
fAtom a = case a of
    INum n     -> FNum n
    IVar x     -> FVar x
    IPlus aa b -> FPlus (fAtom aa) (fAtom b)
    INew n e   -> FNew (fAtom n) (fAtom e)
    IAbs x b   -> FAbs x (final b)
    IApp f e   -> FApp (fAtom f) (fAtom e)
```

Μετά το τέλος του μετασχηματισμού, ο μεταγλώττιστης της γλώσσας μπορεί να προχωρήσει στον έλεγχο/συμπερασμό των τύπων και την παραγωγή τελικού κώδικα προς εκτέλεση. Η ενδιάμεση γλώσσα ουσιαστικά δημιουργεί μια φρέσκια μεταβλητή την οποία θέτει να είναι ίση με το αποτέλεσμα κάθε ανάγνωσης και εγγραφής. Συνεπώς, δεν έχουμε πια εκφράσεις ανάγνωσεις απο μόνες τους και όλες οι αναγνώσεις βρίσκονται σε μια έκφραση let. Η έκφραση αυτή ισοδυναμεί με ένα letdo που έχει μόνο μία ανάγνωση. Το γεγονός ότι έχουμε προσθέσει κάποια let statements μας οδηγεί στο συμπέρασμα ότι η έκφραση έχει μια συγκεκριμένη σειρά που θα γίνουν οι υπολογισμοί πάνω στους πίνακες. Μια βελτιστοποίηση που μειώνει την σειριοποιησιμότητα των πράξεων είναι να προσπαθήσουμε να ενώσουμε τις εκφράσεις letdo μεταξύ τους. Γνωρίζουμε ότι μια έκφραση στην ενδιάμεση γλώσσα είναι μια ακολουθία απο ILet που καταλήγει σε ένα IAtom. Η βελτιστοποιήση υλοποιείται ως εξής. Στην λίστα απο ILet βρίσκουμε τις ακολουθίες συνεχόμενων ILetRead που αναφέρονται στον ίδιο πίνακα και δεν χρησιμοποιούν τις μεταβλητές που ορίζονται στα προηγούμενα ILetRead και τις συνενώνουμε σε ένα letdo με πολλαπλές αναγνώσεις. Λόγω των γραμμικών τύπων γνωρίζουμε ήδη ότι κάθε πίνακας έχει ένα μοναδικό reference πάνω του. Συνεπώς, γνωρίζουμε ότι εαν πριν ή μετα απο μια ακολουθία αναγνώσεων στον πίνακα A υπάρχει μια ακολουθία αναγνώσεων σε διαφορετικό πίνακα B τότε πρόκειται για διαφορετικούς πίνακες οπότε δεν μπορούμε να γράψουμε ένα μεγαλύτερο letdo statement που υλοποιεί και τις δύο λίστες αναγνώσεων.

Κεφάλαιο 5

Συμπερασμός τύπων στην ενδιάμεση γλώσσα

Ο συμπερασμός τύπων είναι η αυτοματοποιημένη διαδικασία με την οποία ο μεταγλωττιστής της γλώσσας προγραμματισμού μετασχηματίζει εκφράσεις χωρίς τύπους(ή με ελλιπείς τύπους) σε εκφράσεις με σωστούς τύπους. Ο στόχος μας είναι να πάρουμε κώδικα σε μια αγνή συναρτησιακή γλώσσα προγραμματισμού(χωρίς επισημειώσεις τύπων) και να υλοποιήσουμε destructive update στους πίνακες όπου είναι ασφαλές. Ο έλεγχος τύπων θα γίνει στην ενδιάμεση γλώσσα, οπότε απαιτείται ένας αλγόριθμος συμπερασμού τύπων.

5.1 Αλγοριθμικός έλεγχος τύπων για την ενδιάμεση γλώσσα

Εφόσον έχουμε ένα σύστημα τύπων για την ενδιάμεση γλώσσα, το επόμενο βήμα είναι να γράψουμε έναν αλγόριθμο ο οποίος θα ελέγχει αν οι τύποι είναι ορθοί, σε ένα πρόγραμμα που έχει επισημειώσεις τύπων. Πρέπει, δηλαδή να γίνει έλεγχος τύπων με αλγοριθμικό τρόπο. Οι κανόνες που δόθηκαν για το γραμμικό σύστημα είναι μη ντετερμινιστικοί και δεν μπορούν να εφαρμοστούν. Το βασικό στοιχείο που προκαλεί μη ντετερμινισμό είναι το context splitting. Όταν έχουμε ενα context Γ το οποίο πρέπει να διαχωριστεί σε δυο contexts Γ1 και Γ2 δεν μπορούμε να γνωρίζουμε ποιά γραμμικά resources θα χρησιμοποιηθούν στο πρώτο και ποιά στο δεύτερο context. Απαιτείται να επαναδιατυπώσουμε τους κανόνες τύπων έτσι ώστε να αποφύγουμε αυτό το μη ντετερμινισμό.

Η βασική ιδέα είναι ότι αντί να διαχωρίσουμε τα contexts σε κομμάτια μπορούμε να περάσουμε ολόκληρο το context στην πρώτη υποέκφραση και να επεκτείνουμε την σχέση ελέγχου τύπων έτσι ώστε να επιστρέφει το αχρησιμοποίητο μέρος του αρχικού context. Το output δηλαδή μπορεί να χρησιμοποιηθεί για τον έλεγχο τύπων της επόμενης έκφρασης. Η ιδέα αυτή μαζί με μικρές αλλαγές στους κανόνες δίνουν ένα αλγόριθμο ελέγχου τύπων στην γλώσσα με γραμμικούς και μη τύπους. Αν εστιάσουμε στον κομμάτι των γραμμικών τύπων, παρατηρούμε ότι στο var rule του συστήματος τύπων έχουμε την αποτυχία σε 2 περιπτώσεις. Η πρώτη είναι όταν μια μεταβλητή έχει ήδη χρησιμοποιηθεί ηδη μια φορά, και συνεπώς θα έχουμε αποτυχία αν χρησιμοποιηθεί δεύτερη φορά επειδή δεν θα υπάρχει πια στο περιβάλλον λόγω του context splitting. Η δεύτερη περίπτωση είναι όταν μια γραμμική μεταβλητή δεν χρησιμοποιηθεί καμία φορά. Στην συγκεκριμένη περίπτωση θα έχουμε ένα μη κενό γραμμικό context στο typing tree οπότε και θα προκληθεί αποτυχία στον έλεγχο τύπων. Τις δύο αυτές περιπτώσεις στο σύστημα τύπων που παρουσιάστηκε τις καλύπτει ο κανόνας LVar και UVar. Ωστόσο, παρατηρούμε ότι στον αλγόριθμο ελέγχου τύπων ο κανόνας Var δεν μπορεί να ελέγξει οτι έχει σίγουρα χρησιμοποιηθεί η μεταβλητή, δηλαδή να ελέγξει την δεύτερη συνθήκη αποτυχίας. Αυτό συμβαίνει γιατί, εάν υπάρχει στο περιβάλλον μια γραμμική μεταβλητή x που δεν χρησιμοποιήθηκε, τότε ο αλγόριθμος λέει ότι ο κανόνας Var θα επιστρέψει μέσα στο output context την μεταβλητή x.

Για να λυθεί αυτό το πρόβλημα απαιτείται να μετατοπιστεί το σημείο που εντοπίζεται η αποτυχία για την μη χρήση μιας γραμμικής μεταβλητής. Εαν επιστρέψουμε στην high level περιγραφή των γραμμικών τύπων, αυτό που θέλουμε ο αλγόριθμος να κάνει είναι όταν ορίζεται μια γραμμική μεταβλητή είναι να ελέγχει όταν βγαίνει out of scope ότι έχει χρησιμοποιηθεί ακριβώς μία φορά. Η συνθήκη αποτυχίας δηλαδή μπορεί να ελεγχθεί στην έκφραση που όρισε την νέα γραμμική μεταβλητή(let, abstraction, pattern matching) ελέγχοντας πάντα στο τελευταίο βήμα ότι η μεταβλητή που έκανε introduce δεν βρίσκεται στο αχρησιμοποίητο context που επιστράφηκε απο τον έλεγχο των υπο-

εκφράσεων. Ο David Walker[Walk04] στο κεφάλαιο που γράφει για substructural type systems στο βιβλίο Advanced topics in Types and Programming Languages[Pier04], παρατηρεί ότι ο έλεγχος αυτός μπορεί να συνενωθεί με την πράξη της αφαίρεσης της νέας μεταβλητής απο το context. Προκειται δηλαδή για μια πράξη που μπορεί να οδηγήσει είτε σε αποτυχία είτε σε επιστροφή του output context. Στην Haskell η συνάρτηση αυτή θα είχε τύπο contextDifference :: Context -> Context -> Either Error Context. Υλποιώντας τον αλγόριθμο που προκύπτει απο το ανανεωμένο σύστημα τύπων μπορούμε να κάνουμε έλεγχο τύπων σε εκφράσεις μιας συγκεκριμένης μορφής της ενδιάμεσης γλώσσας. Παρακάτω μπορούμε να δούμε μια απλή υλοποίηση του παραπάνω αλγορίθμου για μια απλή γλώσσα με γραμμικούς και μη τύπους:

```haskell
check :: Environment -> Expr -> Either Error (Type,Environment)
check gamma (Num _)     = Right (Unrestricted UInt, gamma)
check gamma NewArray = Right (Linear TArray, gamma)
check gamma (Var x) = case M.lookup x gamma of
    Nothing -> Left $ "Variable : " ++ x ++ " doesn't exist in environment"
    Just t  -> case t of
        Linear _        -> Right (t, M.delete x gamma)
        Unrestricted _ -> Right (t, gamma)
check gamma (Let x e e') = do
    (t,gamma') <- check gamma e
    (t',gamma'') <- check (M.insert x t gamma') e'
    result <- differenceWithFailure gamma'' $ M.singleton x t
    return (t', result)
check gamma (Abs x t e) = do
    (t',gamma') <- check (M.insert x t gamma) e
    result <- differenceWithFailure gamma' $ M.singleton x t
    if result == gamma then
        return (Unrestricted (TFun t t'), result)
    else
        return (Linear (TLol t t'),result)
check gamma (App f e) = do
    (ft,gamma') <- check gamma f
    (argType,gamma'') <- check gamma' e
    case ft of
        Linear (TLol t t') -> do
            areEqualType argType t
            return (t',gamma'')
        Unrestricted (TFun t t') -> do
            areEqualType argType t
            return (t',gamma'')
        _ -> throw ( show ft ++ "doesn't have a function type")

-- | Difference with failure implements context difference.
-- This computation can result in failure.
differenceWithFailure :: Environment -> Environment -> Either Error Environment
differenceWithFailure gamma1 gamma2
    | isNonLinear $ M.intersection gamma1 gamma2 =
        Right $ contextDifference gamma1 (M.toList gamma2)
    | otherwise =
        Left $ "Error. The following linear resources were unused:" ++
            show (M.filter linear gamma2)
```

```
-- | contextDifference implements the algorithm assuming no linear resources are
-- shared between gamma_1 and gamma_2. This check is done by differenceWithFailure
contextDifference :: Environment -> [(Var, Type)] -> Environment
contextDifference g []                         = g
contextDifference g ((_,Linear _):xs)          = contextDifference g xs
contextDifference g ((x,Unrestricted _):xs)    = M.delete x (contextDifference g xs)
```

Παρατηρούμε ότι στην περίπτωση έκφρασης Var όταν πρόκειται για μεταβλητή γραμμικού τύπου διαγράφεται απο το περιβάλλον καθώς δεν μπορεί να χρησιμοποιηθεί άλλη φορά. Εαν ο προγραμματιστής επιχειρήσει να χρησιμοποιήσει την μεταβλητή δεύτερη φορά τότε θα προκληθεί σφάλμα τύπων καθώς δεν θα υπάρχει η μεταβλητή αυτή στο περιβάλλον. Παρατηρούμε ότι η πράξη διαφοράς μεταξύ των περιβαλλόντων έχει υλοποιηθεί με την συνάρτηση differenceWithFailure η οποία κάνει πρώτα τον έλεγχο για αποτυχία και έπειτα απλά εφαρμόζει την πράξη στα δύο περιβάλλοντα με την συνάρτηση contextDifference.

Η ιδιαιτερότητα αυτού του αλγορίθμου είναι ότι κάνει έλεγχο τύπων δηλαδή ελέγχει εάν οι τύποι είναι σωστοί σε εκφράσεις της ενδιάμεσης γλώσσας που στην περίπτωση του abstraction δίνουν συντακτικά τον τύπο του argument τους. Εφόσον, δεν γνωρίζουμε τον τύπο των argument στις συναρτήσεις δεν μπορούμε να εφαρμόσουμε αυτό τον αλγόριθμο στην γλώσσα που έχει οριστεί στο προηγούμενο κεφάλαιο. Εαν ο προγραμματιστής επισημείωνε τους τύπους σε κάθε έκφραση τότε θα μπορούσε να εφαρμοστεί ο αλγόριθμος για να γίνει έλεγχος των τύπων.

5.2 Συμπερασμός τύπων

Για τον συμπερασμό τύπων σε μια γλώσσα όπως η OCaml εφαρμόζεται ενας αλγόριθμος συμπερασμού τύπων σε ένα σύστημα τύπων που όνομάζεται Damas Hindley Milner. Η ιδιότητα του συστήματος τύπου αυτού είναι ότι υπάρχει αλγοριθμική διαδικασία με την οποία μπορεί να γίνει συμπερασμός του πίο γενικού τύπου μιας έκφρασης χωρίς να υπάρχει ανάγκη επισημειώσεων απο τον προγραμματιστή. Για τον υπολογισμό του πιο γενικού τύπου σε ένα σύστημα τύπων DHM υπάρχουν αρκετοί αλγόριθμοι.

Το πρόβλημα του συμπερασμού τύπων σε ένα σύστημα τύπων DHM ουσιαστικά αποτελείται απο 2 προβλήματα. Το πρώτο είναι η εύρεση των περιορισμών που προκύπτουν απο το πρόγραμμα. Το δεύτερο πρόβλημα είναι η επίλυση αυτών των περιορισμών έτσι ώστε να δοθεί στο πρόγραμμα ο πιο γενικός τύπος. Ο αλγόριθμος W ακολουθεί μια προσέγγιση στην οποία καθώς διασχίζει το abstract syntax tree δημιουργεί τους περιορισμούς τους οποίους λύνει με μια μέθοδο ενοποίησης αμέσως μετά την παραγωγή τους. Οπότε συνολικά ο αλγόριθμος W επιστρέφει τον τύπο της έκφρασης μαζί με ένα σύνολο αντιστοίχισης type variables σε τύπους. Ο Heereen μαζί με τον Hage και τον Swierstra [Heer02] δείχνουν εναν αλγόριθμο δύο βημάτων που αποτελεί γενίκευση του αλγορίθμου W που μπορεί να δώσει πιο χρήσιμα μηνύματα λάθους στον προγραμματιστή. Είναι γεγονός ότι η παραγωγή και επίλυση των constraints στο ίδιο βήμα προκαλεί το λεγόμενο left to right bias το οποίο είναι υπεύθυνο για περίεργα σφάλματα τύπων που δεν βοηθούν τον προγραμματιστή. Όμως, παρατηρούμε ότι η επίλυση ενός constraint αμέσως αφού παραχθεί ειναι μια ιδιότητα την οποία έχει το σύστημα DHM λόγω απλότητας. Σε ένα πιο σύνθετο σύστημα τύπων δεν υπάρχει σίγουρα αλγόριθμος που να επιλύει τα constraints αμέσως μετά την παραγωγή τους. Συνεπώς, με έμπνευση τον γενικευμένο αλγόριθμο δύο βημάτων του Heeren για την ενδιάμεση γλώσσα που περιέχει γραμμικούς τύπους θα εφαρμόσουμε έναν αλγόριθμο συμπερασμό τύπων δύο βημάτων στην ενδιάμεση γλώσσα. Το πρώτο βήμα θα παράγει ένα συνόλο απο constraints και το δεύτερο βήμα θα επιλύει τα constraints επιστρέφοντας τον τύπο της έκφρασης.

5.3 Περιορισμοί

Το πρώτο βήμα όταν θέλουμε να κάνουμε συμπερασμό τύπων είναι να επεκτείνουμε την γλώσσα των τύπων με μεταβλητές τύπων. Οι μεταβλητές τύπων είναι φρέσκοι τύποι που αναπαριστούν τύπους που δεν γνωρίζουμε μέχρι στιγμής. Ωστόσο στην γλώσσα μας εκτός απο τύπους έχουμε και qualifiers. Συνεπώς, πρεπει να προσθέσουμε και μεταβλητές qualifiers που και αυτοί με την σειρά τους θα αναπαριστούν τους άγνωστους μέχρι στιγμής qualifiers. Η σύνταξη των τύπων φαίνεται στο παρακάτω διάγραμμα.

$$\begin{aligned} \tau, \sigma &::= Array \mid Int \mid q\,(\sigma \to \tau) \mid q\,@n \\ q &::= Linear \mid Unrestricted \mid \#m \\ n, m &: Natural\ numbers \end{aligned}$$

Σχήμα 5.1: Σύνταξη Τύπων

Ο στόχος της διαδικασίας συμπερασμού τύπων είναι μια αντιστοίχιση όλων των άγνωστων τύπων και qualifiers με γνωστούς τύπους. Η διαδικασία έχει δύο στάδια, το πρώτο είναι η παραγωγή των περιορισμών και το δεύτερο είναι η επίλυση των περιορισμών.

Η διαδικασία παραγωγής των περιορισμών θα διασχίζει το AST και θα παράγει σε κάθε βήμα τους απαραίτητους περιορισμούς επιστρέφοντας την λίστα των περιορισμών των τύπων της υποέκφρασης και το νέο περιβάλλον. Απαιτείται κάθε υποέκφραση να επιστρέφει τις μεταβλητές που δεν έχουν χρησιμοποιηθεί όπως στον αλγόριθμο ελέγχου τύπων. Για σχεδιαστικούς λόγους το επόμενο βήμα για να λειτουργήσει ο αλγόριθμος είναι το περιβάλλον να μετράει πόσες φορές χρησιμοποιείται μια μεταβλητή. Στον προηγούμενο αλγόριθμο είχαμε την δυαδική πληροφορία ύπαρξης ή οχι μιας μεταβλητής στο περιβάλλον. Στον νέο αλγόριθμο το περιβάλλον θα περιέχει την πληροφορία τύπου κάθε μεταβλητής καθώς και τον αριθμό που έχει χρησιμοποιηθεί. Συνεπώς στην εφαρμογή του κανόνα Var για την μεταβλητή x θα επιστραφεί το περιβάλλον εισόδο με αυξημένο τον αριθμό που έχει χρησιμοποιηθεί κατα ένα. Η μετατροπή του περιβάλλοντος στην νέα του μορφή είναι απαραίτητη για να δουλέψει αλγόριθμος διότι σε κάθε βήμα δεν είναι γνωστός ο qualifier μιας μεταβλητής οπότε δεν μπορούμε να κωδικοποιήσουμε στο παλιό περιβάλλον την διαφορετική διαχείριση των γραμμικών και την μη γραμμικών μεταβλητών.

Οι περιορισμοί μπορούν να είναι 3 ειδών:

1. EqType τ τ'
   Ο περιορισμός αυτού του είδους ουσιαστικά σημαίνει ότι οι τύποι τ και τ' πρέπει να ενοποιηθούν σε κάποιο στάδιο του αλγορίθμου. Ο περιορισμός αυτός υπάρχει σε κάθε DHM σύστημα.

2. QualConst q Γ Γ'
   Ο περιορισμός αυτός καθορίζει την τιμή του qualifier q ο οποίος είναι ο qualifier ενός τύπου συνάρτησης. Γνωρίζουμε απο το σύστημα τύπων ότι εαν μια συνάρτηση χρησιμοποιεί στο closure της μια τιμή η οποία είναι γραμμικού τύπου τότε ο qualifier της είναι Linear. Ο περιορισμός αυτός δεν μπορεί να λυθεί στην πρώτη διάσχιση του AST καθώς στα περιβάλλοντα Γ και Γ' υπάρχουν μεταβλητές που δεν γνωρίζουμε αν είναι Linear ή Unrestricted.

3. TypeIs τ q
   Ο περιορισμός αυτός καθορίζει ότι ο τύπος τ πρέπει να έχει τον qualifier q. Ενας περιορισμός αυτού του τύπου μπορεί να παραχθεί σε δύο ειδών εκφράσεις, let και abs. Οι εκφράσεις let και abs ορίζουν μια νεα μεταβλητή την οποία χρησιμοποιούν σε ένα νεο block. Ο έλεγχος τύπων στο νέο αυτό block θα επιστρέψει ποσες φορές χρησιμοποιήθηκε η νέα μεταβλητή του περιβάλλοντος. Όταν η νέα μεταβλητή δεν χρησιμοποιηθεί καμία φορά ή όταν χρησιμοποιηθεί 2 ή περισσότερες φορές γνωρίζουμε ότι η μεταβλητή x πρέπει να έχει σίγουρα Unrestricted qualifier καθώς δεν μπορεί να είναι γραμμικός.

Παρακάτω μπορούμε να δούμε τον κώδικα Haskell που παράγει το συνολο των περιορισμών.

```
infer :: Environment -> Expr -> Infer (Environment, Type, [Constraint])
infer gamma expression = case expression of
    Num _     -> return (gamma, TInt, [])
    NewArray -> return (gamma, TArray, [])
    Var x -> case M.lookup x gamma of
        Nothing -> throwError $ "Variable " ++ x ++ " doesn't exist in the environment"
        Just (t,n)  -> return (M.insert x (t,n+1) gamma, t, [])
    Let x e e'  -> do
        (gamma',t,cs) <- infer gamma e
        (gamma'',t',cs') <- infer (M.insert x (t,0) gamma') e'
        let constraint = case M.lookup x gamma'' of
                Nothing        -> undefined
                Just (xType,n) -> [TypeIs xType Unrestricted | n /= 1]
        let result = M.delete x gamma''
        return (result, t',cs ++ cs' ++ constraint)
    App f x -> do
        (gamma',ft,cs) <- infer gamma f
        (gamma'',argt,cs') <- infer gamma' x
        t <- fresh
        q <- freshQual
        return (gamma'',t,cs ++ cs' ++ [EqType ft (TFun q argt t)])
    Abs x e -> do
        t <- fresh
        (gamma',rType,cs) <- infer (M.insert x (t,0) gamma) e
        q <- freshQual
        let
            constraint = case M.lookup x gamma' of
                Nothing    -> undefined
                Just (_,n) -> [TypeIs t Unrestricted | n /= 1]
        let gammaOut = M.delete x gamma'
        let arg = if null constraint then t
            else makeUnrestricted t
        return (gammaOut,TFun q arg rType,
            cs ++ [QualConst q gamma gammaOut] ++ constraint)
    LetDo x (Var r) e e' ->
        case M.lookup r gamma of
            Nothing -> throwError ""
            Just u@(arr,_) -> do
                (gamma',t,cs) <- infer (M.delete r gamma) e
                (gamma'',t',cs') <- infer
                    (M.insert x (TInt,0) (M.insert r u gamma')) e'
                return(M.delete x gamma'',t',
                    cs++[EqType arr TArray]++cs')
```

5.4 Επίλυση των περιορισμών

Η επίλυση των περιορισμών ουσιαστικά αποτελεί μια διαδικασία που δέχεται ως είσοδο μια λίστα απο περιορισμούς και μπορεί είτε να αποτύχει είτε να επιστρέψει μια αντικατάσταση. Η αντικατάσταση θα είναι ενα ζεύγος απο μια αντιστοίχιση των μεταβλητών τύπων σε κανονικούς τύπους και μια αντιστοίχιση των μεταβλητών qualifier σε κανονικά qualifiers. Κάθε περιορισμός εκτός του

QualConst μπορεί να λυθεί σε ένα περάσμα. Για να λυθεί ένας περιορισμός QualConst πρέπει να υπάρχει μεταβλητή που έχει γραμμικό τύπο με διαφορετικό αριθμό χρήσεων στα δύο περιβάλλοντα. Ο μόνος τρόπος να αποφανθούμε ότι το qualifier είναι Unrestricted είναι να γνωρίζουμε ότι δεν χρησιμοποιείται καμία μεταβλητή γραμμικού τύπου μέσα στο closure της συνάρτησης. Αυτό μπορει να γίνει είτε όταν τα 2 περιβάλλοντα ταυτίζονται πλήρως(δηλαδή δεν χρησιμοποιήθηκε τίποτα), είτε όταν γνωρίζουμε ότι κάθε μεταβλητή που χρησιμοποιείται στο closure της συνάρτησης είναι σίγουρα Unrestricted. Ο αλγόριθμος επίλυσης των constraints στο πρώτο πέρασμα της λίστας θα λύσει τους περιορισμούς TypeIs και EqType εφαρμόζοντας αλγορίθμους ενοποίησης των τύπων και των qualifier. Στο τέλος του πρώτου περάσματος θα έχουν απομείνει πιθανόν ορισμένοι QualConst περιορισμοί. Το επόμενο στάδιο προσπαθεί να λύσει τους εναπομείναντες m περιορισμούς QualConst. Ο αλγόριθμος συνεχίζει να προσπαθεί να επιλύσει τα constraints όσο σε κάθε επανάληψη της λίστας το μήκος της μειώνεται. Υπάρχει περίπτωση το μήκος της λίστας να μην μειωθεί. Σε αυτή την περίπτωση αυτό που συμβάινει είναι ότι ορίζονται συναρτήσεις που δεν γνωρίζουμε τον qualifier των παραμέτρων τους οι οποίες δεν καλούνται ποτέ. Σχεδιαστικά, κάναμε την επιλογή για να μπορέσουν να λυθούν αυτα τα αδιέξοδα όταν δεν υπάρχει πρόοδος ο qualifier του πρώτου περιορισμού να γίνεται Unrestricted και έπειτα ο αλγόριθμος να προσπαθεί να επιλύσει τα υπόλοιπα constraints. Η εναλλακτική σχεδιαστική επιλογή θα ήταν να προστεθεί πολυμορφισμός πάνω στους qualifiers.

Είναι γεγονός ότι χωρίς την ύπαρξη πολυμορφισμού είναι απαραίτητο να επιλέγεται ένας προεπιλεγμένος qualifier για να επιλυθούν οι περιορισμοί. Αυτό συμβαίνει γιατί, εάν δεν επιλεγεί τιμή για τον άγνωστο qualifier τότε κάθε συνάρτηση που χρησιμοποιεί την μεταβλητή στο closure της θα έχει και αυτή άγνωστο qualifier. Αυτό μπορούμε να το δούμε στο παρακάτω παράδειγμα

```
let magicFunc = \x -> \y -> x
```

Η παραπάνω συνάρτηση καταλήγει να έχει τύπο $U((\#1@0) \to \#4((U@2) \to (\#1@0)))$. Οι μεταβλητές που ξεκινούν με # είναι qualifiers ενώ οι μεταβλητές που ξεκινούν με @ είναι τύποι. Φαίνεται ότι αν δεν επιλεγεί τιμή για τον qualifier 1 δεν μπορεί να επιλεγεί τιμή για τον qualifier 4, διότι εάν ο qualifier 1 είναι Linear τότε η συνάρτηση λ y -> x πρέπει να έχει Linear qualifier καθώς χρησιμοποιεί την γραμμική μεταβλητή x στο closure της. Αντίθετα εάν ο qualifier 1 είναι Unrestricted τότε και ο qualifier 4 είναι unrestricted. Με αυτό το παράδειγμα μπορούμε να δούμε την αναγκαιότητα της επίλυσης των αδιεξόδων(επιλέγοντας default qualifier) μονάχα όταν όλα τα υπόλοιπα constraints είναι qualConst. Το παρακάτω παράδειγμα προκαλεί σφάλμα τύπων εάν ο qualifier επιλεχθεί στην αρχή αλλά η έκφραση είναι σωστή.

```
let magicFunc = \x -> \y -> x
let z = magicFunc newArray
```

Εαν επιλεγεί ο qualifier 1 στην αρχή να είναι Unrestricted τότε όταν γίνει ενοποίηση του τύπου της τυπικής παραμέτρου x με την πραγματική παράμετρο newArray, θα προκληθεί σφάλμα τύπων αφού δεν θα ταιριάζουν οι qualifiers. Ωστόσο εάν δεν κάνουμε επιλογή για τον qualifier 1 και αφήσουμε τα άλλα constraints να λυθούν πρώτα παίρνουμε το εξής αποτέλεσμα: $L((U@2) \to Array)$ το οποίο είναι το σωστό αποτέλεσμα. Το παράδειγμα αυτό συνεπώς μας δείχνει ότι είναι απαραίτητο ορισμένες φορές να προσπερνάμε τα QualConst που δεν μπορούν να λυθούν εκείνη την στιγμή και ότι η επίλυση των αδιεξόδων πρέπει να γίνεται μόνο όταν υπάρχει αδιέξοδο(δηλαδή όλα οι υπόλοιποι περιορισμοί να είναι qualConst). Ο ψευδοκώδικας που επιλύει την λίστα απο constraints φαίνεται παρακάτω. Δεν είναι λεπτομερής η διαχείριση των σφαλμάτων μέσα απο το Infer monad όπως στον κανονικό κώδικα

```
go :: [Constraint] -> Substitution
go ((EqType t t'):constraints) = go (apply s constraints) 'compose' s
    where
        s = unify t t'

go (TypeIs t q,constraints) = (solve (apply s constraints) 'compose' s)
    where
```

```haskell
        s = applyQual t q

go l@((QualConst q g g'):constraints)
    | all areQualConst l = try 0 (length l) l
    | otherwise = (solve (apply s constraints') `compose` s)
      where
          (constraints',s) = case qualConstraint q g g' of
              Nothing -> (constraints++[head l],emptySubst)
              Just s1 -> (constraints,s1)


qualConstraint :: Qualifier -> Environment -> Environment -> Maybe Substitution
qualConstraint (QVar qn) g g'
    | g == g' = Just (bind q Unrestricted)
    | any inClosure (filter (\(_,(t,_)) -> isLinear t) (M.toList g)) =
      Just (bind q Linear)
    | isEnvNonLinear g && isEnvNonLinear g' = Just (bind q Unrestricted)
    | otherwise = Nothing
    where

        inClosure :: (String,(Type,Int)) -> Bool
        inClosure (x,(t,n)) = n /= n'
            where
                (t',n') = g' M.! x
```

Στην πρώτη περίπτωση έχουμε μια εξίσωση τύπων όπως ακριβώς στο DHM σύστημα. Οι τύποι t και t' πρέπει να ενοποιηθούν με την χρήση της συνάρτησης unify. Η συνάρτηση unify ουσιαστικά εξισώνει τους 2 τύπους, εξισωνοντας και οποιονδήποτε qualifier προκύπτει μέσα στους τύπους. Εάν οι τύποι ή οι qualifiers είναι ασύμβατοι τότε προκαλεί σφάλμα τύπων. Στην δεύτερη περίπτωση θέλουμε ο τύπος t να έχει qualifier q. Έτσι χρησιμοποιείται μια συνάρτηση applyQual που προσπαθεί να κάνει αυτή την λειτουργία στον qualifier του τύπου t. Εάν είναι ήδη γνωστός ο qualifier του τύπου t τότε εάν είναι συμβατοί τότε επιστρέφεται η κενή αντικατάσταση, αλλιώς προκαλείται σφάλμα τύπων.

Η τρίτη περίπτωση είναι η πιο περίπλοκη. Έχουμε έναν qualifier μιας συνάρτησης και τα input και output περιβάλλοντα. Ο κανόνας τύπων ορίζει ότι εάν κάποια γραμμική μεταβλητή χρησιμοποιείται στο closure της συνάρτησης τότε πρέπει ο qualifier της συνάρτησης να είναι Linear. Η συνθήκη αυτή στον αλγόριθμο φαίνεται στο δεύτερο guard που ελέγχει εάν υπάρχει μια γραμμική μεταβλητή που έχει διαφορετικό αριθμό χρήσεων στο input και στο output context. Η πρώτη συνθήκη ελέγχει εάν το input και το output context ταυτίζονται τότε γνωρίζουμε ότι η συνάρτηση δεν χρησιμοποιεί καμία εξωτερική μεταβλητή στο closure της και ως αποτέλεσμα είναι Unrestricted. Η τρίτη συνθήκη ελέγχει ότι αν και τα δύο περιβάλλοντα είναι αμιγώς μη γραμμικά τότε δεν υπάρχουν γραμμικές μεταβλητές για να χρησιμοποιηθούν στο closure οπότε η συνάρτηση είναι Unrestricted. Η τελευταία συνθήκη(otherwise) προσθέτει πολυπλοκότητα στον αλγόριθμο καθώς εάν δεν ίσχυε καμία απο τις παραπάνω συνθήκες σημαίνει ότι δεν μπορεί να λυθεί ο περιορισμός αυτή την στιγμή. Στην περίπτωση που δεν μπορεί να λυθεί ο περιορισμός προστίθεται στο τέλος της λίστας των περιορισμών.

Η υλοποίηση του αλγορίθμου επίλυσης των περιορισμών σε Haskell φαίνεται παρακάτω:

```haskell
solve :: [Constraint] -> Infer Substitution
solve [] = return emptySubst
solve (c:cs) = go (c,cs)


go :: (Constraint,[Constraint]) -> Infer Substitution
go (EqType t t',constraints)      = do
```

```haskell
    s <- unify t t'
    s' <- solve (apply s <$> constraints)
    return (s' `compose` s)
go (a@(QualConst q g g'),constraints) =
    if all areQualConst (a : constraints) then
        -- They are all qualConst so we need to try while the list length
        -- is decreasing. if it doesn't we have a problem.
        try 0 (length (a:constraints)) (a:constraints) else
    do
        maybeS <- qualConstraint q g g'
        let
            (constraints',s) = case maybeS of
                -- Nothing means unsolved, add it to the back of the list
                Nothing -> (constraints++[a],emptySubst)
                -- Just means it is solved and we can move on
                Just s1 -> (constraints,s1)
        s' <- solve (apply s <$> constraints')
        return (s' `compose` s)
go (TypeIs t q,constraints)       = do
    s <- applyQual t q
    s' <- solve (apply s <$> constraints)
    return (s' `compose` s)


qualConstraint :: Qualifier -> Environment -> Environment -> Infer (Maybe Substitution)
qualConstraint (QVar qn) g g'
    | g == g' = Just <$> pureQBind qn Unrestricted
    | any inClosure (filter (\(_,(t,_)) -> isLinear t) (M.toList g)) =
    Just <$> pureQBind qn Linear
    | isEnvNonLinear g && isEnvNonLinear g' = Just <$> pureQBind qn Unrestricted
    | otherwise = return Nothing
    where
        inClosure :: (String,(Type,Int)) -> Bool
        inClosure (x,(t,n)) = n /= n'
            where
                (t',n') = g' M.! x
```

Το τελευταίο θέμα για τον αλγόριθμο συμπερασμού τύπων είναι ο πολυμορφισμός. Όπως αναφέραμε προηγουμένως για λόγους απλότητας στον συμπερασμό τύπων δεν υπάρχει πολυμορφισμός στους qualifiers. Σε ότι αφορά τον πολυμορφισμό τύπων ο αλγόριθμος επιτρέπει κάθε μεταβλητή που έχει qualifier Unrestricted να είναι πολυμορφική(προφανώς στους Unrestricted τύπους). Αυτό υλοποιείται στο τέλος της διαδικασίας, όταν δηλαδή έχει μείνει ένας τύπος της μορφή $U@1$ τότε προστίθεται το εξής $\forall a.U\ a$. Αξίζει να προσέξουμε ότι είναι ιδιότητα του συστήματός ότι δεν μπορεί να προκύψει σε καμία περίπτωση τύπος με qualifier Linear και άγνωστο type variable, αφού για να προκύψει Linear qualifier πρέπει να γίνει ενοποίηση με κάποιο γνωστό τύπο. Αντίθετα ενας Unrestricted qualifier μπορεί να προκύψει στην περίπτωση που κάποια μεταβλητή χρησιμοποιηθεί καμία ή περισσότερες απο μία φορές οπότε θα γνωρίζουμε τον qualifier της ενώ θα είναι άγνωστο το type variable της. Επίσης, πολλές φορές δημιουργείται κάποιος περιορισμός που ουσιαστικά σημαίνει ότι αν ο qualifier 1 είναι γραμμικός τοτε και ο qualifier 3 είναι γραμμικός. Η λάθος αντιμετώπιση εδώ θα ήταν να πούμε ότι ο qualifier 1 ταυτίζεται με τον 3 καθώς εάν ο 1 ειναι Unrestricted δεν γνωρίζουμε σε καμία περίπτωση ότι ο 3 είναι Unrestricted καθώς μπορεί να τον κάνει γραμμικό κάποια άλλη μεταβλητή που χρησιμοποιείται στο closure.

5.5 Παραδείγματα συμπερασμού τύπων

Παρακάτω θα δούμε ορισμένα παραδείγματα του συμπερασμού τύπων στην πράξη. Η συνάρτηση inferType παίρνει μια έκφραση και επιστρέφει μια τιμή τύπου Either TypeError Type. Ενώ έχουμε υλοποιήσει έναν πλήρη compiler που διαβάζει προγράμματα της αρχικής γλώσσας τα μετασχηματίζει σε letdo και έπειτα συμπεραίνει τους τύπους, τα παραδείγματα παραθέτονται στην μορφή τύπου δεδομένων Haskell που αναπαριστά το AST της γλώσσας letdo.

Όταν επιστρέφεται Right έχει επιτύχει ο έλεγχος τύπων. Αντίθετα εάν επιστραφεί Left υπάρχει σφάλμα. Τα μηνύματα σφάλματος δεν είναι πολύ περιγραφικά. Ένας μοντέρνος μεταγλωττιστής μαζί με τα σφάλματα οφείλει να βγάλει και πληροφορίες για την επίλυση τους. Είναι γεγονός ότι στον δικό μας compiler η αποτυχία του ελέγχου γραμμικότητας βγάζει το ίδιο σφάλμα και στην περίπτωση που ένα resource δεν χρησιμοποιηθεί και στην περίπτωση που ένα resource χρησιμοποιηθεί πάνω απο μία φορές. Σε μελλοντικές εκδόσεις, τα μηνύματα σφάλματος πρέπει να γίνουν πιο φιλικά για τον προγραμματιστή.

```
ex1 :: Expr
ex1 = Let "x" NewArray (Var "x")

*Infer> inferType ex1
Right Array
```

Ο πίνακας χρησιμοποιείται ακριβώς μία φορά και η έκφραση επιστρέφει συνολικά έναν πίνακα.

```
ex2 :: Expr
ex2 = Let "x" NewArray (Num 42)

*Infer> inferType ex2
Left "Array Can't be Unrestricted"
```

Ο πίνακας δεν χρησιμοποιείται καμία φορά οπότε έχουμε σφάλμα

```
ex3 :: Expr
ex3 = Let "x" NewArray (Let "y" (Var "x") (Var "x"))

*Infer> inferType ex3
Left "Array Can't be Unrestricted"
```

Χρησιμοποιείται ο πίνακας x δύο φορές οπότε έχουμε σφάλμα

```
ex4 :: Expr
ex4 = Let "f" (Abs "x" NewArray) (Var "f")

*Infer> inferType ex4
Right U((@0)->Array)
```

Η συνάρτηση δεν χρησιμοποιεί κάτι γραμμικό στο closure της οπότε είναι Unrestricted

```
ex5 :: Expr
ex5 = Abs "x" NewArray

*Infer> inferType ex5
Right U((@0)->Array)
```

Ομοίως με παραπάνω είναι Unrestricted συνάρτηση που παίρνει κάτι Unrestricted και επιστρέφει πίνακα

```
ex6 :: Expr
ex6 = Let "f" NewArray (Abs "x" (Var "f"))

*Infer> inferType ex6
Right L((@0)->Array)
```

Η συνάρτηση έχει γραμμικό τύπο γιατί χρησιμοποιεί τον πίνακα f στο closure της.

```
ex7 :: Expr
ex7 = Abs "x" (Abs "y" (Var "x"))

*Infer> inferType ex7
Right U((@0)->#4((@2)->(@0)))
```

Ουσιαστικά είναι η συνάρτηση const πριν την επίλυση του αδιεξόδου.

```
ex10 :: Expr
ex10 = Let "f" (Abs "x" (Var "x")) (Var "x")

*Infer> inferType ex10
Left "Variable x doesn't exist in the environment"
```

Η μεταβλητή x δεν υπάρχει πια στο σημείο που χρησιμοποιείται.

```
ex11 :: Expr
ex11 = Let "f" (Abs "x" (Num 42)) (Let "x" (App (Var "f") (Num 12)) (Var "f"))

*Infer> inferType ex11
Right U(Int->Int)
```

Η συνάρτηση f αποτελεί μια Unrestricted συνάρτηση που λαμβάνει εναν ακέραιο και επιστρέφει έναν ακέραιο.

```
ex12 :: Expr
ex12 = Let "f" (Abs "x" (Var "x")) (App (Var "f") (Var "f"))

*Infer> inferType ex12
Left "Infinite Type"
```

Όπως και στο σύστημα DHM απαγορεύεται να είναι well typed η έκφραση f f καθώς κατασκευάζει άπειρο τύπο.

```
ex13 :: Expr
ex13 = Let "f" (Abs "x" (Var "x")) (Let "x" (App (Var "f") NewArray)
    (App (Var "f") (Num 42)))

*Infer> inferType ex13
Left "Can't unify"
```

Η συνάρτηση f πρώτα χρησιμοποιείται με παράμετρο έναν πίνακα και έπειτα χρησιμοποιείται με παράμετρο εναν ακέραιο οπότε έχουμε σφάλμα.

```
ex18 :: Expr
ex18 = Let "a" NewArray (Let "f" (Abs "x" (Var "x"))
    (Let "result" (App (Var "f") (Var "a")) (Var "a")))

*Infer> inferType ex18
Left "Array Can't be Unrestricted"
```

Στο παραπάνω παράδειγμα ο πίνακας a χρησιμοποιείται δύο φορές οπότε υπάρχει σφάλμα τύπων καθώς οι πίνακες έχουν γραμμικό τύπο και πρέπει να χρησιμοποιηθούν ακριβώς μία φορά.

```
ex19 :: Expr
ex19 = Let "a" (Abs "x" (Var "x")) (Num 42)

*Infer> inferType ex19
Right Int
```

Η εφαρμογή της ταυτοτικής συνάρτησης σε έναν ακέραιο επιστρέφει ακέραιο.

```
ex3 :: Expr
ex3 = Let "x" NewArray (Let "y" (Var "x") (Var "x"))

*Infer> inferType ex3
Left "Array Can't be Unrestricted"
```

Η έκφραση let x = e in e' έχει πάντοτε τον τύπο του e' εφόσον η έκφραση e είναι well typed.

```
ex4 :: Expr
ex4 = Let "f" (Abs "x" NewArray) (Var "f")

*Infer> inferType ex4
Right U((@0)->Array)
```

Η συνάρτηση z "πετάει" την πρώτη παράμετρο της και χρησιμοποιεί την δεύτερη οπότε όταν εφαρμόζεται πάνω σε έναν ακέραιο επιστρέφει την ταυτοτική συνάρτηση.

```
ex22 :: Expr
ex22 = App ex21 NewArray

*Infer> inferType ex22
Right Array
```

Εάν η συνάρτηση που επιστράφηκε στο προηγούμενο παράδειγμα εφαρμοσθεί σε πίνακα επιστρέφει πίνακα.

```
ex23 :: Expr
ex23 = Abs "x" (Abs "y" (App (Var "x") (Var "y")))

*Infer> inferType ex23
Right U(#6((@2)->(@4))->#7((@2)->(@4)))
```

Η συνάρτηση παίρνει μια παράμετρο x τύπου @2 -> @4 και επιστρέφει μια συνάρτηση η οποία αν λάβει μια μεταβλητή @2 επιστρέφει ένα @4. Παρατηρούμε ότι εφόσον δεν έχουμε πληροφορία για τον qualifier της x δεν μπορούμε να συμπεράνουμε τους υπόλοιπους qualifiers(αδιέξοδο).

```
ex24 :: Expr
ex24 = Abs "x" (Abs "y" (Let "z" (App (Var "x") (Num 12))
    (App (Var "x") (Var "y"))))

*Infer> inferType ex24
Right U(U(Int->(@7))->U(Int->(@7)))
```

Ομοίως με το προηγούμενο παράδειγμα αλλα τώρα δίνουμε την πληροφορία οτι το x είναι Unrestricted οπότε μπορούμε να συμπεράνουμε ότι η συνάρτηση που λαμβάνει την παράμετρο y είναι και αυτή Unrestricted.

```
ex25 :: Expr
ex25 = App ex24 (Abs "x" (Var "x"))

*Infer> inferType ex25
Right U(Int->Int)
```

Εάν εφαρμόσουμε την συνάρτηση του προηγούμενο παραδείγματος στην ταυτοτική συνάρτηση παίρνουμε πάλι την ταυτοτική συνάρτηση.

```
ex26 :: Expr
ex26 = Let "x" (Abs "z" (Abs "y" (Var "z"))) (Var "x")

*Infer> inferType ex26
Right U((@0)->#4((@2)->(@0)))
```

Ουσιαστικά η x είναι η συνάρτηση const αλλα επειδή δεν γνωρίζουμε τον qualifier του z έχουμε πάλι αδιέξοδο στην εύρεση του qualifier 4.

```
ex27 :: Expr
ex27 = Let "x" (Abs "z" (Abs "y" (Var "z"))) (App (Var "x") NewArray)

*Infer> inferType ex27
Right L((@2)->Array)
```

Εάν εφαρμόσουμε την const σε μεταβλητή τύπου πίνακα τότε επιστρέφεται μια συνάρτηση που έχει στο closure της χρησιμοποιήσει πίνακα οπότε έχει γραμμικό qualifier.

```
ex28 :: Expr
ex28 = Abs "r" $ LetDo "x" (Var "r") (Num 0) (Var "r")

*Infer> inferType ex28
Right U(Array->Array)
```

Μια συνάρτηση που παίρνει έναν πίνακα τον διαβάζει και τον επιστρέφει έχει ίδιο τύπο με την ταυτοτική συνάρτηση.

```
ex29 :: Expr
ex29 = Let "r" NewArray $ LetDo "x" (Var "r") (Var "r") (Var "x")

*Infer> inferType ex29
Left "Variable r doesn't exist in the environment"
```

Όπως και στα παραδείγματα των προηγούμενων κεφαλαίων η μεταβλητή r δεν υπάρχει μέσα στο letdo block(οπότε δεν μπορεί να διαφύγει).

```
ex30 :: Expr
ex30 = Let "r" NewArray $ LetDo "x" (Var "r") (Num 0) (Var "x")

*Infer> inferType ex30
Left "Array Can't be Unrestricted"
```

Η συγκεκριμένη έκφραση ενώ διαβάζει σωστά τον πίνακα r δεν τον απελευθερώνει πριν επιστρέψει την μεταβλητή x, οπότε πρακτικά ο πίνακας δεν απελευθερώνεται ποτέ.

```
ex31 :: Expr
ex31 = Let "r" (Num 42) $ LetDo "x" (Var "r") (Num 0) (Var "r")

*Infer> inferType ex31
Left "Can't unify"
```

Δεν μπορεί να γίνει η ενοποίηση του πίνακα με τον ακέραιο, και αρα να γίνει ανάγνωση με letdo απο ακέραιο.

Κεφάλαιο 6

Συμπεράσματα και προτάσεις μελλοντικής έρευνας

6.1 Συνεισφορά

Ο στόχος της διπλωματικής ήταν να κάνει ασφαλές το destructive update σε αγνές συναρτησιακές γλώσσες προγραμματισμού. Οι γραμμικοί τύποι προσφέρουν εύκολα το destructive update με μεγάλη δυσκολία στον προγραμματιστή, καθώς το έτοιμο interface απαιτεί threading του νέου πίνακα σε κάθε βήμα ακόμη και αν το βήμα είναι ανάγνωση στον πίνακα. Τα προηγούμενα χρόνια έχει επιχειρηθεί με την βοήθεια των γραμμικών τύπων υπάρχει ένα συναρτησιακό interface το οποίο υλοποιεί destructive update. Το σύστημα τύπων θα εγγυάται ότι ο προγραμματιστής δεν θα μπορεί να χρησιμοποιήσει την ''παλιά'' τιμή η οποία θα έχει καταστραφεί λόγω του destructive update. Οι πρώτες προσεγγίσεις του Wadler και του Odersky έχουν μεγάλους περιορισμούς. Το σύστημα letbang with scopes φαίνεται να λύνει το πρόβλημα, ωστόσο ο προγραμματιστής ειναι υπεύθυνος για την επισημείωση των τύπων(και των scopes). Η γλώσσα letdo λύνει τα παραπάνω προβλήματα, καθώς προσφέρει ενα αγνά συναρτησιακό interface στον προγραμματιστή. Η γλώσσα έχει συμπερασμό τύπων οπότε το πρόγραμμα δεν είναι απαραίτητο να έχει type annotations ή type signatures. Δεν σειριοποιείται κάθε operation πάνω στον πίνακα. Επίσης, μέσω του μετασχηματισμού ο προγραμματιστής μπορεί να γράψει σε μία γλώσσα που γνωρίζει όπως η OCaml και στον τελικό κώδικα θα υλοποιηθεί το destructive update για τους πίνακες. Ωστόσο, η προσέγγιση μας έχει κάποιους περιορισμούς

- Το letdo construct μπορεί να λειτουργήσει μονάχα για eager γλώσσες. Όταν υπάρχει το laziness είναι σίγουρα υποχρεωτικό οι αναγνώσεις που έχουν εκτελεστεί πριν απο εγγραφές να γίνουν πριν τις εγγραφές

- Δεν υπάρχει συνένωση αναγνώσεων σε πολλαπλούς πίνακες. Για παράδειγμα έαν έχουμε τρεις πίνακες α,β,γ και διαβάσουμε το πρώτο στοιχείο του καθενός τότε η γλώσσα letdo θα φτιάξει 3 letdo constructs

6.2 Προτάσεις μελλοντικής έρευνας

Ένα σημείο μελλοντικής έρευνας είναι η μελέτη της σύνδεσης των letbang constructs με την Linear logic. H Linear Logic δεν περιγράφει κανένα construct σαν το let!. Ουσιαστικά, στον κόσμο της λογικής το let! είναι ενας κανόνας που παίρνει ένα γραμμικό resource A και το μετατρέπει σε of course A δηλαδή !A. Είναι γεγονός ότι το let! έχει αρκετούς περιορισμούς και δεν είναι ξεκάθαρη η σύνδεση με την Linear Logic.

Θα μπορούσε να υπάρχει πολυμορφισμός σε επίπεδο qualifier. Ο προγραμματιστής θα ήθελε να γράφει την παρακάτω συνάρτηση να έχει τον επισημειωμένο τύπο

```
id :: forallQ q. forall a. #q@a -> #q@a
id = \x . x
```

Στην γλώσσα letdo δεν υπάρχει καθολικός ποσοδείκτης σε όλους τους qualifiers. Εάν υπήρχε τότε δημιουργεί προβλήματα στον συμπερασμό τύπων αφού μπορεί να δημιουργήσει συνθήκες μη τερματισμού. Θα μπορούσε να υπάρχει πολυμορφισμός στους qualifiers με κόστος τον εγγυημένο τερματισμό του συμπερασμού τύπων. Αυτό σημαίνει ότι στην νέα γλώσσα υπάρχει πολυμορφισμός στους

qualifiers, ο αλγόριθμος συμπερασμού τύπων λύνει τους περιορισμούς και όταν δεν μπορεί, εντοπίζει ότι βρίσκεται σε αδιέξοδο και σταματάει την εκτέλεση του. Ο προγραμματιστής είναι έπειτα υπεύθυνος να δώσει κατάλληλες επισημειώσεις τύπων σε κάποια σημεία έτσι ώστε να λυθούν ορισμένοι περιορισμοί και να τελειώσει η διαδικασία.

Ο αλγόριθμος συμπερασμού τύπων δουλεύει σωστά σε εκφράσεις που δεν έχουν καθόλου τύπους. Ωστόσο, δεν γίνεται χρήση επισημειώσεων τύπων. Είναι γεγονός ότι στις σύγχρονες συναρτησιακές γλώσσες προγραμματισμού, όπως η Haskell κάθε συνάρτηση συνοδεύεται απο ένα type signature το οποίο καθορίζει τους τύπους των παραμέτρων και του αποτελέσματος της συνάρτησης. Μια επέκταση της τωρινής υλοποίησης θα ήταν να υπάρχουν επισημειώσεις τύπων ή γενικότερα type signatures και ο μεταγλωττιστής να περνάει αυτή την πληροφορία στο annotated AST. Όταν υπάρχει πληροφορία τύπων απο τον προγραμματιστή στο στάδιο παραγωγής περιορισμών δεν είναι απαραίτητο να φτιάχνουμε νέες μεταβλητές τύπων για κάθε μεταβλητή, μπορούμε να χρησιμοποιήσουμε τους τύπους που προσφέρει ο προγραμματιστής όπου υπάρχει η πληροφορία και να παράγουμε νέες μεταβλητές τύπων όπου δεν υπάρχει η πληροφορία. Σε κάθε αλγόριθμο συμπερασμού τύπων η επιπλέον πληροφορία απο τον προγραμματιστή μπορεί να βελτιώσει την επίδοση, και αυτό το φαινόμενο είναι εντονότερο σε αλγορίθμους 2 βημάτων παραγωγής περιορισμών. Ωστόσο, ένα μειονέκτημα είναι ότι θα θέλαμε το destructive update να γίνεται σαν optimisation χωρίς να γνωρίζει ο προγραμματιστής τις λεπτομέρειες των γραμμικών τύπων και ως συνέπεια να γράφει type signatures με lollipop arrows.

Ένα ακόμη κομμάτι που μπορεί να βελτιωθεί στον αλγόριθμο συμπερασμού τύπων είναι τα μηνύματα λάθους που διαβάζει ο προγραμματιστής. Όταν ένας πίνακας χρησιμοποιείται δύο φορές τότε βγαίνει το μήνυμα λάθους ”Array can’t be Unrestricted”. Η πρόταση αυτή είναι αληθής αλλά δεν βοηθάει τον προγραμματιστή να λύσει το πρόβλημα. Θα μπορούσε να υπάρχει ειδικό μήνυμα όταν μια μεταβλητή δεν χρησιμοποιηθεί καμία φορά και ειδικό μήνυμα όταν μια μεταβλητή χρησιμοποιηθεί δύο φορές. Επίσης, για να είναι βοηθητικά τα μηνύματα θα ήταν χρήσιμο να αποθηκεύεται η θέση στο πρόγραμμα που χρησιμοποιείται κάθε μεταβλητή έτσι ώστε να μπορούν να παραχθούν σωστά μηνύματα λάθους.

Ένα κομμάτι που δεν έχει μελετηθεί αρκετά σε σχέση με τους γραμμικούς τύπους είναι τι συμβαίνει στην περίπτωση exception. Το πρόβλημα μπορεί να φανεί στο παρακάτω κομμάτι κώδικα

```
let r = newArray in
let x = 42 / 0 in
free r
```

Εδώ βλέπουμε ότι η διαίρεση με το μηδέν θα προκαλέσει σφάλμα στο πρόγραμμα. Η εγγύηση των γραμμικών τύπων είναι ότι αν δεν προκληθεί exception τότε κάθε γραμμική μεταβλητή θα χρησιμοποιηθεί ακριβώς μια φορά. Επειδή οι γραμμικοί τύποι προσφέρουν αυτή την εγγύηση δεν υπάρχει garbage collector για τις γραμμικές τιμές οπότε ο πίνακας δεν θα ελευθερωθεί ποτέ. Στην περίπτωση που το πρόγραμμα σταματήσει την εκτέλεση του λόγω του σφάλματος το λειτουργικό σύστημα θα πάρει πίσω την δοσμένη μνήμη. Στην περίπτωση όμως που ο προγραμματιστής θέλει να ”πιάσει” το exception τότε έχει χαθεί κάθε reference πάνω στον πίνακα r και άρα η μνήμη δεν μπορεί να αποδεσμευθεί. Μια λύση είναι για τα σφάλματα που ο προγραμματιστής θέλει να διαχειρίζεται να κωδικοποιεί την αποτυχία μέσα στους τύπους. Όμως, πρέπει να μελετηθεί πως ακριβώς μπορεί να γίνει η διαχείριση της αποτυχίας σε μια γλώσσα με γραμμικούς τύπους και τι συνεπάγεται αυτό για τα χαμένα references σε γραμμικές τιμές.

Στην γλώσσα Letdo δεν έχει οριστεί ακριβώς πως γίνεται το pattern matching σε αλγεβραϊκούς τύπους δεδομένων. Εάν ενας τύπος δεδομένων περιέχει κάποιο γραμμικό τύπο δεδομένων τότε ειναι υποχρεωτικά γραμμικός(διότι αν δεν ήταν μπορεί να χρησιμοποιηθεί δύο φορές μια γραμμική τιμή).

Επίσης πρέπει να μελετηθεί πως μπορεί να γενικευτεί η γλώσσα για να υποστηρίζει πίνακες πολλών διαστάσεων. Φαίνεται ότι είναι προτιμότερο να έχουμε διαφορετικό τύπο για κάθε πιθανή διάσταση διότι θέλουμε να αποφύγουμε να έχουμε πίνακες που οι τιμές είναι γραμμικές.

Τέλος, ενώ έχουμε υλοποιήσει εναν μεταγλωττιστή για την γλώσσα letdo (μαζί με τον μετασχηματισμό) δεν έχουμε αποδείξει τυπικά τα θεωρήματα προόδου και διάτηρησης. Το θεώρημα προόδου

είναι ότι κάθε well typed όρος είναι τιμή ή μπορεί να κάνει βήμα εκτέλεσης. Το θεώρημα διατήρησης λέει ότι εάν ένας well typed όρος e : τ κάνει βήμα στο e' τότε και το e' πρέπει να έχει τον ίδιο τύπο τ. Η απόδειξη αυτών των θεωρημάτων προκύπτει απο την απόδειξη ότι κάθε γραμμική μεταβλητή χρησιμοποιείται ακριβώς μια φορά, δηλαδή δεν υπάρχει τρόπος να διαφύγει γραμμική μεταβλητή. Όπως παλίοτερα κάνανε οι Παπασπύρου-Παπακυριάκου[Papa07] μπορούμε να κάνουμε χρήση του Isabelle/HOL proof assistant για να πάρουμε αυτόματα ορισμένες αποδείξεις για την ασφάλεια της γλώσσας letdo.

Βιβλιογραφία

[Bern17] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones and Arnaud Spiwack, “Linear Haskell: practical linearity in a higher-order polymorphic language”, *CoRR*, vol. abs/1710.09756, 2017.

[BIRD97] RICHARD BIRD, GERAINT JONES and OEGE DE MOOR, “More haste, less speed: lazy versus eager evaluation”, *Journal of Functional Programming*, vol. 7, no. 5, p. 541–547, 1997.

[Gira87] Jean-Yves Girard, “Linear Logic”, *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–102, 1987.

[Heer02] Bastiaan Heeren and Jurriaan Hage, “Generalizing Hindley-Milner Type Inference Algorithms”, in *””*, 2002.

[Morr05] Greg Morrisett, Amal Ahmed and Matthew Fluet, “L3: A Linear Language with Locations”, in Pawe Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pp. 293–307, Berlin, Heidelberg, 2005, Springer Berlin Heidelberg.

[Papa07] Michalis A. Papakyriakou and Nikolaos S. Papaspyrou, “Mechanized Proofs of Type Safety for a Family of Lambda Calculi with References”, in *””*, 2007.

[Papa10] Michalis A. Papakyriakou and Nikolaos S. Papaspyrou, “From Linear to Unrestricted and Back: Type Safety and the Let-bang Construct”, in *””*, 2010.

[Pier04] Benjamin C. Pierce, *Advanced Topics in Types and Programming Languages*, The MIT Press, 2004.

[Pipp96] Nicholas Pippenger, “Pure Versus Impure Lisp”, in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’96, pp. 104–109, New York, NY, USA, 1996, ACM.

[Wadl90] Philip Wadler, “Linear Types Can Change the World!”, in *PROGRAMMING CONCEPTS AND METHODS*, North, 1990.

[Walk04] David Walker, “Substructural Type Systems”, *Advanced Topics in Types and Programming Languages*, 2004.