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

Σχεδίαση, Σημασιολογική Περιγραφή και Υλοποίηση της Γλώσσας Προγραμματισμού Cantor

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΥΡΙΑΚΟΣ Γ. ΓΚΙΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2003

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

Σχεδίαση, Σημασιολογική Περιγραφή και Υλοποίηση της Γλώσσας Προγραμματισμού Cantor

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΥΡΙΑΚΟΣ Γ. ΓΚΙΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιουλίου 2003.

....	....	....
Εμμανουήλ Σκορδαλάκης Καθηγητής Ε.Μ.Π.	Ευστάθιος Ζάχος Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2003

....

Κυριάκος Γ. Γκίνης

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

Περίληψη

Ο σκοπός της εργασίας ήταν η σχεδίαση μιας μικρής αλλά ιδιαίτερα εκφραστικής συναρτησιακής γλώσσας προγραμματισμού, η τυπική περιγραφή της σημασιολογίας της και η κατασκευή ενός διερμηνέα και ενός μεταγλωττιστή για αυτήν. Συγκεκριμένα, σχεδιάστηκε η γλώσσα Cantor, τα κυριότερα χαρακτηριστικά της οποίας είναι οι συναρτήσεις υψηλής τάξης με δυνατότητα κλήσης με όνομα και με τιμή, οι τοπικές μεταβλητές με δυνατότητα ανάθεσης, οι εγγραφές (records) των οποίων τα πεδία μπορούν να έχουν τύπο συνάρτησης, και η αναδρομή. Τα χαρακτηριστικά αυτά έχουν ενσωματωθεί στη γλώσσα Cantor με τέτοιο τρόπο ώστε να είναι κατά το δυνατό ορθογώνια.

Αφού ορίστηκε η σημασιολογία τύπων της Cantor, το σύστημα τύπων της οποίας επιτρέπει τον πλήρη στατικό έλεγχο των προγραμμάτων, υλοποιήθηκε ο ελεγκτής τύπων. Η υλοποίηση αυτή όπως και οι υλοποιήσεις των υπόλοιπων τμημάτων της εργασίας, έγινε στη γλώσσα Haskell και βασίζεται στη χρήση μονάδων. Στη συνέχεια ορίστηκε η δυναμική σημασιολογία της Cantor με τη βοήθεια μονάδων και υλοποιήθηκε ένας διερμηνέας μέσω απλής κωδικοποίησης της σημασιολογίας αυτής σε Haskell. Το επόμενο στάδιο της εργασίας ήταν η κατασκευή ενός μεταγλωττιστή για την Cantor. Ο μεταγλωττιστής που κατασκευάστηκε παράγει κώδικα για μια παραλλαγή της αφηρημένης μηχανής υπολογισμού G-machine. Προκειμένου να δοκιμαστεί η ορθότητα του μεταγλωττιστή, κατασκευάστηκε και ένας διερμηνέας για τον κώδικα G-machine.

Κατά τη διάρκεια της εργασίας έγινε φανερή αφενός η χρησιμότητα των μονάδων τόσο στη φάση του ορισμού της δυναμικής σημασιολογίας όσο και στη φάση της υλοποίησης μιας γλώσσας προγραμματισμού, αφετέρου το γεγονός ότι μια συναρτησιακή γλώσσα όπως η Haskell που υποστηρίζει μονάδες είναι ιδιαίτερα κατάλληλη για την υλοποίηση διερμηνέων και μεταγλωττιστών.

Λέξεις κλειδιά

Σημασιολογία γλωσσών προγραμματισμού, συναρτησιακός προγραμματισμός, Haskell, μονάδες, διερμηνέας, μεταγλωττιστής, ελεγκτής τύπων, G-machine.

Abstract

The aim of this project was the design of a small but highly expressive functional programming language, the formal description of its semantics and the implementation of an interpreter and a compiler for this language. More specifically, the language Cantor was designed, whose main characteristics are higher-order functions supporting call-by-value and call-by-name, local variables supporting assignment, records whose fields can be functions, and recursion. These characteristics have been incorporated in Cantor in such a way that they are as orthogonal as possible.

After the typing semantics of Cantor was defined (Cantor's type system allows the full static check of programs), a type checker was implemented. This implementation, as well as the implementations of all other parts of this project, was done in Haskell and is based on the use of monads. Subsequently, the dynamic semantics of Cantor was defined and an interpreter was implemented by a simple encoding of this semantics in Haskell. The next stage of the project was the implementation of a compiler for Cantor. The compiler that was built produces code for a variation of the abstract computing machine known as G-machine. In order to verify the correctness of the compiler, an interpreter for G-machine code was also implemented.

During this project it was evident that monads are very useful both for the definition of dynamic semantics and for the implementation of a programming language. On the other hand, it was evident that a functional language supporting monads, such as Haskell, is highly appropriate for the implementation of interpreters and compilers.

Key words

Programming language semantics, dynamic semantics, functional programming, Haskell, monads, interpreter, compiler, type checker, G-machine.

Ευχαριστίες

Πριν από όλους θα ήθελα να ευχαριστήσω τους γονείς μου, για την αμέριστη συμπαράσταση που έδειξαν κατά τη διάρκεια όχι μόνο της εκπόνησης της διπλωματικής μου εργασίας, αλλά και των σπουδών μου γενικότερα. Επίσης θα ήθελα να ευχαριστήσω τον επιβλέποντα την διπλωματική μου κ. Νίκο Παπασπύρου για την καθοδήγηση και τις συμβουλές που μού έδωσε, καθώς και για το γεγονός ότι ήταν πάντα διαθέσιμος και φιλικός οποτεδήποτε τον χρειάστηκα. Τέλος, θα ήθελα να ευχαριστήσω τα μέλη της εξεταστικής επιτροπής κ.κ. Στάθη Ζάχο και Μανόλη Σκορδαλάκη, τον πρώτο γιατί με τα μαθήματα που δίδασκε κατά τη διάρκεια του πρώτου έτους στο Πολυτεχνείο με έφερε σε επαφή για πρώτη φορά με τον προγραμματισμό και την Πληροφορική γενικά, και τον δεύτερο διότι με το μάθημά του «Μεταγλωττιστές» κέντρισε το ενδιαφέρον μου για το συγκεκριμένο αντικείμενο και συνετέλεσε στο να επιλέξω το θέμα της παρούσας διπλωματικής εργασίας.

Κυριάκος Γ. Γκίνης,

Αθήνα, 22 Ιουλίου 2003.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-03, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2003.

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

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σκοπός της παρούσας διπλωματικής εργασίας είναι η σχεδίαση μιας μικρής συναρτησιακής γλώσσας προγραμματισμού, η σημασιολογική ανάλυση της γλώσσας αυτής, η κατασκευή διερμηνέα με βάση τη δυναμική σημασιολογία και τέλος η κατασκευή μεταγλωττιστή για τη γλώσσα. Το προγραμματιστικό κομμάτι της εργασίας έγινε στη συναρτησιακή γλώσσα προγραμματισμού Haskell.

1.2 Γλώσσα Cantor

Για τις ανάγκες της εργασίας σχεδιάστηκε η μικρή συναρτησιακή γλώσσα Cantor, στα πρότυπα της Idealized Algol που προτείνει ο Reynolds στο [1]. Η Cantor διαθέτει ποικιλία χαρακτηριστικών, όπως εκφράσεις λάμβδα, τοπικές μεταβλητές με δυνατότητα ανάθεσης, records κ.ά., κάτι που καθιστά ενδιαφέρουσα τόσο τη σημασιολογική ανάλυσή της, όσο και την κατασκευή του διερμηνέα και του μεταγλωττιστή. Επίσης διαθέτει σύστημα τύπων που καθιστά δυνατό τον πλήρη στατικό έλεγχο τύπων ενός προγράμματος Cantor.

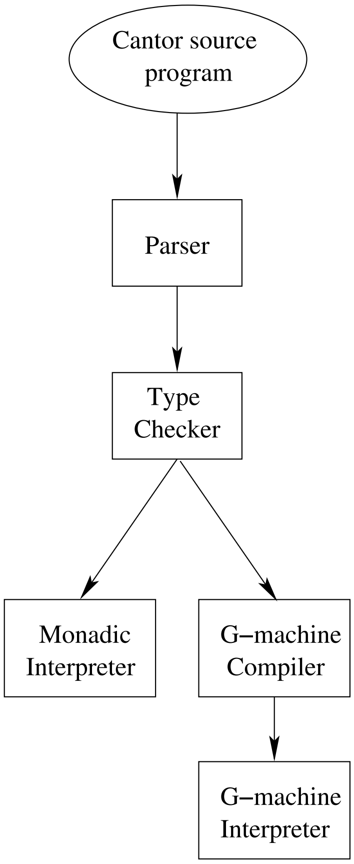
1.3 Γλώσσα Haskell

Όλες οι υλοποιήσεις που έχουν γίνει στα πλαίσια της παρούσας εργασίας, έχουν γραφτεί στη γλώσσα Haskell. Η Haskell είναι μια καθαρή (pure), lazy συναρτησιακή γλώσσα, βασισμένη στο λάμβδα λογισμό, με πολυμορφικό σύστημα τύπων (polymorpicly typed). Μια συναρτησιακή γλώσσα σαν τη Haskell μάς προσφέρει τη δυνατότητα να γράφουμε κώδικα μικρότερο σε μέγεθος, πιο εύκολα κατανοητό και με λιγότερα λάθη από ότι με μια συμβατική προστακτική γλώσσα.

Το κείμενο της παρούσας εργασίας προϋποθέτει ότι ο αναγνώστης διαθέτει τουλάχιστον βασικές γνώσεις Haskell. Σε αντίθετη περίπτωση, ως μια εισαγωγή στη γλώσσα Haskell προτείνεται το [2], το οποίο μαζί με πληθώρα υλικού για τη γλώσσα Haskell, υπάρχει διαθέσιμο στην ιστοσελίδα της γλώσσας, `http://www.haskell.org`.

1.4 Μονάδες

Μεγάλο μέρος των υλοποιήσεων που έχουμε κάνει, αλλά και η δυναμική σημασιολογία, βασίζονται στη χρήση μονάδων (monads). Οι μονάδες έχουν έρθει στο προσκήνιο τα τελευταία χρόνια στο χώρα του συναρτησιακού προγραμματισμού, καθώς δίνουν τη δυνατότητα σε καθαρές (pure) συναρτησιακές γλώσσες, όπως η Haskell, να προσεγγίσουν με επιτυχία προγραμματιστικά προβλήματα όπως ο χειρισμός λαθών και εξαιρέσεων, η είσοδος/έξοδος ή οι stateful αλγόριθμοι, στα οποία εμφανίζονταν να υστερούν, τόσο έναντι των προστακτικών (imperative) γλωσσών, όσο και έναντι μη καθαρών (impure) συναρτησιακών γλωσσών όπως η Standard ML. Ταυτόχρονα, χάρη στις μονάδες οι καθαρές συναρτησιακές γλώσσες αντιμετωπίζουν τα



Σχήμα 1.1: Δομή υλοποιήσεων.

προβλήματα αυτά με τρόπο που διατηρεί ανέπαφα τα πλεονεκτήματά τους (referential transparency, equational reasoning, ανεξαρτησία από την σειρά αποτίμησης, lazy evaluation).

Επιπλέον η χρήση μονάδων κάνει εύκολη την εκ των υστέρων αλλαγή ενός συναρτησιακού προγράμματος ή την προσθήκη νέων χαρακτηριστικών. Έστω για παράδειγμα ότι θέλουμε να κατασκευάσουμε ένα διερμηνέα με μια καθαρή συναρτησιακή γλώσσα. Αν θελησουμε να προσθεσουμε στο διερμηνέα δυνατότητα χειρισμού λαθών, θα πρέπει να τροποποιήσουμε τον τύπο της κύριας συνάρτησης του διερμηνέα, ώστε να συμπεριλαμβάνει και τα πιθανά μηνύματα λάθους. Θα πρέπει επίσης σε κάθε αναδρομική κλήση να φροντίσουμε ώστε τα λάθη να αντιμετωπίζονται με τον κατάλληλο τρόπο. Αντίθετα σε μια μη καθαρή συναρτησιακή γλώσσα με δυνατότητα χειρισμού εξαιρέσεων, μια τέτοια προσθήκη θα ήταν πολύ απλούστερη. Παρομοίως, αν θελήσουμε να προσθέσουμε στο διερμηνέα ένα μετρητή, θα πρέπει και πάλι να τροποποιήσουμε τον τύπο επιστροφής και να φροντίσουμε ώστε σε κάθε αναδρομική κλήση να περνάει ο μετρητής ως όρισμα. Σε μια μη καθαρή συναρτησιακή γλώσσα θα αρκούσε η χρήση μιας global μεταβλητής. Οι μονάδες, όπως θα δούμε παρακάτω, έρχονται να λύσουν αυτά τα προβλήματα.

1.5 G-Machine

Η μεταγλώττιση συναρτησιακών γλωσσών απαιτεί τη χρήση διαφορετικών τεχνικών από τη μεταγλώττιση προστακτικών γλωσσών. Μερικές τέτοιες τεχνικές, που παρουσιάζονται στο [3], είναι η “Template instantiation”, η “G-Machine” και η “Three Instruction Machine (TIM)”. Εμείς για το μεταγλωττιστή μας επιλέξαμε την G-machine, τροποποιώντας την ωστόσο σε σχέση με το μοντέλο που προτείνεται στο [3], ώστε να προσαρμοστεί καλύτερα στις ιδιαιτερότητες της γλώσσας Cantor. Η G-machine πρώτα αναπτύχθηκε από τους Johnsson & Augustsson [4, 5] και με κάποιες παραλλαγές και βελτιώσεις χρησιμοποιείται σήμερα σε μεταγλωττιστές όπως ο GHC της γλώσσας Haskell. Η κεντρική ιδέα πίσω από την G-machine, είναι η μεταγλώττιση κάθε συνάρτησης και τελικά ολόκληρου του συναρτησιακού προγράμματος σε μια σειρά από εντολές, οι οποίες ανήκουν σε μια ενδιάμεση αφηρημένη «γλώσσα μηχανής». Η γλώσσα αυτή πρέπει να έχει επιλεγεί κατά τέτοιο τρόπο, ώστε οι εντολές της να είναι εύκολο να μεταφραστούν σε κανονική γλώσσα μηχανής για διάφορες αρχιτεκτονικές, αλλά ταυτόχρονα να μην καθιστά δύσκολη τη μετάβαση από τον αρχικό κώδικα στον ενδιάμεσο.

Στην υλοποίηση που έχουμε κάνει, έχουμε φτάσει μέχρι το στάδιο της ενδιάμεσης γλώσσας. Σε μια κανονική υλοποίηση, η ενδιάμεση αυτή γλώσσα θα μεταγλωττιζόταν σε κανόνική γλώσσα μηχανής, ώστε να παραχθεί ένα εκτελέσιμο αρχείο που θα αντιστοιχούσε στον πηγαίο κώδικα που δόθηκε ως είσοδος. Αντιθέτως στην υλοποίηση μας, υπάρχει ένας διερμηνέας για την ενδιάμεση γλώσσα.

Το σχήμα 1.1 παρουσίαζει τη δομή των υλοποιήσεων που έχουμε κάνει.

1.6 Δομή εργασίας

Η εργασία μας έχει την ακόλουθη δομή:

Κεφάλαιο 2. Στο κεφάλαιο αυτό περιγράφεται η γλώσσα Cantor.

Κεφάλαιο 3. Σε αυτό το κεφάλαιο ορίζεται η σημασιολογία τύπων της γλώσσας Cantor και παράλληλα περιγράφεται η υλοποίηση του ελεγκτή τύπων. Επίσης στην αρχή του κεφαλαίου γίνεται μια σύντομη περιγραφή των μονάδων.

Κεφάλαιο 4. Στο κεφάλαιο αυτό ορίζεται η δυναμική σημασιολογία της Cantor και περιγράφεται η υλοποίηση του διερμηνέα.

Κεφάλαιο 5. Στο κεφάλαιο αυτό δίνεται ο ορισμός της G-Machine, καθώς και η περιγραφή της υλοποίησης του διερμηνέα της G-Machine.

Κεφάλαιο 6. Σε αυτό το κεφάλαιο δίνονται οι κανόνες μεταγλώττισης σε κώδικα G-Machine της γλώσσας Cantor και η αντίστοιχη υλοποίηση.

Κεφάλαιο 7. Στο κεφάλαιο αυτό τέλος, περιέχονται τα συμπεράσματα της εργασίας μας.

Κεφάλαιο 2

Περιγραφή γλώσσας Cantor

Στο κεφάλαιο αυτό δίνεται η περιγραφή της γλώσσας Cantor. Στην παράγραφο 2.1 γίνεται μια γενική περιγραφή της γλώσσας, ενώ στην παράγραφο 2.2 δίνεται η γραμματική της Cantor.

2.1 Γενική περιγραφή της γλώσσας Cantor

Η Cantor είναι μια απλή γλώσσα προγραμματισμού που βασιζεται στο λάμβδα-λογισμό και ακολουθεί τις αρχές της Idealized Algol [1]. Κατά τον αρχικό σχεδιασμό υπήρξε πρόνοια ώστε τα διάφορα χαρακτηριστικά της γλώσσας να είναι ορθογώνια, κάτι το οποίο έκανε δυνατή τη σταδιακή προσθήκη χαρακτηριστικών, αλλά βοήθησε και στην κατασκευή των διάφορων τμημάτων του διερμηνέα και του μεταγλωττιστή. Η Cantor είναι strongly-typed γλωσσα, το σύστημα τύπων της δηλαδή διασφαλίζει ότι όσα προγράμματα περάσουν το στατικό έλεγχο τύπων θα εκτελεστούν χωρίς λάθος τύπων.

Τα κυριότερα χαρακτηριστικά της είναι εν συντομία τα εξής:

- απλοί τύποι δεδομένων (int, bool)
- σύνθετοι τύποι δεδομένων (records)
- συναρτήσεις με κλήση με τιμή (call-by-value) ή όνομα (call-by-name)
- απλή εντολή συνθήκης if-then-else
- τοπικές μεταβλητές με δυνατότητα ανάθεσης
- σειριακή εκτέλεση
- αναδρομή

Στις επόμενες παραγράφους δίνονται περισσότερες λεπτομέρειες της γλώσσας.

2.1.1 Αναγνωριστικά

Τα αναγνωριστικά (identifiers) της γλώσσας Cantor αποτελούνται από γράμματα (A..Z, a..z) και αριθμούς (0..9). Είναι υποχρεωτικό ωστόσο ένα αναγνωριστικό να ξεκινά από γράμμα. Τα μικρά και τα κεφαλαία γράμματα θεωρούνται διαφορετικά, ενώ δεν υπάρχει περιορισμός στο μήκος του αναγνωριστικού.

Τα παρακάτω αναγνωριστικά είναι δεσμευμένα ως λέξεις κλειδιά και δεν επιτρέπεται η χρήση τους ως κοινά αναγνωριστικά:

and	**bool**	**deref**	**else**	**empty**
exp	**false**	**if**	**in**	**int**
lambda	**neg**	**not**	**or**	**rec**
record	**true**	**var**		

2.1.2 Τύποι

Οι βασικοί τύποι δεδομένων (data types) που υποστηρίζει η γλώσσα Cantor είναι οι εξής:

- **int** : ακέραιοι αριθμοί[1]
- **bool** : λογικός τύπος δεδομένων, που μπορεί να πάρει τις τιμές **true** ή **false**

Ένας τύπος δεδομένων με την προσθήκη του προσδιοριστικού **var** ή **exp** (π.χ. **int exp**) μετατρέπεται σε τύπο φράσης (phrase type). Το προσδιοριστικό **exp** αντιστοιχεί σε σταθερές, ενώ το προσδιοριστικό **var** σε μεταβλητές. Επιπλέον των βασικών τύπων δεδομένων, η Cantor υποστηρίζει τύπο συνάρτησης (από τύπο φράσης σε τύπο φράσης, π.χ. **int exp** –> **int exp**) και το σύνθετο τύπο δεδομένων **record**. Ένα record περιέχει ένα μη διατεταγμένο σύνολο από ετερογενή πεδία, για καθένα από τα οποία πρέπει να δηλωθεί ένας τύπος φράσης και τα οποία αναγνωρίζονται από το όνομά τους, π.χ. **record**(i:**int var**, j:**int exp**, f:(**int exp** –> **int exp**) **exp**). Η δεσμευμένη λέξη **empty** συμβολίζει το κενό record.

2.1.3 Δομή προγράμματος

Ένα πρόγραμμα γραμμένο στη γλώσσα Cantor αποτελείται από μια έκφραση (expression). Μια έκφραση όπως θα δούμε παρακάτω, μπορεί να είναι μια εντολή συνθήκης, μια σειρά από εκφράσεις χωρισμένες με semicolon, μια συνάρτηση, κ.λπ.

2.1.4 Εκφράσεις

Η Cantor υποστηρίζει τα παρακάτω είδη εκφράσεων:

- Σταθερά τύπου **int**, π.χ. `5, 42`.
- Σταθερά τύπου **bool**, π.χ. `true, false`.
- Αναγνωριστικό, π.χ. `x, i3`.
- Τελεστής **not** ή **neg** ακολουθούμενος από έκφραση, π.χ. `neg 5, not true`.
- Δύο εκφράσεις μεταξύ των οποίων παρεμβάλλεται ένας από τους παρακάτω δυαδικούς τελεστές: +, −, ∗, /, %, ==, !=, <=, >=, <, >, **and**, **or**, ;. Π.χ.

  ```
  3+5, x;5, true and false, x<0 or y<0
  ```

  Από τους παραπάνω τελεστές, αυτοί των αριθμητικών πράξεων και των συγκρίσεων έχουν την ίδια σημασιολογία με αυτήν που έχουν στη γλώσσα C. Στην περίπτωση του τελεστή “;” αποτιμάται πρώτα το πρώτο ορισμά του και στη συνέχεια το δεύτερο όρισμά του, του οποίου η τιμή και επιστρέφεται. Οι τελεστές “**and**” και “**or**” (λογική σύζευξη και διάζευξη αντίστοιχα) *δεν* αποτιμούν το δεύτερο όρισμά τους, αν το αποτέλεσμα της λογικής έκφρασης μπορεί να συναχθεί από την τιμή του πρώτου ορίσματος.

- Εντολή συνθήκης: **if** $\langle$έκφραση$\rangle_1$ **then** $\langle$έκφραση$\rangle_2$ **else** $\langle$έκφραση$\rangle_3$, π.χ.

  ```
  if x==0 then 3 else 42
  ```

  Η σημασιολογία της έκφρασης είναι ή ίδια με την αντίστοιχη της γλώσσας C.

[1] Στο διερμηνέα και στο μεταγλωττιστή που έχουμε υλοποιήσει, χρησιμοποιείται ο τύπος Int της Haskell, που αντιστοιχεί σε ακέραιο σταθερού μήκους και σύμφωνα με τον ορισμό της Haskell98 [6, §6.4] καλύπτει τουλάχιστον τους ακεραίους από -2^{29} έως $2^{29}-1$.

- Λάμβδα έκφραση: **lambda** ⟨αναγνωριστικό⟩ : ⟨τύπος φράσης⟩ . ⟨έκφραση⟩, για συναρτήσεις κλήσης με τιμή (call-by-value), ή **lambda** ! ⟨αναγνωριστικό⟩ : ⟨τύπος φράσης⟩ . ⟨έκφραση⟩, για συναρτήσεις κλήσης με όνομα (call-by-name), π.χ.

```
lambda x:int exp. x+1
```

```
lambda !x:int exp. if x<0 then true else false
```

- Κλήση συνάρτησης: $\langle$έκφραση$\rangle_1$ $\langle$έκφραση$\rangle_2$, π.χ.

```
(lambda x:int exp. neg x) 5
```

```
f 42
```

- Δήλωση τοπικής μεταβλητής: **var** ⟨αναγνωριστικό⟩ : ⟨τύπος δεδομένων⟩ **in** ⟨έκφραση⟩. π.χ.

```
var i:int in i := 0; deref i
```

- Πρόσβαση στη τιμή μεταβλητής: **deref** ⟨αναγνωριστικό⟩, π.χ.

```
deref i
```

- Απόδοση τιμής σε μεταβλητή: $\langle$έκφραση$\rangle_1$:=$\langle$έκφραση$\rangle_2$. Πρώτα αποτιμάται η $\langle$έκφραση$\rangle_1$, στη συνέχεια η $\langle$έκφραση$\rangle_2$ και τέλος αποδίδεται η τιμή της $\langle$έκφρασης$\rangle_2$ στη θέση μνήμης που αντιστοιχεί στην $\langle$έκφραση$\rangle_1$. Όλη η έκφραση έπιστρέφει την τιμή της $\langle$έκφρασης$\rangle_2$. Π.χ.

```
i := 3
```

```
(3+5 ; i) := 42
```

```
f := (lambda x:int exp. x+42)
```

```
var f:bool exp -> bool exp in
       f := (lambda x:bool exp. not x);
       (deref f) true
```

- Αναδρομή: **rec** (⟨έκφραση⟩). Για παράδειγμα με την παρακάτω έκφραση ορίζουμε την παραγοντική συνάρτηση:

```
rec (lambda !f:(int exp -> int exp) exp.
                       lambda x:int exp. if x<=1 then 1 else x*f(x-1))
```

- Κενό record: **empty**.

- Επέκταση record: $\langle$έκφραση$\rangle_1$ [⟨αναγνωριστικό⟩ := $\langle$έκφραση$\rangle_2$], π.χ.

```
empty[i:=3]
```

```
var r:record(i:int exp, j:bool exp) in r:=empty[i:=3][j:=true]
```

- Πρόσβαση σε πεδίο record: ⟨έκφραση⟩ :: ⟨αναγνωριστικό⟩, π.χ.

```
x :: i

(deref r) :: x

empty[i:=3] :: i
```

2.1.5 Προτεραιότητα και προσεταιριστικότητα

Στον παρακάτω πίνακα δίνεται η προτεραιότητα των τελεστών και εκφράσεων της γλώσσας Cantor, ξεκινώντας από τη μεγαλύτερη και πηγαίνοντας προς τη μικρότερη προτεραιότητα (τελεστές ή εκφράσεις που βρίσκονται στην ίδια γραμμή του πίνακα έχουν την ίδια προτεραιότητα). Στον ίδιο πίνακα δίνεται επίσης και η προσεταιριστικότητα των τελεστών.

Τελεστής/Έκφραση	**Προσεταιριστικότητα**
rec	όχι
Κλήση συνάρτησης, ::, [:=]	←
neg, **not**	→
deref	όχι
∗, /, %	←
+, −	←
==, !=, <=, >=, <, >	όχι
and	←
or	←
:=	→
;	←

2.2 Η γραμματική της γλώσσας Cantor

Ακολουθεί ο ορισμός της γραμματικής της γλώσσας Cantor. Στην παράγραφο 2.2.1 δίνεται η αφηρημένη σύνταξη της γλώσσας, η οποία χρησιμοποιείται και στα επόμενα κεφάλαια για τον ορισμό των κανόνων τύπων, της δυναμικής σημασιολογίας, κ.λπ. Για λόγους πληρότητας ωστόσο, στην παράγραφο 2.2.2 δίνεται και η συγκεκριμένη σύνταξη (concrete syntax) της Cantor.

2.2.1 Αφηρημένη σύνταξη σε μορφή EBNF

Ακολουθεί η αφηρημένη σύνταξη της γλώσσας Cantor σε μορφή EBNF. Τα τερματικά σύμβολα αναγράφονται μέσα σε διπλά εισαγωγικά. Μέσα σε γωνιακές αγκύλες (⟨ ⟩) αναγράφονται τα μη τερματικά σύμβολα καθώς και οι ακόλουθες λεκτικές μονάδες:

⟨n⟩ : ακέραια σταθερά (χωρίς πρόσημο)
⟨id⟩ : αναγνωριστικό

Η αναγραφή μιας παράστασης μέσα σε παρενθέσεις ακολουθούμενες από αστερίσκο σημαίνει "μηδέν ή περισσότερες" επαναλήψεις της παράστασης αυτής.

⟨expr⟩	::=	⟨n⟩ | “**true**” | “**false**” | ⟨id⟩
	|	⟨unop⟩ ⟨expr⟩ | ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
	|	“**if**” ⟨expr⟩ “**then**” ⟨expr⟩ “**else**” ⟨expr⟩
	|	“**var**” ⟨id⟩ “:” ⟨data⟩ “**in**” ⟨expr⟩
	|	“**deref**” ⟨id⟩ | ⟨expr⟩ “:=” ⟨expr⟩
	|	“**lambda**” ⟨mode⟩ ⟨id⟩ “:” ⟨type⟩ “.” ⟨expr⟩ | ⟨expr⟩ ⟨expr⟩
	|	“**rec**” “(” ⟨expr⟩ “)”
	|	“**empty**” | ⟨expr⟩ “[” ⟨id⟩ “:=” ⟨expr⟩ “]” | ⟨expr⟩ “::” ⟨id⟩
⟨unop⟩	::=	“**neg**” | “**not**”
⟨binop⟩	::=	“+” | “−” | “∗” | “/” | “%”
	|	“==” | “! =” | “<” | “>” | “<=” | “>=”
	|	“**and**” | “**or**” | “;”
⟨data⟩	::=	“**int**” | “**bool**” | ⟨type⟩ “−>” ⟨type⟩
	|	“**record**” “(” ⟨id⟩ “:” ⟨type⟩ (“,” ⟨id⟩ “:” ⟨type⟩)* “)”
⟨type⟩	::=	⟨data⟩ “**exp**” | ⟨data⟩ “**var**”
⟨mode⟩	::=	“!” | ϵ

2.2.2 Συγκεκριμένη σύνταξη σε μορφή EBNF

Παρακάτω παρατίθεται η συγκεκριμένη σύνταξη της γλώσσας Cantor, όπως χρησιμοποιήθηκε για την κατασκευή του συντακτικού αναλυτή της γλώσσας. Σημειώνουμε ότι η παρακάτω γραμματική έχει υποστεί τους κατάλληλους μετασχηματισμούς ώστε να απαλειφθεί η (άμεση και έμμεση) αριστερή αναδρομικότητα.

Όπως και στην περίπτωση της αφηρημένης σύνταξης, μέσα σε γωνιακές αγκύλες παρατίθενται τα μη τερματικά σύμβολα, καθώς και οι παρακάτω λεκτικές μονάδες:

⟨n⟩ : ακέραια σταθερά (χωρίς πρόσημο)
⟨id⟩ : αναγνωριστικό

Μέσα σε διπλά εισαγωγικά παρατίθενται τα τερματικά σύμβολα.

⟨expr⟩	::=	⟨expr0⟩ ⟨expr0list⟩
⟨expr0list⟩	::=	“;” ⟨expr0⟩ ⟨expr0list⟩ | ϵ
⟨expr0⟩	::=	⟨expr1⟩ ⟨expr1list⟩
⟨expr1list⟩	::=	“:=” ⟨expr0⟩ | ϵ
⟨expr1⟩	::=	⟨expr2⟩ ⟨expr2list⟩
⟨expr2list⟩	::=	“**or**” ⟨expr2⟩ ⟨expr2list⟩ | ϵ
⟨expr2⟩	::=	⟨expr3⟩ ⟨expr3list⟩
⟨expr3list⟩	::=	“**and**” ⟨expr3⟩ ⟨expr3list⟩ | ϵ
⟨expr3⟩	::=	⟨expr4⟩ ⟨expr4list⟩
⟨expr4list⟩	::=	⟨compop⟩ ⟨expr4⟩ ⟨expr4list⟩ | ϵ
⟨compop⟩	::=	“==” | “! =” | “<” | “>” | “<=” | “>=”
⟨expr4⟩	::=	⟨expr5⟩ ⟨expr5list⟩
⟨expr5list⟩	::=	“+” ⟨expr5⟩ ⟨expr5list⟩ | “−” ⟨expr5⟩ ⟨expr5list⟩ | ϵ

⟨expr5⟩	::=	⟨expr6⟩ ⟨expr6list⟩
⟨expr6list⟩	::=	“∗” ⟨expr6⟩ ⟨expr6list⟩ | “/” ⟨expr6⟩ ⟨expr6list⟩
	|	“%” ⟨expr6⟩ ⟨expr6list⟩ | ϵ
⟨expr6⟩	::=	“**neg**” ⟨expr6⟩ | “**not**” ⟨expr6⟩ | ⟨expr7⟩
⟨expr7⟩	::=	“**if**” ⟨expr0⟩ “**then**” ⟨expr0⟩ “**else**” ⟨expr0⟩
	|	“**lambda**” “!” ⟨id⟩ “:” ⟨type⟩ “.” ⟨expr⟩
	|	“**lambda**” ⟨id⟩ “:” ⟨type⟩ “.” ⟨expr⟩
	|	“**var**” ⟨id⟩ “:” ⟨data⟩ “**in**” ⟨expr⟩ | “**deref**” ⟨aexpr⟩ ⟨aexprlist⟩
	|	⟨aexpr⟩ ⟨aexprlist⟩
⟨aexprlist⟩	::=	⟨aexpr⟩ ⟨aexprlist⟩ | “::” ⟨id⟩ ⟨aexprlist⟩ | “[” ⟨id⟩ “:=” ⟨expr⟩ “]” | ϵ
⟨aexpr⟩	::=	⟨n⟩ | “**true**” | “**false**” | ⟨id⟩ | “**empty**” | “(” ⟨expr⟩ “)” | “**rec**” “(” ⟨expr⟩ “)”
⟨data⟩	::=	⟨intboolrec⟩ ⟨data’⟩ | “(” ⟨data⟩ “)” ⟨data’⟩
	|	“(” ⟨data⟩ ⟨varexp⟩ “)” “–>” “(” ⟨data⟩ ⟨varexp⟩ “)” ⟨data’⟩
	|	“(” ⟨data⟩ ⟨varexp⟩ “)” “–>” ⟨data⟩ ⟨varexp⟩ ⟨data’⟩
⟨data’⟩	::=	⟨varexp⟩ “–>” ⟨data⟩ ⟨varexp⟩ ⟨data’⟩
	|	⟨varexp⟩ “–>” “(” ⟨data⟩ ⟨varexp⟩ “)” ⟨data’⟩ | ϵ
⟨intboolrec⟩	::=	“**int**” | “**bool**” | “**record**” “(” ⟨varlist⟩ “)” | “(” ⟨intboolrec⟩ “)”
⟨varexp⟩	::=	“**var**” | “**exp**” | “(” ⟨varexp⟩ “)”
⟨type⟩	::=	⟨data⟩ ⟨varexp⟩ | “(” ⟨type⟩ “)”
⟨varlist⟩	::=	⟨decl⟩ ⟨vars⟩
⟨decl⟩	::=	⟨id⟩ “:” ⟨type⟩
⟨vars⟩	::=	“,” ⟨decl⟩ | ϵ

Κεφάλαιο 3

Έλεγχος τύπων

Στο παρόν κεφάλαιο περιγράφονται η σημασιολογία τύπων της γλώσσας Cantor και ο ελεγκτής τύπων που έχει κατασκευαστεί με βάση αυτή τη σημασιολογία (παράγραφος 3.2). Καθώς η υλοποίηση του ελεγκτή τύπων (αλλά και των διερμηνέα & μεταγλωττιστή όπως θα δούμε παρακάτω) χρησιμοποιεί μονάδες, προηγείται στην παράγραφο 3.1 μια συνοπτική περιγραφή των μονάδων.

3.1 Μονάδες

3.1.1 Ορισμός

Οι μονάδες μπορούν να οριστούν είτε με αυστηρά μαθηματικό τρόπο μέσω της θεωρίας κατηγοριών [7], είτε μέσω της συναρτησιακής προσέγγισης που έχει ακολουθήσει ο Wadler στο [8]. Καθώς σκοπός της παρούσας παραγράφου είναι μόνο η συνοπτική περιγραφή των μονάδων, ώστε να διευκολυνθεί η κατανόηση των υλοποιήσεων που έχουμε κάνει, παρακάτω παρατίθεται μόνο η συναρτησιακή προσέγγιση του ορισμού της μονάδας. Ο αναγνώστης μπορεί να ανατρέξει στο [7] για μια αναλυτικότερη εισαγωγή στις μονάδες.

Ως μονάδα ορίζουμε μια τριάδα `(M, unit, bind)`, η οποία αποτελείται από έναν type constructor `M` και δύο πολυμορφικές συναρτήσεις[1]:

```
unit   ::   a -> M a
bind   ::   M a -> (a -> M b) -> M b
```

για τις οποίες πρέπει να ισχύουν οι παρακάτω σχέσεις, γνωστές και ως νόμοι των μονάδων (monad laws):

```
(unit a) `bind` k      = k a
m `bind` unit          = m
m `bind` (\a -> (k a) `bind` (\b -> h b)) =
                            (m `bind` (\a -> k a)) `bind` (\b -> h b)
```

Οι τρεις παραπάνω σχέσεις αντιστοιχούν στο αριστερό ουδέτερο στοιχείο, στο δεξί ουδέτερο στοιχείο και στην προσεταιριστική ιδιότητα.

Διαισθητικά μπορεί να ειπωθεί ότι το νόημα της `unit v` είναι η μετατροπή μιας τιμής `v::Value` σε έναν υπολογισμό (computation) `m :: M Value`, ενώ η σημασία της

```
m `bind` (\v -> k)
```

είναι κατά σειρά η αποτίμηση του `m`, η ανάθεση του αποτελέσματος στο `v` και τέλος η αποτίμηση του `k`.

[1] Χρησιμοποιούμε τον συμβολισμό της γλώσσας Haskell για συναρτήσεις και εκφράσεις λάμβδα.

3.1.2 Μονάδες και Haskell

Η Haskell ορίζει την Monad class, η οποία για την συνάρτηση `unit` χρησιμοποιεί την ονομασία `return`, ενώ για την `bind` χρησιμοποιεί το σύμβολο `>>=`. Επιπλέον, η Haskell παρέχει τον do-συμβολισμό (do notation, [6, §3.14]), προκειμένου να διευκολύνει την χρήση των μονάδων. Έτσι η έκφραση

```
m >>= (\a -> k a >>= (\b -> return (h b)))
```

μπορεί να γραφεί και ως:

```
do { a <- m ; b <- k a ; return (h b) }
```

ή κάνοντας χρήση της στοίχισης (layout, [6, §2.7]):

```
do a <- m
   b <- k a
   return (h b)
```

O do-συμβολισμός είναι αυτός που χρησιμοποιείται στα αποσπάσματα κώδικα που παρουσιάζονται στο υπόλοιπο της εργασίας.

3.2 Ελεγκτής τύπων

3.2.1 Ορισμός απαραίτητων μονάδων

O ελεγκτής τύπων πρέπει αφενός να μπορεί να αντιστοιχεί έναν τύπο σε κάθε αναγνωριστικό που συναντά (στατικό περιβάλλον), αφετέρου να έχει τη δυνατότητα όταν εντοπίσει κάποιο λάθος να τυπώνει το κατάλληλο μήνυμα. Ως εκ τούτου, είναι αναγκαίο να οριστούν οι κατάλληλες μονάδες, δηλ. μια μονάδα χειρισμού λαθών και μια μονάδα περιβάλλοντος. Επιπλέον, θα πρέπει οι μονάδες αυτές να μπορούν να συνδυαστούν μεταξύ τους. Έτσι, όπως περιγράφεται στο [9], ορίζουμε μια ταυτοτική μονάδα (`Id`), και δύο μετασχηματιστές μονάδων `ErrorT` και `EnvT`.

O ορισμός της ταυτοτικής μονάδας είναι απλός:

```
newtype Id a            = MkId a

instance Monad Id where
        return          = MkId
        MkId m >>= k    = k m
```

Προχωρώντας στη μονάδα χειρισμού λαθών, ορίζουμε τον τύπο Error:

```
data Error a            = Ok a | Error String
```

με το `String` να αντιστοιχεί στο εκάστοτε μήνυμα λάθους. Εν συνεχεία ορίζουμε το μετασχηματιστή μονάδας `ErrorT`, ο οποίος ανήκει στην class `Monad` και για τον οποίον ορίζονται οι συναρτήσεις `return` και `>>=`:

```
newtype ErrorT m a      = MkErrT {fromErrT :: m (Error a) }

instance Monad m => Monad (ErrorT m) where
        return a        = MkErrT (return (Ok a))
        MkErrT m >>= k  = MkErrT (m >>= r)
                          where r (Error s)     = return (Error s)
                                r (Ok x)        = fromErrT (k x)
```

Ορίζουμε ακόμη την class ErrMonad, η οποία έχει μοναδικό μέλος τη συνάρτηση err, που θα καλείται από τον ελεγκτή τύπων όταν εντοπίζεται λάθος. Καθιστούμε το ErrorT μέλος της class ErrMonad:

```
class Monad m => ErrMonad m where
        err     :: String -> m a

instance Monad m => ErrMonad (ErrorT m) where
        err s   = MkErrT (return (Error s))
```

Τέλος, ακολουθώντας πάντα το [9], είναι απαραίτητο για κάθε μετασχηματιστή μονάδας t να οριστεί η συνάρτηση lift :: m a -> t m a, όπου m μονάδα. Για το σκοπό αυτό ορίζουμε την class MonadT, με μοναδικό μέλος τη συνάρτηση lift, και καθιστούμε το ErrorT μέλος της class αυτής. Σημειώνουμε εδώ ότι η συνάρτηση map_m που χρησιμοποιείται στο [9] για τον ορισμό της lift, παρέχεται από το Monad module της Haskell με την ονομασία liftM.

```
class MonadT t where
        lift    :: Monad m => m a -> t m a

instance MonadT ErrorT where
        lift    = MkErrT . liftM Ok
```

Με εντελώς παρόμοιο τρόπο ορίζουμε και τη μονάδα χειρισμού περιβάλλοντος, την οποία ονομάζουμε EnvT, και την class EnvMonad, με μέλη τις συναρτήσεις rdEnv :: m env (επιστρέφει το τρέχον περιβάλλον) και inEnv :: env -> m a -> m a (αποτιμά[2] το δεύτερο όρισμά της χρησιμοποιώντας ως περιβάλλον το πρώτο ορισμά της).

```
newtype EnvT env m a    = EnvT {fromEnvT :: env -> m a}

instance Monad m => Monad (EnvT env m) where
        return a        = EnvT (\e -> return a)
        EnvT m >>= k    = EnvT (\e -> do a <- m e
                                         fromEnvT (k a) e)

class Monad m => EnvMonad env m where
        rdEnv           :: m env
        inEnv           :: env -> m a -> m a

instance Monad m => EnvMonad env (EnvT env m) where
        rdEnv           = EnvT (\r -> return r)
        inEnv env m     = EnvT (\env' -> fromEnvT m env)

instance MonadT (EnvT env) where
        lift c          = EnvT (\env -> c)
```

3.2.2 Ορισμός τύπων

Μεταφέροντας με εντελώς ευθύ τρόπο την αφηρημένη σύνταξη που ορίστηκε στην § 2.2.1, και αντιστοιχίζοντας τα μη τερματικά σύμβολα ⟨expr⟩, ⟨unop⟩, ⟨binop⟩, ⟨data⟩, ⟨type⟩, ⟨mode⟩ στους τύπους Expr, UnOp, BinOp, Data, Type και Mode ορίζουμε τους παρακάτω τύπους:

```
type    I       = String
```

[2] Σωστότερα επιστρέφει έναν υπολογισμό (computation) που αντιστοιχεί σε αυτή την αποτίμηση.

```
data    Expr    = Con Int                --σταθερά τύπου Int
                | MyTrue                 --σταθερά τύπου bool
                | MyFalse
                | Ide I                  --αναγνωριστικό
                | Unexp UnOp Expr        --unary έκφραση (neg ή not)
                | Bexp BinOp Expr Expr   --binary έκφραση
                | If Expr Expr Expr      --εντολή συνθήκης
                | Var I Data Expr        --δήλωση τοπικής μεταβλητής
                | Deref Expr             --πρόσβαση στην τιμή μεταβλητής
                | Assign Expr Expr       --ανάθεση τιμής σε μεταβλητή
                | Lam Mode I Type Expr   --λάμδα έκφραση
                | Call Expr Expr         --κλήση συνάρτησης
                | MyRec Expr             --αναδρομή
                | RecAccess Expr I       --πρόσβαση σε πεδίο record
                | RecExtend Expr I Expr  --επέκταση record
                | Empty                  --κενό record
                deriving Show

data    UnOp    = Not | Neg
                deriving (Eq, Show)

data    BinOp   = Plus | Minus | Mul | Div | Mod
                | Eq' | Ne | Gt | Lo | Ge | Le
                | And | Or | Semicolon
                deriving (Eq, Show)

data    Data    = MyInt
                | MyBool
                | Func Type Type
                | Recor [(I, Type)]
                deriving (Eq, Show)

data    Type    = Exp Data
                | Vari Data
                deriving (Eq, Show)

data    Mode    = BV   --κλήση με τιμή (call-by-value)
                | BN   --κλήση με όνομα (call-by-name)
                deriving Show
```

Σημειώνουμε εδώ, ότι η ένταξη του τύπου `Data` στην class `Eq`, συνεπάγεται ότι τα μέλη της λίστας `[(I, Type)]` του data constructor `Recor` είναι ταξινομημένα ως προς το πεδίο `I`. Αυτό γίνεται μόνο εσωτερικά για λόγους διευκόλυνσης της υλοποίησης του ελεγκτή τύπων, και δεν αλλάζει όσα ορίσαμε στην παράγραφο 2.1.2, σύμφωνα με τα οποία τα records είναι σύνολα *μη διατεταγμένων* πεδίων.

3.2.3 Τυπική σημασιολογία και ελεγκτής τύπων

Μετά τον ορισμό των απαραίτητων μονάδων και βοηθητικών τύπων, είμαστε έτοιμοι να προχωρήσουμε στην υλοποίηση του κυρίως ελεγκτή τύπων. Ο έλεγχος τύπων εκτελείται από τη συνάρτηση

```
gettype          :: Expr -> Typecheck Type
```

όπου `Typecheck` κατάλληλη μονάδα, και πιο συγκεκριμένα

```
type Typecheck a         = EnvT StaticEnv (ErrorT Id) a
```

με

```
type StaticEnv           = I -> Maybe Type
```

Βλέπουμε ότι η μονάδα που χρησιμοποιούμε είναι σύνθεση της μονάδας περιβάλλοντος `EnvT` και της μονάδας διαχείρισης λαθών `ErrorT`. Για να μπορούμε ωστόσο να καλούμε τη συνάρτηση `err` της μονάδας διαχείρισης λαθών, αναγκαία προϋπόθεση είναι να κάνουμε την παρακάτω δήλωση:

```
instance ErrMonad m => ErrMonad (EnvT StaticEnv m) where
           err           = lift . err
```

Πριν προχωρήσουμε στην τυπική σημασιολογία της Cantor, είναι απαραίτητο να ορίσουμε τα στατικά σύνολα (static domains):

$\tau : \mathbf{TypeDat} = int \mid bool \mid func(\theta_1, \theta_2) \mid record\ [(I, \theta)]$
$\theta : \mathbf{TypePhr} = exp(\tau) \mid var(\tau)$
$e : \mathbf{TypeEnv} = \mathbf{Ide} \to \mathbf{TypePhr} \oplus \mathbf{U}$

όπου **Ide** το σύνολο των αναγνωριστικών, και **U** το σύνολο με μοναδικό στοιχείο το $\perp$. Στα πεδία τ, θ και e αντιστοιχούν οι τύποι `Data`, `Type` και `StaticEnv` που ορίσαμε παραπάνω, δηλαδή οι τύποι δεδομένων, φράσεων και στατικού περιβάλλοντος. Έστω επίσης οι συναρτήσεις

$\{\!|\langle id\rangle|\!\} : \mathbf{Ide}$
$\{\!|\langle data\rangle|\!\} : \mathbf{TypeDat}$
$\{\!|\langle type\rangle|\!\} : \mathbf{TypePhr}$

που απεικονίζουν τα συντακτικά σύνολα των αναγνωριστικών, τύπων δεδομένων και τύπων φράσεων στα αντίστοιχα στατικά σύνολα.

- Για σταθερές τύπου int και bool ισχύουν οι παρακάτω κανόνες τύπων:

  $$\frac{}{e \vdash \langle n\rangle : exp(int)}$$

  $$\frac{}{e \vdash \mathbf{true} : exp(bool)} \qquad \frac{}{e \vdash \mathbf{false} : exp(bool)}$$

  οι οποίοι μεταφράζονται με ευθύ τρόπο στον παρακάτω κώδικα Haskell:

  ```
  gettype (Con n)         = return (Exp MyInt)
  gettype MyTrue          = return (Exp MyBool)
  gettype MyFalse         = return (Exp MyBool)
  ```

- Αναγνωριστικά:

  $$\frac{Id = \{\!|\langle id\rangle|\!\} \qquad e\ Id = inl\ \theta}{e \vdash \langle id\rangle : \theta}$$

  Κώδικας Haskell:

```
gettype (Ide i)      = do e <- rdEnv
                          case e i of
                            Just t -> return t
                            Nothing -> err ("unbound variable: " ++ i)
```

Παρατηρούμε ότι ο παραπάνω κώδικας καλεί την `rdEnv` η οποία επιστρέφει το τρέχον περιβάλλον. Σε περίπτωση που δεν υπάρχει το αναγνωριστικό "`i`" στο περιβάλλον αυτό, καλείται η `err` με όρισμα το κατάλληλο μήνυμα λάθους. Αξιοποιούνται έτσι οι μονάδες που ορίσαμε στην § 3.2.1.

- Εκφράσεις **neg** και **not**:

$$\frac{e \vdash \langle expr \rangle : exp(bool)}{e \vdash \textbf{not}\ \langle expr \rangle : exp(bool)} \qquad \frac{e \vdash \langle expr \rangle : exp(int)}{e \vdash \textbf{neg}\ \langle expr \rangle : exp(int)}$$

Κώδικας Haskell:

```
gettype (Unexp Not e)
                 = do t <- gettype e
                      case t of
                        Exp MyBool -> return t
                        _ -> err ("not: argument should be boolean")
gettype (Unexp Neg e)
                 = do t <- gettype e
                      case t of
                        Exp MyInt -> return t
                        _ -> err ("neg: argument should be integer")
```

- Δυαδικές εκφράσεις με έναν από τους τελεστές $+$, $-$, $*$, $/$, $\%$: Τα ορίσματά τους όπως και τα αποτελέσματά τους πρέπει να είναι τύπου $exp(int)$, οπότε προκύπτουν οι παρακάτω κανόνες τύπων:

$$\frac{\begin{array}{c} e \vdash \langle expr \rangle_1 : exp(int) \\ e \vdash \langle expr \rangle_2 : exp(int) \end{array}}{e \vdash \langle expr \rangle_1 + \langle expr \rangle_2 : exp(int)} \qquad \frac{\begin{array}{c} e \vdash expr_1 : exp(int) \\ e \vdash \langle expr \rangle_2 : exp(int) \end{array}}{e \vdash \langle expr \rangle_1 - \langle expr \rangle_2 : exp(int)}$$

$$\frac{\begin{array}{c} e \vdash \langle expr \rangle_1 : exp(int) \\ e \vdash \langle expr \rangle_2 : exp(int) \end{array}}{e \vdash \langle expr \rangle_1 * \langle expr \rangle_2 : exp(int)} \qquad \frac{\begin{array}{c} e \vdash \langle expr \rangle_1 : exp(int) \\ e \vdash \langle expr \rangle_2 : exp(int) \end{array}}{e \vdash \langle expr \rangle_1 / \langle expr \rangle_2 : exp(int)}$$

$$\frac{e \vdash \langle expr \rangle_1 : exp(int) \qquad e \vdash \langle expr \rangle_2 : exp(int)}{e \vdash \langle expr \rangle_1 \% \langle expr \rangle_2 : exp(int)}$$

Ο αντίστοιχος κώδικας σε Haskell είναι:

```
gettype (Bexp Plus e1 e2)
                     = do t1 <- gettype e1
                          case t1 of
                            Exp MyInt ->
                                do t2 <- gettype e2
                                   case t2 of
                                     Exp MyInt -> return (Exp MyInt)
                                     _ -> err ("+ : illegal type")
                            _ -> err ("+ : illegal type")
```

Ο κώδικας είναι αντίστοιχος και για τους τελεστές $-$, $*$, $/$ και $\%$ (`Minus`, `Mul`, `Div` κα `Mod` αντίστοιχα).

- Δυαδικές εκφράσεις με έναν από τους τελεστές **and** και **or**: Σε αυτήν την περίπτωση ορίσματα και αποτελέσματα πρέπει να είναι τύπου $exp(bool)$.

  Κανόνες τύπων:

$$\frac{e \vdash \langle expr\rangle_1 : exp(bool) \quad e \vdash \langle expr\rangle_2 : exp(bool)}{e \vdash \langle expr\rangle_1 \textbf{ and } \langle expr\rangle_2 : exp(bool)} \qquad \frac{e \vdash \langle expr\rangle_1 : exp(bool) \quad e \vdash \langle expr\rangle_2 : exp(bool)}{e \vdash \langle expr\rangle_1 \textbf{ or } \langle expr\rangle_2 : exp(bool)}$$

  Κώδικας Haskell:

```
gettype (Bexp And e1 e2)
                    = do t1 <- gettype e1
                         case t1 of
                           Exp MyBool ->
                               do t2 <- gettype e2
                                  case t2 of
                                    Exp MyBool -> return (Exp MyBool)
                                    _ -> err ("and: illegal type")
                         _ -> err ("and: illegal type")
```

  Αντίστοιχος είναι ο κώδικας για τον τελεστή **or** (`Or`).

- Δυαδικές εκφράσεις με έναν από τους τελεστές $<=$, $>=$, $<$, $>$: Τα ορίσματα πρέπει να είναι τύπου $exp(int)$, ο τύπος όμως όλης της έκφρασης είναι $exp(bool)$.

  Κανόνες τύπων:

$$\frac{e \vdash \langle expr\rangle_1 : exp(int) \quad e \vdash \langle expr\rangle_2 : exp(int)}{e \vdash \langle expr\rangle_1 <= \langle expr\rangle_2 : exp(bool)} \qquad \frac{e \vdash \langle expr\rangle_1 : exp(int) \quad e \vdash \langle expr\rangle_2 : exp(int)}{e \vdash \langle expr_1\rangle >= \langle expr\rangle_2 : exp(bool)}$$

$$\frac{e \vdash \langle expr\rangle_1 : exp(int) \quad e \vdash \langle expr\rangle_2 : exp(int)}{e \vdash \langle expr\rangle_1 < \langle expr\rangle_2 : exp(bool)} \qquad \frac{e \vdash \langle expr\rangle_1 : exp(int) \quad e \vdash \langle expr\rangle_2 : exp(int)}{e \vdash \langle expr\rangle_1 > \langle expr\rangle_2 : exp(bool)}$$

  Κώδικας ελεγκτή τύπων:

```
gettype (Bexp Le e1 e2) = do t1 <- gettype e1
                             case t1 of
                               Exp MyInt ->
                                 do t2 <- gettype e2
                                    case t2 of
                                      Exp MyInt -> return (Exp MyBool)
                                      _ -> err ("<= : illegal type")
                               _ -> err ("<= : illegal type")
```

  Παρόμοιος είναι ο κώδικας για τους τελεστές $>=$, $<$ και $>$ (`Ge`, `Lo` και `Gt` αντίστοιχα).

- Δυαδικές εκφράσεις σύγκρισης $==$ και $!=$: Εδώ ο τύπος της έκφρασης είναι $exp(bool)$, ο τύπος των ορισμάτων όμως μπορεί να είναι είτε $exp(int)$ είτε $exp(bool)$, με την προϋπόθεση ότι τα δύο ορίσματα πρέπει να έχουν τον ίδιο τύπο.

  Κανόνες τύπων:

$$\frac{e \vdash \langle expr\rangle_1 : exp(int) \quad e \vdash \langle expr\rangle_2 : exp(int)}{e \vdash \langle expr\rangle_1 == \langle expr\rangle_2 : exp(bool)} \qquad \frac{e \vdash \langle expr\rangle_1 : exp(bool) \quad e \vdash \langle expr\rangle_2 : exp(bool)}{e \vdash \langle expr\rangle_1 == \langle expr\rangle_2 : exp(bool)}$$

$$\frac{e \vdash \langle expr\rangle_1 : exp(int) \quad e \vdash \langle expr\rangle_2 : exp(int)}{e \vdash \langle expr\rangle_1 != \langle expr\rangle_2 : exp(bool)} \qquad \frac{e \vdash \langle expr\rangle_1 : exp(bool) \quad e \vdash \langle expr\rangle_2 : exp(bool)}{e \vdash \langle expr\rangle_1 != \langle expr\rangle_2 : exp(bool)}$$

Κώδικας Haskell:

```
gettype (Bexp Eq' e1 e2) = do t1 <- gettype e1
                              t2 <- gettype e2
                              case (t1, t2) of
                                (Exp MyInt, Exp MyInt) ->
                                        return (Exp MyBool)
                                (Exp MyBool, Exp MyBool) ->
                                        return (Exp MyBool)
                                _ -> err ("== : illegal type")
```

Παραλείπεται ο κώδικας που αντιστοιχεί στον τελεστή $!=$ (`Ne`), που είναι εντελώς παρόμοιος.

- Δυαδική έκφραση με τον τελεστή ";" : Στην προκειμένη περίπτωση, δεν υπάρχει περιορισμός για τον τύπο των δύο ορισμάτων (εκτός φυσικά ότι πρέπει να είναι κάποιος τύπος φράσης $\theta \in$ **TypePhr**). Ο τύπος της έκφρασης είναι ο τύπος του δεύτερου ορίσματος.

  Κανόνας τύπων:

$$\frac{e \vdash \langle expr\rangle_1 : \theta_1 \quad e \vdash \langle expr\rangle_2 : \theta_2}{e \vdash \langle expr\rangle_1 \; ; \; \langle expr\rangle_2 : \theta_2}$$

  Κώδικας Haskell:

```
gettype (Bexp Semicolon e1 e2) = do _ <- gettype e1
                                    t2 <- gettype e2
                                    return t2
```

- Εντολή συνθήκης: Η έκφραση που ακολουθεί το **if** πρέπει προφανώς να είναι τύπου $exp(bool)$, ενώ οι εκφράσεις που ακολουθούν τα **then** και **else** πρέπει να έχουν τον ίδιο τύπο $\theta \in$ **TypePhr**, ο οποίος θα είναι και ο τύπος όλης της έκφρασης.

  Κανόνας τύπων:

$$\frac{e \vdash \langle expr\rangle_1 : exp(bool) \quad e \vdash \langle expr\rangle_2 : \theta \quad e \vdash \langle expr\rangle_3 : \theta}{e \vdash \textbf{if } \langle expr\rangle_1 \textbf{ then } \langle expr\rangle_2 \textbf{ else } \langle expr\rangle_3 : \theta}$$

  Κώδικας Haskell:

```
gettype (If e1 e2 e3)
                  = do t1 <- gettype e1
                       case t1 of
                         Exp MyBool ->
                            do t2 <- gettype e2
                               t3 <- gettype e3
                               if t2==t3
                                then return t3
                                else err("if : illegal types")
                         _ -> err ("if : boolean expression expected")
```

- Λάμβδα εκφράσεις: Η δήλωση της παραμέτρου πρέπει να ακολουθείται από κάποιον τύπο φράσης θ_1, και αν, μετά την κατάλληλη επέκταση του στατικού περιβάλλοντος[3], θ_2 είναι ο τύπος που προκύπτει για το «σώμα» της έκφρασης λάμβδα, τότε ο τύπος όλης της έκφρασης είναι $exp(func(\theta_1, \theta_2))$. Σημειώνουμε εδώ ότι ο τρόπος κλήσης (με τιμή ή με όνομα) είναι αδιάφορος κατά τον έλεγχο τύπων, καθώς υπάρχει ένας και μόνο τύπος συνάρτησης.

  Κανόνας τύπων:

$$\frac{Id = [\![\langle id \rangle]\!] \quad \theta_1 = [\![\langle type \rangle]\!] \quad e\{Id \mapsto \theta_1\} \vdash \langle expr \rangle : \theta_2}{e \vdash \textbf{lambda}\ \langle mode \rangle\ \langle id \rangle : \langle type \rangle . \langle expr \rangle : exp(func(\theta_1, \theta_2))}$$

  Κώδικας Haskell:

```
gettype (Lam m x t v)= do e <- rdEnv
                          t2 <- inEnv (extendEnv e (x, t)) (gettype v)
                          return (Exp (Func t t2))
```

  Η βοηθητική συνάρτηση `extendEnv` ορίζεται ως εξής:

```
extendEnv              :: StaticEnv -> (I, Type) -> StaticEnv
extendEnv e (i, t)     = \i' -> if i==i' then Just t else e i'
```

- Κλήση συνάρτησης:

  Κανόνας τύπων:

$$\frac{e \vdash \langle expr \rangle_1 : exp(func(\theta_1, \theta_2)) \quad e \vdash \langle expr \rangle_2 : \theta_1}{e \vdash \langle expr \rangle_1\ \langle expr \rangle_2 : \theta_2}$$

  Κώδικας Haskell:

```
gettype (Call f u)
           = do t1 <- gettype f
                case t1 of
                  Exp (Func x y) ->
                     do t2 <- gettype u
                        if t2 == x
                         then return y
                         else err ("call : illegal type of argument")
                     _ -> err("call: " ++ show f ++ " not a function")
```

- Δήλωση τοπικής μεταβλητής: Το όνομα της μεταβλητής πρέπει να ακολουθείται από ένα νόμιμο τύπο δεδομένων τ, ενώ το στατικό περιβάλλον ενημερώνεται κατάλληλα όπως στην περίπτωση των εκφράσεων λάμβδα.

  Κανόνας τύπων:

$$\frac{Id = [\![\langle id \rangle]\!] \quad \tau = [\![\langle data \rangle]\!] \quad e\{Id \mapsto var(\tau)\} \vdash \langle expr \rangle : \theta}{e \vdash \textbf{var}\ \langle id \rangle : \langle data \rangle\ \textbf{in}\ \langle expr \rangle : \theta}$$

  Κώδικας Haskell:

[3] Ο συμβολισμός $e\{Id \mapsto \theta\}$ χρησιμοποιείται για να συμβολίσει την επέκταση του υπάρχοντος περιβάλλοντος e με την αντιστοίχιση του τύπου θ στο αναγνωριστικό Id.

```
gettype (Var i d exp)   = do e <- rdEnv
                             inEnv (extendEnv e (i, Vari d)) (gettype exp)
```

- Πρόσβαση στην τιμή τοπικής μεταβλητής: Το όρισμα του τελεστή **deref** θα πρέπει να έχει τύπο $var(\tau)$, με $\tau \in$ **TypeDat**. Ο τύπος όλης της έκφρασης θα είναι τότε $exp(\tau)$.

  Κανόνας τύπων:

  $$\frac{e \vdash \langle expr \rangle : var(\tau)}{e \vdash \textbf{deref}\ \langle expr \rangle : exp(\tau)}$$

  Κώδικας Haskell:

```
gettype (Deref exp)     = do i <- gettype exp
                               case i of
                                 Vari t -> return (Exp t)
                                 _ -> err ("deref: illegal argument")
```

- Ανάθεση τιμής σε τοπική μεταβλητή: Το πρώτο τελούμενο του τελεστή := θα πρέπει να έχει τύπο $var(\tau)$ και το δεύτερο $exp(\tau)$, ενώ όλη η έκφραση θα έχει τύπο $exp(\tau)$.

  Ο κανόνας τύπων που προκύπτει είναι ο:

  $$\frac{e \vdash \langle expr \rangle_1 : var(\tau) \qquad e \vdash \langle expr \rangle_2 : exp(\tau)}{e \vdash \langle expr \rangle_1 := \langle expr \rangle_2 : exp(\tau)}$$

  Ο αντίστοιχος κώδικας του ελεγκτή τύπων έχει ως εξής:

```
gettype (Assign exp1 exp2)
                        = do i <- gettype exp1
                               case i of
                                 Vari t ->
                                   do t1 <- gettype exp2
                                      if t1 == Exp t
                                       then
                                        return t1
                                       else
                                        err ("assignment: illegal rvalue")
                                 _ -> err ("assignment: illegal lvalue")
```

- Αναδρομή **rec** $\langle expr \rangle$: Απαιτούμε το $\langle expr \rangle$ να έχει τύπο $exp(func(\theta,\theta))$. Τότε όλη η έκφραση θα έχει τύπο θ.

  Κανόνας τύπων:

  $$\frac{e \vdash \langle expr \rangle : exp(func(\theta,\theta))}{e \vdash \textbf{rec}\ (\langle expr \rangle) : \theta}$$

  Κώδικας Haskell:

```
gettype (MyRec e)  = do t <- gettype e
                          case t of
                            Exp (Func x y) ->
                                       if x==y then return x
                                       else err ("rec: illegal type")
                            _ -> err ("rec: illegal type")
```

- Κενό record (**empty**): Έχει τύπο $exp(record[])$. Με [] συμβολίζουμε την κενή λίστα.

  Κανόνας τύπων:

$$\frac{}{e \vdash empty : exp(record\;[])}$$

  Κώδικας Haskell:

```
gettype Empty           = return (Exp (Recor []))
```

- Επέκταση record $\langle expr_1\rangle[\langle id\rangle := \langle expr_2\rangle]$: Απαιτούμε η $expr_1$ να έχει τύπο $exp(record\;\pi)$ και η λίστα π να *μην* περιέχει ζεύγος με πρώτο μέλος το αναγνωριστικό $\langle id\rangle$. Αν ισχύουν οι προϋποθέσεις αυτές, κι αν υποθέσουμε ότι η $expr_2$ έχει τύπο $\theta \in$ **TypePhr**, τότε όλη η έκφραση θα έχει τύπο $exp(record\;\pi')$. Η λίστα π' προκύπτει αν προσθέσουμε στην π το ζεύγος $(\langle id\rangle,\;\theta)$, διατηρώντας όμως τη λίστα ταξινομημένη ως προς το πρώτο στοιχείο κάθε ζεύγους[4]. Η ταξινόμηση είναι αναγκαία, ώστε δύο records που έχουν ακριβώς τα ίδια πεδία (ως προς το όνομά τους *και* ως προς τον τύπο τους), αλλά με διαφορετική διάταξη, να έχουν τον ίδιο τύπο.

  Κανόνας τύπων:

$$\frac{\begin{array}{cc} e \vdash \langle expr\rangle_1 : exp(record\;\pi) \quad Id = \{\!|\langle id\rangle|\!\} & (Id,\;\theta') \notin \pi \\ e \vdash \langle expr\rangle_2 : \theta & \pi' = (Id,\;\theta) \mapsto_s \pi \end{array}}{e \vdash \langle expr\rangle_1[\langle id\rangle := \langle expr\rangle_2] : exp(record\;\pi')}$$

  Κώδικας Haskell:

```
gettype (RecExtend exp1 id exp2)
       = do r <- gettype exp1
            case r of
              Exp (Recor pi)
                 -> case lookup id pi of
                      Just d ->
                        err ("record extension: field already exists")
                      Nothing ->
                        do t <- gettype exp2
                           return (Exp (Recor (a ++ [(id, t)] ++ b)))
                           where (a,b) = span ((< id).fst) pi
              _ -> err ("record extension: not a record")
```

- Πρόσβαση σε πεδίο record $\langle expr\rangle :: \langle id\rangle$: Πρέπει το $\langle expr\rangle$ να είναι τύπου $exp(record\;\pi)$ και το αναγνωριστικό $\langle id\rangle$ να υπάρχει στη λίστα π μαζί με τον τύπο του, $\theta \in$ **TypePhr**. Ο τύπος όλης της έκφρασης είναι θ.

  Κανόνας τύπων:

$$\frac{e \vdash \langle expr\rangle : exp(record\;\pi) \quad Id = \{\!|\langle id\rangle|\!\} \quad (Id,\;\theta) \in \pi}{e \vdash \langle expr\rangle :: \langle id\rangle : \theta}$$

  Κώδικας Haskell:

```
gettype (RecAccess exp i)
       = do r <- gettype exp
            case r of
```

[4] Ο συμβολισμός $\mapsto_s$ χρησιμοποιείται για την εισαγωγή στοιχείου σε ταξινομημένη λίστα.

```
Exp (Recor l) ->
    let v = lookup i l in
        case v of
          Just t -> return t
          Nothing ->err ("record access: no field "++ i)
_ -> err ("record access: not a record")
```

Κεφάλαιο 4

Δυναμική σημασιολογία και διερμηνέας

Σκοπός του κεφαλαίου αυτού είναι να περιγράψει την υλοποίηση του διερμηνέα (interpreter) της γλώσσας Cantor. Ορίζει επίσης τη δυναμική σημασιολογία της γλώσσας, στην οποία βασίζεται ο διερμηνέας.

4.1 Ορισμός μονάδων

Ήδη από το Κεφάλαιο 3 έχουμε ορίσει μια μονάδα χειρισμού λαθών και μια μονάδα διαχείρισης περιβάλλοντος. Η τελευταία χρησιμοποιήθηκε για τη διαχείριση του *στατικού* περιβάλλοντος κατά τη διάρκεια ελέγχου τύπων, και θα χρησιμοποιηθεί και από το διερμηνέα, για τη διαχείριση του *δυναμικού* περιβάλλοντος αυτή τη φορά. Καθώς όμως η γλώσσα Cantor διαθέτει τοπικές μεταβλητές με δυνατότητα ανάθεσης τιμών σε αυτές, είναι απαραίτητο ο διερμηνέας να μπορεί να υποστηρίξει αυτό το χαρσκτηριστικό. Για το σκοπό αυτό θα ορίσουμε και μια μονάδα κατάστασης (state monad), ακολουθώντας το [9].

Καταρχήν, ορίζουμε τον τύπο `StateT`, που είναι ένας μετασχηματιστής μονάδας που προσθέτει δυνατότητες χειρισμού κατάστασης σε μια οποιαδήποτε μονάδα `m`:

```
newtype StateT s m a    = StateT { runStateT :: s -> m (a, s)}
```

Στη συνέχεια, δηλώνουμε ότι το `StateT` ανήκει στην class `Monad` και στην class `MonadT`, και ορίζουμε τις απαραίτητες συναρτήσεις `return`, `>>=` και `lift`.

```
instance Monad m => Monad (StateT s m) where
        return a                = StateT (\s -> return (a, s))
        StateT m >>= k          = StateT (\s0 -> do (a, s1) <- m s0
                                                    runStateT (k a) s1)

instance MonadT (StateT s) where
        lift c                  = StateT (\s -> do {x <- c; return (x, s)})
```

Κατόπιν ορίζουμε μια νέα class `StateMonad`, με μέλη τις συναρτήσεις `fetch` (επιστρέφει την τρέχουσα κατάσταση) και `update` (παίρνει ως όρισμα μια συνάρτηση την οποία εφαρμόζει στην τρέχουσα κατάσταση). Δηλώνουμε ότι το `StateT` ανήκει στην class αυτή και ορίζουμε τις αντίστοιχες συναρτήσεις:

```
class Monad m => StateMonad s m where
        fetch           :: m s
        update          :: (s -> s) -> m ()

instance Monad m => StateMonad s (StateT s m) where
        fetch           = StateT (\s -> return (s,s))
        update f        = StateT (\s -> return ((), f s))
```

Τέλος ορίζουμε τα απαραίτητα liftings:

```
instance ErrMonad m => ErrMonad (StateT s m) where
        err             = lift . err

instance ErrMonad m => ErrMonad (EnvT Env m) where
        err             = lift . err

instance StateMonad s m => StateMonad s (EnvT Env m) where
        fetch = lift fetch
        update = lift . update
```

4.2 Ορισμός συνόλων και τύπων

Έστω $\mathcal{E}$ η συνάρτηση αποτίμησης (evaluation) του διερμηνέα, η οποία δέχεται ως ορίσματα εκφράσεις (expressions) και επιστρέφει υπολογισμούς (computations). Αν ονομάσουμε **Value** το σύνολο στο οποίο θα ανήκουν τα αποτελέσματα–τιμές αυτών των υπολογισμών, ο τύπος της $\mathcal{E}$ θα είναι

$\mathcal{E} : Expr \rightarrow Compute\ \mathbf{Value}$

όπου *Compute* μια κατάλληλα ορισμένη μονάδα. Το σύνολο **Value** θα περιλαμβάνει ακεραίους αριθμούς (σύνολο **N**), λογικές (boolean) σταθερές (σύνολο **T**), συναρτήσεις (σύνολο **Fun**), records (σύνολο **Record**) και διευθύνσεις στη μνήμη/store (σύνολο **Loc**). Μπορούμε συνεπώς να γράψουμε

$\mathbf{Value} = \mathbf{N} \oplus \mathbf{T} \oplus \mathbf{Fun} \oplus \mathbf{Record} \oplus \mathbf{Loc}$

Λαμβάνοντας υπόψη ότι η Cantor περιλαμβάνει τόσο συναρτήσεις call-by-value όσο και call-by-name, μπορούμε να συμβολίσουμε το σύνολο **Fun** σαν μια συνάρτηση από υπολογισμούς σε υπολογισμούς:

$\mathbf{Fun} = Compute\ \mathbf{Value} \rightarrow Compute\ \mathbf{Value}$

Το σύνολο των records ακόμη, έχει ως στοιχεία λίστες, οι οποίες περιέχουν ζεύγη αναγνωριστικών και τιμών. Αν **Ide** είναι το σύνολο των αναγνωριστικών, θα έχουμε

$\mathbf{Record} = (\mathbf{Ide} \times \mathbf{Value})^*$

Το σύνολο **Loc** τέλος, μπορούμε να θεωρήσουμε ότι ταυτίζεται με το **N**.

Αν μεταφέρουμε τους παραπάνω ορισμούς στη Haskell, και αν ονομάσουμε τη συνάρτηση αποτίμησης `ev`, προκύπτουν τα παρακάτω[1]:

```
data Value      = Num Int
                | Mbool Bool
                | Fun (Compute Value -> Compute Value)
                | Record [(I, Value)]
                | Loc Location

type Location   = Int

ev              :: Expr -> Compute Value
```

[1] Για τον τύπο `Expr` ισχύει ο ορισμός που έχει δοθεί στην §3.2.2

Απομένει τώρα να ορίσουμε τη μονάδα *Compute*. Η μονάδα αυτή όπως προαναφέραμε πρέπει να έχει δυνατότητες χειρισμού λαθών, χειρισμού κατάστασης και χειρισμού περιβάλλοντος. Ορίζουμε λοιπόν:

```
type Compute a          = EnvT Env (StateT Store (ErrorT Id)) a

newtype Env             = MkE {fromE :: I -> Maybe (Compute Value)}

type Store              = ([Maybe Value], Location)
```

4.3 Δυναμική σημασιολογία και υλοποίηση

Είμαστε έτοιμοι να προχωρήσουμε στον ορισμό της δυναμικής σημασιολογίας, για τις εκφράσεις της Cantor, και στη μεταφορά της σημασιολογίας αυτής στη γλώσσα Haskell[2].

- Σταθερές τύπου int και bool: Αρκεί να κληθεί η "*unit*", ώστε να επιστραφεί ένας στοιχειώδης υπολογισμός με αποτέλεσμα τις αντίστοιχες σταθερές. Σημειώνουμε ότι αν και στις τρεις πρώτες εξισώσεις καλούμε τις συναρτήσεις $\mathbf{in_i}$ (injections) που μετατρέπουν τις σταθερές ακεραίου ή λογικού τύπου σε στοιχεία του συνόλου **Value**, στις επόμενες εξισώσεις που παραθέτουμε οι κλήσεις των συναρτήσεων αυτών, καθώς και των αντιστρόφων τους, παραλείπονται.

  $\mathcal{E}[\![n]\!] = unit(\mathbf{in_1}\ n)$
  $\mathcal{E}[\![\mathbf{true}]\!] = unit(\mathbf{in_2}\ \mathbf{true})$
  $\mathcal{E}[\![\mathbf{false}]\!] = unit(\mathbf{in_2}\ \mathbf{false})$

  Ο αντίστοιχος κώδικας του διερμηνέα, που προκύπτει αν μεταφράσουμε τις παραπάνω εξισώσεις σε κώδικα Haskell, είναι ο εξής:

```
ev (Con n)        = return (Num n)
ev (MyTrue)       = return (Mbool True)
ev (MyFalse)      = return (Mbool False)
```

- Αναγνωριστικά: Καλείται η "*rdEnv*" που επιστρέφει το τρέχον περιβάλλον ρ, και κατόπιν καλείται το ρ με όρισμα το αναγνωριστικό:

  $\mathcal{E}[\![id]\!] = rdEnv * (\lambda\rho.\rho\ id)$

  Κώδικας Haskell:

```
ev (Ide i)        = do e <- rdEnv
                       fromJust (fromE e i)
```

- Εκφράσεις **neg** και **not**: Στην περίπτωση αυτή αποτιμάται πρώτα το τελούμενο του τελεστή **neg** ή **not** και το αποτέλεσμα εφαρμόζεται η ανάλογη συνάρτηση:

  $\mathcal{E}[\![\mathbf{not}\ e]\!] = \mathcal{E}[\![e]\!] * (\lambda u.unit(not\ u))$
  $\mathcal{E}[\![\mathbf{neg}\ e]\!] = \mathcal{E}[\![e]\!] * (\lambda u.unit(-u))$

  Ο κώδικας Haskell είναι ο παρακάτω:

[2] Στις παρακάτω εξισώσεις συμβολίζουμε με "$*$" τη συνάρτηση "bind" της μονάδας *Compute*.

```
ev (Unexp Not e)  = do u <- ev e
                       not' u
ev (Unexp Neg e)  = do u <- ev e
                       neg u

not'              :: Value -> Compute Value
not' (Mbool b)    = return (Mbool (not b))

neg               :: Value -> Compute Value
neg (Num n)       = return (Num (-n))
```

- Δυαδικές εκφράσεις με έναν από τους τελεστές $+$, $-$, $*$, $/$, $\%$, $==$, $!=$, $<=$, $>=$, $<$, $>$: Με εντελώς παρόμοιο με την προηγούμενη περίπτωση τρόπο, τα δύο τελούμενα αποτιμούνται και με τη χρήση της συνάρτησης *bind* περνούν ως ορίσματα στην αντίστοιχη συνάρτηση.

$$\mathcal{E}[\![e_1 + e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u + v)))$$
$$\mathcal{E}[\![e_1 - e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u - v)))$$
$$\mathcal{E}[\![e_1 * e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u * v)))$$
$$\mathcal{E}[\![e_1 / e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u / v)))$$
$$\mathcal{E}[\![e_1 \% e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u \% v)))$$
$$\mathcal{E}[\![e_1 == e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u == v)))$$
$$\mathcal{E}[\![e_1 != e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u != v)))$$
$$\mathcal{E}[\![e_1 <= e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u <= v)))$$
$$\mathcal{E}[\![e_1 >= e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u >= v)))$$
$$\mathcal{E}[\![e_1 < e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u < v)))$$
$$\mathcal{E}[\![e_1 > e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u > v)))$$

Ακολουθεί ο κώδικας Haskell που αντιστοιχεί σε μερικές από τις παραπάνω εξισώσεις. Στις υπόλοιπες είναι παρόμοιος.

```
ev (Bexp Plus e1 e2)    = do {n1 <- ev e1; n2 <- ev e2; plus n1 n2}

ev (Bexp Div e1 e2)     = do {n1 <- ev e1; n2 <- ev e2; div' n1 n2}

ev (Bexp Eq' e1 e2)     = do {n1 <- ev e1; n2 <- ev e2; eq n1 n2}

ev (Bexp Gt e1 e2)      = do {n1 <- ev e1; n2 <- ev e2; gt n1 n2}

plus                    :: Value -> Value -> Compute Value
plus (Num n1) (Num n2)  = return (Num (n1+n2))

div'                    :: Value -> Value -> Compute Value
div' (Num n1) (Num n2)  = if n2 == 0
                             then err ("divide by zero")
                             else return (Num (div n1 n2))

eq                      :: Value -> Value -> Compute Value
eq (Num n1) (Num n2)    = return (Mbool (n1 == n2))
eq (Mbool b1) (Mbool b2)= return (Mbool (b1 == b2))
```

```
gt                        :: Value -> Value -> Compute Value
gt (Num n1) (Num n2)      = return (Mbool (n1 > n2))
```

- Δυαδικές εκφράσεις με έναν από τους τελεστές **and** και **or**: Η δυναμική σημασιολογία στην περίπτωση των δύο αυτών τελεστών διαφοροποιείται από αυτήν των υπόλοιπων δυαδικών τελεστών, καθώς ο διερμηνέας, όταν μπορεί να υπολογίσει το αποτέλεσμα της λογικής σύζευξης ή διάζευξης από την τιμή του πρώτου τελούμενου, *δεν* αποτιμά το δεύτερο τελούμενο. Οι σχετικές εξισώσεις έχουν ως εξής:

  $\mathcal{E}[\![e_1 \textbf{ and } e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.u \rightarrow \mathcal{E}[\![e_2]\!],\ \textbf{false})$
  $\mathcal{E}[\![e_1 \textbf{ or } e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.u \rightarrow \textbf{true}\ , \mathcal{E}[\![e_2]\!])$

  Τα παραπάνω μεταφέρονται στην Haskell ως εξής:

```
ev (Bexp And e1 e2)       = do u <- ev e1
                               case u of
                                 Mbool True -> ev e2
                                 Mbool False -> return (Mbool False)

ev (Bexp Or e1 e2)        = do u <- ev e1
                               case u of
                                 Mbool True -> return (Mbool True)
                                 Mbool False -> ev e2
```

- Δυαδική έκφραση με τον τελεστή ";" : Αποτιμάται το πρώτο τελούμενο, η τιμή του αγνοείται και κατόπιν αποτιμάται το δεύτερο τελούμενο. Η τιμή όλης της έκφρασης είναι η τιμή του δεύτερου τελούμενου.

  $\mathcal{E}[\![e_1\ ; e_2]\!] = \mathcal{E}[\![e_1] * (\lambda\_.\mathcal{E}[\![e_2]\!])$

  Κώδικας Haskell:

```
ev (Bexp Semicolon e1 e2)   = do ev e1
                                 ev e2
```

- Εντολή συνθήκης: Αποτιμάται η έκφραση που ακολουθεί το **if** και αν είναι αληθής επιστρέφεται η τιμή της έκφρασης που ακολουθεί το **then**, αλλιώς η τιμή της έκφρασης μετά το **else**.

  $\mathcal{E}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] = \mathcal{E}[\![e_1]\!] * (\lambda b.b \rightarrow \mathcal{E}[\![e_2]\!], \mathcal{E}[\![e_3]\!])$

  Κώδικας Haskell:

```
ev (If e1 e2 e3)          = do b <- ev e1
                               case b of
                                 Mbool True -> ev e2
                                 Mbool False -> ev e3
```

- Λάμβδα εκφράσεις: Εδώ διακρίνουμε τις δυο περιπτώσεις της κλήσης κατά τιμή και της κλήσης κατά όνομα. Και στις δύο περιπτώσεις το όρισμα της λάμβδα έκφρασης είναι ένας υπολογισμός. Στην πρώτη περίπτωση, της κλήσης κατά τιμή, ο υπολογισμός αυτός (arg) αποτιμάται και το αποτέλεσμα (v) περνά ως όρισμα στη συνάρτηση· το περιβάλλον ενημερώνεται κατάλληλα συνδέοντας το όνομα x της παραμέτρου της συνάρτησης με το $unitv$:

$$\mathcal{E}[\![\textbf{lambda}\ x : type.e]\!] = rdEnv * (\lambda\rho.unit(\lambda arg.arg * (\lambda v.inEnv\ \rho[unitv/x])\ \mathcal{E}[\![e]\!]))$$

Στην περίπτωση της κλήσης κατα όνομα, ο ίδιος ο υπολογισμός–όρισμα (arg) συνδέεται με το όνομα της παραμέτρου, έτσι ώστε το arg να αποτιμάται *αν* και *όποτε* συναντάται στο σώμα της λάμβδα έκφρασης:

$$\mathcal{E}[\![\textbf{lambda}\ !\ x : type.e]\!] = rdEnv * (\lambda\rho.unit(\lambda arg.inEnv\ \rho[arg/x]\ \mathcal{E}[\![e]\!]))$$

Τα παραπάνω μεταφέρονται εύκολα σε κώδικα Haskell ως εξής (η ενημέρωση–επέκταση του περιβάλλοντος γίνεται με τη συνάρτηση `extendEnv`):

```
ev (Lam m x t expr)
        = do e <- rdEnv
             case m of
               BV -> return (Fun (\arg ->
                      do v <- arg
                         inEnv (extendEnv (x, return v) e) (ev expr)))
               BN -> return (Fun (\arg ->
                      inEnv (extendEnv (x, arg) e) (ev expr)))

extendEnv          :: (I, Compute Value) -> Env -> Env
extendEnv (i, v) e = MkE (\i' -> if i==i' then Just v else fromE e i')
```

- Κλήση συνάρτησης: Αποτιμάται η πρώτη έκφραση (συνάρτηση) και στο αποτέλεσμα της αποτίμησης περνά ως όρισμα ένας υπολογισμός που αντιστοιχεί στην αποτίμηση της δεύτερης έκφρασης (όρισμα). Ένα σημείο που χρήζει προσοχής είναι ότι η αποτίμηση του ορίσματος γίνεται μέσα στο περιβάλλον που υπάρχει κατά τη στιγμή που καλείται η συνάρτηση:

  $$\mathcal{E}[\![e_1\ e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda f.rdEnv * (\lambda\rho.f\ (inEnv\ \rho\ \mathcal{E}[\![e_2]\!])))$$

  Κώδικας Haskell:

```
ev (Call t u)       = do f <- ev t
                         case f of
                           Fun k -> do e <- rdEnv
                                       k (inEnv e (ev u))
```

- Δήλωση τοπικής μεταβλητής: Καλείται η βοηθητική συνάρτηση $alloc : Compute\ \textbf{Loc}$, η οποία επιστρέφει μια νέα διεύθυνση στο χώρο αποθήκευσης, και ενημερώνεται το περιβάλλον συνδέοντας το όνομα της μεταβλητής με τη διεύθυνση αυτή:

  $$\mathcal{E}[\![\textbf{var}\ i : data\ \textbf{in}\ e]\!] = rdEnv * (\lambda\rho.alloc * (\lambda l.inEnv\ \rho[l/i]\ \mathcal{E}[\![e]\!]))$$

Μεταφέροντας την παραπάνω σχέση σε κώδικα Haskell, ορίζουμε τη συνάρτηση `alloc` η οποία επιστρέφει την πρώτη ελεύθερη διεύθυνση στο χώρο αποθήκευσης. Υπενθυμίζουμε ότι έχουμε ορίσει τον τύπο `Store` ως ένα ζεύγος λίστας και ακεραίου αριθμού: Η λίστα χρησιμεύει ως χώρος αποθήκευσης, ενώ ο αριθμός ισούται με την πρώτη ελεύθερη διεύθυνση (ουσιαστικά ο αριθμός αυτός μας δίνει το τρέχον μήκος της λίστας). Η `alloc` επιπλέον τοποθετεί την τιμή `Nothing` στη νέα θέση.

```
ev (Var i d exp) = do e <- rdEnv
                      l <- alloc
                      inEnv (extendEnv (i, return (Loc l)) e) (ev exp)

alloc           :: Compute Location
alloc           = do update (cross ((++ [Nothing]), (+1)))
                      store <- fetch
                      return ((snd store) - 1)

cross               :: (a -> b, c -> d) -> (a, c) -> (b, d)
cross (f, g) (x, y)  = (f x, g y)
```

- Πρόσβαση στην τιμή τοπικής μεταβλητής: Αποτιμάται το όρισμα του τελεστή **deref** και το αποτέλεσμα (που είναι μια διεύθυνση στο χώρο αποθήκευσης) δίνεται ως όρισμα στη βοηθητική συνάρτηση $getValue : \mathbf{Loc} \to (Compute\ \mathbf{Value} \oplus \mathbf{U})$, η οποία επιστρέφει το περιεχόμενο της διεύθυνσης αυτής:

  $\mathcal{E}[\![\mathbf{deref}\, e]\!] = \mathcal{E}[\![e]\!] * (\lambda l.getValue\ l)$

  Ο κώδικας Haskell που υλοποιεί τα παραπάνω είναι ο εξής:

```
ev (Deref exp)  = do l <- ev exp
                     getValue (unLoc l)

getValue        :: Location -> Compute Value
getValue i      = do store <- fetch
                     case ((fst store) !! i) of
                       Just v -> return v
                       Nothing -> err "uninitialized variable"

unLoc           :: Value -> Location
unLoc (Loc a)   = a
```

- Ανάθεση τιμής σε τοπική μεταβλητή: Το αριστερό τελούμενο του τελεστή ":=" αποτιμάται και επιστρέφει μια διεύθυνση η οποία μαζί με την τιμή του δεξιού τελούμενου δίνεται ως όρισμα στη συνάρτηση $putValue :: \mathbf{Loc} \to \mathbf{Value} \to \mathbf{Value}$, που αναθέτει την τιμή/δεύτερο όρισμά της στη διεύθυνση/πρώτο όρισμά της, και επιστρέφει την τιμή που ανέθεσε.

  $\mathcal{E}[\![e_1 := e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda l.\mathcal{E}[\![e_2]\!] * (\lambda v.putValue\ l\ v))$

  Σε κώδικα Haskell τα προηγούμενα υλοποιούνται ως εξής:

```
ev (Assign exp1 exp2)   = do l <- ev exp1
                             v <- ev exp2
                             putValue (unLoc l) v

putValue        :: Location -> Value -> Compute Value
putValue i v    = do update (cross (modifyVar i v, id))
                     return v

modifyVar       :: Location -> Value -> [Maybe Value] -> [Maybe Value]
modifyVar l v s = let (a, b) = splitAt l s in
                      a ++ [Just v] ++ tail b
```

- Αναδρομή **rec** ($\langle expr\rangle$): Αποτιμάται το όρισμα του τελεστή **rec** και δίνεται ως όρισμα στη συνάρτηση $fix : (\alpha \to \alpha) \to \alpha$ (least fixed point operator), για την οποία ισχύει $fix\ f = f\ (fix\ f)$.

  $\mathcal{E}[\![\mathbf{rec}\ (exp)]\!] = \mathcal{E}[\![exp]\!] * (\lambda f.fix\ f)$

  Κώδικας Haskell:

```
ev (MyRec exp)          = do e <- ev exp
                             case e of
                               Fun k -> fix k

fix                     :: (a -> a) -> a
fix f                   = f (fix f)
```

- Κενό record: Όπως και στην περίπτωση των ακεραίων και λογικών σαθερών, αρκεί να επιστραφεί ένας στοιχειώδης υπολογισμός με αποτέλεσμα το κενό record, δηλαδή:

  $\mathcal{E}[\![\mathbf{empty}]\!] = unit([])$

  Κώδικας Haskell:

```
ev Empty                = return (Record [])
```

- Επέκταση record $\langle expr_1\rangle[\langle id\rangle := \langle expr_2\rangle]$: Αφού αποτιμηθούν τα $\langle expr_1\rangle$ και $\langle expr_2\rangle$, προστίθεται[3] το ζευγάρι (id, τιμή $\langle expr_2\rangle$) στο record. Υπενθυμίζουμε ότι ένα record είναι μια λίστα ζευγών αναγνωριστικών και τιμών. Επίσης σημειώνουμε εδώ, ότι δεν χρειάζεται να γίνει έλεγχος για την πιθανή ύπαρξη πεδίου $\langle id\rangle$ στο record, καθώς αυτός ο έλεγχος έχει ήδη γίνει στο στατικό έλεγχο τύπων.

  $\mathcal{E}[\![e_1[id := e_2]]\!] = \mathcal{E}[\![e_1]\!] * (\lambda r.\mathcal{E}[\![e_2]\!] * (\lambda v.unit((id,\ v) \mapsto r)))$

```
ev (RecExtend exp1 i exp2)
       = do r <- ev exp1
            v <- ev exp2
```

[3] Με $a \mapsto l$ συμβολίζουμε την εισαγωγή στοιχείου a σε λίστα l.

```
            return (Record ((i, v):unRec r))

unRec                   :: Value -> [(I, Value)]
unRec (Record r)        = r
```

- Πρόσβαση σε πεδίο record $\langle expr \rangle :: \langle id \rangle$: Αποτιμάται η έκφραση πριν τον τελεστή "::" και η τιμή της μαζί με το αναγνωριστικό id δίνεται ως όρισμα στη συνάρτηση $lookup$: $\mathbf{Ide} \rightarrow \mathbf{Record} \rightarrow (\mathbf{Value} \oplus \mathbf{U})$:

$$\mathcal{E}[\![e :: id]\!] = \mathcal{E}[\![e]\!] * (\lambda r.[unit,\ err(\text{"}field\ does\ not\ exist\text{"})]\ (lookup\ id\ r))$$

```
ev (RecAccess exp i)
               = do r <- ev exp
                    case lookup i (unRec r) of
                      Just v -> return v
                      Nothing -> err ("record: field does not exist")
```

Κεφάλαιο 5

Περιγραφή και διερμηνέας G-machine

Στο κεφάλαιο αυτό περιγράφουμε την G-machine, στην οποία βασίζεται ο μεταγλωττιστής της Cantor που κατασκευάσαμε. Παράλληλα με την τυπική περιγραφή της G-machine, δίνουμε και την υλοποίηση στη γλώσσα Haskell του διερμηνέα της G-machine.

5.1 Γενικά

Η G-machine που χρησιμοποιεί ο μεταγλωττιστής της Cantor, βασίζεται σε αυτή που παρουσιάζουν οι Peyton Jones & Lester στο [3]. Λόγω όμως των διαφορών μεταξύ της Cantor και της υποθετικής γλώσσας "Core" που χρησιμοποιούν οι τελευταίοι στο βιβλίο τους, ήταν αναγκαίες κάποιες τροποποιήσεις στο μοντέλο της G-machine. Εκτός από τις αναγκαίες προσθήκες στη G-machine λόγω της ύπαρξης records, η άλλη κύρια αιτία τροποποιήσεων ήταν η ύπαρξη στην Cantor τοπικών μεταβλητών με δυνατότητα ανάθεσης, οι οποίες μάλιστα μπορεί να έχουν και τύπο συνάρτησης, όπως επίσης και η υποστήριξη κλήσης με όνομα αλλά και κλήσης με τιμή. Τέλος, μια ακόμη σημαντική διαφορά προκύπτει από το τρόπο χειρισμού των εκφράσεων λάμδα, όπου αντί για την τεχνική του "lambda lifting" η οποία οδηγεί στην απαλοιφή των εκφράσεων αυτών, ο μεταγλωττιστής της Cantor ακολουθεί μια τακτική παρόμοια με αυτή που παρουσιάζεται στο [10], που περιγράφει την G-machine στη οποία βασίζεται ο GHC.

5.2 Κλεισίματα (closures)

Πριν προχωρήσουμε στην τυπική περιγραφή της G-machine, είναι αναγκαίο να περιγράψουμε μερικά γενικά στοιχεία της λειτουργίας της G-machine. Η G-machine αποτελείται από τις στοίβες μεταβλητών και αποτίμησης, από το χώρο αποθήκευσης του συνολικού κώδικα, από το χώρο αποθήκευσης του τρέχοντα κώδικα, καθώς και από το σωρό (heap).

Κεντρικό ρόλο στη λειτουργία της G-machine έχει ο σωρός, ο οποίος περιέχει δύο κατηγοριών αντικείμενα: Τιμές εκφράσεων (values), π.χ. ακεραίους αριθμούς, συναρτήσεις, και μη αποτιμημένες εκφράσεις, δηλαδή ορίσματα συναρτήσεων ή κλήσεις συναρτήσεων. Οι συναρτήσεις (δηλ. οι εκφράσεις λάμδα) καθώς και τα ορίσματα των συναρτήσεων, αναπαριστώνται στο σωρό ως *κλεισίματα (closures)*. Πιο συγκεκριμένα, αναπαριστούμε ένα κλείσιμο ως ένα δείκτη σε ένα κομμάτι κώδικα, ο οποίος συνοδεύεται από μια λίστα δεικτών (διευθύνσεων στο σωρό) προς τις ελεύθερες μεταβλητές του συγκεκριμένου τμήματος κώδικα. Όταν ξεκινά η αποτίμηση ενός κλεισίματος, το κομμάτι κώδικα στο οποίο δείχνει ο δείκτης προστίθεται στην αρχή του τρέχοντος κώδικα, και οι διευθύνσεις των ελεύθερων μεταβλητών (μαζί με το όρισμα αν ο κλείσιμο αναπαριστά συνάρτηση) φορτώνονται στην στοίβα μεταβλητών. Η φόρτωση των διευθύνσεων των ελεύθερων μεταβλητών στη στοίβα μεταβλητών γίνεται ώστε να μειωθούν οι κατά τεκμήριο ακριβές χρονικά αναγνώσεις και αναζητήσεις στο σωρό. Καθώς εκτελούνται διαδοχικά οι εντολές που αποτελούν τον τρέχοντα κώδικα, «κατασκευάζεται» σταδιακά στη στοίβα αποτίμησης η έκφραση η οποία αντιστοιχεί στο κλείσιμο. Όταν ολοκληρωθεί η αποτίμηση της έκφρασης/κλεισίματος, στην κορυφή της στοίβας αποτίμησης υπάρχει πλέον το αποτέλεσμα της αποτίμησης αυτής, ενώ οι διευθύνσεις των ελεύθερων μεταβλητών αφαιρούνται από τη στοίβα μεταβλητών.

Σημειώνουμε εδώ ότι οι διευθύνσεις των ελευθερων μεταβλητών είναι αναγκαίο να αποθηκευθούν μαζί με το κλείσιμο στο σωρό, διότι όταν χρειαστεί να αποτιμηθεί το κλείσιμο αυτό, είναι πιθανό να μην υπάρχουν πλέον στη στοίβα μεταβλητών. Π.χ. στον παρακάτω κώδικα,

```
var f:int exp -> int exp in
    (lambda x:int exp. f:=(lambda y:int exp. x+y)) 42;
    (deref f) 5
```

όταν κληθεί η συνάρτηση `f`, η οποία στο G-machine θα αναπαρασταθεί ως ένα κλείσιμο (δείκτης σε κώδικα & διεύθυνση της ελεύθερης μεταβλητής `x`), η μεταβλητή x δεν θα υπάρχει πια στην στοίβα μεταβλητών, αφού θα έχει ολοκληρωθεί η αποτίμηση της συνάρτησης της οποίας ήταν παράμετρος.

5.3 Συστήματα μετάβασης κατάστασης (state transition systems)

Για την τυπική περιγραφή της G-machine θα χρησιμοποιήσουμε ένα *σύστημα μετάβασης κατάστασης*, όπως και οι Peyton Jones & Lester στο [3]. Το σύστημα μετάβασης κατάστασης είναι μια μέθοδος περιγραφής του τρόπου λειτουργίας μιας σειριακής μηχανής. Αποτελείται από μια *αρχική κατάσταση* s_0 και ένα σύνολο *κανόνων μετάβασης* (state transition rules) $s_a \Rightarrow s_b$. Κάθε χρονική στιγμή, το σύστημα (δηλ. η μηχανή) βρίσκεται σε μια συγκεκριμένη *κατάσταση* s_i. Αν υπάρχει κανόνας μετάβασης $s_a \Rightarrow s_b$, με $s_i = s_a$, ο κανόνας αυτός ενεργοποιείται και η μηχανή περνά στην επόμενη κατάσταση s_b. Αν όχι, η μηχανή τερματίζει.

Τα συστήματα μετάβασης κατάστασης μάς παρέχουν την αναγκαία αφαίρεση (abstraction) που χρειάζεται ώστε να περιγράψουμε τη λειτουργία της G-machine, χωρίς όμως να μάς αποκρύπτουν σημαντικές λεπτομέρειες. Επίσης, όπως θα δούμε, είναι αρκετά απλή η μεταφορά των κανόνων μετάβασης σε μια συναρτησιακή γλώσσα όπως η Haskell.

5.4 Κατάσταση του G-machine

Η *κατάσταση* του G-machine μπορεί να οριστεί ως μια πεντάδα

(current code, codestore, variable stack, evaluation stack, heap)

ή εν συντομία (c, cs, vs, es, h).

- Ο τρέχων κώδικας (current code) είναι μια σειρά από εντολές που έχουν «φορτωθεί» σε πρωτύτερη φάση της εκτέλεσης του προγράμματος στην G-machine και εκτελούνται η μία μετά την άλλη. Ο συμβολισμός $i_1 : is$ δηλώνει μια σειρά από εντολές που ξεκινά με την i_1. Όπως θα δούμε παρακάτω, όταν μια εντολή εκτελείται, αφαιρείται από τον τρέχοντα κώδικα. Όταν στο πεδίο του τρέχοντα κώδικα δεν υπάρχουν άλλες εντολές, η G-machine τερματίζει.

- Ο χώρος αποθήκευσης κώδικα (codestore) είναι μια λίστα που περιέχει όλο το G-machine κώδικα που αντιστοιχεί στο πρόγραμμα και που έχει παραχθεί από το μεταγλωττιστή. Κάθε μέλος της λίστας είναι ένα τμήμα κώδικα (code segment), που όπως θα δούμε στο επόμενο κεφάλαιο μπορεί να αντιστοιχεί σε μια έκφραση λάμδα, σε μια έκφραση–όρισμα συνάρτησης ή σε ένα σκέλος μιας εντολής συνθήκης **if**–**then**–**else**.

- Η στοίβα μεταβλητών (variable stack) περιλαμβάνει τις διευθύνσεις στο σωρό (heap) που αντιστοιχούν στις ελεύθερες μεταβλητές του τρέχοντος κλεισίματος, στο όρισμα, αν το κλείσιμο αντιστοιχεί σε συνάρτηση, και στις τυχόν τοπικές μεταβλητές που έχουν δημιουργηθεί μέσα στο τρέχον κλείσιμο. (Μπορεί να ειπωθεί ότι η στοίβα μεταβλητών είναι το αντίστοιχο του δυναμικού περιβάλλοντος του διερμηνέα.) Ο συμβολισμός $\alpha : \alpha s$ υποδηλώνει ότι στην κορυφή της στοίβας υπάρχει η διεύθυνση α, και η υπόλοιπη στοίβα συμβολίζεται με το αs.

- Η στοίβα αποτίμησης (evaluation stack) περιέχει επίσης διευθύνσεις στο σωρό, και χρησιμοποιείται από τις εντολές αριθμητικών και λογικών πράξεων, διακλάδωσης, κλήσης συνάρτησης κ.λπ. της G-machine. Κάθε έκφραση προκειμένου να αποτιμηθεί, «κατασκευάζεται» στην κορυφή της στοίβας απότίμησης.

- Ο σωρός (heap) χρησιμοποιείται από την G-machine ως χώρος αποθήκευσης, και περιέχει τιμές (values) και κλεισίματα (closures). Ο συμβολισμός $h[\alpha : Value]$ σημαίνει ότι το περιεχόμενο της διεύθυνσης α του σωρού είναι "$Value$".

5.5 Αρχική κατάσταση

Η αρχική κατάσταση του G-machine είναι αυτή που προκύπτει αν φορτώσουμε στο πεδίο του τρέχοντα κώδικα το πρώτο κομμάτι κώδικα του codestore. Οι στοίβες μεταβλητών και αποτίμησης, καθώς και ο σωρός, είναι αρχικά κενές:

$$(\text{code}, \text{code:cs}, [], [], [])$$

5.6 Υλοποίηση διερμηνέα για την G-machine

Η υλοποίηση του διερμηνέα σε μια συναρτησιακή γλώσσα όπως είναι η Haskell είναι αρκετά απλή διαδικασία. Ξεκινάμε τον ορισμό των απαραίτητων τύπων από την κατάσταση (state) της G-machine:

```
type Evalstate     = (GMCodeSegment, GMCodeStore, VarStack,
                                              EvalStack, Heap Value)

type GMCodeSegment = [GMInstruction]

type GMCodeStore   = [GMCodeSegment]

type VarStack      = [HAddr]

type EvalStack     = [HAddr]
```

Ο τύπος `GMInstruction` αναφέρεται στις εντολές της G-machine και ο ορισμός του θα δοθεί παρακάτω. Ο τύπος `HAddr` αναπαριστά τις διευθύνσεις στο σωρό (`Heap`), οι λεπτομέρειες της υλοποίησης του οποίου παραλείπονται. Οι συναρτήσεις χειρισμού του σωρού που θα χρησιμοποιήσουμε είναι οι παρακάτω:

```
hInitial        :: Heap a
hAlloc          :: Heap a -> a -> (Heap a, HAddr)
hUpdate         :: Heap a -> HAddr -> a -> Heap a
hLookup         :: Heap a -> HAddr -> a
```

Η `hInitial` επιστρέφει έναν άδειο σωρό και χρησιμοποιείται για την αρχικοποίηση του σωρού. Η `hAlloc` χρησιμοποιείται για την εισαγωγή νέων στοιχείων στο σωρό· δέχεται ως όρισμα τον υπάρχοντα σωρό και το νέο στοιχείο, και επιστρέφει ένα ζευγάρι με μέλη το νέο σωρό και τη διεύθυνση στην οποία τοποθετήθηκε το νέο στοιχείο. Η `hUpdate` χρησιμοποιείται για να αλλάξει το περιεχόμενο μιας διεύθυνσης στο σωρό· δέχεται ως ορίσματα τον υπάρχοντα σωρό, τη διεύθυνση και το νεό περιεχόμενο και επιστρέφει το νέο σωρό. Η `hLookup` τέλος, επιστρέφει το περιεχόμενο μιας διαύθυνσης στο σωρό· δέχεται ως όρισμα το σωρό και τη διέυθυνση και επιστρέφει το αντίστοιχο περιεχόμενο.

Ο τύπος `Value` ορίζεται ως εξής:

```
data Value  = Num Int               --σταθερές ακεραίου τύπου
            | MBool Bool            --σταθερές λογικού τύπου
            | Record [(I, HAddr)]   --records
            | Loc HAddr             --τοπικές μεταβλητές
            | Fun Int [HAddr]       --συνάρτηση
            | Arg Int [HAddr]       --όρισμα συνάρτησης
            | App HAddr HAddr       --κλήση συνάρτησης
```

Από τα παραπάνω, το `Loc HAddr` είναι ουσιαστικά ένας δείκτης προς το περιεχόμενο μιας τοπικής μεταβλητής. Τα `Fun Int [HAddr]` και `Arg Int [HAddr]` αναπαριστούν κλεισίματα. Το πρώτο τους πεδίο είναι η θέση στη λίστα `GMCodeStore` του κώδικα που αντιστοιχεί στο κλείσιμο, ενώ το δεύτερο πεδίο είναι μια λίστα με τις διευθύνσεις των ελεύθερων μεταβλητών.

Η συνάρτηση `execute` που εκτελεί ένα πρόγραμμα σε κώδικα G-machine και τυπώνει το αποτέλεσμα, ορίζεται ως εξής:

```
execute         :: GMCodeStore -> String
execute cs      = showValue finalheap value
                  where (value, _, _, finalheap)
                                = eval (head cs, cs, [], [], hInitial)

showValue       :: Heap Value -> Value -> String
--η υλοποίηση της showValue παραλείπεται

eval             :: Evalstate -> Output
eval state       = if final state then getresult state
                   else eval (step state)

type Output      = (Value, VarStack, EvalStack, Heap Value)

final                        :: Evalstate -> Bool
final ([], cs, as, es, h)    = True
final _                      = False

getresult                    :: Evalstate -> Value
getresult (i, cs, as, es, h) = hLookup h (head es)
```

Απο τα παραπάνω βλέπουμε η μηχανή τερματίζει όταν είναι κενό το πεδίο του τρέχοντος κώδικα, και το αποτέλεσμα της αποτίμησης είναι το περιεχόμενο της διεύθυνσης που βρίσκεται στην κορυφή της στοίβας αποτίμησης στο τέλος της εκτέλεσης. Η συνάρτηση `step` έχει τύπο `Evalstate -> Evalstate` και υλοποιεί τους κανόνες μετάβασης που θα ορίσουμε παρακάτω.

5.7 Εντολές – Κανόνες μετάβασης

Στον παρακάτω πίνακα δίνεται το σύνολο εντολών της G-machine, ενώ ακολουθεί ο αντίστοιχος κώδικας Haskell, όπου δηλώνεται ο τύπος `GMInstruction`.

Alloc	Assig	Cond Int Int	Dereff
Eval	Iadd	Idiv	Ieq
Ige	Igt	Ile	Ilo
Imod	Imul	Ine	Ineg
Inot	Isub	Mkap	MkFix
PopVar Int	Pop Int	PushArg Int Int	PushFun Int Int
Push Int	PushVal Value	RecAcc String	RecExt String
Update			

```
type I                  = String

data GMInstruction      = PushVal Value
                        | Inot | Ineg
                        | Iadd | Isub | Imul | Idiv | Imod
                        | Ieq | Ine | Igt | Ilo | Ige | Ile
                        | Cond Int Int
                        | PushArg Int Int | PushFun Int Int | Push Int
                        | Mkap | MkFix
                        | Eval | Update
                        | Pop Int | PopVar Int
                        | Alloc | Assig | Dereff
                        | RecAcc I | RecExt I
```

Ακολουθούν οι κανόνες μετάβασης και ο αντίστοιχος κώδικας Haskell για κάθε εντολή:

- `Alloc` : Η εντολή αυτή χρησιμοποιείται για να δεσμεύσει χώρο στο σωρό για μια νέα τοπική μεταβλητή. Χρησιμοποιεί για αρχικοποίηση την ειδική διεύθυνση του σωρού `hNull`, στην οποία δεν αντιστοιχεί κανένα αντικείμενο του σωρού. Τοποθετεί τη διεύθυνση που δέσμευσε στην κορυφή της στοίβας μεταβλητών:

$$\begin{array}{|llllll|}\hline & \texttt{Alloc}:is & cs & vs & es & h \\ \Rightarrow & is & cs & v:vs & es & h[v:\mathsf{Loc}\ \texttt{hNull}] \\ \hline\end{array}$$

  Ο αντίστοιχως κώδικας Haskell είναι ο εξής:

```
step (Alloc:is, cs, vs, es, h)  = (is, cs, v:vs, es, h')
                                  where (h', v) = hAlloc h (Loc hNull)
```

- `Assig` : Η εντολή αυτή χρησιμοποιείται για την ανάθεση τιμής σε τοπική μεταβλητή. Περιμένει να έχουν τοποθετηθεί στην κορυφή της στοίβας αποτίμησης η διεύθυνση της τοπικής μεταβλητής καθώς και η διεύθυνση στο σωρό του νέου περιεχομένου, και ενημερώνει κατάλληλα το σωρό. Αφήνει στην κορυφή της στοίβας αποτίμησης μόνο τη διεύθυνση του νέου περιεχομένου (υπενθυμίζουμε ότι με βάση τη σημασιολογία της Cantor η τιμή μιας παράστασης $\langle\text{expr}\rangle_1{:=}\langle\text{expr}\rangle_2$ είναι ίση με την τιμή του δεξιού τελούμενου).

$$\begin{array}{|llllll|}\hline & \texttt{Assig}:is & cs & vs & rval:lval:es & h[lval:\mathsf{Loc}\ oldcontent] \\ \Rightarrow & is & cs & vs & rval:es & h[lval:\mathsf{Loc}\ rval] \\ \hline\end{array}$$

  Κώδικας Haskell:

```
step (Assig:is, cs, vs, rval:lval:es, h)
                              = (is, cs, vs, rval:es, h')
                                where h' = hUpdate h lval (Loc rval)
```

- `Cond n m` : Η εντολή αυτή εκτελεί άλμα υπό συνθήκη. Αναμένει να υπάρχει στη κορυφή της στοίβας αποτίμησης διευθυνση στο σωρό η οποία περιέχει λογική (boolean) τιμή. Αν η τιμή είναι αληθής, τότε φορτώνεται στην αρχή του τρέχοντος κώδικα το κομμάτι κώδικα που βρίσκεται στη θέση `n` του codestore, αλλιώς αυτό που βρίσκεται στη θέση `m`. Στην ενολή αυτή αντιστοιχούν δύο κανόνες μετάβασης:

	$\mathtt{Cond\,n\,m} : is$	$c_0 : c_1 : ... : c_n : ... : c_m : cs$	vs	$a : es$	$h[\mathtt{MBool\ True}]$
$\Rightarrow$	$c_n +\!\!+ is$	$c_0 : c_1 : ... : c_n : ... : c_m : cs$	vs	es	h

Ενώ αν το περιεχόμενο της διεύθυνσης που είναι στην κορυφή της στοίβας αποτίμησης είναι `False`, ενεργοποιείται ο κανόνας:

	$\mathtt{Cond\,n\,m} : is$	$c_0 : c_1 : ... : c_n : ... : c_m : cs$	vs	$a : es$	$h[\mathtt{MBool\ False}]$
$\Rightarrow$	$c_m +\!\!+ is$	$c_0 : c_1 : ... : c_n : ... : c_m : cs$	vs	es	h

Κώδικας Haskell:

```
step ((Cond i1 i2):is, cs, vs, a:es, h)
                              = (cs!!i ++ is, cs, vs, es, h)
                                where i = case hLookup h a of
                                          MBool True -> i1
                                          MBool False -> i2
```

- `Dereff` : Η εντολή αυτή χρησιμοποιείται για να διαβάσει το περιεχόμενο μιας τοπικής μεταβλητής :

	$\mathtt{Dereff} : is$	cs	vs	$a : es$	$h[a : \mathtt{Loc}\ l]$
$\Rightarrow$	is	cs	vs	$l : es$	h

Κώδικας Haskell:

```
step (Dereff:is, cs, vs, a:es, h)
                              = case hLookup h a of
                                  Loc l -> (is, cs, vs, l:es, h)
```

- `Eval` : Η εντολή αυτή είναι βασική στη λειτουργία της G-machine. Ελέγχει αν η διεύθυνση που βρίσκεται στην κορυφή της στοίβας αποτίμησης περιέχει αποτιμημένη ή μη έκφραση, και στην δεύτερη περίπτωση ξεκινά την αποτίμηση φορτώνοντας τον αντίστοιχο κώδικα στην αρχή του τρέχοντα κώδικα. Οι παρακάτω 5 κανόνες αναφέρονται στην περίπτωση που δεν χρειάζεται αποτίμηση, οπότε η μηχανή απλώς περνά στην επόμενη εντολή χωρίς να κάνει τίποτα:

	$\mathtt{Eval} : is$	cs	vs	$a : es$	$h[a : \mathtt{Num}\ n]$
$\Rightarrow$	is	cs	vs	$a : es$	h

	$\mathtt{Eval} : is$	cs	vs	$a : es$	$h[a : \mathtt{MBool}\ b]$
$\Rightarrow$	is	cs	vs	$a : es$	h

$$\begin{array}{|ll|}\hline & \texttt{Eval} : is \quad cs \quad vs \quad a : es \quad h[a : \texttt{Record}\, r] \\ \Rightarrow & \qquad\; is \quad cs \quad vs \quad a : es \quad h \\ \hline\end{array}$$

$$\begin{array}{|ll|}\hline & \texttt{Eval} : is \quad cs \quad vs \quad a : es \quad h[a : \texttt{Loc}\, l] \\ \Rightarrow & \qquad\; is \quad cs \quad vs \quad a : es \quad h \\ \hline\end{array}$$

$$\begin{array}{|ll|}\hline & \texttt{Eval} : is \quad cs \quad vs \quad a : es \quad h[a : \texttt{Fun}\, n\, l] \\ \Rightarrow & \qquad\; is \quad cs \quad vs \quad a : es \quad h \\ \hline\end{array}$$

Σημειώνουμε εδώ ότι στην περίπτωση του `Fun n l`, το περιεχόμενο της διεύθυνσης στο σωρό είναι συνάρτηση και όχι κλήση συνάρτησης, επομένως δεν χρειάζεται να γίνει αποτίμηση. Αντίθετα αποτίμηση γίνεται στις δύο παρακάτω περιπτώσεις, που αντιστοιχούν σε όρισμα συνάρτησης και σε κλήση συνάρτησης. Στην πρώτη περίπτωση, το αντίστοιχο κομμάτι κώδικα από το codestore προστίθεται στην αρχή του κώδικα, ενώ και οι διευθύνσεις των ελεύθερων μεταβλητών προστίθενται στην κορυφή της στοίβας μεταβλητών:

$$\begin{array}{|lllllll|}\hline & \texttt{Eval} : is & c_0 : ... : c_n : ... : cs & vs & a : es & h[a : \texttt{Arg}\, n\, l] & \\ \Rightarrow & c_n +\!\!+ is & c_0 : ... : c_n : ... : cs & l +\!\!+ vs & es & h & \\ \hline\end{array}$$

Στη δεύτερη περίπτωση, η εξεταζόμενη διεύθυνση στο σωρό περιέχει κόμβο κλήσης συνάρτησης. Μέσω του κόμβου αυτού η G-machine βρίσκει τη διεύθυνση `f` στο σωρό που περιέχει την καλούμενη συνάρτηση και τη διεύθυνση `arg` που περιέχει το αντίστοιχο όρισμα. Ο κώδικας που αντιστοιχεί στη συνάρτηση τοποθετείται στην αρχή του τρέχοντος κώδικα, ενώ η διεύθυνση του ορίσματος ακολουθούμενη από τη λίστα των ελεύθερων μεταβλητών της συνάρτησης τοποθετούνται στην κορυφή της στοίβας μεταβλητών:

$$\begin{array}{|llllll|}\hline & \texttt{Eval} : is & c_0 : .. : c_n : .. : cs & vs & a : es & h[a : \texttt{App}\, f\, arg,\ f : \texttt{Fun}\, n\, l] \\ \Rightarrow & c_n +\!\!+ is & c_0 : .. : c_n : .. : cs & arg : (l +\!\!+ vs) & es & h \\ \hline\end{array}$$

Ο κώδικας Haskell για όλες τις παραπάνω περιπτώσεις έχει ως εξής:

```
step (Eval:is, cs, vs, a:es, h)
              = case hLookup h a of
                  Num n -> (is, cs, vs, a:es, h)
                  MBool b -> (is, cs, vs, a:es, h)
                  Record r -> (is, cs, vs, a:es, h)
                  Loc l -> (is, cs, vs, a:es, h)
                  Fun n l -> (is, cs, vs, a:es, h)
                  Arg n l -> (cs !! n ++ is, cs, l++vs, es, h)
                  App f arg -> case hLookup h f of
                    Fun n l -> (cs !! n ++ is, cs, arg:(l++vs), es, h)
```

- Εντολές αριθμητικών πράξεων `Iadd`, `Isub`, `Imul`, `Idiv` και `Imod` : Οι εντολές αυτές αντιστοιχούν στην πρόσθεση, αφαίρεση, πολλαπλασιασμό, πηλίο διαίρεσης και υπόλοιπο διαίρεσης. Αναμένουν οι δύο διευθύνσεις που βρίσκονται στην κορυφή της στοίβας αποτίμησης να περιέχουν ακεραίους αριθμούς. Αφού εκτελέσουν την πράξη, τοποθετούν το αποτέλεσμα σε μια νέα διεύθυνση στο σωρό, η οποία μπαίνει στην κορυφή της στοίβας αποτίμησης. Οι δλυο διευθύνσεις που περιείχαν τα τελούμενα της πράξης αφαιρούνται από τη στοίβα αποτίμησης.

$$\begin{array}{|llllll|}\hline & \texttt{Iadd} : is & cs & vs & a:b:es & h[a:\texttt{Num}\,n,\ b:\texttt{Num}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{Num}\,(m+n)] \\ \hline\end{array}$$

$$\begin{array}{|llllll|}\hline & \texttt{Isub} : is & cs & vs & a:b:es & h[a:\texttt{Num}\,n,\ b:\texttt{Num}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{Num}\,(m-n)] \\ \hline\end{array}$$

$$\begin{array}{|llllll|}\hline & \texttt{Imul} : is & cs & vs & a:b:es & h[a:\texttt{Num}\,n,\ b:\texttt{Num}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{Num}\,(m*n)] \\ \hline\end{array}$$

$$\begin{array}{|llllll|}\hline & \texttt{Idiv} : is & cs & vs & a:b:es & h[a:\texttt{Num}\,n,\ b:\texttt{Num}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{Num}\,(m/n)] \\ \hline\end{array}$$

$$\begin{array}{|llllll|}\hline & \texttt{Imod} : is & cs & vs & a:b:es & h[a:\texttt{Num}\,n,\ b:\texttt{Num}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{Num}\,(m\%n)] \\ \hline\end{array}$$

Παρακάτω δίνεται ο κώδικας Haskell για την εντολή `Iadd`. Για τις υπόλοιπες εντολές είναι εντελώς παρόμοιος και παραλείπεται:

```
step (Iadd:is, cs, vs, a:b:es, h)
        = (is, cs, vs, c:es, h')
          where (h', c) = hAlloc h (hAdd (hLookup h b) (hLookup h a))

hAdd                    :: Value -> Value -> Value
hAdd (Num a) (Num b)    = Num (a+b)
```

- Εντολές ισότητας και ανισότητας `Ieq` και `Ine` : Όπως και στην περίπτωση των εντολών αριθμητικών πράξεων, οι εντολές ισότητας και ανισότητας περιμένουν οι διευθύνσεις που αντιστοιχούν στα δύο τελούμενα να βρίσκονται στην κορυφή της στοίβας αποτίμησης, και τις αντικαθιστούν με τη διεύθυνση του αποτελέσματος:

$$\begin{array}{|llllll|}\hline & \texttt{Ieq} : is & cs & vs & a:b:es & h[a:\texttt{Num}\,n,\ b:\texttt{Num}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{MBool}\,(m==n)] \\ \hline\end{array}$$

$$\begin{array}{|llllll|}\hline & \texttt{Ieq} : is & cs & vs & a:b:es & h[a:\texttt{MBool}\,n,\ b:\texttt{MBool}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{MBool}\,(m==n)] \\ \hline\end{array}$$

$$\begin{array}{|llllll|}\hline & \texttt{Ine} : is & cs & vs & a:b:es & h[a:\texttt{Num}\,n,\ b:\texttt{Num}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{MBool}\,(m\,!=n)] \\ \hline\end{array}$$

$$\begin{array}{|llllll|}\hline & \texttt{Ine} : is & cs & vs & a:b:es & h[a:\texttt{MBool}\,n,\ b:\texttt{MBool}\,m] \\ \Rightarrow & is & cs & vs & c:es & h[c:\texttt{MBool}\,(m\,!=n)] \\ \hline\end{array}$$

Παρακάτω παρατίθεται ο κώδικας Haskell για την εντολή ισότητας `Ieq`. Για την εντολή `Ine` είναι παρόμοιος και παραλείπεται:

```
step (Ieq:is, cs, vs, a:b:es, h)
          = (is, cs, vs, c:es, h')
          where (h', c) = hAlloc h (hEq (hLookup h b) (hLookup h a))

hEq                     :: Value -> Value -> Value
hEq (Num a) (Num b)     = MBool (a==b)
hEq (MBool a) (MBool b) = MBool (a==b)
```

- Εντολές σύγκρισης `Ige`, `Igt`, `Ile`, και `Ilo`. Είναι παρόμοιες με τις εντολές αριθμητικών πράξεων, πλην φυσικά του γεγονότος ότι το αποτέλεσμα είναι λογικού τύπου:

$$\boxed{\begin{array}{lllll} \quad \texttt{Ige} : is & cs & vs & a : b : es & h[a : \texttt{Num}\, n,\ b : \texttt{Num}\, m] \\ \Rightarrow \qquad is & cs & vs & c : es & h[c : \texttt{MBool}\ (m >= n)] \end{array}}$$

$$\boxed{\begin{array}{lllll} \quad \texttt{Igt} : is & cs & vs & a : b : es & h[a : \texttt{Num}\, n,\ b : \texttt{Num}\, m] \\ \Rightarrow \qquad is & cs & vs & c : es & h[c : \texttt{MBool}\ (m > n)] \end{array}}$$

$$\boxed{\begin{array}{lllll} \quad \texttt{Ile} : is & cs & vs & a : b : es & h[a : \texttt{Num}\, n,\ b : \texttt{Num}\, m] \\ \Rightarrow \qquad is & cs & vs & c : es & h[c : \texttt{MBool}\ (m <= n)] \end{array}}$$

$$\boxed{\begin{array}{lllll} \quad \texttt{Ilo} : is & cs & vs & a : b : es & h[a : \texttt{Num}\, n,\ b : \texttt{Num}\, m] \\ \Rightarrow \qquad is & cs & vs & c : es & h[c : \texttt{MBool}\ (m < n)] \end{array}}$$

  Ο αντίστοιχος κώδικας σε Haskell για την εντολή `Ige` (για τις υπόλοιπες είναι παρόμοιος) έχει ως εξής:

```
step (Ige:is, cs, vs, a:b:es, h)
          = (is, cs, vs, c:es, h')
            where (h', c) = hAlloc h (hGe (hLookup h b) (hLookup h a))

hGe                      :: Value -> Value -> Value
hGe (Num a) (Num b)      = MBool (a >= b)
```

- Εντολή αλλαγής προσήμου `Ineg` : Αλλάζει το πρόσημο του περιεχομένου της διεύθυνσης που βρίσκεται στην κορυφή της στοίβας αποτίμησης κσι τοποθετεί το αποτέλεσμα σε μια νέα διεύθυνση στο σωρό. Τοποθετεί τη διεύθυνση του αποτελέσματος στην κορυφή της στοίβας, ενώ αφαιρεί απο εκεί τη διεύθυνση του τελούμενου:

$$\boxed{\begin{array}{lllll} \quad \texttt{Ineg} : is & cs & vs & a : es & h[a : \texttt{Num}\, n] \\ \Rightarrow \qquad is & cs & vs & b : es & h[b : \texttt{Num}\,(-n)] \end{array}}$$

  Κώδικας Haskell:

```
step (Ineg:is, cs, vs, a:es, h)
           = (is, cs, vs, b:es, h')
             where (h', b) = hAlloc h (hNeg (hLookup h a))

hNeg                      :: Value -> Value
hNeg (Num a)              = Num (-a)
```

- Εντολή λογικής άρνησης `Inot` : Παρόμοια με την εντολή αλλαγής προσήμου· υπολογίζει το λογικά αντίθετο του περιεχομένου της διεύθυνσης που βρίσκεται στην κορυφή της στοίβας, το αποθηκεύει σε μια νέα διεύθυνση στο σωρό την οποία τοποθετεί στην κορυφή της στοίβας στη θέση της διεύθυνσης του τελούμενου:

$$\boxed{\begin{array}{lllll} \quad \texttt{Ineg} : is & cs & vs & a : es & h[a : \texttt{MBool}\, n] \\ \Rightarrow \qquad is & cs & vs & b : es & h[b : \texttt{MBool}\ (not\ n)] \end{array}}$$

  Κώδικας Haskell:

```
step (Inot:is, cs, vs, a:es, h)
          = (is, cs, vs, b:es, h')
            where (h', b) = hAlloc h (hNot (hLookup h a))

hNot                    :: Value -> Value
hNot (MBool a)          = MBool (not a)
```

- Εντολή δημιουργίας κόμβου κλήσης συνάρτησης `Mkap` : Η εντολή αναμένει στην κορυφή της στοίβας αποτίμησης να βρίσκονται μια διεύθυνση στο σωρό που περιέχει το κλείσιμο/όρισμα, και μια διεύθυνση που περιέχει το κλείσιμο που αντιστοιχεί στη συνάρτηση. Αφαιρεί τις δύο αυτές διευθύνσεις και τοποθετεί στη θέση τους μια νέα, που περιέχει ένα κόμβο κλήσης συνάρτησης (application node):

$$\begin{array}{|llllll|}\hline & \mathtt{Mkap} : is & cs & vs & a : f : es & h \\ \Rightarrow & is & cs & vs & app : es & h[app : \mathtt{App}\ f\ a] \\ \hline\end{array}$$

Κώδικας Haskell:

```
step (Mkap:is, cs, vs, a:f:es, h)
          = (is, cs, vs, app:es, h')
            where (h', app) = hAlloc h (App f a)
```

- Εντολή αναδρομής `MkFix` : Η εντολή αυτή υποθέτει ότι στην κορυφή της στοίβας αποτίμησης υπάρχει μια διεύθυνση που περιέχει ένα κλείσιμο–συνάρτηση `f`. Σκοπός της εντολής είναι να φτιάξει ένα κόμβο κλήσης συνάρτησης που να αντιστοιχεί στη συνάρτηση `fix f`. Όπως έχουμε δει στο κεφάλαιο 4, για τη συνάρτηση αυτή ισχύει $fix\ f = f\ (fix\ f)$. Επομένως στη θέση της διεύθυνσης του ορίσματος, πρέπει να υπάρχει η διεύθυνση του ίδιου του κόμβου κλήσης:

$$\begin{array}{|llllll|}\hline & \mathtt{MkFix} : is & cs & vs & r : es & h \\ \Rightarrow & is & cs & vs & app : es & h[app : \mathtt{App}\ r\ app] \\ \hline\end{array}$$

Κώδικας Haskell:

```
step (MkFix:is, cs, vs, r:es, h)
          = (is, cs, vs, app:es, h')
            where (h', app) = hAlloc h (App r app)
```

- `PopVar` n : Πρόκειται για μια απλή εντολή, η οποία δέχεται ως όρισμα έναν ακέραιο αριθμό n και αφαιρεί n διευθύνσεις από την κορυφή της στοίβας μεταβλητών:

$$\begin{array}{|llllll|}\hline & \mathtt{PopVar\ n} : is & cs & v_1 : v_2 : \ldots : v_n : vs & es & h \\ \Rightarrow & is & cs & vs & es & h \\ \hline\end{array}$$

Κώδικας Haskell:

```
step ((PopVar n):is, cs, vs, es, h)  = (is, cs, drop n vs, es, h)
```

- `Pop` n : Είναι μια εντολή παρόμοια με την προηγούμενη· αφαιρεί n διευθύνσεις από την κορυφή της στοίβας αποτίμησης:

$$\boxed{\begin{array}{llllll} & \texttt{Pop n} : is & cs & vs & a_1 : a_2 : ... : a_n : es & h \\ \Rightarrow & is & cs & vs & es & h \end{array}}$$

Και ο αντίστοιχος κώδικας Haskell:

```
step ((Pop n):is, cs, vs, es, h)     = (is, cs, vs, drop n es, h)
```

- `PushArg` n m : Η εντολή αυτή κατασκευάζει ένα κλείσιμο που αντιστοιχεί σε όρισμα συνάρτησης. Το κλείσιμο που κατασκευάζεται περιέχει τον αριθμό n, που είναι η θέση του αντίστοιχου κώδικα στη λίστα codestore, και μία λίστα με m μέλη, που είναι οι διευθύνσεις στο σωρό των ελεύθερων μεταβλητών του κώδικα αυτού. Η εντολή αναμένει ότι οι m αυτές διευθύνσεις υπάρχουν στην κορυφή της στοίβας αποτίμησης, από όπου και τις αφαιρεί. Στην κορυφή της στοίβας αποτίμησης τοποθετεί τη διεύθυνση του κλεισίματος που κατασκεύασε:

$$\boxed{\begin{array}{llllll} & \texttt{PushArg n m} : is & cs & vs & a_1 : ... : a_m : es & h \\ \Rightarrow & is & cs & vs & addr : es & h[addr : \texttt{Arg}\ n\ (a_1 : ... : a_m)] \end{array}}$$

Ο κώδικας Haskell που υλοποιεί τον παραπάνω κανόνα έχει ως εξής:

```
step ((PushArg n m):is, cs, vs, es, h)
             = (is, cs, vs, addr: drop m es, h')
               where freevars = take m es
                     (h', addr) = hAlloc h (Arg n freevars)
```

- `PushFun` n m : Η εντολή αυτή είναι εντελώς παρόμοια με την προηγούμενη, με τη διαφορά ότι ότι κατασκευάζει ένα κλείσιμο που αντιστοιχεί σε συνάρτηση:

$$\boxed{\begin{array}{llllll} & \texttt{PushFun n m} : is & cs & vs & a_1 : ... : a_m : es & h \\ \Rightarrow & is & cs & vs & addr : es & h[addr : \texttt{Fun}\ n\ (a_1 : ... : a_m)] \end{array}}$$

Κώδικας Haskell:

```
step ((PushFun n m):is, cs, vs, es, h)
             = (is, cs, vs, addr: drop m es, h')
               where freevars = take m es
                     (h', addr) = hAlloc h (Fun n freevars)
```

- `Push` n : Τοποθετεί στην κορυφή της στοίβας αποτίμησης τη διεύθυνση που βρίσκεται στη n-οστή (ξεκινώντας από το μηδέν) θέση της στοίβας μεταβλητών:

$$\boxed{\begin{array}{llllll} & \texttt{Push n} : is & cs & v_0 : v_1 : ... : v_{n-1} : vs & es & h \\ \Rightarrow & is & cs & v_0 : v_1 : ... : v_{n-1} : vs & v_{n-1} : es & h \end{array}}$$

Ο κώδικας σε Haskell έχει ως εξής:

```
step ((Push n):is, cs, vs, es, h)    = (is, cs, vs, vs!!n : es, h)
```

- `PushVal val` : Δημιουργεί ένα νέο κόμβο στο σωρό στον οποίον τοποθετεί την τιμή val. Στην κορυφή της στοίβας αποτίμησης τοποθετεί τη διεύθυνση αυτού του νέου κόμβου:

$$\boxed{\begin{array}{llllll} & \texttt{PushVal val} : is & cs & vs & es & h \\ \Rightarrow & is & cs & vs & a : es & h[a : \texttt{val}] \end{array}}$$

Κώδικας Haskell:

```
step ((PushVal val):is, cs, vs, es, h)
                   = (is, cs, vs, a:es, h')
                     where (h', a) = hAlloc h val
```

- `RecAcc` id : Η εντολή αυτή δίνει πρόσβαση στο πεδίο id ενός record. Αναμένει να βρίσκεται στην κορυφή της στοίβας αποτίμησης η διεύθυνση που περιέχει το record. Αναζητά στο record το πεδίο id, και τοποθετεί τη διεύθυνση του τελευταίου στην κορυφή της στοίβας αποτίμησης:

$$\boxed{\begin{array}{llllll} & \texttt{RecAcc id} : is & cs & vs & addr : es & h[addr : \texttt{Record}\ (\langle id_1,\ a_1\rangle : ... : \langle id,\ a\rangle : ...)] \\ \Rightarrow & is & cs & vs & a : es & h \end{array}}$$

Κώδικας Haskell:

```
step ((RecAcc id):is, cs, vs, addr:es, h)
                 = (is, cs, vs, a:es, h)
                   where a = looookup id (unRec (hLookup h addr))

looookup                :: Eq a => a -> [(a, b)] -> b
looookup x ((y, i):xs)   = if x==y then i else looookup x xs

unRec                   :: Value -> [(I, HAddr)]
unRec (Record r)         = r
```

- `RecExt` id : Αυτή η εντολή σε ένα υπάρχον record προσθέτει ένα πεδιό id, του οποίου το περιεχόμενο βρίσκεται στη διεύθυνση που είναι στην κορυφή της στοίβας αποτίμησης τη στιγμή που εκτελείται η εντολή. Στη δεύτερη θέση της στοίβας αποτίμησης βρίσκεται η διεύθυνση του αρχικού record. Το νέο record τοποθετείται σε μια νέα θέση στο σωρό, η διεύθυνση της οποίας τοποθετείται στην κορυφή της στοίβας αποτίμησης:

$$\boxed{\begin{array}{llllll} & \texttt{RecExt id} : is & cs & vs & f : addr : es & h[addr : \texttt{Record}\ r] \\ \Rightarrow & is & cs & vs & a : es & h[a : \texttt{Record}\ (\langle id,\ f\rangle : r)] \end{array}}$$

Κώδικας Haskell:

```
step ((RecExt id):is, cs, vs, f:addr:es, h)
           = (is, cs, vs, a:es, h')
             where (h', a) = hAlloc h (hRecExt (hLookup h addr) id f)

hRecExt                 :: Value -> I -> HAddr -> Value
hRecExt (Record r) i a   = Record ((i, a):r)
```

- `Update` : Η εντολή αυτή, η οποία όπως θα δούμε στο επόμενο κεφάλαιο χρησιμεύει στην κλήση κατά τιμή, αντικαθιστά τη διεύθυνση που βρίσκεται στην κορυφή της διεύθυνσης μεταβλητών με αυτήν που βρίσκεται στην κορυφή της στοίβας αποτίμησης:

	$\texttt{Update} : is$	cs	$v : vs$	$a : es$	h
$\Rightarrow$	is	cs	$a : vs$	es	h

Ο κώδικας Haskell είναι ο ακόλουθος:

```
step (Update:is, cs, vs, a:es, h)   = (is, cs, a: tail vs, es, h)
```

Κεφάλαιο 6

Μεταγλωττιστής σε κώδικα G-machine

Στο παρόν κεφάλαιο παρουσιάζουμε τον μεταγλωττιστή που δέχεται ως είσοδο ένα πρόγραμμα Cantor και στην έξοδο παράγει G-machine κώδικα.

6.1 Γενικά

Ο μεταγλωττιστής φέρει κάποιες ομοιότητες με το διερμηνέα που παρουσιάσαμε στο κεφάλαιο 4. Χρησιμοποιεί μια μονάδα περιβάλλοντος και μια μονάδα κατάστασης. Η έξοδός του είναι μια λίστα αποτελούμενη από τμήματα κώδικα, όπου κάθε τμήμα κώδικα είναι μια σειρά από εντολές G-machine. Στην πρώτη θέση της λίστας είναι το τμήμα κώδικα που αντιστοιχεί στην «κύρια» έκφραση του προγράμματος που δόθηκε στην είσοδο, ενώ στις επόμενες θέσεις υπάρχουν τμήματα κώδικα που αντιστοιχούν σε υποεκφράσεις της έκφρασης αυτής/. Στην πρώτη θέση της λίστας είναι το τμήμα κώδικα που αντιστοιχεί στην «κύρια» έκφραση του προγράμματος που δόθηκε στην είσοδο, ενώ στις επόμενες θέσεις υπάρχουν τμήματα κώδικα που αντιστοιχούν σε υποεκφράσεις της έκφρασης αυτής. Για κάθε έκφραση λάμδα που συναντά, δημιουργεί ένα καινούριο τμήμα κώδικα στη λίστα εξόδου. Το ίδιο κάνει και για τα ορίσματα συναρτήσεων, καθώς η κλήση με όνομα απαιτεί το όρισμα να αποτιμηθεί όχι κατά την κλήση της συνάρτησης, αλλά όπου συναντάται μέσα στο «σώμα» της συνάρτησης η τυπική παράμετρος της τελευταίας. Δηλαδή τα ορίσματα συναρτήσεων αντιμετωπίζονται ουσιαστικά σαν συναρτήσεις μηδενικού αριθμού τυπικών παραμέτρων. Επίσης, ξεχωριστά τμήματα κώδικα δημιουργούνται για κάθε εναλλακτικό σκέλος σε μια εντολή συνθήκης (**if–then–else**). Τέλος, καθώς κατά τη διάρκεια της μεταγλώττισης ο μεταγλωττιστής πρέπει να γνωρίζει ποιές είναι οι ελεύθερες μεταβλητές κάθε έκφρασης λάμδα, ή οι μεταβλητές που υπάρχουν μέσα σε μια έκφραση–όρισμα συνάρτησης, πριν αρχίσει η μεταγλώττιση γίνεται ανάλυση της έκφρασης–εισόδου από μια κατάλληλη συνάρτηση.

6.2 Τύποι

Η είσοδος του μεταγλωττιστή, όπως και στην περίπτωση του ελεγκτή τύπων και του διερμηνέα, είναι τύπου `Expr`. Τον τύπο αυτόν τον ορίσαμε στην παράγραφο 3.2.2 και δεν επαναλαμβάνουμε τον ορισμό του εδώ. Επειδή ωστόσο πριν τη μεταγλώττιση πρέπει να εντοπιστούν οι ελεύθερες μεταβλητές, χρειαζόμαστε έναν «εμπλουτισμένο» τύπο στον οποίο να μπορούν συνυπάρξουν η έκφραση εισόδου μαζί με επιλέον πληροφορίες, όπως είναι οι ελεύθερες μεταβλητές κάθε υποέκφρασης. Ορίζουμε έτσι τον τύπο `FreeVarExpr` και τον παραμετρικό τύπο `AnnotExpr a`:

```
data FreeVarExpr     = FV {freevs :: [I], annexpr :: AnnotExpr FreeVarExpr}

data AnnotExpr a     = ACon Int
                     | AMyTrue
                     | AMyFalse
                     | AIde I
                     | AUnexp UnOp a
```

```
                    | ABexp BinOp a a
                    | AIf a a a
                    | AVar I Data a
                    | ADeref a
                    | AAssign a a
                    | ALam Mode I Type a
                    | ACall a a
                    | ARec a
                    | ARecAccess a I
                    | ARecExtend a I a
                    | AEmpty
```

Για τους τύπους I, UnOp, BinOp, Data, Mode και Type ισχύουν οι ορισμοί που δώσαμε στην παράγραφο 3.2.2.

6.3 Ελεύθερες μεταβλητές

Η εύρεση των ελεύθερων μεταβλητών στην έκφραση εισόδου γίνεται από τη συνάρτηση

```
freevar             :: Expr -> FreeVarExpr
```

Προκειμένου να εντοπίσει τις ελεύθερες μεταβλητές μιας έκφρασης λάμδα, η συνάρτηση freevar εντοπίζει πρώτα το σύνολο των μεταβλητών που βρίσκονται μέσα στο «σώμα» της έκφρασης λάμδα, και αφαιρεί στη συνέχεια από το σύνολο αυτό την τυπική παράμετρο της έκφρασης λάμδα.

Παρακάτω δίνεται ο κώδικας για τη συνάρτηση freevar για ορισμένες ενδεικτικές εκφράσεις της Cantor. Καθώς ο κώδικας για τις υπόλοιπες είναι παρόμοιος, παραλείπεται:

- Σταθερές ακεραίου τύπου: Προφανώς δεν υπάρχει μετάβλητή:

  ```
  freevar (Con i)     = FV [] (ACon i)
  ```

- Αναγνωριστικά: Είναι φανερό ότι περιέχουν μια μεταβλητή, δηλ. τον εαυτό τους:

  ```
  freevar (Ide x)     = FV [x] (AIde x)
  ```

- Δυαδικές εκφράσεις: Οι μεταβλητές είναι η ένωση του συνόλου των μεταβλητών του πρώτου τελούμενου με το σύνολο των μεταβλητών του δεύτερου τελούμενου:

  ```
  freevar (Bexp binop u v)
                   = let d1 = freevar u
                         d2 = freevar v
                     in FV (freevs d1 ++ freevs d2) (ABexp binop d1 d2)
  ```

- Εκφράσεις λάμδα: Οι ελεύθερες μεταβλητές είναι το σύνολο των μεταβλητών του «σώματος» της έκφρασης λάμδα, αν αφαιρέσουμε το αναγνωριστικό που αντιστοιχεί στην τυπική παράμετρο της έκφρασης λάμδα:

  ```
  freevar (Lam m x t u)= let d = freevar u
                         in FV (filter (/= x) (freevs d)) (ALam m x t d)
  ```

- Δήλωση τοπικών μεταβλητών: Παρόμοια περίπτωση με τις εκφράσεις λάμδα:

  ```
  freevar (Var i t u)  = let d = freevar u
                         in FV (filter (/= i) (freevs d)) (AVar i t d)
  ```

6.4 Συνάρτηση μεταγλώττισης

Συμβολίζουμε με $\mathcal{C}$ τη συνάρτηση μεταγλώττισης, η οποία θα δέχεται ως είσοδο μια έκφραση τύπου $FreeVarExpr$ και θα επιστρέφει ένα τμήμα κώδικα μέσω μιας κατάλληλης μονάδας $GMCompile$. Θα είναι δηλαδή:

$$C : FreeVarExpr \rightarrow GMCompile\ \ GMCodeSegment$$

Σημειώνουμε ότι στις παρακάτω εξισώσεις γράφουμε τις εκφράσεις τύπου $FreeVarExpr$ ως ζεύγη, με μέλη μια λίστα αναγνωριστικών και μια έκφραση τύπου $AnnExpr\ FreeVarExpr$. Για να είναι εύκολα αναγνώσιμες δεν χρησιμοποιούμε τους data constructors `ACon`, `AIde` κ.λπ. και κάνουμε την παραδοχή ότι οι συμβολισμοί exp, exp_1, exp_2, exp_3 αντιστοιχούν σε εκφράσεις τύπου $FreeVarExpr$.

Αν μεταφέρουμε τον ορισμό της $\mathcal{C}$ στη Haskell, ονομάζουμε τη συνάρτηση μεταγλώττισης "`compile`". Η `compile` δέχεται ως είσοδο το αποτέλεσμα της συνάρτησης `freevar` και παράγει ως έξοδο ένα τμήμα κώδικα. Έχει επομένως τύπο

```
compile                  :: FreeVarExpr -> GMCompile GMCodeSegment
```

Απομένει τώρα να ορίσουμε τη μονάδα `GMCompile`. Η μονάδα αυτή πρέπει να έχει δυνατότητες χειρισμού περιβάλλοντος και κατάστασης. Συνεπώς συνδυάζοντας τη μοναδα περιβάλλοντος με τη μονάδα κατάστασης ορίζουμε:

```
type GMCompile a         = EnvT GMEnv (StateT GMCodeStore Id) a

type GMEnv               = [(I, Int)]
```

Για τα `GMCodeSegment` και `GMCodeStore` ισχύουν οι ορισμοί που δώσαμε στην παράγραφο 5.6. Επίσης χρειαζόμαστε μια ακόμη βοηθητική συνάρτηση, η οποία θα δέχεται ως όρισμα ένα νέο τμήμα κώδικα και θα το τοποθετεί στο τέλος του GMCodeStore (δηλ. του state), επιστρέφοντας τη θέση στην οποία τοποθετήθηκε το τμήμα αυτό. Θα είναι δηλαδή

$$newCodeSeg : GMCodeSegment \rightarrow GMCompile\ Int$$

Όταν ολοκληρωθεί η μεταγλώττιση, το state θα περιέχει όλο τον κώδικα που παράχθηκε. Είμαστε έτοιμοι να προχωρήσουμε στην παρουσίαση των κανόνων μεταγλώττισης για κάθε έκφραση της Cantor:

- Σταθερές ακέραιου ή λογικού τύπου : Ο κώδικας που παράγεται έχει μία και μόνη εντολή, την $PushVal\ val$. Είναι επομένως

  $$\mathcal{C}[\![l,\ n]\!] = unit([PushVal\ (Num\ n)])$$
  $$\mathcal{C}[\![l,\ \mathbf{true}]\!] = unit([PushVal\ (MBool\ True)]$$
  $$\mathcal{C}[\![l,\ \mathbf{false}]\!] = unit([PushVal\ (MBool\ False)]$$

  Σε κώδικα Haskell εύκολα γράφουμε:

  ```
  compile (FV l (ACon i)) = return [PushVal (Num i)]
  compile (FV l AMyTrue)  = return [PushVal (MBool True)]
  compile (FV l AMyFalse) = return [PushVal (MBool False)]
  ```

- Αναγνωριστικά : Καλείται η "$rdEnv$" που επιστρέφει το τρέχον περιβάλλον $\rho : \mathbf{Ide} \rightarrow \mathbf{N}$. Ο κώδικας που παράγεται έχει μια και μόνη εντολή, την $Push\ (\rho\ id)$.

$\mathcal{C}[\![l,\ id]\!] = rdEnv * (\lambda\rho.unit([Push\ (\rho\ id)]))$

Κώδικας Haskell:

```
compile (FV l (AIde x)) = do env <- rdEnv
                             return [Push (loookup x env)]

loookup                 :: Eq a => a -> [(a, b)] -> b
loookup x ((y, i):xs)   = if x==y then i else loookup x xs
```

- Εκφράσεις **neg** και **not** : Πρώτα μεταγλωττίζεται η έκφραση–τελούμενο και στον παραγόμενο κώδικα προστίθενται οι εντολές $Eval$ και $Ineg$ ή $Inot$. Η $Eval$ είναι απαραίτητη, καθώς ο δείκτης που θα υπάρχει στην κορυφή της στοίβας αποτίμησης μετά την εκτέλεση του κώδικα που αντιστοιχεί στο τελούμενο, είναι πιθανό να δείχνει σε μη αποτιμημένη έκφραση. Έχουμε λοιπόν:

$\mathcal{C}[\![l,\ \mathbf{neg}\ exp]\!] = \mathcal{C}[\![exp]\!] * (\lambda c.unit(c \mathbin{+\!\!+} [Eval,\ Ineg]))$
$\mathcal{C}[\![l,\ \mathbf{not}\ exp]\!] = \mathcal{C}[\![exp]\!] * (\lambda c.unit(c \mathbin{+\!\!+} [Eval,\ Inot]))$

Κώδικας Haskell:

```
compile (FV l (AUnexp unop b))
                       = do c <- compile b
                            case unop of
                              Not -> return (c ++ [Eval] ++ [Inot])
                              Neg -> return (c ++ [Eval] ++ [Ineg])
```

- Δυαδική έκφραση με τον τελεστή ";" : Ο παραγόμενος κώδικας είναι ο κώδικας που προκύπτει από τη μεταγλώττιση του πρώτου τελούμενου, ακολουθούμενος από τις εντολές $Eval$ και $Pop\ 1$ και τον κώδικα που προκύπτει από τη μεταγλώττιση του δεύτερου τελούμενου. Η εντολή $Pop\ 1$ χρειάζεται ώστε να απομακρυνθεί από τη στοίβα αποτίμησης η διεύθυνση του αποτελέσματος της αποτίμησης του πρώτου τελούμενου. Η αλληλουχία εντολών "$Eval,\ Pop\ 1$" αντικατοπτρίζει το γεγονός, ότι σε μια έκφραση "`a;b`" στη γλώσσα Cantor η τιμή της έκφρασης "`a`", αν και υπολογίζεται, αγνοείται.

$\mathcal{C}[\![l,\ exp_1; exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 \mathbin{+\!\!+} [Eval,\ Pop\ 1] \mathbin{+\!\!+} c_2)))$

Κώδικας Haskell:

```
compile (FV l (ABexp Semicolon u v))
                         = do c1 <- compile u
                              c2 <- compile v
                              return (c1 ++ [Eval, Pop 1] ++ c2)
```

- Δυαδικές εκφράσεις με τους τελεστές **and** και **or** : Οι εκφράσεις αυτές, στις οποίες το δεύτερο τελούμενο αποτιμάται μόνο όταν αυτό είναι απαραίτητο, μεταγλωττίζονται μέσω του μετασχηματισμού τους σε μια έκφραση συνθήκης:

$\mathcal{C}[\![l,\ exp_1\ \mathbf{and}\ exp_2]\!] = \mathcal{C}[\![l,\ \mathbf{if}\ exp_1\ \mathbf{then}\ exp_2\ \mathbf{else}\ ([],\ \mathbf{false})]\!]$

$\mathcal{C}[\![l,\ exp_1 \textbf{ or } exp_2]\!] = \mathcal{C}[\![l,\ \textbf{if } exp_1 \textbf{ then } ([],\ \textbf{true}) \textbf{ else } exp_2]\!]$

Κώδικας Haskell:

```
compile (FV l (ABexp And u v))
                         = compile (FV l (AIf u v (FV [] AMyFalse)))
compile (FV l (ABexp Or u v))
                         = compile (FV l (AIf u (FV [] AMyTrue) v))
```

- Δυαδικές εκφράσεις με έναν από τους τελεστές $+\ -\ *\ /\ \%\ \ ==\ !=\ <=\ >=\ <\ >$: Ο κώδικας που παράγεται είναι ο κώδικας που αντιστοιχεί στο πρώτο τελούμενο, η εντολή *Eval*, ο κώδικας του δεύτερου τελούμενου και η εντολή *Eval*, και τέλος μια από τις εντολές *Iadd*, *Isub*, *Imul*, *Idiv*, *Imod*, *Ieq*, *Ine*, *Ile*, *Ige*, *Ilo* και *Igt*:

$$\mathcal{C}[\![l,\ exp_1 + exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Iadd])))$$
$$\mathcal{C}[\![l,\ exp_1 - exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Isub])))$$
$$\mathcal{C}[\![l,\ exp_1 * exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Imul])))$$
$$\mathcal{C}[\![l,\ exp_1 / exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Idiv])))$$
$$\mathcal{C}[\![l,\ exp_1 \% exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Imod])))$$
$$\mathcal{C}[\![l,\ exp_1 == exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Ieq])))$$
$$\mathcal{C}[\![l,\ exp_1 != exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Ine])))$$
$$\mathcal{C}[\![l,\ exp_1 <= exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Ile])))$$
$$\mathcal{C}[\![l,\ exp_1 >= exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Ige])))$$
$$\mathcal{C}[\![l,\ exp_1 < exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Ilo])))$$
$$\mathcal{C}[\![l,\ exp_1 > exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Igt])))$$

Κώδικας Haskell:

```
compile (FV l (ABexp binop u v)
               = do c1 <- compile u
                    c2 <- compile v
                    return (c1 ++ [Eval] ++ c2 ++ [Eval] ++ [i])
                    where i = case binop of
                                Plus -> Iadd
                                Minus -> Isub
                                Mul -> Imul
                                Div -> Idiv
                                Mod -> Imod
                                Eq' -> Ieq
                                Ne -> Ine
                                Gt -> Igt
                                Lo -> Ilo
                                Ge -> Ige
                                Le -> Ile
```

- Εντολή συνθήκης : Μεταγλωττίζονται η συνθήκη και τα δύο εναλλακτικά σκέλη της εντολής. Ο κώδικας για καθένα από αυτά τα σκέλη τοποθετείται σε ένα νέο τμήμα κώδικα. Στο τρέχον τμήμα κώδικα, προστίθενται ο κώδικας της συνθήκης ακολουθούμενος από τις εντολές *Eval* και *Cond*:

$\mathcal{C}[\![l,\ \textbf{if } exp_1 \textbf{ then } exp_2 \textbf{ else } exp_3]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.\mathcal{C}[\![exp_3]\!] * (\lambda c_3.$

$newCodeSeg\ c_2 * (\lambda s_1.newCodeSeg\ c_3 * (\lambda s_2.unit(c_1 +\!\!+ [Eval,\ Cond\ s_1\ s_2]))))))$

Κώδικας Haskell:

```
compile (FV l (AIf e1 e2 e3))   = do c1 <- compile e1
                                     c2 <- compile e2
                                     c3 <- compile e3
                                     s1 <- newCodeSeg c2
                                     s2 <- newCodeSeg c3
                                     return (c1 ++ [Eval, Cond s1 s2])

newCodeSeg              :: GMCodeSegment -> GMCompile Int
newCodeSeg codeseg      = do cs <- fetch
                                 update (++ [codeseg])
                                 return (length cs + 1)
```

- Λάμβδα εκφράσεις : Στο τρέχον τμήμα κώδικα προστίθενται οι κατάλληλες εντολές $Push$, που φορτώνουν τις διευθύνσεις των ελεύθερων μεταβλητών στη στοίβα αποτίμησης, μαζί με την εντολή $PushFun\ s\ n$, όπου n το πλήθος των ελεύθερων μεταβλητών και s το τμήμα κώδικα στο οποίο τοποθετείται ο κώδικας του «σώματος» της έκφρασης λάμβδα. Η $PushFun$ όταν εκτελεστεί θα αποθηκεύσει στο σωρό ένα κλείσιμο, δηλ. ένα ζεύγος με μέλη το s (δείκτης στον κώδικα) και τη λίστα διευθύνσεων των ελεύθερων μεταβλητών. Ο τρόπος μεταγλώττισης του «σώματος» της έκφρασης λάμβδα διαφέρει μεταξύ των εκφράσεων με κλήση με τιμή και αυτών με κλήση με όνομα. Και στις δύο περιπτώσεις, το «σώμα» της έκφρασης μεταγλωττίζεται μέσα στο περιβάλλον που προκύπτει, αν αντιστοιχήσουμε την τυπική παράμετρο στη θέση μηδέν και τις n ελεύθερες μεταβλητές στις θέσεις 1 έως n. Στο τέλος του παραγόμενου κώδικα προστίθεται η εντολή $Eval$ και η εντολή $PopVar\ (n+1)$. Η τελευταία εντολή έχει σκοπό να αφαιρέσει από τη στοίβα μεταβλητών τις διευθύνσεις του ορίσματος και των ελεύθερων μεταβλητών της έκφρασης λάμβδα. Αν η έκφραση λάμβδα καλείται με όνομα, ο κώδικας αυτός τοποθετείται σε ένα νέο τμήμα κώδικα ως έχει. Αν όμως πρόκειται για έκφραση λάμβδα που καλείται με τιμή, στην αρχή του κώδικα προστίθεται η αλληλουχία εντολών $Push\ 0$, $Eval$, $Update$. Οι δύο πρώτες εντολές αποτιμούν το όρισμα της έκφρασης λάμβδα. Η τρίτη εντολή ($Update$) αντικαθιστά στην κορυφή της στοίβας μεταβλητών τη διεύθυνση του μη αποτιμημένου ορίσματος με τη διεύθυνση του αποτελέσματος της αποτίμησης.

$$
\begin{aligned}
&\mathcal{C}[\![(fv_1,...,fv_n),\ \mathbf{lambda}\,!\,x : type.exp]\!] = rdEnv * (\lambda\rho.\\
&\qquad inEnv\ [x \to 0, fv_1 \to 1, ..., fv_n \to n]\ \mathcal{C}[\![exp]\!] * (\lambda f.\\
&\qquad newCodeSeg\ (f +\!\!+ [Eval,\ PopVar\ (n+1)]) * (\lambda s.\\
&\qquad unit([Push\ (\rho\ fv_n), ..., Push\ (\rho\ fv_1),\ PushFun\ s\ n]))))\\
&\mathcal{C}[\![(fv_1,...,fv_n),\ \mathbf{lambda}\ x : type.exp]\!] = rdEnv * (\lambda\rho.\\
&\qquad inEnv\ [x \to 0, fv_1 \to 1, ..., fv_n \to n]\ \mathcal{C}[\![exp]\!] * (\lambda f.\\
&\qquad newCodeSeg\ ([Push\ 0,\ Eval,\ Update] +\!\!+ f +\!\!+ [Eval,\ PopVar\ (n+1)]) * (\lambda s.\\
&\qquad unit([Push\ (\rho\ fv_n), ..., Push\ (\rho\ fv_1),\ PushFun\ s\ n]))))
\end{aligned}
$$

Κώδικας Haskell:

```
compile (FV l (ALam m x t v))
            = do env <- rdEnv
                 let freevs = nub l
                     inds = map (flip loookup env) freevs
```

```
                saves = reverse $ map Push inds
                env' = (x, 0) : zipWith (\x y -> (x, y))
                                        freevs [1..length freevs]
            f <- inEnv env' (compile v)
            let c = case m of
                     BV -> [Push 0, Eval, Update]
                     BN -> []
            seg <- newCodeSeg (c ++ f ++
                              [Eval, PopVar (length freevs + 1)])
            return (saves ++ [PushFun seg (length saves)])
```

- Κλήση συνάρτησης "exp_1 exp_2": Μεταγλωττίζεται η exp_2 μέσα στο κατάλληλο περιβάλλον και τοποθετείται σε ένα νέο τμήμα κώδικα "s". Επίσης μεταγλωττίζεται η exp_1 και επιστρέφεται ο παραγόμενος κώδικας ακολουθούμενος από μια εντολή $Eval$, τις εντολές $Push$ που χρειάζονται για να φορτωθούν στη στοίβα αποτίμησης οι διευθύνσεις των μεταβλητών που χρειάζεται η exp_2, την εντολή $PushArg$ που δημιουργεί το κλείσιμο που αντιστοιχεί στο όρισμα, και τέλος την εντολή $Mkap$ που δημιουργεί τον κόμβο κλήσης συνάρτησης (application node) στο σωρό:

$$\begin{aligned}&\mathcal{C}[\![l,\ exp_1\ ((v_1,...,v_n),\ exp_2)]\!] = rdEnv * (\lambda\rho.\mathcal{C}[\![exp_1]\!] * (\lambda c_1.\\&\qquad inEnv\ [v_1 \to 0,...,v_n \to n-1]\ \mathcal{C}[\![exp_2]\!] * (\lambda c_2.\\&\qquad newCodeSeg\ (c_2 +\!\!+ [Eval,\ PopVar\ n)]) * (\lambda s.\\&\qquad unit(c_1 +\!\!+ [Eval,\ Push\ (\rho\ v_n),...,Push\ (\rho\ v_1),\ PushArg\ s\ (n-1),\ Mkap])))))\end{aligned}$$

Κώδικας Haskell:

```
compile (FV l1 (ACall e1 (FV l2 e2)))
          = do env <- rdEnv
               c1 <- compile e1
               let freevars = nub l2
                   inds = map (flip loookup env) freevs
                   saves = reverse $ map Push inds
                   env' = zipWith (\x y -> (x, y))
                                      freevs [0..length freevs - 1]
               c2 <- inEnv env' (compile (FV l2 e2))
               seg <- newCodeSeg (c2 ++
                                    [Eval, PopVar (length freevs)])
               return (c1 ++ [Eval] ++ saves ++
                              [PushArg seg (length saves), Mkap])
```

- Δήλωση τοπικής μεταβλητής : Προστίθεται στο περιβάλλον η νέα μεταβλητή, και μέσα στο νέο περιβάλλον μεταγλωττίζεται η έκφραση που ακολουθεί τη λέξη–κλειδί **in**. Στην αρχή του παραγόμενου κώδικα μπαίνει η εντολή $Alloc$, που αναθέτει χώρο στο σωρό για τη νέα μεταβλητή και στο τέλος η $PopVar$ 1, που αφαιρεί από τη στοίβα μεταβλητών τη διεύθυνση της μεταβλητής.

$$\begin{aligned}&\mathcal{C}[\![l,\ \textbf{var}\ i : data\ \textbf{in}\ exp]\!] = rdEnv * (\lambda\rho.inEnv\ (extend\ i\ \rho)\ \mathcal{C}[\![exp]\!] * (\lambda c.\\&\qquad\qquad unit([Alloc] +\!\!+ c +\!\!+ [PopVar\ 1])))\end{aligned}$$

όπου:

$$extend\ i\ [x_0 \to 0, x_1 \to 1, ..., x_n \to n] = [i \to 0, x_0 \to 1, x_1 \to 2, ..., x_n \to n+1]$$

Κώδικας Haskell:

```
compile (FV l (AVar i d exp))  = do env <- rdEnv
                                    let env' = extend i env
                                    c <- inEnv env' (compile exp)
                                    return ([Alloc] ++ c ++ [PopVar 1])

extend                  :: a -> [(a, Int)] -> [(a, Int)]
extend a env            = (a, 0) : incsnd env

incsnd                  :: [(a,Int)] -> [(a, Int)]
incsnd []               = []
incsnd ((a, n):xs)      = (a, n+1) : incsnd xs
```

- Πρόσβαση στην τιμή τοπικής μεταβλητής : Μεταγλωττίζεται η έκφραση που βρίσκεται μετά τον τελεστή **deref** και επιστρέφεται ο παραγόμενος κώδικας μαζί με τις εντολές $Eval$ και $Dereff$.

  $\mathcal{C}[\![l,\ \mathbf{deref}\, exp]\!] = \mathcal{C}[\![exp]\!] * (\lambda c.unit(c +\!\!+[Eval,\ Dereff]))$

  Κώδικας Haskell:

```
compile (FV l (ADeref exp))    = do c <- compile exp
                                    return (c ++ [Eval, Dereff])
```

- Ανάθεση τιμής σε τοπική μεταβλητή :

  $$\mathcal{C}[\![l,\ exp_1 := exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.\ unit(c_1 +\!\!+[Eval] +\!\!+c_2 + [Eval,\ Assig])))$$

  Κώδικας Haskell:

```
compile (FV l (AAssign e1 e2))
                      = do c1 <- compile e1
                           c2 <- compile e2
                           return (c1 ++ [Eval] ++ c2 ++ [Eval, Assig])
```

- Αναδρομή : Μεταγλωττίζεται η έκφραση exp που ακολουθεί τον τελεστή **rec**, και επιστρέφεται ο παραγόμενος κώδικας ακολουθούμενος από την εντολή $MkFix$. Υπενθυμίζουμε ότι η $MkFix$ ουσιαστικά κατασκευάζει την $fix\ exp$.

  $\mathcal{C}[\![l,\ \mathbf{rec}\ (exp)]\!] = \mathcal{C}[\![exp]\!] * (\lambda c.unit(c +\!\!+[MkFix]))$

  Κώδικας Haskell:

```
compile (FV l1 (ARec exp))     = do c <- compile exp
                                    return (c ++ [MkFix])
```

- Κενό record : Όπως και στην περίπτωση των σταθερών ακέραιου και λογικού τύπου, ο κώδικας που παράγεται αποτελείται μόνο από την εντολή $PushVal$ ($Record$ []), που αποθηκεύει ένα κενό record στο σωρό και τοποθετεί τη διεύθυνσή του στην κορυφή της στοίβας αποτίμησης.

$\mathcal{C}[\![l,\ \mathbf{empty}]\!] = unit([PushVal\ (Record\ [])])$

Κώδικας Haskell:

```
compile (FV l AEmpty)          = return [PushVal (Record [])]
```

- Πρόσβαση σε πεδίο record : Η έκφραση πριν τον τελεστή "::" μεταγλωττίζεται και επιστρέφεται ο παραγόμενος κώδικας ακολουθούμενος από την εντολή *Eval* και την εντολή *RecAcc i*, που φορτώνει στη στοίβα αποτίμησης τη διεύθυνση του περιεχομένου του πεδίου "i".

  $\mathcal{C}[\![l,\ exp :: id]\!] = \mathcal{C}[\![exp]\!] * (\lambda c.unit(c +\!\!+ [Eval,\ RecAcc\ i]))$

  Κώδικας Haskell:

```
compile (FV l (ARecAccess exp i))
                     = do c <- compile exp
                          return (c ++ [Eval, RecAcc i])
```

- Επέκταση record : Μεταγλωττίζονται οι εκφράσεις exp_1 και exp_2, και επιστρέφεται ο κώδικας που προκύπτει μαζί με τις απαραίτητες εντολές *Eval* και την εντολή *RecExt i* που προσθέτει το νέο πεδίο στο υπάρχον record.

  $$\mathcal{C}[\![l,\ exp_1[id := exp_2]]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.\ unit(c_1 +\!\!+ [Eval] +\!\!+ c2 +\!\!+ [Eval,\ RecExt\ i])))$$

  Κώδικας Haskell:

```
compile (FV l (ARecExtend exp1 i exp2))
                  = do c1 <- compile exp1
                       c2 <- compile exp2
                       return (c1 ++ [Eval] ++ c2 ++ [Eval, RecExt i])
```

6.5 Απαλοιφή περιττών εντολών Eval

Μια αδυναμία των παραπάνω κανόνων μεταγλώττισης είναι η χρήση της εντολής *Eval* σε σημεία που δεν είναι απαραίτητη. Τέτοια σημεία είναι αυτά για τα οποία είναι βέβαιο ότι κατά την εκτέλεση του κώδικα στην κορυφή της στοίβας αποτίμησης θα βρίσκεται διεύθυνση που αντιστοιχεί σε ήδη αποτιμημένη έκφραση. Για παράδειγμα, αν μεταγλωττίσουμε την έκφραση Cantor

```
(lambda x:int exp.x+3) 5
```

ο παραγόμενος κώδικας G-machine αποτελείται από τα εξής 3 τμήματα κώδικα:

```
0: [PushFun 1 0,Eval,PushArg 2 0,Mkap,Eval]
1: [Push 0,Eval,Update,Push 0,Eval,PushVal (Num 3),Eval,Iadd,Eval,PopVar 1]
2: [PushVal (Num 5),Eval,PopVar 0]
```

Είναι φανερό, ότι για παράδειγμα μετά την εκτέλεση της "$PushVal$ (Num 3) στο τμήμα κώδικα "1", η διεύθυνση που θα βρίσκεται στην κορυφή της στοίβας θα περιέχει την τιμή "`Num 3`" και επομένως η εντολή $Eval$ που ακολουθεί είναι περιττή.

Για να εξαλείψουμε την παραπάνω αδυναμία, αφού ολοκληρωθεί η μεταγλώττιση περνάμε τον κώδικα που παράχθηκε από τη συνάρτηση `optimize`, που απαλείφει τις περιττές εντολές $Eval$. Περιττές είναι όσες εντολές $Eval$ βρίσκονται αμέσως μετά από εντολές $PushVal$, $PushFun$, $Assig$, $Dereff$ καθώς και μετά από όλες τις εντολές αριθμητικών πράξεων και σύγκρισης. Ο κώδικας Haskell για την `optimize` έχει ως εξής:

```
optimize                :: GMCodeStore -> GMCodeStore
optimize                = map scavenge

scavenge                             :: GMCodeSegment -> GMCodeSegment
scavenge []                          = []
scavenge (x : [])                    = [x]
scavenge (PushVal v   : Eval : xs)   = (PushVal v) : scavenge xs
scavenge (PushFun i n : Eval : xs)   = (PushFun i n) : scavenge xs
scavenge (Assig : Eval : xs)         = Assig : scavenge xs
scavenge (Dereff : Eval : xs)        = Dereff : scavenge xs
scavenge (Iadd : Eval : xs)          = Iadd : scavenge xs
scavenge (Isub : Eval : xs)          = Isub : scavenge xs
scavenge (Imul : Eval : xs)          = Imul : scavenge xs
scavenge (Idiv : Eval : xs)          = Idiv : scavenge xs
scavenge (Imod : Eval : xs)          = Imod : scavenge xs
scavenge (Ieq : Eval : xs)           = Ieq : scavenge xs
scavenge (Ine : Eval : xs)           = Ine : scavenge xs
scavenge (Igt : Eval : xs)           = Igt : scavenge xs
scavenge (Ige : Eval : xs)           = Ige : scavenge xs
scavenge (Ilo : Eval : xs)           = Ilo : scavenge xs
scavenge (Ile : Eval : xs)           = Ile : scavenge xs
scavenge (x : y : xs)                = x : scavenge (y:xs)
```

Μετά τη μικρή αυτή βελτιστοποίηση, ο κώδικας για την έκφραση Cantor που δώσαμε πιο πάνω παίρνει την παρακάτω μορφή:

```
0: [PushFun 1 0,PushArg 2 0,Mkap,Eval]
1: [Push 0,Eval,Update,Push 0,Eval,PushVal (Num 3),Iadd,PopVar 1]
2: [PushVal (Num 5),PopVar 0]
```

Κεφάλαιο 7

Συμπεράσματα

Μπορούμε να συνοψίσουμε τα συμπεράσματα της παρούσας εργασίας στα παρακάτω:

- Ξεκινώντας από τη γλώσσα Cantor, η πρωτοτυπία της γλώσσας έγκειται στο ότι αν και είναι μια μινιμαλιστική γλώσσα, περιέχει στοιχεία που πολλές φορές δεν συναντώνται ακόμη και σε μεγαλύτερες γλώσσες, όπως για παράδειγμα η συνύπαρξη κλήσης με τιμή και κλήσης με όνομα. Επίσης η δυνατότητα ένα record να έχει πεδίο τύπου συνάρτησης παρέχει στοιχειώδεις δυνατότητες αντικειμενοστραφούς προγραμματισμού, αφού ένα τέτοιο record θα είναι ουσιαστικά ένα αντικείμενο και η συνάρτηση μια μέθοδος–μέλος του αντικειμένου. Τέλος, η ορθογωνιότητα των χαρακτηριστικών της Cantor διευκόλυνε σημαντικά την υλοποίηση διερμηνέα και μεταγλωττιστή.

- Ένα άλλο συμπέρασμα που προκύπτει, είναι ότι η χρήση των μονάδων στη δυναμική σημασιολογία απλοποιεί την κατασκευή ενός διερμηνέα. Όπως είδαμε στο κεφάλαιο 4, ο διερμηνέας προέκυψε με εντελώς ευθύ και φυσικό τρόπο από τη δυναμική σημασιολογία, χάρη στη χρήση των μονάδων.

- Η κατασκευή του διερμηνέα (αλλά και του ελεγκτή τύπων) διευκολύνθηκε περαιτέρω από τη χρήση μιας συναρτησιακής γλώσσας όπως η Haskell. Στα κεφάλαια 3 και 4 είδαμε πόσο εύκολη ήταν η μεταφορά της σημασιολογίας τύπων και της δυναμικής σημασιολογίας αντίστοιχα σε κώδικα. Επίσης το σύστημα τύπων της Haskell έκανε δυνατό τον έγκαιρο εντοπισμό πολλών προγραμματιστικών λαθών.

- Η χρήση των μονάδων στον κώδικα έκανε δυνατή την τμηματική υλοποίηση του ελεγκτή τύπων, του διερμηνέα και του μεταγλωττιστή σε κώδικα G-Machine. Κατά την ανάπτυξη του κώδικα ήταν φανερή η μεγάλη διευκόλυνση που προσέφερε η χρήση των μονάδων τόσο στην σταδιακή προσθήκη χαρακτηριστικών όσο και στις διάφορες αλλαγές που χρειάστηκε να γίνουν.

- Τέλος, ένα ενδιαφέρον στοιχείο είναι οι τυχόν αναλογίες μεταξύ του διερμηνέα και του μεταγλωττιστή. Αν ξεκινήσουμε από τις απλούστερες εκφράσεις, π.χ. τις ακέραιες σταθερές:

  $$\mathcal{E}[\![n]\!] = unit(Num\ n)$$
  $$\mathcal{C}[\![l,\ n]\!] = unit([PushVal\ (Num\ n)])$$

  βλέπουμε ότι οι διαφορές μεταξύ τους είναι το στάδιο εύρεσης των ελεύθερων μεταβλητών που έχει προηγηθεί στο μεταγλωττιστή και η προσθήκη μιας εντολής PushVal. Αν πάμε στα αναγνωριστικά:

  $$\mathcal{E}[\![id]\!] = rdEnv * (\lambda\rho.\rho\ id)$$
  $$\mathcal{C}[\![l,\ id]\!] = rdEnv * (\lambda\rho.unit([Push\ (\rho\ id)]))$$

  παρατηρούμε ότι η διαφορά έγκειται στην προσθήκη μιας εντολής Push. Στην περίπτωση των δυαδικών εκφράσεων, π.χ. της πρόσθεσης:

$$
\begin{aligned}
&\mathcal{E}[\![e_1 + e_2]\!] = \mathcal{E}[\![e_1]\!] * (\lambda u.\mathcal{E}[\![e_2]\!] * (\lambda v.unit(u+v)))\\
&\mathcal{C}[\![l,\ exp_1 + exp_2]\!] = \mathcal{C}[\![exp_1]\!] * (\lambda c_1.\mathcal{C}[\![exp_2]\!] * (\lambda c_2.\\
&\qquad\qquad\qquad\quad unit(c_1 +\!\!+ [Eval] +\!\!+ c_2 +\!\!+ [Eval,\ Iadd])))
\end{aligned}
$$

βλέπουμε ότι η διαφορά εντοπίζεται στην προσθήκη εντολών Eval στον κώδικα που αντιστοιχεί στο κάθε τελούμενο και στην αντικατάσταση της συμβατικής πρόσθεσης από την εντολή Iadd. Σε πολυπλοκότερες περιπτώσεις όμως, όπως είναι οι εκφράσεις λάμβδα, οι διαφορές είναι αρκετά μεγαλύτερες:

$$
\begin{aligned}
&\mathcal{E}[\![\textbf{lambda}\ !\ x : type.e]\!] = rdEnv * (\lambda\rho.unit(\lambda arg.inEnv\ \rho[arg/x]\ \mathcal{E}[\![e]\!]))\\
&\mathcal{C}[\![(fv_1, ..., fv_n),\ \textbf{lambda}\ !\ x : type.exp]\!] = rdEnv * (\lambda\rho.\\
&\qquad\qquad\qquad\quad inEnv\ [x \to 0, fv_1 \to 1, ..., fv_n \to n]\ \mathcal{C}[\![exp]\!] * (\lambda f.\\
&\qquad\qquad\qquad\quad newCodeSeg\ (f +\!\!+ [Eval,\ PopVar\ (n+1)]) * (\lambda s.\\
&\qquad\qquad\qquad\quad unit([Push\ (\rho\ fv_n), ..., Push\ (\rho\ fv_1),\ PushFun\ s\ n]))))
\end{aligned}
$$

Θα μπορούσε να αποτελέσει αντικείμενο μελλοντικής έρευνας το κατά πόσο είναι δυνατό οι διαφορές αυτές να γεφυρωθούν ώστε ο διερμηνέας και ο μεταγλωττιστής να προκύπτουν από ένα ενιαίο «πρότυπο» (template).

Βιβλιογραφία

[1] J. C. Reynolds, "The essence of Algol," in *Algorithmic Languages* (J. W. de Bakker and J. C. van Vliet, eds.), (Amsterdam), pp. 345–372, North-Holland, 1981.

[2] P. Hudak, J. Peterson, and J. Fasel, "A gentle introduction to Haskell 98," Oct. 1999.

[3] S. L. Peyton Jones and D. R. Lester, "Implementing functional languages: A tutorial," 1991.

[4] T. Johnsson, *Compiling Lazy Functional Languages.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[5] L. Augustsson, *Compiling Lazy Functional Languages, Part II.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[6] S. Peyton Jones, ed., *Haskell 98 Language and Libraries, The Revised Report.* Sept. 2002.

[7] E. Moggi, "An abstract view of programming languages," Tech. Rep. ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, 1990.

[8] P. Wadler, "The essence of functional programming," in *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Albequerque, New Mexico), pp. 1–14, 1992.

[9] S. Liang and P. Hudak, "Modular denotational semantics for compiler construction," in *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, vol. 1058, pp. 219–234, Springer-Verlag, 1996.

[10] S. L. Peyton Jones, "Implementing lazy functional languages on stock hardware: The spineless tagless g-machine," *Journal of Functional Programming*, vol. 2, no. 2, pp. 127–202, 1992.

[11] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995* (ACM, ed.), (New York, NY, USA), pp. 333–343, ACM Press, 1995.

[12] P. D. Mosses, "Denotational semantics," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. B, ch. 11, pp. 577–631, Elsevier Science Publishers B.V., 1990.

[13] N. Papaspyrou and V. Vescoukis, "Facilitating the definition of programming languages by using parametric context-free grammars," in *Proceedings of the 7th Hellenic Conference on Informatics*, vol. II, (Ioannina, Greece), pp. 91–98, Aug. 1999.

[14] N. Papaspyrou, *A formal semantics for the C programming language.* PhD thesis, National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, Feb. 1998.

[15] R. Bird, *Introduction to Functional Programming using Haskell, second edition.* London: Printice Hall Europe, 1998.

[16] Ν. Σ. Παπασπύρου and Ε. Σ. Σκορδαλάκης, *Μεταγλωττιστές.* Αθήνα: Εκδόσεις Συμμετρία, 2002.