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

Μετασχηματισμός Συναρτήσεων Haskell σε Μοναδιαία Μορφή

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΕΡΖΙΔΑΚΗ ΑΛΕΞΑΝΔΡΑ

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

Αθήνα, Σεπτέμβριος 2019

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

Μετασχηματισμός Συναρτήσεων Haskell σε Μοναδιαία Μορφή

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΕΡΖΙΔΑΚΗ ΑΛΕΞΑΝΔΡΑ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20η Σεπτεμβρίου 2019.

Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Αριστείδης Παγουρτζής Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2019

Τερζιδάκη Αλεξάνδρα

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

Περίληψη

Οι αγνές συναρτησιακές γλώσσες προγραμματισμού, σε αντίθεση με διάφορες άλλες κατηγορίες γλωσσών (προστακτικές, λογικές, κτλ.), χρησιμοποιούν τις συναρτήσεις όπως ορίζονται από το μαθηματικό πρότυπο. Έτσι, μία συνάρτηση για μία δεδομένη είσοδο θα δώσει μία και μόνο μία έξοδο, όσες φορές και αν καλεστεί. Το γεγονός αυτό, παρά τα πλεονεκτήματα που μπορεί να έχει, δημιουργεί κάποια προβλήματα που σχετίζονται με τον χειρισμό μη αγνών (impure) συναρτησιακών χαρακτηριστικών, όπως τα side effects, η αποτυχία τερματισμού, ο μη ντετερμινισμός, κ.α.

Στην Haskell, το εργαλείο που είναι υπεύθυνο για την ενσωμάτωση μη αγνών συναρτησιακά χαρακτηριστικά ονομάζεται monads και η λογική του στηρίζεται στον κλάδο των μαθηματικών που καλείται category theory. Ο ορισμός των monads περιέχει τις συναρτήσεις return και bind ($>>=$) που βοηθούν στην διαχείριση τιμών που δεν είναι αγνές, τις οποίες συνηθίζεται να καλούμε υπολογισμούς, και συμβολίζονται με *M a*. Αν και τα monads είναι ιδιαίτερα επιτακτικά στην πράξη, η ιδιόμορφη φύση τους αποθαρρύνει τους προγραμματιστές να τα κατανοήσουν και κατά συνέπεια να τα χρησιμοποιήσουν με κατάλληλο τρόπο.

Ένα σημαντικό εμπόδιο που τίθεται, σχετίζεται με το γεγονός ότι δεν ορίζεται μαθηματικά η καθολική πράξη της σύνθεσης των monads. Ωστόσο, στον πραγματικό κόσμο είναι συχνά απαραίτητη η συνδιαλλαγή με πολλάπλα και, πιθανώς, διαφορετικού σκοπού προστακτικά γνωρίσματα. Έτσι, το επιστημονικό ενδιαφέρον στράφηκε στην ανάπτυξη πρακτικών μοντέλων που καταφέρνουν να επιτύχουν την επιθυμητή ”μείξη” των impure στοχείων. Δύο τέτοιες μέθοδοι είναι οι monad Transformers και τα Extensible Effects.

Στο πλαίσιο της παρούσας διπλωματικής επιχειρήσαμε να επεκτείνουμε τον τρόπο με τον οποίο οι προγραμματιστές της Haskell αντιμετωπίζουν τα monads. Το εργαλείο που αναπτύξαμε διευκολύνει τη χρήση των monads και παρέχει επιπλέον δυνατότητες σχετικά με την ταυτόχρονη επεξεργασία πολλαπλών και ετερόκλητων impure στοιχείων. Για τους σκοπούς αυτούς, επιλέξαμε να πειραματιστούμε με ένα υποσύνολο της Haskell, τη γλώσσα RouReg. Με τη βοήθεια του framework των Extensible Effects επιδιώξαμε τη μετατροπή των συναρτήσεων των επικείμενων προγραμμάτων σε monadic form. Με αυτόν τον τρόπο, επιτύχαμε όλες οι συναρτήσεις του μετασχηματισμένου κώδικα, να έχουν τη μορφή *Eff r (a → Eff r (...))*, δηλαδή να βρίσκονται εντός του *Eff r* monad, μέσω του οποίου μπορούν να συνδιαστούν ταυτόχρονα διαφορετικά effects, ως ένωση στο σύνολο r. Ένας τέτοιος μετασχηματισμός, πριν τη διαδικαδία της μεταγλώττισης, επιτρέπει στον ανεπεξέργαστο κώδικα να περιέχει monadic σύνταξη που, μέχρι πρότινος, δεν ήταν αποδεκτή.

Λέξεις κλειδιά

Αγνά Συναρτησιακές Γλώσσες Προγραμματισμού, Haskell, Μονάδες, Σύνθεση monads, Extensible Effects, Monad Transformers, Μετασχηματισμός Κώδικα.

Abstract

In contrast to Procedural Languages, Functional Programming Languages treats functions as their mathematical definition does. Thus, calling a function with the same value for an argument always produces the same result. Despite all the advantages of such a property, there are a lot of difficulties related with the handling of impure functional features, like side effects, exceptions and non-determinism.

Monads, based on Category Theory, constructed in Haskell as a convenient framework for simulating effects found in other languages. Non functional values that we are usually called computations, are manipulated by monads' elementary functions $return$ and $bind$ ($>>=$). If a is a pure value, symbol *M a* stands for the computation which wraps value a. Although monads are an essential feature of Haskell for building well-developed software, it is generally accepted that it comprises a hard to use object.

An important barrier which prevents monad from being a wieldy tool is that they prove difficult to compose. In the real world, however, the combination of multiple and possibly diverse imperative features is often deemed essential. Thus, the scientific community has attempted to investigate alternative methods that have succeeded in overcoming said limitations. Monad Transformers and Extensible Effects are two such approaches.

In this thesis we attempt to enrich the way in which Haskell's programmers are able to utilize monads. The framework we have developed facilitates the use of monads and provides additional options regarding the concurrent processing of multiple and diverse impure elements. For this purpose, we have chosen to experiment on a subset of Haskell syntax, the RouReg language. Using Extensible Effects we attempted, and succeeded in, the transformation of RouReg programs' functions into monadic form. Consequently, the form *Eff r (a → Eff r (...))*, was achieved, in which, through the *Eff r* monad the simultaneous combination of different effects, as a union on the r set, is made possible. Through this model, the previously limited monadic syntax, is now enriched to accommodate the programmers' needs.

Key words

Purely Functional Programming Languages, Haskell, Monads, Monad Composition, Monad Transformers, Extensible-Effects, Code Transformation.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Νικόλαο Παπασπύρου για την καθοδήγηση και την υποστήριξη που μου προσέφερε. Επίσης, θα ήθελα να ευχαριστήσω τους γονείς μου και τις μοναδικές αδερφές μου για την αγάπη που μου δίνουν. Τέλος, θέλω να πω ένα τεράστιο ευχαριστώ στα φιλαράκια μου και στα άλλα υπέροχα πλάσματα που είχα την τύχη να συναντήσω στη ζωή μου. Ιδιαίτερα, θέλω να ευχαριστήσω τη Ντίνα, τον Αντώνη, τη Μάνη, τον Θανάση, τον Τζο, την Ανδριάνα και τον εξαίσιο επιστήμονα Γεώργιο Πετράκη.

Η εργασία αυτή είναι αφιερωμένη στη μνήμη της γιαγιάς μου, Ιουλίας.

Τερζιδάκη Αλεξάνδρα,

Αθήνα, 20η Σεπτεμβρίου 2019

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-6-19, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Σεπτέμβριος 2019.

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

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Περιεχόμενα 11

Κατάλογος πινάκων 13

Κατάλογος σχημάτων 15

1. Εισαγωγή 17
1.1 Αντικείμενο της Εργασίας 17
1.2 Διάρθρωση Εργασίας 17

2. Θεωρητική Εισαγωγή στις Συναρτησιακές Γλώσσες 19
2.1 Κίνητρα Ανάπτυξης των Συναρτησιακών Γλωσσών 19
2.2 Χαρακτηριστικά των Συναρτησιακών Γλωσσών. 21
2.2.1 Referential Transparency 21
2.2.2 Higher Order Functions 21
2.2.3 Λάμδα-Λογισμός 22
2.2.4 Lazy evaluation 22
2.3 Επιχειρηματολογία για τον Συναρτησιακό Προγραμματισμό 23
2.3.1 Πλεονεκτήματα 23
2.3.2 Μειονεκτήματα 26
2.4 Η Γλώσσα Haskell 27

3. Monads 29
3.1 Θεωρητικό Υπόβαθρο 29
3.1.1 Οι συναρτήσεις *Return* και *Bind* 29
3.1.2 Do notation 30
3.2 Maybe και State Monad 31
3.3 Input\Output 34
3.3.1 Εφαρμογή των monads για αλληλεπίδραση με εξωτερικό κόσμο 34
3.3.2 Συνδιάζοντας I\O λειτουργίες 36
3.4 Σχολιασμός 36

4. Συνδιασμός των effects 39
4.1 Σύνθεση monads 39
4.2 Monad Transformers 40
4.2.1 Ορισμός των Monad Transformers 40
4.2.2 State Monad Transformer 42

4.2.3 Η συνάρτηση *lift* . . . 42
4.2.4 Αδυναμίες των Monad Transformers . . . 44
4.3 Extensible Effects . . . 46
4.3.1 Κεντρική ιδέα . . . 46
4.3.2 Περιήγηση στο Framework των Extensible Effects . . . 47
4.3.3 Lift handler . . . 50
4.3.4 Σύγκριση με Monad Transformers . . . 50

5. Υλοποίηση του Μετασχηματισμού . . . 53
5.1 Εισαγωγή . . . 53
5.1.1 Σχεδιαστικές αποφάσεις . . . 53
5.1.2 Περιγραφή ροής εκτέλεσης . . . 54
5.2 Η Γλώσσα RouReg . . . 54
5.3 Λεκτική και Συντακτική Ανάλυση . . . 55
5.4 Το Transformation . . . 59
5.4.1 Μετασχηματισμός TypeSignature . . . 59
5.4.2 Μετασχηματισμός Εκφράσεων . . . 61
5.5 Παραδείγμα Transformation . . . 63

6. Συμπεράσματα . . . 69
6.1 Συνεισφορά . . . 69
6.2 Σχετικές Εργασίες - Μελλοντική Έρευνα . . . 69

Βιβλιογραφία . . . 71

Παράρτημα 73

A. Predefined Συναρτήσεις & Δομή του Μετασχηματισμένου Κώδικα . . . 73

Κατάλογος πινάκων

5.1 Η γραμματική της RouReg. 56
5.2 Tokens και Regular Expressions για τον Lexer της RouReg. 57
5.3 Παραδείγματα κανόνων παραγωγής 58
5.4 Κανόνες Μετασχηματισμού Τύπου 60
5.5 Συναρτήσεις της οικογένειας MConvert, τάξεως ≤ 3 61
5.6 Πίνακας Μετασχηματισμού Εκφράσεων 62
5.7 Πίνακας Μετασχηματισμού Εκφράσεων για Pure και Impure Όρους 63

Κατάλογος σχημάτων

3.1 Σχηματική Αναπαράσταση του τύπου IO 34

4.1 Αναπαράσταση στοίβας για ”StateT s Error” monad 43
4.2 Το interface της βιβλιοθήκης των Extensible Effects 48

5.1 Διάγραμμα Ροής για το interface. 54
5.2 Παράδειγμα αρχείου εισόδου 59
5.3 Εκτέλεση αρχείου εισόδου 5.2 64
5.4 Τροποποιημένος κώδικας για το παράδειγμα αρχείου εισόδου του σχήματος 5.2 . . . 65
5.5 Εκτέλεση αρχείου εισόδου 5.4 66
5.6 Σύνθετο παράδειγμα αρχείου εισόδου 67

Κεφάλαιο 1

Εισαγωγή

1.1 Αντικείμενο της Εργασίας

Τα monads αποτελούν το εργαλείο που έχει η Haskell, ως αγνά συναρτησιακή γλώσσα, για να ενσωματώνει με προστακτικά χαρακτηριστικά, όπως η ύπαρξη κατάστασης, οι εξαιρέσεις, η είσοδος/έξοδος κ.τ.λ. Πάρ'οτι, όμως τα monads είναι πολύ συχνά βοηθητικά και άλλοτε απαραίτητα - και αναντικατάστατα- στην ανάπτυξη λογισμικού, η ιδιόμορφη φύση τους τα μετατρέπει σε ένα αρκετά "στρυφνό" και δύσχρηστο αντικείμενο για τούς προγραμματιστές του συναρτησιακού κόσμου. Επιπλέον δυσκολίες, συντρέχουν και από το γεγονός πως μαθηματικά δεν ορίζεται η σύνθεση τους, και συνεπώς η, συχνά, επιθυμητή "μείξη" διαφορετικών impure γνωρισμάτων, καθίσταται επίπονη. Η αναγκαιότητα της διερεύνησης πιστικών λύσεων στη διευκόλυνση της χρήσης των monads αποτέλεσε το βασικότερο κίνητρο για την εκπόνηση της παρούσας διπλωματικής. Σκοπός μας ήταν να κατασκευάσουμε ένα μοντέλο προγραμματισμού στο οποίο η έννοια των monads θα ακολουθεί το συμβατικό πρότυπο θεώρησης των αντικειμένων στη Haskell και ως εκ τούτου θα εξυπηρετείται η αντιμετώπιση τους.

Για την υλοποίηση της παραπάνω ιδέας ήταν απαραίτητη η εισαγωγή ενός επιπλέον επιπέδου αυτόματων μετασχηματισμών (desugaring) πριν τη μεταγλώττιση του πηγαίου κώδικα. Με στόχο την αποδείξη της θεώρησης μας, εργαστήκαμε με ένα υποσύνολο της Haskell που ονομάσαμε RouReg. Για τη γλώσσα αυτή δημιουργήσαμε κατάλληλο λεκτικό και συντακτικό αναλυτή με την βοήθεια των οποίων καταφέραμε να παράγουμε το Αφηρημένο Συντακτικό Δέντρο (AST) του προγράμματος προς επεξεργασία. Ο αυτόματος μετασχηματισμός του AST που επιτύχαμε, έγινε με χρήση του framework των Extensible Effects και θεμελίωσε την ικανότητα συγγραφής προγραμμάτων που περιέχουν monads, με απλούστερα, μη επιτρεπτά μέχρι πρότινος, μοτίβα.

1.2 Διάρθρωση Εργασίας

Το υπόλοιπο της εργασίας διαμορφώνεται ως εξής:

Κεφάλαιο 2:

Παρέχουμε μία θεωρητική εισαγωγή στις συναρτησιακές γλώσσες προγραμματισμού. Συγκεκριμένα, έπειτα από μία σύντομη ιστορική αναδρομή, εξετάζουμε τα κυριότερα χαρακτηριστικά του συναρτησιακού μοντέλου για τα οποία και επιχειρηματολογούμε. Τέλος, σκιαγραφούμε με περισσότερη λεπτομέρεια κάποια ειδικά γνωρίσματα της Haskell που θα μας απασχολήσουν στη συνέχεια.

Κεφάλαιο 3:

Συζητάμε για τον τρόπο με τον οποίο η Haskell καταφέρνει να αλληλεπιδρά με μη αγνά συναρτησιακά στοιχεία, διατηρώντας παράλληλα την αγνά συναρτησιακή της φύση. Αφού κάνουμε μία εκτενή ανάλυση στις δυνατότητες που προσφέρουν τα monads, επιχειρούμε να αποδώσουμε τη λειτουργική σημασία που έχουν, μέσω παραδειγμάτων.

Κεφάλαιο 4:

Παρουσιάζουμε τις τεχνικές με τις οποίες μπορεί να γίνει εφικτή η ταυτόχρονη συνδιαλλαγή με ένα σύνολο από monads, που, πιθανώς, είναι υπεύθυνα για διαφορετικά impure γνωρίσματα. Αρχικά, αναφερόμαστε στους αποτρεπτικούς παράγοντες της σύνθεσης των monads, ενώ στη συνέχεια εξετάζουμε και επιχειρηματολογούμε για τις εναλλακτικές μεθόδους των Monad Transformers και Extensible Effects.

Κεφάλαιο 5:

Περιγράφουμε την υλοποίηση της εργασίας. Σε πρώτο χρόνο, αναφέρουμε τις σχεδιατικές αποφάσεις που λήφθησαν και παρουσιάζουμε το διάγραμμα ροής της εφαρμογής. Μετέπειτα, περιγράφουμε με λεπτομέρειες τα στάδια της υλοποίησης, ξεκινώντας από την ίδια τη γλώσσα RouReg, και συνεχίζοντας με το Lexer, τον Parser, και το Transformation. Στο τέλος του κεφαλαίου, δίνονται παραδείγματα προς απόδειξη της χρησιμότητας του εργαλείου.

Κεφάλαιο 2

Θεωρητική Εισαγωγή στις Συναρτησιακές Γλώσσες

Σε αυτό το κεφάλαιο θα κάνουμε μία θεωρητική εισαγωγή στις συναρτησιακές γλώσσες προγραμματισμού. Αρχικά, θα εξετάσουμε τα κίνητα ανάπτυξης των γλωσσών αυτών και στη συνέχεια θα περιγράψουμε κάποια από τα κυριότερα χαρακτηριστικά τους. Έπειτα, θα θέσουμε τα πλεονεκτήματα και μειονεκτήματα που μπορεί να έχει το συναρτησιακό μοντέλο προγραμματισμού, ενώ στο τέλος θα περιγράψουμε πιο στοχευμένα κάποια χαρακτηριστικά της γλώσσας Haskell. Όλα τα παραδείγμα που τίθενται είναι γραμμένα σε Haskell, η οποία πέρα από μία χαρακτηριστικά συναρτησιακή γλώσσα, αποτελεί και την γλώσσα ενδιαφέροντος της παρούσας διπλωματικής.

2.1 Κίνητρα Ανάπτυξης των Συναρτησιακών Γλωσσών

Η διάλεξη των Turing Award που πραγματοποιήθηκε το έτος 1978, δώθηκε από τον J.W.Backus, και πραγματευόταν το κατά πόσο ο Προγραμματισμός μπορεί να απελευθερωθεί από το von Neumann Style. Η συζήτηση γύρω από αυτό το επίδικο, νηματοδοτήθηκε από τα κενά και τα προβλήμματα που αντιμετωπίζονταν στις εως τότε πρακτικές προγραμματισμού, και αποτέλεσε την κυριότερη επιχειρηματολογία για την έρευνα γύρω από τις συναρτησιακές γλώσσες.[Cunn14]

Το σχετικό άρθρο που δημοσιεύθηκε, με τίτλο "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs"[Back78], αναφέρει χαρακτηριστικά πως για να κατανοήσουμε τα προβλήματα των συμβατικών γλωσσών προγραμματισμού, είναι απαραίτητο να εξετάσουμε την αρχιτεκτονική von Neumann, στην οποία στηρίζονται. Στην πιο απλή του μορφή, ένας υπολογιστής von Neumann περιέχει τρία μέρη: μία κεντρική μονάδα επεξεργασίας (CPU), έναν αποθηκευτικό χώρο (store) και έναν συνδετικό δίαυλο - ονόματι von Neumann bottleneck - που μπορεί να μεταφέρει μία μόνο λέξη μεταξύ CPU και store (και να στείλει μία διεύθυνση στη μνήμη).

Η αποστολή ενός προγράμματος είναι να τροποποιήσει τα περιεχόμενα της μνήμης με έναν συγκεκριμένο τρόπο, μετακινώντας τιμές και διευθύνσεις μεταξύ CPU και store εξωλοκλήρου, μέσω του bottleneck - εξού και το όνομα του. Όπως είναι ευνόητο, ένας μεγάλος όγκος της κίνησης που δημιουργείται, αποτελείται από δεδομένα τα οποία δεν περιέχουν χρήσιμη πληροφορία, αλλά απλώς ονόματα δεδομένων ή λειτουργίες και data που χρησιμεύουν στο να υπολογιστούν αυτά τα ονόματα.

Ο Backus σημείωσε επίσης, πως οι συμβατικές[1] γλώσσες προγραμματισμού χρησιμοποιούν μεταβλητές (variables) για να μιμηθούν τις θέσεις μνήμης του υπολογιστή, εντολές ελέγχου (control statements) για να παραστήσουν τα άλματα (jumps) και εντολές ανάθεσης (assignment) που αντιστοιχούνται στις αριθμητικές εντολές και εντολές αλληλεπίδρασης με τη μνήμη (fetch και store). Οι εντολές ανάθεσης, είναι το von Neumann bottleneck των γλωσσών προγραμματισμού, δεδομένου ότι ακολουθούν τη λογική του word-at-a-time με τον ίδιο τρόπο που το κάνει και ένας υπολογιστής της ίδιας αρχιτεκτονικής.

Ας εξετάσουμε ένα τυπικό πρόγραμμα: το περιεχόμενο του αποτελείται από μία σειρά εντολών ανάθεσης που περιέχουν κάποιες δεικτοδοτούμενες μεταβλητές, ενώ κάθε εντολή ανάθεσης παράγει ένα αποτέλεσμα μίας λέξης (one-word-result). Η εκτέλεση του προγράμματος προκαλεί την -ίσως-

[1] Συμβατικές γλώσσες προγραμματισμού, ονόμασε ο Backus τις γλώσσες που ακολουθούν τη λογική της αρχιτεκτονική von Neumann, γνωστές και ως προστακτικές (imperative) γλώσσες.

πολλάπλή εκτέλεση των εντολών αυτών, ενώ μεταβάλλει την τιμή των δεικτοδοτούμενων μεταβλητών (one-word-at-a-time) προκειμένου να φέρει το επιθυμητό αποτέλεσμα στη θέση μνήμης. Επομένως, ο προγραμματιστής ενδιαφέρεται για τη ροή των λέξεων εντός του bottleneck, καθώς σχεδιάζει το πρόγραμμα του με βάση την αλληλουχία των αλλάγών της μνήμης.

Όπως προκύπτει, το προστακτικό μοντέλο είναι πολύ κοντά σε αυτό που πράγματι συμβαίνει στον υπολογιστή όταν εκτελείται ένα πρόγραμμα: διάφορες οδηγίες που επεξεργάζεται ο υπολογιστής έχουν ως αποτέλεσμα την αλλαγή των περιεχομένων της μνήμης του. Ως εκ τούτου, οι γλώσσες von Neumann διαχωρίζουν το πεδίο ενδιαφέροντος σε δύο ξεχωριστά τμήματα. Το πρώτο, αφορά το κομμάτι των statements - commands -, που δεν αποτελούν τίποτα άλλο παρά requests στη CPU για την τροποποιήση κάποιας θέσης στη μνήμη. Το δεύτερο, είναι το σημείο στο οποίο λαμβάνουν χώρα οι περισσότεροι χρήσιμοι υπολογισμοί, και καταπιάνεται με το δεξί μέλος μιάς εντολής ανάθεσης που αποτελείται από μία έκφραση, σαφώς ορισμένη, με αλγεβρικές ιδιότητες που μπορούν δυνητικά να καταστραφούν από κάποια παρενέργεια - side-effect. Κατά τον υπολογισμό μιάς έκφρασης είναι πιθανό να κληθεί κάποια υπορουτίνα ή διαδικασία (procedure), η οποία θα επιδρά στη μνήμη μέσω κάποιας ανάθεσης, και θα επηρεάζει το αποτέλεσμα. Ακόμη και εκφράσεις όπως η:

$$x = x$$

ενδεχομένως να αποτιμόνται σε False. Σαν παράδειγμα μπορούμε να σκεφτούμε τις συναρτήσεις *random* ή *readinteger*.

Από τα παραπάνω συμπεραίνουμε ότι στις προστακτικές γλώσσες η σειρά εκτέλεσης των εντολών είναι άρρηκτα συνδεδεμένη με τη σημασιολογία του προγράμματος - κάτι που προκαλεί προβλήματα αποδοτικότητας, καθώς δυσχαιρένονται τεχνικές reordering ή parallelization. Επιπλέον, το γεγονός ότι δεν ισχύει η ανακλαστική (reflexive) ιδιότητα της ισότητας ($x = x$) σημαίνει πως δεν μπορούμε να χρησιμοποιούμε τους κλασσικούς νόμους των συμβατικών μαθηματικών για την απόδειξη προγραμμάτων γραμμένων σε γλώσσες von Neumann. Στο σημείο αυτό είναι που εισάγονται οι συναρτησιακές γλώσσες[2] ως μία πολλά υποσχόμενη αντιπρόταση.[Davi92]

Στην περίπτωση του συναρτησιακού προγραμματισμού, το παραδοσιακό μοντέλο των εντολών και παρενεργειών εγκαταλείπεται χάριν μίας υψηλού επιπέδου αφαίρεσης, αυτήν της μαθηματικής συνάρτησης. Ο υπολογισμός πλέον περιγράφεται σα μία συνάρτηση που συσχετίζει τις παραμέτρους της (είσοδος) με το αποτέλεσμά της (έξοδος). Η έννοια της κατάστασης προγράμματος και της παρενέργειας δεν υπάρχει σε αυτό το μοντέλο υπολογισμού. [Κα]

Ας δούμε τώρα τους βασικούς λόγους για τους οποίους το συναρτησιακό μοντέλο απέρριψε τις έννοιες της κατάστασης και της παρενέργειας (impure χαρακτηριστικά). Το διαρκές ζητούμενο στον προγραμματισμό είναι η παραγωγή ορθού και εύκολα συντηρήσιμου κώδικα. Για την επίτευξη αυτού του στόχου, είναι σημαντικό ο κώδικας να είναι:

- Καλογραμμένος και ευανάγνωστος.
- Εύκολος στη χρήση από μαθηματικές θεωρίες ορθότητας.
- Διαχωρισμένος σε όσο το δυνατόν πιο ανεξάρτητα μεταξύ τους κομμάτια, που μπορούν να δημιουργηθούν, να ελεγχθούν και να αποσφαλματωθούν ανεξάρτητα το ένα από το άλλο. Η σημαντική αυτή ιδέα ονομάζεται τμηματοποίηση (modularization) του κώδικα.

Ο συναρτησιακός προγραμματισμός, χάρη στην απλότητα του μαθηματικού του μοντέλου, επιτυγχάνει τους δύο πρώτους στόχους αρκετά καλύτερα από τον προστακτικό προγραμματισμό. Όσον αφορά τον τρίτο στόχο, στο συναρτησιακό προγραμματισμό έχουν αναπτυχθεί ιδιώματα που προσφέρουν δυνατότητες τμηματοποίησης που είναι ανέφικτες με τους κλασσικούς μηχανισμούς του προστακτικού προγραμματισμού (τμήματα (modules), πακέτα (packages), κλάσεις (classes) κτλ.). Τέτοια ιδιώματα είναι οι συναρτήσεις ως τιμές πρώτης κατηγορίας (functions as first-class values) (και κατά συνέπεια οι συναρτήσεις υψηλότερης τάξης (higher order functions)) και η οκνηρή αποτίμηση (lazy evaluation), που θα εξετάσουμε λεπτομερώς στη συνέχεια.[Hugh89]

[2] Συναρτησιακές γλώσσες σαν τη Lisp, έχουν κάνει την εμφάνιση τους από το 1950. Όμως η διάλεξη του Backus θεωρείται ότι θεμελίωσε την αξίας τους και κινητοποίησε την περεταίρω έρευνα γύρω από αυτές.

2.2 Χαρακτηριστικά των Συναρτησιακών Γλωσσών.

Όπως αναφέρθηκε, το βασικότερο χαρακτηριστικό των συναρτησιακών γλωσσών και η διαφοροποίηση τους από τις γλώσσες von Neumann είναι ότι δεν περιέχουν statements - εντολές ανάθεσης που προκαλούν αλλαγές στη μνήμη. Στην ενότητα 2.4 θα δούμε αναλυτικά τη γλώσσα Haskell που αποτελεία ένα αντιπροσωπευτικό παράδειγμα command-free γλώσσας. Προς το παρόν πάμε να εξηγήσουμε κάποιες σημαντικές ιδιότητες των functional (ή applicative) γλωσσών.

2.2.1 Referential Transparency

Έστω ότι έχουμε την έκφραση $f + g$ και θέλουμε να υπολογίσουμε τις υποεκφράσεις f και g (που μπορεί να αποτελούν αρκετά σύνθετους υπολογισμούς, συμπεριλαμβανομένων μεταξύ άλλων και πολλών κλήσεων συναρτήσεων - function calls) Ένας compiler που κάνει βελτιστοποιήσεις, μπορεί να αποφασίσει να υπολογίσει πρώτα την f και μετά την g ή αντίστροφα ή ακόμη, αν υπάρχει κατάλληλο hardware να τις υπολογίσει παράλληλα την ίδια στιγμή. Αν ο υπολογισμός της μίας υποέκφρασης επηρέαζε το αποτέλεσμα της άλλης, μέσω κάποιου side effect, θα υπήρχε πρόβλημα καθώς το αποτέλεσμα θα είχε σχέση με την αλληλουχία (time-order) της διασικασίας αποτίμησης των υποεκφράσεων. Επομένως, θα βρισκόμασταν σε μία κατάσταση που ο υπολογισμός μιας έκφρασης δεν θα εξαρτώνταν μόνο από τα συστατικά της (δηλαδή τις υποεκφράσεις της), αλλά και από άλλους παράγοντες όπως η σειρά της αποτίμησης τους.

Αν παρόλα αυτά είχαμε ένα σύστημα το οποίο θα είχε σχεδιαστεί κάτα τρόπο που θα απαγόρευε ολωσδιόλου τα side effects, τότε η αλληλουχία της επεξεργασίας και αποτίμησης των υποεκφράσεων θα μπορούσε να πραγματοποιηθεί με οποιαδήποτε σειρά ή και παράλληλα. Μία τέτοια γλώσσα, στην οποία η τιμή της αποτιμήσης μιας έκφρασης εξαρτάται μονάχα απο τις τιμές των καλώς ορισμένων υποεκφράσεων της ονομάζεται referencial transparent - οι applicative γλώσσες ανήκουν σε αυτή την κατηγορία. Ένας διαφορετικός τρόπος να το σκεφτούμε είναι πως οποιαδήποτε καλώς ορισμένη υποέκφραση μπορεί να αντικατασταθεί από μία άλλη με την ίδια τιμή, χωρίς να επηρεάζεται το συνολικό αποτέλεσμα. Το γεγονός αυτό, βοηθάει στην απόδειξη της μαθηματικής ορθότητας ενός προγράμματος, καθώς αρκεί η αποδείξη της ορθότητας των επιμέρους συστατικών του που μπορούν να ενοποιηθούν με τη χρήση τεχνικών αποδείξεων, συνθέτοντας την τελική απόδειξη.

2.2.2 Higher Order Functions

Εφόσον, δεν μπορούμε να ορίσουμε κατηγορηματικά τη χρονική σειρά των γεγονότων στο πρόγραμμα μας, τι μπορεί να αντικαταστήσει αυτή τη δυνατότητα όταν χρειάζεται σημασιολογικά; Για να απαντήσουμε σ'αυτό το ερώτημα θα αντιπαραβάλουμε τη μαθηματική ερμηνεία που μπορεί να έχει, με ένα παράδειγμα. Έστω ότι έχουμε την έκφραση $f(g(x))$, η οποία δηλώνει εμμέσως την αλληλουχία των υπολογισμών που πρέπει να τηρηθεί, δηλαδή για τον υπολογισμό της τελικής έκφρασης, απαιτείται πρώτα η εφαρμογή της συνάρτησης g στο x , και έπειτα η εφαρμογή της f στο αποτέλεσμα που έχει προκύψει.[3]

Η παραπάνω έκφραση μπορεί επίσης να αναπαρασταθεί στα μαθηματικά με τον τελεστή της σύνθεσης συναρτήσεων, ως $(f \circ g)\ (x)$. Στη μορφή αυτή μπορούμε πλέον να αναφερθούμε σε μία μοναδική συνάρτηση $(f \circ g)$ που είναι εμφανώς διαχωρισμένη απο το όρισμα (argument) της. Επιπλέον, μπορούμε να δούμε τον τελεστή σύνθεσης σαν μία συνάρτηση που δέχεται ως όρισμα δύο συναρτήσεις και επιστρέφει μία τελική συνάρτηση, τη σύνθεση τους. Δεν υπάρχει κανένας λόγος ώστε να αποτρέπεται η εφαρμογή των συναρτήσεων σε άλλες συναρτήσεις ή/και η επιστροφή αποτελεσμάτων που είναι και αυτά συναρτήσεις. Οι συναρτήσεις τέτοιου τύπου, συνηθίζεται να καλούνται higher order functions (συναρτήσεις υψηλής τάξης) , και παίζουν σπουδαίο ρόλο στο συναρτησιακό προγραμματισμό, καθώς βοηθούν στη δημιουργία κομψού κώδικα. Για παράδειγμα μπορούμε να έχουμε

[3] Αν η συνάρτηση g ήταν μία σταθερή συνάρτηση, έστω $g(x)=3$, δεν θα υπήρχε λόγος να εφαρμόσουμε την g σε κάποιο όρισμα. Το όλο ζήτημα στα μαθηματικά τίθειται με έναν διαφορετικό τρόπο, δεδομένου ότι δεν υπάρχει η λογική της εκτέλεσης υπολογισμών και η σειριοποίηση τους.

μία συνάρτηση for, η οποία θα παίρνει σαν όρισμα έναν ακέραιο n, μία συνάρτηση f και ένα argument x και θα επιστρέφει την εφαρμογή της f, n φορές στο argument:

$$for(n)(f)(x) = f(f(...f(x)...))$$

Έτσι μπορούμε εύκολα να γράψουμε:

$$thrice = for(3) \qquad \text{, όπου } thrice f = f \circ f \circ f$$

2.2.3 Λάμδα-Λογισμός

Ο λάμδα λογισμός είναι ένας συνεπές μαθηματικό σύστημα, που προτάθηκε από τον Church την δεκαετία του 1930 και αποτελεί το θεωρητικό υπόβαθρο όλων των σύγχρονων γλωσσών συναρτησιακού προγραμματισμού. Ταυτόχρονα, αποτελεί το μέσο με το οποίο επιτυγχάνεται η υλοποίηση των συναρτησιακών γλωσσών. Η χρησιμότητα αυτή του λάμδα λογισμού εντοπίζεται στο γεγονός ότι, όντας ένα σύστημα με πολύ απλή σύνταξη και σημασιολογία, αλλά την ίδια στιγμή εκφραστικά ισχυρότατο, μπορεί να λειτουργήσει ως ενδιάμεση μορφή, στην οποία να μετασχηματίζεται ένα οποιοδήποτε συναρτησιακό πρόγραμμα. Οπότε, αυτό που χρειάζεται πλέον είναι η υλοποίηση ενός πολύ περιορισμένου συνόλου από κανόνες μετασχηματισμού, και ειδικότερα από κανόνες αναγωγής, για τον υπολογισμό εκφράσεων. [Στ15]

Οι εκφράσεις στον λ-λογισμό ονομάζονται λ-εκφράσεις, μία κατηγορία από τις οποίες αποτελούν οι λάμδα αφαιρέσεις. Μια λάμδα αφαίρεση είναι ο ορισμός μιας ανώνυμης συνάρτησης. Με την εισαγωγή των ανώνυμων συναρτήσεων, μπορούμε πλέον να αντιμετωπίζουμε τις συναρτήσεις ως τιμές πρώτης κατηγορίας (functions as first-class values), δηλαδή με τον ίδιο ακριβώς τρόπο που αντιμετωπίζουμε οποιοδήποτε άλλη βαθμωτή τιμή. Για παράδειγμα η έκφραση:

$$let f(x) = x + 3$$

μπορεί να αναπαρασταθεί πλέον από τη λ-έκφραση:

$$let f = \lambda x.(x + 3)$$

Ο τελεστής λ, αποτελεί έναν αφαιρετικό τελεστή που μας πληροφορεί ότι αφενός το *x* αποτελεί μεταβλητή (variable ή formal parameter), και αφετέρου ότι υπάρχει ο ορισμός μίας συνάρτησης μαζί με τον τρόπο που υπολογίζεται το αποτέλεσμα της (x+3). [Davi92]

2.2.4 Lazy evaluation

Σε αυτό το σημείο θα μιλήσουμε για την αγνή [4] συναρτησιακή (purely functional) φύση γλωσσών όπως η Haskell, η οποία τους προσδίδει ένα ακόμα χαρακτηριστικό, την οκνυρή αποτίμηση (lazy evaluation) των εκφράσεων τους. Το παραδοσιακό μοντέλο αποτίμησης, υπολογίζει τα ορίσματα αμέσως μόλις οριστούν ή περαστούν σαν παράμετροι σε μία συνάρτηση και ονομάζεται πρόθυμη αποτίμηση (eager ή strict evaluation). Αντίθέτως, στην οκνηρή αποτίμηση οι τιμές που κατασκευάζονται από έναν ορισμό ή που περνάνε σε μία συνάρτηση δεν αποτιμώνται αμέσως. Υπολογίζονται μονάχα όταν και αν χρειαστούν (call by need) από κάποιον υπολογισμό, ενώ παράλληλα η αποτίμηση τους γίνεται το πολύ μία φορά. Σε περιπτώσεις, που το τελικό αποτέλεσμα δεν χρησιμοποιεί καθόλου κάποια έκφραση, αυτή δεν θα αποτιμηθεί ποτέ, ενώ αν μια έκφραση χρειάζεται περισσότερες από μια φορές, τότε θα υπολογιστεί μονάχα την πρώτη από αυτές (sharing) [Seba11]

Η καινοτόμα ιδέα του lazy evaluation, οδήγησε στη δημιουργία νέων στρατηγικών προγραμματισμού για την επίλυση αλγοριθμικών προβλημάτων. Χαρακτηριστικό είναι ότι επέτρεψε την αλληλεπίδραση με μη πεπερασμένες (άπειρες) δομές δεδομένων που με το συμβατικό μοντέλο του strict

[4] Με τον όρο αγνή (pure), αναφερόμαστε στο γεγονός ότι διατηρείται η ιδιότητα της μη ύπαρξης προστακτικών χαρακτηριστικών στη γλώσσα. Αντιθέτως, υπάρχουν περιπτώσεις συναρτησιακών γλωσσών, όπως η LISP, ML, Scheme, οι οποίες κατατάσσονται στις impure συναρτησιακές γλώσσες καθώς μπορεί να περιέχουν εντολές ανάθεσης, state, while-loops.

evaluation δεν μπορούσαν να υπάρξουν. Παρακάτω, βλέπουμε μία απλή συνάρτηση, γραμμένη σε Haskell, για τη δημιουργία μιας λίστας με άπειρα στοιχεία.

```
listOfOnes::[Int]
listOfOnes = [1] ++ listOfOnes
```

Εδώ ο τελεστής ++ είναι ο τελεστής συνένωσης (concatenation) λιστών. Παρατηρούμε ότι μέσω της αναδρομής κατασκευάζεται μία λίστα που περιέχει το στοιχείο 1 άπειρες φορές. Κάτι τέτοιο δεν θα μπορούσε να συμβεί σε μία γλώσσα με πρόθυμη αποτίμηση καθώς ο υπολογιστής θα προσπαθούσε να κατασκευάσει αμέσως τη λίστα και αυτό θα οδηγούσε σε κάποιο σφάλμα (overflow στη μνήμη, ή μη τερματισμός προγράμματος).

Στη Haskell αυτό δε συμβαίνει, καθώς η δομή listOfOnes δεν κατασκευάζεται αμέσως.Έτσι, κάθε στοιχείο της δομής θα αποτιμηθεί όταν και εφόσον ζητηθεί ρητά. Για παράδειγμα, αν ζητήσουμε από τη Haskell να υπολογίσει το πρώτο στοιχείο της λίστας, τότε η λίστα θα αποτιμηθεί μέχρι το πρώτο της στοιχείο ώστε να δωθεί απάντηση στο ερώτημα. Η ερώτηση που θα κάνουμε στη Haskell είναι $headlistOfOnes$ και η απάντηση που θα πάρουμε είναι, φυσικά το 1. Ομοίως, αν ζητήσουμε το τρίτο στοιχείο, η λίστα θα αποτιμηθεί μέχρι το τρίτο στοιχείο, με την ερώτηση $head(tail(taillistOfOnes))$ και η απάντηση θα είναι και πάλι 1

Όπως προκύπτει, το δεδομένο σχήμα αποτίμησης είναι χρήσιμο γιατί διαχωρίζει τη διαδικασία κατασκευής μίας πολύπλοκης και πιθανώς άπειρης δομής δεδομένων, από τη διαδικασία χρήσης αυτής της δομής σε ένα πρόγραμμα. Ο προγραμματιστής μπορεί να κατασκευάσει τη δομή, αδιαφορώντας για το μέγεθός της, καθώς ξέρει ότι στην πραγματικότητα θα αποτιμηθούν μόνο όσα τμήματά της χρειάζονται. Συνεπώς, μπορεί να ασχοληθεί με τον αλγοριθμό του, αδιαφορώντας για την κατασκευή της δομής, την οποία μπορεί να χειριστεί σαν να ήταν ήδη κατασκευασμένη. Αυτό δεν είναι δυνατό με την πρόθυμη αποτίμηση, στην οποία είμαστε αναγκασμένοι να παράγουμε τη δομή μαζί με τον κώδικα που τη χρησιμοποιεί.[Κα]

2.3 Επιχειρηματολογία για τον Συναρτησιακό Προγραμματισμό

Στο σημείο αυτό θα συνοψίσουμε τα βασικότερα πλεονεκτήματα των συναρτησιακών γλωσσών και έπειτα θα αναπτύξουμε τον αντίλογο που τίθεται.

2.3.1 Πλεονεκτήματα

Συμφωνία με μαθηματικό πρότυπο

Όπως αναφέρθηκε και προηγουμένως, ένα από τα πιο σημαντικά γνωρίσματα των συναρτησιακών γλωσσών, είναι πως καθιστούν ευκολότερη τη μαθηματική διαχείριση των προγραμμάτων τους. Αυτό, σχετίζεται με το γεγονός πως στο εσωτερικό τους ενυπάρχει η ιδιότητα του referential transparency, η οποία κατοχυρώνοντας πως μία μεταβλητή σε ένα πρόγραμμα θα αναπαριστά πάντα την ίδια τιμή, επιτρέπει την ταύτιση της προγραμματιστικής έννοιας της μεταβλητής, με αυτή της μαθηματική έννοια της μεταβλητής. Κάτι τέτοιο δεν συμβαίνει στις προστακτικές γλώσσες, καθώς η μεταβλητή μπορεί να θεωρηθεί σαν μία διεύθυνση μνήμης ή ένας ”container”, στο εσωτερικό του οποίου αποθηκεύεται μία τιμή, η οποία μπορεί ανά πάσα στιγμή να αλλάξει, μέσω κάποιας εντολής ανάθεσης.

Η διατήρηση της τιμής μίας μεταβλητής καθόλη τη διάρκεια ενός προγραμμάτος, μας δίνει τη δυνατότητα να δομίσουμε μαθηματικές αποδείξεις για την ορθότητα προγραμματιστικών συναρτήσεων, με τον ίδιο ακριβώς τρόπο που θα το κάναμε για οποιαδήποτε άλλη μαθηματική έκφραση. Συνεπώς, μπορούμε να χρησιμοποιήσουμε τους νόμους της άλγεβρας ως έχουν ή να δημιουργήσουμε επιπλέον νόμους για λειτουργίες που ορίζονται εντός των προγραμμάτων μας, προκειμένου, για παράδειγμα, να εξασφαλίσουμε την υπάρξη κάποιας ιδιότητας. Επίσης, όπως διαφαίνεται, υπάρχει ενδογενώς -

λόγω referential transparency - η τεχνική του equational reasoning, δηλαδή η δυνατότητα αποδείξεων μέσω της αντικατάστασης (substitution) των ίσων με τα ίσα. Αυτό αποτελεί έναν ιδιαίτερα χρήσιμο πυλώνα των συναρτησιακών γλωσσών, καθώς επιτρέπει την μετατροπή προγραμμάτων σε αντίστοιχα πιο αποδοτικά, τα οποία αποδεδειγμένα επιτελούν τον ίδιο ρόλο (για οποιαδήποτε είσοδο θα δώσουν την ίδια ακριβώς έξοδο, ή εξαίρεση ή θα τρέχουν για πάντα - loop for ever -)

Ένα επιπρόσθετο εργαλείο, που είναι ιδιαίτερα δημοφιλές στον κόσμο του συναρτησιακού προγραμματισμού, είναι η απόδειξη μεσω επαγωγής. Η τεχνική της επαγωγής είναι εύκολο να χρησιμοποιηθεί, δεδομένου ότι οι συναρτήσεις χτίζονται βάση αναδρομικών κανόνων. Ως γνωστών η αναδρομή ενυπάρχει στον ορισμό της επαγωγής, οπότε η δημιουργία επαγωγικών αποδείξεων πάνω σε αναδρομικές συναρτήσεις αποτελεί ένα τετριμμένο πρόβλημα.[Bird95]

Υψηλές δυνατότητες αφαίρεσης

Μπορούμε να πούμε ότι σημασιολογικά, μία συνάρτηση αποτελεί την αφαίρεση (abstraction) του μοτίβου μιάς λειτουργίας - συμπεριφοράς. Αν για παράδειγμα, σε ένα πρόγραμμα γραμμένο σε μία προστακτική γλώσσα όπως η C ή η Pascal παρατηρήσουμε ότι κάποιες λειτουργίες θα πραγματοποιηθούν με τον ίδιο ακριβώς τρόπο για τα στοιχεία δύο παρόμοιων δομών δεδομένων, τότε είθισται να ενσωματώνουμε τις λειτουργίες αυτές σε κάποια συνάρτηση ή διαδικασία (procedure). Έτσι, η συνάρτηση ή η διαδικασία αποτελεί μία αφαίρεση της εφαρμογής της λειτουργίας πάνω σε έναν δοσμένο τύπο δεδομένων. Αν από την άλλη, οι λειτουργίες πρέπει να πραγματοποιηθούν με παρόμοιο και όχι ακριβώς ίδιο τρόπο, με τις συμβατικές μεθόδους δημιουργούνται προβλήματα στη μοντελοποίηση μίας μοναδικής συνάρτησης που να τις ικανοποιεί.

Ας υποθέσουμε ότι θέλουμε να υπολογίσουμε είτε το άθροισμα είτε το γινόμενο των στοιχείων ένος πίνακα ακεραίων. Οι λειτουργίες της πρόσθεσης και του πολλαπλασιασμού αποτελούν δύο παρόμοιες λειτουργίες, καθώς είναι και οι δύο προσεταιριστικές, δυαδικές (binary), υπό την εννοια των δύο ορισμάτων, και αριθμητικές λειτουργίες που δέχονται ως είσοδο και δίνουν ως έξοδο απλούς αριθμούς. Οι συναρτησιακές γλώσσες μας δίνουν τη δυνατότητα να υλοποιήσουμε μία μοναδική συνάρτηση για τον υπολογισμό του παραπάνω ζητουμένου, καθώς με τις higher order functions, η διαφοροποίηση των δύο λειτουργιών (πρόσθεσης και πολλάπλασιασμού) μπορεί πλέον να πραγματοποιείται με την εισαγωγή ενός επιπλέον ορίσματος στη συνάρτηση.

Όπως έχει υποθεί, στις συναρτησιακές γλώσσες οι συναρτήσεις αντιμετωπίζονται ως first class values και επομένως μπορούν να χρησιμοποιηθούν τόσο ως ορίσματα όσο και ως αποτελέσματα άλλων συναρτήσεων. Αύτο αποδεικνύεται ένα πολύ ισχυρό εργαλείο για τη δημιουργία κομψού και μικρού σε έκταση κώδικα, ο οποίος περιέχει υψηλά επίπεδα αφαίρεσης. Κάτι τέτοιο είναι ιδιαίτερα σημαντικό, καθώς διευκολύνει τόσο τη συγγραφή όσο και τη συντήρηση του σχεδιαζόμενου λογισμικού.

Εναλλακτικές αλγοριθμικές τεχνικές

Η οκνυρή αποτίμηση, που χρησιμοποιούν οι αγνά συναρτησιακές γλώσσες όπως η Haskell, έφερε στο προσκήνιο μία σειρά από νέες ιδέες για την επίλυση αλγοριθμικών προβλημάτων. Στο μοντέλο αυτό, είναι πλέον θεμιτές δομές απείρου μήκους ή ακόμα και δομές που είναι μη δεσμευμένες (unbounded). Εφόσον η κλήση μίας συνάρτησης μπορεί να μην χρειαστεί ποτέ να προβεί στον υπολογισμό τέτοιων μη σαφώς ορισμένων στοιχείων ή ολόκληρων των μη πεπερασμένων δομών, το αποτέλεσμα που παράγεται μπορεί να είναι ανεξαρτήτως ορθό.

Η παραπάνω συλλογιστική οδηγεί στην ανάπτυξη αλγορίθμων που διαχωρίζουν τα δεδομένα από την διαχείριση τους. Οι προγραμματιστές των συναρτησιακών γλωσσών που χρησιμοποιούν lazy evaluation, μπορούν να ορίζουν δεδομένα, χωρίς να ενδιαφέρονται για τον τρόπο με τον οποίο αυτά θα επεξεργαστούν, και συναρτήσεις που θα μεταχειρίζονται δομές δεδομένων, αγνοώντας το μέγεθος ή τον τρόπο που κατασκευάζονται. Σαν συνέπεια, έχουμε τη δημιουργία προγραμμάτων που ικανοποιούν την απαίτηση για τμηματοποίηση (modularization) του κώδικα.

Ένα πάραδειγμα που αντικατοπτρίζει το πως συνδέεται ο διαχωρισμός των δεδομένων από την επεξεργασία τους με την τμηματοποίηση του κώδικα είναι το εξής: Έστω ότι έχουμε το σύνολο των φυσικών αριθμών (μη πεπερασμένη δομή). Με βάση αυτό μπορούμε να ορίσουμε το σύνολο των περιττών αριθμών, εφαρμόζοντας του μία συνάρτηση που φιλτράρει τους αριθμούς που θέλουμε (περιττούς). Η κατασκευή της νέας δομής δεδομένων δεν έχει κανέναν λειτουργικό έλεγχο εντός της, αλλά μπορεί εύκολα να συνδεθεί με άλλες συναρτήσεις που επεξεργάζονται περιττούς αριθμούς.

Εναλλακτικές προσεγγίσεις για program development

Η μαθηματική φύση των συναρτησιακών γλωσσών και η ευκολία να μετατρέπουμε προγράμματα στα ισοδύναμα τους, ανοίξε νέους δρόμους και στον τρόπο με τον οποίο αναπτύσσουμε το λογισμικό. Έτσι, ενώ μπορεί να είναι δύσκολη η συγγραφή ενός προγράμματος με την παραδοσιακή λογική, πιθανόν να υπάρχουν τρόποι να εξετάσεις το πρόβλημα από διαφορετικές σκοπιές. Πολλές φορές μπορείς να καταλήξεις στη λύση του ζητουμένου επιλύοντας απλούστερα προβλήματα με τα οποία συνδέεται το αρχικό πρόβλημα μέσω ιδιοτήτων, ακριβώς με τον ίδιο τρόπο που αυτό συμβαίνει στα μαθηματικά. Σαν παράδειγμα μπορούμε να σκεφτούμε ότι θέλουμε να δημιουργήσουμε μία συνάρτηση, ονόματι *lines*, για επεξεργασία κειμένου που θα διαβάζει ένα κείμενο και θα το μετατρέπει σε μία λίστα από γραμμές:

```
lines :: Text → [Lines]
```

Το να γράψεις αποδοτικά το πρόγραμμα lines δεν είναι τόσο εύκολο. Όμως, είναι εύκολο να γράψεις ένα πρόγραμμα *unlines*, που θα κάνει το αντίθετο[5], δηλαδή θα μετατρέπει μία λίστα από lines σε κείμενο.

```
unlines :: [Lines] → Text
unlines xss = foldr1 (insert "\n") xss
```

Η συνάρτηση *unlines* προσθέτει στο τέλος κάθε γραμμής τον τελεστή " \n" (newline) για την παραγωγή του κειμένου, και υλοποιείται εύκολα με τη χρήση της βοηθητικής συνάρτησης υψηλής τάξης foldr1[6]. Δοθέντος του ορισμού της *unlines* μπορούμε να σκιαγραφίσουμε τη λογική με την οποία θα οδηγηθούμε στη *lines*, ως εξής:

```
lines :: Text → [Line]
lines = foldr op a
```

Πλέον χρειάζεται μόνο να ορίσουμε τα *op, a*, από τις ιδιαιτερότητες της *lines*. Έτσι καταλήγουμε στην τελική λύση, όπου η breakOn συνάρτηση αντιστοιχεί στην εύρεση του " \n" που αποτελεί και σημείο ενδιαφέροντος για την διάσπαση των γραμμών του κειμένου.

```
lines :: Text → [Line]
lines xs = foldr (breakOn "\n") [[]] xs
```

Δυνατότητα παράλληλης εκτέλεσης

[5] Για τις *lines* και *unlines* ισχύει ότι: *lines (unlines xss) = xss*, για οποιοδήποτε *xss* που δεν είναι null

[6] η συνάρτηση foldr1 δέχεται ως όρισμα μία συνάρτηση δύο παραμέτρων, και μία λίστα και λειτουργεί ως εξής: αρχικά, εφαρμόζει τη συνάρτηση στο τελευταίο και προτελευταίο στοιχείο της λίστας. Έπειτα εφαρμόζεο τη συνάρτηση στο τρίτο από τέλος στοιχείο της λίστας και στο αποτέλεσμα που είχε προκύψει από το προηγούμενο βήμα. Τελικά επαναλαμβάνει με τον ίδιο τρόπο για όλα τα στοιχεία της λίστας. Η foldr1 όπως και η foldr ορίζεται στην Prelude της Haskell

Η χρήση πολυεπεξεργαστή απαιτεί την ικανότητα αποσύνθεσης ενός προγράμματος σε components (συστατικά) που μπορούν να εκτελεστούν παράλληλα. Στις προστακτικές γλώσσες ο επεξεργαστής χρειάζεται να έχει γνώση των διαμειραζόμενων πόρων των συστατικών ώστε να μπορέσει να συγχρονίσει την εκτέλεση τους χωρίς να υπάρξει αλλοίωση του αποτελέσματος. Αντίθετα, στις συναρτησιακές γλώσσες το referential transparency εγγυάται τη χρονική ανεξαρτησία των συστατικών ενός προγράμματος. Η χρονική αλληλουχία που ικανοποιεί την επιθυμητή σημασιολογία, από μεριάς προγραμματιστή, μπορεί να επιβληθεί μόνο με χρήση εμφωλευμένων εκφράσεων.

Ως αποτέλεσμα συνάγεται πως για την εκτέλεση προγραμμάτων σε πολυπύρινους επεξεργαστές οι compilers των συναρτησιακών γλωσσών δεν χρειάζεται να δαπανήσουν ιδιαίτερο κόπο και ευφευρετικότητα. Εντούτοις ένας ”έξυπνος” μεταγλωττιστής μπορεί, με χρήση των αλγεβρικών νόμων της γλώσσας, να μετατρέψει ένα πρόγραμμα σε κάποιο ισοδύναμο που εκτελείται πιο αποδοτικά. Από την άλλη πλευρά οι compilers των συμβατικών γλωσσών, υποχρεούνται να καταβάλλουν αξιέπαινες προσπάθιες προκειμένου να μπορέσουν να εντοπίσουν την ανούσια σειριοποίηση εκφράσεων και να την αποβάλλουν ασφαλώς.

Σε αυτό το σημείο αξίζει να σημειωθεί πως οι συναρτησιακές και οι προστακτικές γλώσσες επιβάλλουν δύο διαφορετικές προγραμματιστικές προσεγγίσεις. Αφενός, η κλασσική σχολή των προστακτικών γλωσσών δίνει στον προγραμματιστή μέρος της ευθύνης για την επίτευξη παραλληλοποίησης. Αφετέρου, η συναρτησιακή προσέγγιση θεωρεί θεμιτή την πλήρη άγνοια, από μεριάς προγραμματιστή, σχετικά με την πολυεπεξεργασία των προγραμμάτων του. [Cunn14]

2.3.2 Μειονεκτήματα

Μη αποδοτικότητα

Τα προγράμματα που είναι γραμμένα σε κάποια συναρτησιακή γλώσσα είναι κατά κανόνα πιο αργά σε χρόνο, έχουν περισσότερες απαιτήσεις σε μνήμη και περιορισμένες δυνατότητες. Η αποδοτικότητα τους επηρεάζεται από το γεγονός ότι συναγωνίζονται προστακτικές γλώσσες οι οποίες έχουν χτιστεί με την λογική της αρχιτεκτονική Von Neumann, επί της οποίας και χρησιμοποιούνται. Παρ’όλα αυτά, η έρευνα γύρω από καινούργιες αρχιτεκτονικές, πιο συμβατές με το συναρτησιακό πρότυπο, φαίνεται ελπιδοφόρα για τη βελτίωση της αποδοτικότητας του συναρτησιακού προγραμματισμού. Ως παράδειγμα μπορούμε να αναφέρουμε τους υπολογιστές που στηρίζονται σε ροές δεδομένων (dataflow) και γραφικά μοντέλα αναγωγής υπολογισμών (graph reduction models of computations), οι οποίοι εκτελούν αποδοτικότερα προγράμματα συναρτησιακών γλωσσών, παρά προστακτικών, ενώ επίπλέον έδαφος στις συναρτησιακές γλώσσες δίνουν και τα συστήματα που υποστηρίζουν παραλληλοποίηση.

Είναι σημαντικό να ειπωθεί πως με την πάροδο του χρόνου και καθώς οι τεχνολογίες που αφορούν το υλικό (hardware) τόσο σε επίπεδο ταχύτητας επεξεργασίας, όσο και σε επίπεδο μνήμης, έχουν εξελιχθεί ριζικά, είναι αρκετά πιθανό, κατά περίπτωση, οι απαιτήσεις σε απόδοση να θυσιάζονται για χάρη της μαθηματικής πιστότητας ή και της ποιότητας του κώδικα. Σε τέτοιες συνθήκες, οι συναρτησιακές γλώσσες γίνονται όλο και πιο δημοφιλείς.[Cunn14]

Προβλήματα στην αλληλεπίδραση με τον πραγματικό κόσμο

Με τον τρόπο που αναπτύχθηκαν τα χαρακτηριστικά των συναρτησιακών γλωσσών, παραμένει ένα πολύ σπουδαίο ερώτημα: πώς οι συναρτησιακές γλώσσες αλληλεπιδρούν με τον εξωτερικό κόσμο, ο οποίος περιέχει impure γνωρίσματα όπως εξαιρέσεις, input/output, μη ντετερμινισμό; Το ζητούμενο αυτό διαχώρισε το συναρτησιακό κόσμο σε δύο μεγάλες κατηγορίες γλωσσών, τις αγνά συναρτησιακές όπως η Miranda και η Haskell -που θα εξεταστεί βαθύτερα στο πλαίσιο της παρούσας διπλωματικής- και τις μη αγνά συναρτησιακές όπως η ML και η Scheme.

Οι impure συναρτησιακές γλώσσες για την αλλήλεπίδραση με τον εξωτερικό κόσμο θυσιάζουν την ιδιότητα του referential transparency, και επομένως επιτρέπουν την ύπαρξη προστακτικών χαρακτηριστικών όπως τα side effects. Από την άλλη μεριά, οι pure γλώσσες ακολουθούν πιστά το

μαθηματικό μοντέλο και για τη συνδιαλλαγή με impure χαρακτηριστικά χρησιμοποιούν ένα εργαλείο που ονομάζεται monads το οποίο θα συζητηθεί εκτενώς στο επόμενο κεφάλαιο. Για την ώρα, αυτό που αξίζει να σημειωθεί είναι πως αν και τα monads αποτελούν μια πιστική λύση στην αλληλεπίδραση με τον εξωτερικό κόσμο, η έρευνα γύρω από αυτά μπορεί, μελλοντικά, να παράσχει επιπλέον δυνατότητες και διευκολύνσεις στον προγραμματιστή.

Μη επαρκής κατάρτιση των προγραμματιστών

Ο προγραμματισμός σε κάποια συναρτησιακή γλώσσα φαίνεται κάτι παράδοξο σαν πρώτη επαφή. Αυτό πιθανώς να σχετίζεται με το ότι η προστακτική προσέγγιση τοποθετείται πιο κοντά στην ανθρώπινη αντίληψη. Ομολογουμένως, η αναδρομική διαδικασία που χρησιμοποιείται κατά κόρον στις συναρτησιακές γλώσσες, δεν αντιστοιχεί στον καθιερωμένο τρόπο επίλυσης προβλημάτων. Για τους υποστηρικτές όμως των συναρτησιακών γλωσσών κυριότερη αιτία αποτελεί το γεγονός πως η παραδοσιακή διδασκαλία δεν καταπιάνεται με το συναρτησιακό μοντέλο και ως εκ τούτου οι προγραμματιστές δεν εκπαιδεύονται να σκέφτονται ”συναρτησιακά”.

Πρόβλημα στην εκτίμηση της πολυπλοκότητας

Για τις συναρτησιακές γλώσσες που χρησιμοποιούν οκνυρή αποτίμηση, η δυσκολία των προγραμματιστών να κατανοήσουν τη διαδιακασία εκτέλεσης των υπολογισμών (intentional properties) αποτελεί σκόπελο στη συγγραφή αποδοτικών προγραμμάτων. Ένα επιπρόσθετο πρόβλημα είναι πως λόγω της non-strict στρατηγικής αποτίμησης, δισχαιρένεται η εύρεση της πολυπλοκότητας των αλγορίθμων. Η επιστημονική κοινότητα αντιλαμβανόμενη τα δύο αυτά ζητήματα, έχει προβεί στην έρευνα γύρω από την ανάπτυξη τόσο θεωρητικών όσο και πρακτικών προσεγγίσεων με στόχο τη διευκόλυνση των προγραμματιστών.

Από θεωρητικής σκοπιάς έγιναν προσπάθειες ανάπτυξης ενός framework που θα επιτρέπει στους προγραμματιστές να αποκτούν γνώση επί της διαδικασίας εκτέλεσης των εκφρασεων, κάνοντας χρήση αλγεβρικών μεθόδων. Η έρευνα αφορούσε την χρονική πολυπλοκοτητα των προγραμμάτων, ωστόσο δεν προχώρησε σε κάποιο αξιοσημείωτο αποτέλεσμα καθώς αποδείχτηκε ένα αρκετά δύσκολο πρόβλημα. Από την άλλη πλευρά, η πρακτική προσεγγιση αφορά την ανάπτυξη εργαλείων απεικόνησης καραγραφής δεδομένων κατά την εκτέλεση των προγραμμάτων. Εντούτοις, φαίνεται πως μέτρησεις όπως το πλήθος των αναγωγών (reductions) που σημειώθηκαν δεν σχετίζονται απόλυτα με το χρόνο εκτέλεσης (σε δευτερόλεπτα), γιατί κάποια είδη αναγωγών μπορεί να είναι πολύ πιο αργά από άλλα - ας σκεφτούμε αναγωγές που περιέχουν garbage collection. Αντίστοιχα, ούτε το πλήθος των θέσεων μνήμης του σωρού (heap) που χρησιμοποιήθηκαν φαίνεται να αντιστοιχεί αμφιμονοσήμαντα στην ολική μνήμη που χρειάστηκε. Για αυτούς τους λόγους υπάρχει ακόμα ένα μεγάλο πεδίο ενδιαφέροντος προς την έρευνα νέων και πιο αξιόπιστων, ως προς τη συσχέτιση των παραγόντων, εργαλείων. [Coli93]

2.4 Η Γλώσσα Haskell

Τόσο η θεμελίωση, όσο και η σχεδίαση, αλλά και η υλοποίηση, της Haskell είναι βασισμένες στον λάμδα λογισμό. Συγκεκριμένα, η γλώσσα αποτελείται από ένα αρκετά διευρυμένο και υψηλού επιπέδου συντακτικό σύνολο, το οποίο όμως μετατρέπεται με κατάλληλο desugaring[7] σε εκφράσεις του λ-λογισμού. Επιπλέον, ως αγνά συναρτησιακή γλώσσα εναρμονίζεται με όλες τις προδιαγραφές που περιγράφησαν στην προηγούμενη ενότητα, δηλαδή διατηρεί την ιδιότητα του referential transparency, δίνει τη δυνατότητα της μεταχείρησης των συναρτήσεων ως τιμές πρώτης τάξης, επιτρέποντας και ενθαρρύνοντας την ύπαρξη higher order functions, ενώ τέλος αποτιμά οκνυρά της εκφράσεις της.

[7] Με τον όρο desugaring αναφερόμαστε στους μετασχηματισμούς που μπορεί να συμβούν, ώστε να μεταβούμε από ένα πρόγραμμα σε κάποιο άλλο ισοδύναμο.

Η Haskell ανήκει στην κατηγορία των γλωσσών με στατικό σύστημα τύπων (statically typed). Πρακτικά αυτό σημαίνει πως ο τύπος όλων των συναρτήσεων και μεταβλητών γίνεται γνωστός κατά το χρόνο μεταγλώττισης και όχι κατά το χρόνο εκτέλεσης όπως συμβαίνει σε άλλες περιπτώσεις. Η διερεύνηση των τύπων βοηθά σε δύο βασικά ζητήματα. Αφενός, επιβάλλει τους επιθυμητούς περιορισμούς (constraints) τόσο στις παραμέτρους της συνάρτησης όσο και στο αποτέλεσμα της - από μαθηματικής σκοπιάς καθορίζει το πεδίο ορισμού και το πεδίο τιμών. Αφετέρου βοηθά στην περιγραφή της σημασιολογίας της συνάρτησης, δηλαδή στον τρόπο με τον οποίο πρέπει να χρησιμοποιηθεί και την αφαιρετική διαδικασία που χρειάζεται να ακολουθηθεί ώστε να δώσει το επιθυμητό αποτέλεσμα.

Για τους παραπάνω σκοπούς, η Haskell προσφέρει ένα αξιόπιστο σύστημα εξαγωγής τύπων (type inference), το οποίο απελευθερώνει τους προγραμματιστές από την ανάγκη της ρητής περιγραφής αυτών. Ο τύπος που ανατίθεται αυτόματα σε μία έκφραση είναι, κατά κανόνα, ο πιο γενικός - ώστε να υποστηρίζεται πολυμορφισμός - εκτός και αν ορίζεται διαφορετικά απο τον προγραμματιστή. Επιπλέον, για την ασφάλεια των προγραμμάτων, κατά τη διαδικασία της μεταγλώττισης χρησιμοποιείται κατάλληλος type checker που βοηθάει στην έγκαιρη αποφυγή σφαλμάτων τύπων (type errors).

Ένα επιπρόσθετο γνώρισμα της γλώσσας είναι πως όλες οι συναρτήσεις της θεωρούνται συναρτήσεις μίας παραμέτρου. Η αναπαράσταση μιάς συναρτήσης πολλών μεταβλητών - έστω n - γίνεται μέσω μίας συναρτήσης ενός ορίσματος που επιστρέφει ως αποτέλεσμα μία συνάρτηση n-1 μεταβλητών, η οποία με τη σειρά της θα αναπαρασταθεί με τον ίδιο τρόπο. Η ως άνω τεχνική ονομάζεται Currying και σχετίζεται άμεσα με τη δυνατότητα ύπαρξης συναρτήσεων υψηλής τάξης. Από άποψη, τύπων το Currying φαίνεται απο την δεξιά προσεταιριστικότητα (right associativity) του τελεστή $\rightarrow$. Παρακάτω μπορούμε να δούμε ένα παράδειγμα για τη συνάρτηση *sum*, η οποία αναπαριστά την λειτουργία της πρόσθεση δύο ακεραίων.

```
sum :: Int → Int → Int
sum a b = a + b
```

Λόγω της προσεταιριστικόστητας του τελεστή $\rightarrow$, ο τύπος της *sum* ισοδυναμεί με τον παρακάτω τύπο:

```
sum :: Int → (Int → Int)
```

Η δεύτερη αυτή αναπαράσταση του τύπου της *sum* μας υποδεικνύει ότι μιλάμε για μία συνάρτηση η οποία δέχεται ως όρισμα έναν ακέραιο και επιστρέφει ως αποτέλεσμα μια άλλη συνάρτηση με τύπο **Int** $\rightarrow$ **Int**.

Το Currying αποδεικνύεται ιδιαίτερα χρήσιμο συστατικό της Haskell καθώς επιτρέπει την μερική εφαρμογή (partial application) των συναρτησεων. Με τον παραπάνω όρο, αναφερόμαστε στο γεγονός πως μπορούμε να καλέσουμε μία συνάρτηση, χωρίς να είναι απαραίτητο εφαρμόσουμε όλα τα ορίσματα της. Στο παράδειγμα που συζητήκε, η *sum* μπορεί να καλεστεί με ένα αρχικό όρισμα (π.χ το 1) ως εξής:

```
plus1 :: Int → Int
plus1 = sum 1
```

Παρατηρούμε ότι με το partial application μπορούμε να δημιουργήσουμε συναρτήσεις που να επιτελούν διάφορους σκοπούς, μέσω άλλων πιο γενικών συναρτήσεων. Στην περίπτωση μας η plus1, παίρνει έναν αριθμό και τον αυξάνει κατά 1. Συνεπώς δίνονται επιπλέον δυνατότητες στη συγγραφή καθαρού και κομψού κώδικα. [Thom11]

Κεφάλαιο 3

Monads

Στο παρόν κεφάλαιο και θα εξετάσσουμε τον τρόπο με τον οποίο η Haskell καταφέρνει να διαχειρίζεται μη αγνά συναρτησιακά χαρακτηριστικά, διατηρώντας παράλληλα την αγνά συναρτησιακή της φύση. Αρχικά, θα αναφερθούμε στα θεωρητικά στοιχεία των monads και θα επιχερήσουμε να τα ορίσουμε προγραματιστικά. Εν συνεχεία, θα παρουσιάσουμε τα Maybe και State monad, ως δύο αντιπροσωπευτικά instances της monad Class. Μέσω παραδειγμάτων θα περιηγηθούμε στις δυνατότητες που προσφέρουν και τον τρόπο με τον οποίο χρησιμοποιούνται πρακτικά. Το ίδιο θα επιχειρήσουμε και για το IO monad που είναι υπεύθυνο για την αλληλεπίδραση με τον εξωτερικό κόσμο. Στην τελευταία ενότητα, θα συνοψίσουμε τα πλεονεκτήματα των monads.

3.1 Θεωρητικό Υπόβαθρο

Τα monads αποτελούν τον τρόπο με τον οποίο η Haskell καταφέρνει να αλληλεπιδρά με τον εξωτερικό κόσμο και να διαχειρίζεται προστακτικά χαρακτηριστικά, χωρίς να θυσιάζει την ιδιότητα του referential transparency. Η κύρια ιδέα πίσω από τα monads είναι ο διαχωρισμός των τιμών (values) από τους υπολογισμούς (computations). Αν m είναι ένα monad, τότε ένα αντικείμενο με τύπο (m a) αναπαριστά έναν υπολογισμό που αναμένεται να παράξει μία τιμή με τύπο a. Με αυτόν τον τρόπο μπορούμε να σκιαγραφίσουμε το γεγονός πως η χρήση ενός συγκεκριμένου χαρακτηριστικού της γλώσσας εντός μίας συνάρτησης, αποτελεί ιδιότητα του ίδιου του υπολογισμού που πρόκειται να εκτελεστεί και όχι του αποτιμηθέντος αποτελέσματος που αυτή θα δώσει. [Laun92]

Η βάση των monads, προέρχεται από τον κλάδο των μαθηματικών που ονομάζεται Category Theory. Ο Wadler παρατήρησε πως τα monads μπορούν να χρησιμοποιηθούν για τη δημιουργία συναρτησιακών προγραμμάτων, με τον ίδιο ακριβώς τρόπο που χρησιμοποιήθηκαν από τον Moggi για την κατασκευή της δηλωτικής σημασιολογίας (denotational semantics) των γλωσσών προγραμματισμού.

3.1.1 Οι συναρτήσεις *Return* και *Bind*

Στο πλαίσιο της παρούσας διπλωματικής, δεν θα εμπλακούμε με την θεωρητική μαθηματική βάση των monads. Θα εξετάσουμε μονάχα κάποιες βασικές ιδιότητες τους, προκειμένου να μπορέσουμε να περιγράψουμε σαφώς το προγραμματιστικό μοντέλο που παράγουν. Η αναπαράσταση των monads στην Haskell στηρίζεται στο σχηματισμό της Kleisli τριπλέτας (triple), η οποία μαθηματικά ορίζεται ως εξής:

> Μία Kleisli τριπλέτα πάνω σε μία κατηγορία C είναι της μορφής (M, η, _ *), όπου
> $M : |C| \rightarrow |C|$, η$A : A \rightarrow MA$ για $A \in |C|$, $\quad f^* : MA \rightarrow MB$ για $f : A \rightarrow MB$

Στη Haskell, *C* είναι η κατηγορία με τους τύπους της Haskell ως αντικείμενα (objects) και τις συναρτήσεις της Haskell ώς arrows, το *M* αντιστοιχεί σε έναν παραμετροποιημένο τύπο,το *η* ονομάζεται *return*, και το * ονομάζεται *bind* και συμβολίζεται με τον τελεστή $>>=$ [1]. Παρακάτω μπορούμε να

[1] Συνηθίζεται να χρησιμοποιούμε τον τελεστη $>>=$, σε εκφράσεις της μορφής $e1 \ >>= \ \backslash x \ \rightarrow \ e2$. Διαισθητικά, μπορούμε να πούμε ότι σε αυτή την έκφραση το αποτέλεσμα της έκφρασης $e1$ δένεται (bind) με την μεταβλητή x για να χρησιμοποιηθεί εντός της έκφρασης $e2$ ως μία pure τιμή

δούμε τους τύπους για της συναρτήσεις *return* και *bind*, όπου τα a, b αποτελούν τύπους μεταβλητών (type variables) της Haskell, ώστε να υπάρχει πολυμορφισμός. .

```
return  :: a → M a
(>>=)   :: M a → (a → M b) → M b
```

Έτσι, για κάθε monad χρειάζεται να υπάρχει μία σαφής περιγραφή για τον τρόπο που μία τιμή μπορεί να εισαχθεί σε έναν υπολογισμό. Τη λειτουργεία αυτή επιτελεί η συνάρτηση *return*. Ο δεύτερος κανόνας που πρέπει να περιγράφεται ρητά, αφορά τον τρόπ με τον οποιό μπορούν να συνδιαστούν περισσότεροι του ενός υπολογισμοί, και ικανοποιείται από τη συνάρτηση *bind*. Επιπλέον υπάρχουν τρεις monadic laws (νόμους) που πρέπει να ικανοποιεί κάθε monad[Wadl92]:

$$Left\quad unit: \qquad (return\quad a) >>= k \quad = \quad k\quad a$$

$$Right\quad unit: \qquad m >>= (return\quad a) \quad = \quad m$$

$$Associativity: m >>= \backslash a \rightarrow (k\quad a >>= h) \quad = \quad (m >>= k) >>= h$$

Επειδή στη γλώσσα υπάρχουν πολλά ήδη monads που το καθένα εξυπηρετεί διαφορετικούς σκοπούς, η υλοποίηση των συναρτήσεων *bind* και *return* αλλάζει ανάλογα με το εκάστοτε monad. Για να μην υπάρχει η ανάγκη της διαφορετικής ονοματοδοσίας των παραπάνω συναρτήσεων για κάθε ξεχωριστή υλοποίηση, έχει δημιουργηθεί μία κλάση (class) που ονομάζεται Monad και βοηθάει στην υπερφόρτωση (overloading)τους. Με τον όρο υπερφόρτωση αναφερόμαστε στη δυνατότητα που παράχεται στις συναρτήσεις να έχουν το ίδιο όνομα και τύπο, αλλά διαφορετική υλοποίηση για κάθε instance της κλάσης. Ένα επιπλέον θετικό της υπερφόρτωσης είναι πως επιτρέπει τη δημιουργία κώδικα που λειτουργεί πολυμορφικά, δηλαδή για κάθε monad. Για παράδειγμα, μπορούμε να δημιουργήσουμε μία συνάρτηση που να συνδιάζει δύο monadic υπολογισμούς που παράγουν ακεραίους, σε έναν υπολογισμό του αθροισματος τους:

```
addM :: (Monad m) ⇒ m Int → m Int → m Int
addM a b = a >>= \m →
           b >>= \n →
           return (m+n)
```

Καθώς τίποτα στον παραπάνω ορισμό της συνάρτησης δεν καθορίζει κάποιο συγκεκριμένο τύπο monad, η *addM* μπορεί να χρησιμοποιηθεί για οποιοδήποτε monad. Το "(Monad m) =>" καλείται context, και επιβάλλει, οποιαδήποτε αντικατάσταση του m, να συμβεί αποκλειστικά από αντικείμενο που είναι instance της κλάσης Monad.[Nick00]

3.1.2 Do notation

Προς διευκόλυνση των προγραμματιστών, η Haskell παρέχει μία επιπλέον ειδική σύνταξη , που καλείται do notation - για τη συγγραφή κώδικα που σχετίζεται με monadic υπολογισμούς. Χρησιμόποιώντας τη δεδομένη σύνταξη η *addM* της προηγούμενης ενότητας, μπορεί να γραφτεί με τον ακόλουθο τρόπο:

```
addM a b =do m ← a
            n ← b
            return (m+n)
```

Η παραπάνω σύνταξη είναι αρκετά εύχρηστη και χρησιμοποιείται κατα κόρον στην πράξη, δεδομένου ότι προσιδιάζει στο προστακτικό μοντέλο, με το οποίο οι περισσότεροι προγραμματιστές είθισται να είναι περισσότερο εξοικειωμένοι. Ωστόσο, στην πραγματικότητα αποτελεί μονάχα έναν συντακτικό καλλωπισμό (syntactic sugar) για τη γλώσσα και δεν ανήκει στη σύνταξη της. Έτσι, ο compiler μεταφράζει της do-εκφράσεις σε bind-εκφράσεις (>>=) κατά τη φάση του desugaring. Οι κανόνες με τους οποίους επιτυγχάνεται αυτό φαίνονται παρακάτω[2]:

```
do { x ← e; s } = e >>= \x→ do { s }
do { e; s } = e >> do { s }
do { e } = e
```

Όπως προκύπτει, μία έκφραση της μορφής $x \leftarrow e$, εντός ενός do block, δεσμεύει (binds) τη μεταβλητή x. Αντίθετα, σε μία προστακτική γλώσσα η αντίστοιχει έκφραση $x = e$ θα δημιουργούσε μία ανάθεση (assign) στη μνήμη. Παρακάτω παρατίθεται ένα παράδειγμα που υπογραμμίζει τη συγκεκριμένη διαφορά:

```
do    c ← getChar      -- c :: Char
      c ← putChar c    -- c :: ()
      return c
```

Εδώ, χρησιμοποιούμε το ίδιο όνομα μεταβλητής δύο φορές στα αριστερά της έκφρασης (left hand side) και με αυτό τον τρόπο δεσμεύουμε δύο ξεχωριστές μεταβλητές. Η πρώτη γραμμή δεσμεύει το c με την τιμή που δίνει η συνάρτηση *getChar*. Η δεύτερη γραμμή τροφοδοτεί αυτό το c στη συνάρτηση *putChar* και δεσμεύει μία δεύτερη μεταβλητή με όνομα c με την τιμή () που είναι το αποτελέσμα που επιστρέφει η *putChar*.[3] [Jone01]

3.2 Maybe και State Monad

Στο σημείο αυτό θα εξετάσουμε δύο περιπτώσεις monad και τον τρόπο με τον οποίο υλοποιούν τις συναρτήσεις *return* και *bind* ως instances της κλάσης Monad. Αρχικά θα προβούμε στον ορισμό ενός τύπου για την αναπαράσταση ενός υπολογισμού που ενδέχεται να αποτύχει. Αυτό θα το ονομάσουμε *monad of partiality*.

```
data Maybe a = Just a | Nothing
```

Με τον παραπάνω τρόπο, δημιουργούμε ένα παραμετροποιημένο τύπο, ονόματι *Maybe*, που τα στοιχεία του δημιουργούνται είτε με τον constructor *Just* για οποιοδήποτε στοιχείο *x* με τύπο *a*, αναπαριστώντας έναν επιτυχημένο υπολογισμό, είτε με τον constructor *Nothing* για την απεικόνιση ενός υπολογισμού που απέτυχε. Η αποτυχία αποτελεί ένα imperative χαρακτηριστικό, το οποίο για να προσαρτιστεί στη Haskell χρειάζεται να οριστεί κατάλληλο instance της κλάσης Monad.

```
instance Monad Maybe where
        return a = Just a
        m >>= f  = case m of
                     Just a → f a
                     Nothing → Nothing
```

[2] Στη Haskell επιτρέπονται επίσης let-εκφράσεις εντός ενός do block, αλλά δεν ασχολούμαστε μαζί τους για λόγους απλότητας.

[3] Οι συνάρτησεις *getChar* και *putChar* αποτελούν primitives συναρησεις που απαντώνται στο IO monad και αφορούν, αντίστοιχα, την ανάγνωση ενός χαρακτήρα από την είσοδο και την εγγραφή ενός χαρακτήρα στην έξοδο. Το IO monad εξετάζεται λεπτομερώς στην ενότητα 3.3

Ένα παράδειγμα χρήσης του *Maybe* monad αποτελέι η συνάρτηση *divide* που αναπαριστά τη λειτουγεία της διαίρεσης δύο ακεραίων, η οποία όπως είναι γνωστό αποτυγχάνει σε περίπτωση που ο διαιρέτης είναι μηδέν.

```
divide :: Int → Int → Maybe Int
divide a b = if b==0 then Nothing
                     else return (a `div` b)
```

Ένα δεύτερο πιο σύνθετο monad, είναι αυτό που χρησιμοποιείται για την αναπαράσταση της μνήμης και των side-effects που λαμβάνουν χώρα. Παρακάτω ορίζεται ο τύπος του συγκεκριμένου monad, που ονομάζεται State και απεικονίζει τους υπολογισμούς που παράγουν ένα στοιχείο τύπου *a*, προκαλώντας παρενέργειες στη κατάσταση της μνήμης με τύπο s:

```
newtype State s a= State (s → (a,s))
```

Με τη βοήθεια του newtype δημιουργούμε έναν καινούργιο τύπο State s a, ξεχωριστό από όλους τους άλλους, ο οποίος όμως είναι ισομορφικός[4] με τον s → (s,a). Τα στοιχεία του καινούργιου αυτού τύπου περιγράφονται ως State f, ώστε να ξεχωρίζουν από τις συναρτήσεις. Αντίστοιχα με προηγουμένως, τα side-effects αποτελούν μη αγνό συναρτησιακό γνώρισμα, επομένως, για την εισαγωγή τους στη γλώσσα πρέπει να οριστεί instance της κλάσης Monads:

```
instance Monad (State s) where
        return a       = State (\s → (a,s))
        State m >>= f = State (\s → let (a,s') = m s
                                        State m' = f a
                                    in m' s')
```

Εδώ παρατηρούμε ότι παρόλο που ο τύπος State έχει δύο παραμέτρους, και η κλάση Monad απαιτεί οι τύποι που είναι instances της να έχουν μία παράμετρο, η Haskell μας επιτρέπει να δημιουργήσουμε τον τύπο που χρειαζόμαστε, με μερική εφαρμογή του τύπου State στην πρώτη του παράμετρο.[5] [Nick00] Για την διαχείριση της κατάστασης (state) της μνήμης, υπάρχουν κατάλληλες συναρτήσεις:

```
get :: State s s
get = State (\s → (s,s))

put :: s → State s ()
put s = State (\_ → ((),s))
```

Με τη *get* παίρνουμε το περιεχόμενο της κατάστασης της μνήμης εντός ενός State monad όπου η κατάσταση ταυτίζεται με την τιμή επιστροφής. Πρακτικά αυτό σημαίνει πως αφενός η κατάσταση παραμένει αμετάβλητη και αφετέρου επιστρέφεται σαν αποτέλεσμα. Από την άλλη, η συνάρτηση *put* μας επιτρέπει να τροποποιήσουμε την κατάσταση ένος State monad, αγνοώντας την προηγούμενη κατάσταση του. Εφόσον ο μοναδικός στόχος της *put* είναι να επηρεάσει την κατάσταση, θέτουμε το πρώτο στοιχείο του ζεύγους (tuple) ως () -the universal placeholder value - ώστε να δείξουμε ότι το αποτέλεσμα του υπολογισμού είναι αδιάφορο. Ένα παράδειγμα χρήσης των παραπάνω συναρτήσεων αποτελέι η συνάρτηση *increment*, η οποία, οπως αποκαλύπτει και το όνομα της, αυξάνει την κατάσταση - εν προκειμένω έναν μετρητη ακεραίων - κατά ένα.

[4] Με τον όρο ισομορφικός περιγράφεται η ιδιότητα ασφαλής μετατροπής μεταξύ των τύπων, χωρίς απώλεια πληροφορίας. Για δύο τύπους A και B, λέμε ότι είναι ισομορφικοί αν υπάρχουν συναρτήσεις μετατροπής με τύπους: $f :: A \to B$, $g :: B \to A$, τέτοιους ώστε να ισχύει $f \circ g = idB$ και $g \circ f = idA$

[5] Όπως αναφέρθηκε και στο προηγούμενο κεφάλαιο, οι συναρτήσεις με πολλά ορίσματα στη Haskell θεωρούνται curried. Αντίστοιχα, curried είναι και οι τύποι με πολλές παραμέτρους.

```
increment :: State Int ()
increment = get >>= \s →
            put (s+1)
```

Στο σημείο αυτό θα παρουσιάσουμε ένα πιο ολοκληρωμένο παράδειγμα monadic προγραματισμού.[Nick00] Έστω έχουμε μία δομή για την περιγραφή ενός δέντρου, όπως η ακόλουθη:

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

Έστω τώρα ότι θέλουμε να δημιουργήσουμε μία συνάρτηση *unique* με τύπο :

```
unique :: Tree a → Tree (a, Int)
```

Η *unique* θέλουμε να παίρνει ως όρισμα ένα δέντρο (Tree) t και να επιστρέφει ένα δέντρο t' το οποίο να είναι ίδιο με το t με τη διαφορά ότι κάθε κόμβος του θα έχει ένα επιπλέον πεδίο που θα αντιστοιχεί σε μία δεδομένη αρίθμηση[6]:

```
unique (Bin (Bin (Leaf 'a') (Leaf 'b')) (Leaf 'c'))
= Bin (Bin (Leaf ('a',1)) (Leaf ('b',2))) (Leaf ('c',3))
```

Διαισθητικά, μπορούμε να θεωρήσουμε ότι ο αριθμός που αντιστοιχεί σε κάθε φύλλο αντιστοιχεί σε μία κατάσταση για το δέντρο, η οποία αυξάνεται κάθε φορά που καταμετράται ένα φύλλο. Σε μία προστακτική γλώσσα, το πρόβλημα αυτό θα λυνόταν εύκολα με τη χρήση μία global μεταβλητής που θα αναπαραστούσε την αυξανόμενη κατάσταση (μετρητή). Στη Haskell όμως δεν υπάρχουν τέτοιες μεταβλητές, οπότε θα επιδιώξουμε να επιλύσουμε αποδοτικά το ζητούμενο χρησιμοποιώντας το State monad. Θα χρησιμοποιήσουμε μια ενδιάμεση συνάρτηση *unique'* με τύπο:

```
unique' :: Tree a → State Int (Tree (a, Int))
```

Αρχικά μπορούμε να ορίσουμε μία συνάρτηση για την αύξηση του μετρητή, όμοια με την *increment* που εξετάσαμε προηγουμένως:

```
tick :: State Int Int
tick = do s ← get
          put (s+1)
          return s
```

Τώρα μπορούμε να ορίσουμε εύκολα τη *unique'* με τον ακόλουθο τρόπο:

```
unique' (Leaf a) = do n ← tick
                      return (Leaf (a,n))
unique' (Bin t1 t2) = liftM2 Bin (unique' t1) (unique' t2)
```

Η *liftM2* αποτελεί μία συνάρτηση υψηλής τάξης της Haskell για την αλληλεπίδραση με monadic στοιχεία. Η *liftM2* έχει τύπο:

```
liftM2 :: Monad m ⇒ (a → b → c) → m a → m b → m c
```

[6] Η συγκεκριμένη αρίθμηση ξεκινάει από το 1 και πραγματοποιείται από αριστερά προς τα δεξιά για όλα τα φύλλα.

Εν προκειμένω, χρησιμοποιείται για να μετατρέψει τον constructor Bin σε μία συνάρτηση που δέχεται ως ορίσματα δύο υπολογισμούς που προέρχονται από την αναδρομή της *unique'*. Πλέον η *unique* ορίζεται απλά με τον παρακάτω τρόπο:

```
unique t = runState (unique' t) 1
```

Όπου *runState* είναι η συνάρτηση που εξάγει την pure τιμή από το monad, ως ακολούθως:

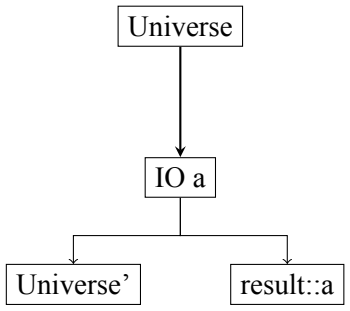
```
runState (State f) s = fst (f s)
```

3.3 Input\Output

3.3.1 Εφαρμογή των monads για αλληλεπίδραση με εξωτερικό κόσμο

Οι λειτουργίες της Εισόδου \Εξόδου (I \O) αποτελούν τα πιο ασυμβίβαστα συστατικά που καλείται να εγκολπώσει το συναρτησιακό μοντέλο προγραμματισμού. Ο λόγος είναι πως οι παραπάνω λειτουργίες ταυτίζονται με την έννοια του action (συχνά το ονομάζουμε και υπολογισμό), το οποίο δεν υπάρχει ενδογενώς στις αγνά συναρτησιακές γλώσσες. Με την εισαγωγή των monads, όμως, δημιουργήθηκε η ικανότητα διαχωρισμού του being (είμαι κάτι) με το doing (κάνω κάτι): μία έκφραση της Haskell αναπαριστά μία τιμή (value) ενώ μία εντολή I \O αφορά την εκτέλεση ενός action.

Σκεφτόμενοι ως παράδειγμα την εγγραφή ή ανάγνωση ενός αρχείου, μπορούμε να πούμε πως τα actions αφορούν την αλληλεπίδραση με τον εξωτερικό κόσμο (Universe) και πιθανώς την τροποποίηση του μέσω κάποιου side-effect. Επομένως, με τις υπάρχουσες γνώσεις μας για τα monads θα μπορούσαμε να μοντελοποιήσουμε το παραπάνω ζητούμενο ως εξής: ένα πρόγραμμα θεωρείται μία συνάρτηση από την κατάσταση (state) του Universe πριν αυτό ξεκινήσει, στην κατάσταση που θα έχει το Universe όταν το πρόγραμμα τερματίσει και δώσει αποτέλεσμα.Μία πιθανή υλοποίηση είναι τα προγράμματα που εξαρτώνται από την κατάσταση του εξωτερικού κόσμου, να τον λαμβάνουν ως παράμετρο και τα προγράμματα που επεμβαίνουν σε αυτόν, να τον επιστρέφουν ως μέρος του αποτελέσματος τους. Σχηματικά, μπορούμε να το φανταστούμε ως εξής[Jone01]:



Σχήμα 3.1: Σχηματική Αναπαράσταση του τύπου IO

Για παράδειγμα, ένα πρόγραμμα που αντιγράφει τα περιεχόμενα ενός αρχείου σε ένα άλλο αρχείο μπορεί να γραφτεί με τον παρακάτω τρόπο:

```
copy :: String → String → Universe → Universe
copy from to universe =
  let contents = readFile from universe
      universe' = writeFile to contents universe
  in universe'
```

Ένα τέτοιο πρόγραμμα ικανοποιεί όλες τις προϋποθέσει του αγνά συναρτησιακού μοντέλου, αλλά δεν είναι εύκολο να υλοποιηθεί λόγω των δυσκολιών που ενέχει η αναπαράσταση του Universe. Από την άλλη πλευρά, θα μπορούσαμε να δημιουργήσουμε ένα παρόμοιο πρόγραμμα που θα χρησιμοποιεί το State monad για την απεικόνιση του Universe, ορίζοντας τον παραπάκτω τύπο:

```
type IO a = State Universe a
```

Έτσι οι primitives συναρτήσεις για την ανάγνωση και εγγραφή σε κάποιο αρχείο θα έχουν τους τύπους:

```
readFile :: String → IO String
writeFile :: String → String → IO ()
```

Ο τύπος IO () υποδεικνύει ότι δεν επιστρέφεται καμία τιμή ως αποτέλεσμα. Βάση αυτών η συνάρτηση *copy* μπορεί να μετατραπεί σε ένα πρόγραμμα που μοιάζει με προστακτικό[7]:

```
copy :: String → String → IO ()
copy from to = readFile from >>= \contents →
               writeFile to contents
```

Η παραπάνω λύση είναι τόσο αγνά συναρτησιακή, όσο και αποδοτική. Παρόλα αυτά επιτρέπει στον προγραμματιστή να ορίσει συναρτήσεις που θα δημιουργούν αντιφάσεις, όπως η παρακάτω:

```
snapshot :: IO Universe
snapshot = State (\univ → (univ, univ))
```

Το πρόβλημα αυτό μπορεί να αποφευχθεί με τη δημιουργία ενός absract type IO για την αναπαράσταση της αλληλεπίδρασης με τον εξωτερικό κόσμο. Έτσι υπάρχει η εγγύηση πως όσο οι primitives λειτουργίες του τύπου IO αλληλενεργούν με το Universe με κάποιον συγκεκριμένο μονονηματικό τρόπο, το ίδιο θα κάνει και οποιοδήποτε πρόγραμμα τις χρησιμοποιεί. [Nick00]

Στο σημείο αυτό είναι σημαντικό να εστιάσουμε στη διαφορά που έχει ένα action από την εκτέλεση του (performance). Μπορούμε να σκεφτόμαστε τα action σαν scripts, τα οποία γίνονται performed όταν τρέξουμε το script. Όμως τα actions αποτελούν *πολίτες-πρώτης-τάξης* για τη Haskell. Πως, λοιπόν, μπορούν να γίνουν performed? Στο σύστημα μας η τιμή ολόκληρου του προγράμματος είναι ένα μοναδικό -πιθανώς μεγάλο- action, που ονομάζεται mainIO, και το πρόγραμμα εκτελείται κάνοντας perform αυτό το action. Παρακάτω μπορούμε να δούμε ένα πολύ απλό παράδειγμα:

```
mainIO :: IO ()
mainIO = putcIO '!'
```

Εδώ διαφαίνεται καθαρά, η διαφορά μεταξύ being και doing: όταν το πρόγραμμα εκτελεστεί, η putcIO action θα γίνει permformed και θα τυπώσει ένα θαυμαστικό στην έξοδο.[Peyt93]

[7] Η κύρια διαφορά είναι πως εδώ, ολόκληρο το περιεχόμενο του φακέλου διαβάζεται ή γράφεται μονομιάς και όχι με τη byte-by-byte μέθοδο που θα γινόταν στην περίπτωση μίας προστακτική γλώσσας. Αυτό θα μπορούσε να φανεί σαν κατάχρηση υπερβολικού χώρου στη μνήμη, αλλά λόγω της οκνυρής αποτίμησης οι χαρακτήρες του αρχείου εισόδου εισάγωνται πραγματικά στη μνήμη μονάχα όταν χρειάστούν για τον υπολογισμό τη εξόδου. Συνεπώς, οι χωρικές απαιτήσεις είναι μικρές και σταθερές (constant), όπως ακριβώς συμβαίνει και σε μία imperative γλώσσα.

3.3.2 Συνδιάζοντας I\O λειτουργίες

Πώς μπορούμε να συνδιάσουμε περισσότερες λειτουργίες που σχετίζονται με είσοδο και έξοδο, προκειμένου να αναπτύξουμε πιο περίπλοκα προγράμματα; Όπως συμβαίνει για όλα το monads, έτσι και για τα IO monad, έχουμε στη διάθεση μας τις συναρτήσεις *return* και *bind*:

```
return :: a → IO a
(>>=):: IO a → (a → IO b) → IO b
```

Ο τύπος της return υποδεικνύει ότι υπάρχει η δυνατότητα ένα action να επιστρέψει μια τιμή. Αν έχουμε μία μεταβλητή *x* τύπου *a*, τότε η *return x* αποτελεί τον υπολογισμό που όταν εκτελεστεί δεν θα κάνει τίποτα άλλο από το να επιστρέψει το *x*. Από την άλλη, αν $m :: IOa$ και $k :: a \rightarrow IOb$ τότε το m»=k υποδηλώνει το action που όταν γίνει performed θα συμπεριφερθεί με τον ακόλουθο τρόπο: αρχικά θα εκτελέσει τον υπολογισμό m, παράγοντας μία τιμή *x* με τύπο *a*. Έπειτα θα εκτελέσει το action *k x*, δημιουργόντας μία τιμή *y* με τύπο *b* την οποία και θα επιστρέψει. Παρακάτω μπορούμε να δούμε μία συνάρτηση για την ανάγνωση μία ολόκληρης γραμμής εισόδου:

```
getLine :: IO [Char]
getLine = getChar >>= \c →
          if c == "\n" then
             return []
          else
             getLine >>= \cs →
             return (c : cs)
```

Το *bind* είναι ο μοναδικός τρόπος που υπάρχει για να συνδιαστούν I\O actions. Μέσω αυτού η αλληλεπίδραση με τον εξωτερικό κόσμο αντιμετωπίζεται σαν μία μονονηματική διαδικασία. Η ουσιαστική λειτουργία που επιτελεί το bind είναι ότι παίρνει τον κόσμο που παράγεται από το αρχικό action και τον περνάει στο επόμενο action. Κατά τον τρόπο αυτό το Universe ποτέ δεν διαγράφεται ούτε αποκτά κάποιο αντίγραφο, ανεξάρτήτως του κώδικα που παράγει ο προγραμματιστής. [Jone01]

Από τα παραπάνω καταλαβαίνουμε πως η αλληλεπίδραση με τον εξωτερικό κόσμο, είναι δυνατόν να υπάρξει σε μία αγνά συναρτησιακή γλώσσα. Έτσι καμία έκφραση δεν έχει side-effects, αλλά υπάρχουν εκφράσεις που υποδηλώνουν την ύπαρξη παρενεργειών -μέσω του τύπου τους. Με τον τρόπο αυτό οι αλληλουχία των εκφράσεων συνεχίζει να μην επενεργει στο αποτέλεσμα. Ωστόσο είναι σημαντικό να σημειωθεί πως οι monadic υπολογισμοί έχουν αυστηρή σειρά εκτέλεσης, την οποία οι νόμοι των monads προστατεύουν ρητά.[Nick00]

3.4 Σχολιασμός

Μέχρι στιγμής, έχουμε αναφερθεί στον τρόπο με τον οποίο η Haskell καταφέρνει να διαχειρίζεται την αποτυχία τερματισμού, να χρησιμοποιεί μεταβλητές που θυμίζουν τις global μεταβλητές των προστακτικών γλωσσών και να αλληλεπιδρά με τον εξωτερικό κόσμο. Αυτά είναι μόνο μερικά από τα παραδείγματα που μπορούμε να δώσουμε για τα monads. Στην πραγματικότητα, με τη βοήθεια των monads οι αγνά συναρτησιακές γλώσσες καταφέρνουν να περιγράψουν με σαφήνεια όλα τα impure στοιχεία που επιθυμούν να ενσωματώσουν. Υπάρχει μία σειρά από monads που έχουν υλοποιηθεί για να διαχειριστούν καταστάσεις όπως ο μη-ντετερμινισμός, τα side effects, τα exceptions, το concurrency, η αλληλεπίδραση με άλλες γλώσσες προγραμματισμού ή και με το ίδιο το λειτουργικό σύστημα και ένα τεράστιο ακόμη σύνολο παρόμοιων προστακτικών γνωρισμάτων.

Στο σημείο αυτό θα παραθέσουμε συγκεντρωτικά τους λόγους για τους οποίους τα monads αποτελούν ένα τόσο ισχυρό χαρακτηριστικό των αγνά συναρτησιακών γλωσσών:

- Η σύνθεση (composition) μεταξύ ομοειδών monads είναι θεμιτή.[8]. Για παράδειγμα, μεγάλα

[8] Όπως θα δούμε στο επόμενο κεφάλαιο η σύνθεση μεταξύ διαφορετικών monads δεν ορίζεται

προγράμματα που επιτελούν I\O λειτουργίες, δομούνται μέσω της συνένωσης και σύνθεσης μικρότερων και απλούστερων προγραμμάτων που διαχειρίζονται κάποια συγκεκριμένη τέτοια λειτουργία. Το γεγονός αυτό, μαζί με τις higher order συναρτήσεις και την οκνυρή αποτίμηση που παρέχει η Haskell χαρίζουν στους προγραμματιστές μεγάλη ευχέρεια στην υλοποίηση κώδικα υψηλής ποιότητας.

- Είναι ένα εύκολα επεκτάσιμο (extensible) εργαλείο. Η Haskell δίνει στους προγραμματιστές τη δυνατότητα να δημιουργήσουν έυκολα νέα monads, με μόνη απαίτηση τη δημιουργία instance που να περιγράφει τις εκάστοτε λειτουργίες των συναρτήσεων *return* και *bind* της κλάσης Monad.

- Είναι αποδοτικά καθώς επιτρέπουν στον compiler την ασφαλή μετατροπή κώδικα με σκοπό τη βελτιστοποιήση του προγράμματος. Επίσης υπάρχουν περιπτώσεις, που ο μεταγλωττιστής μετατρέπει τον monadic κώδικα σε κάποια άλλη γλώσσα. Για παράδειγμα, υπάρχουν βιλιογραφικές αναφορές που υποστηρίζουν πως Haskell κώδικας που αφορά IO actions, μετατρέπεται αυτόματα σε ισοδύναμο πρόγραμμα της C.

- Τα monads βασίζονται μονάχα στο Hindley-Milner σύστημα τύπων, σε αντίθεση με κάποιες άλλες προσεγγίσεις που απαιτούν γραμμικούς(linear) ή existential τύπους. [Peyt93]

Κεφάλαιο 4

Συνδιασμός των effects

Όπως είδαμε στο προηγούμενο κεφάλαιο, τα monads αποτελούν ένα μέσο για τη συγγραφή συναρτησιακών προγραμμάτων που περιέχουν προστακτικά χαρακτηριστικά. Οι περισσότερες πραγματικές εφαρμογές, ωστόσο, απαιτούν κάτι περισσότερο από αυτό. Πολλές φορές υπάρχει η ανάγκη της συνδιαλλαγή με ένα συνδιασμό από monads για την ταυτόχρονη διαχείριση διαφορετικών impure γνωρισμάτων. Στο παρόν κεφάλαιο θα εξετάσουμε τους τρόπους με τους οποίους μπορεί να γίνει ευφυκτή μία τέτοια ”μείξη”. Αρχικά, θα υπογραμμίσουμε την αδυναμία ορισμού της πράξης της σύνθεσης των monads και στη συνέχεια θα παρουσιάσουμε δύο εναλλακτικές προσεγγίσεις, τους monad Transformers και τα Extensible Effects. Τέλος, θα αναπτύξουμε μία συγκριτική μελέτη που θα αφορά τις εν λόγω μεθόδους.

4.1 Σύνθεση monads

Θα επιχειρήσουμε να προσεγγίσουμε το ζήτημα της σύνθεσης των monads μέσω ενός παραδείγματος από τον πραγματικό κόσμο. Ας σκεφτούμε πως θέλουμε να δημιουργήσουμε έναν διερμηνέα (interpreter), ο οπόιος για κάθε έκφραση που υπολογίζει, επιστρέφει πέρα από το αποτέλεσμα και τα βήματα (reduction steps) που χρειάστηκαν για να φτάσει σε αυτό. Εν συνεχεία, έστω ότι θέλουμε ο διερμηνέας μας να επεκταθεί με τρόπο που σε περίπτωση αποτυχίας να εμφανιζει κατάλληλο μήνυμα σφάλματος.

Κάτι τέτοιο θα ήταν ιδιαίτερα επίπονο στην περίπτωση που η υλοποίηση δεν ήταν προσαρμοσμένη στο monadic πρότυπο. Με τα monads όμως, οι μόνες αλλαγές που πρέπει να γίνουν αφορούν το ίδιο το monad που χρησιμοποιήθηκε. Στο σημείο αυτό, μπορούμε να παραθέσουμε μία αναλογία με τα, ευρέως χρησιμοποιούμενα, υπολογιστικά φύλλα (spreadsheet). Όπως όταν σε ένα υπολογιστικό φύλλο που υπάρχουν γραμμές, οι οποίες σχετίζονται με κάποια φόρμουλα, αρκεί να αλλάξουμε τη φόρμουλα για να προξενήσουμε επιθυμητές αλλάγες στα κελιά τη γραμμής, έτσι και στον διερμηνέα μας αρκεί να τροποποιήσουμε το monad με το οποίο έχει υλοποιηθεί, για να προσθέσουμε επιπλέον λειτουργίες.

Στο συγκεκριμένο παράδειγμα, θα θέλαμε να χρησιμοποιήσουμε ένα monad που θα συνδιάζει την ύπαρξη μίας κατάστασης (μετρητή) με την πιθανότητα αποτυχίας ενός υπολογισμού. Ωστόσο, είναι αποδεδειγμένο ότι μαθηματικά δεν είναι δυνατόν να οριστεί η πράξη της σύνθεσης δύο οποιονδήποτε monads ώστε το αποτέλεσμα που προκύπτει να είναι και αυτό monad[1]- δηλαδή να υπακούει στους monadic laws. Επομένως, το επιστημονικό ενδιαφέρον στράφηκε στην ανάπτυξη ενός συστηματικού τρόπου με σκοπό τον συνδιασμό συγκεκριμένων monads. Αναπτύχθηκαν τεχνικές οι οποίες διαχωρίζουν τα πιο γνωστά και συχνά χρησιμοποιούμενα monads σε κατηγορίες, σύμφωνα με τις ιδιότητες που επηρεάζουν τη σύνθεση τους. Για κάθε κατηγορία μπορεί να περιγραφεί μία ξεχωριστή ”συνταγή”, για τον τρόπο με τον οποίο τα στοιχεία της μπορούν να συνδιαστούν με άλλα monads. Για παράδειγμα, οι λίστες, τα δέντρα και τα σύνολα (sets) φαίνεται να έχουν κοινή αντιμετώπιση στην σύνθεση τους με άλλα monads. [King93]

Μιά διαφορετική προσέγγιση δόθηκε από τους Mark P. Jones και Luc Duponcheel στο άρθρο τους

[1] Μία πιο εκτενής εξήγηση για τη μη-ύπαρξη της καθολικής πράξης της σύνθεσης των monads, δίνεται στο paper ”Composing Monads ”[Jone93]

'Composing Monads', όπου εξήγούν πως η γνώση επιπλέον πληροφοριών για τη σύνδεση συστατικών μπορεί να βοηθήσει στην ανάπτυξη στρατηγικών σύνθεσης. Αναφέρουν τέσσερις διαφορετικές τεχνικές, τις οποιες υλοποιούν με τη δημιουργία αντίστοιχων κλάσεων. Κάθε ζεύγος monad που είναι instance μιάς εκ των κλάσεων αυτών, επιβεβαιώνει τις εκάστοτε ιδιότητες του μέσω της υλοποίησης των προβλεπόμενων συναρτήσεων της κλάσης. Ως εκ τούτου, οι επιπλέον πληροφορίες που υποδεικνύει η κάθε κλάση επιτρέπουν τον, υπό όρους, καθόρισμό της πράξης της σύνθεσης των συστατικών που συνδέει. Συνεπώς, ενώ η σύνθεση δύο αυθαίρετων monad M και N είναι αδύνατο να οριστεί με καθολικό τρόπο, η σύνθεση ενός συγκεκριμένου monad M, με κάποιο άλλο monad N[2] το οποίο δεν είναι καθορισμένο μπορεί να είναι θεμιτή, εφόσον γνωρίζουμε ότι το N ανήκει σε μία συγκεκριμένη οικογένεια monad.

Ως παράδειγμα, μπορούμε να σκεφτούμε τη σύνθεση ενός αυθαίρετου monad m, με το Maybe monad για τη δημιουργία ενός καινούργιου monad. Στο μοντέλο που παρουσίασαν οι Mark P. Jones και Luc Duponcheel, για να οριστεί η σύνθεση $Maybe \circ n$, μέ την τεχνική του prod constructions, θα έπρεπε να υπάρχει υλοποίηση της συνάρτησης *prod* της κλάσης PComposable:

```
prod :: n (Maybe (n a)) → Maybe (n a).
```

Παρ'όλα αυτά μία τετοια συνάρτηση δεν μπορεί να οριστεί, όσο το monad n είναι αόριστο. Από την άλλη, για τη σύνθεση $m \circ Maybe$ η *prod* θα έχει τύπο:

```
prod :: Maybe (m (Maybe a)) → m (Maybe a).
```

Στην περίπτωση αυτή, έχουμε πληροφορίες για τη δομή των αντικειμένων που δομούνται με τη χρήση του Maybe. Επομένως, μπορούμε να ορίσουμε την *prod* ως ακολούθως:

```
instance PComposable m Maybe where
    prod (Just m) = m
    prod Nothing = return Nothing
```

4.2 Monad Transformers

Καθώς η ανάγκη εξεύρεσης πιο συγκεκριμένων και απτών προγραμματιστικών λύσεων, στο ζήτημα της σύνθεσης διαφορετικών προστακτικών χαρακτηριστικών, γινόταν όλα και πιο επιτακτική, ανακαλύφθηκε η έννοια των monad Transformers. Στη δημοσίευση τους 'Monad Transformers and Modular Interpreters' οι S. Liang and P. Hudak και M. Jones, ορίζουν τους monad Transformers σε μία προσπάθεια να υλοποιήσουν έναν σύνθετο διερμηνέα. Στόχος ήταν η δημιουργία του διερμηνέα με modular-way, δηλαδή μέσω της συνένωσης μικρότερων προγραμμάτων που το καθένα θα επιτελούσε μία ξεχωριστή λειτουργία. Η σύνδεση των συστατικών του, ωστόσο, πυροδότησε την ανάγκη για τη σύνδεση μη αγνών χαρακτηριστικών όπως η διαχείρηση σφαλμάτων, ο μη ντετερμινισμός, η ύπαρξη περιβάλλοντος και μνήμης και η αλληλεπίδραση με τον εξωτερικό κόσμο (I\O).

4.2.1 Ορισμός των Monad Transformers

Πρίν δώσουμε τον ακριβή ορισμό των monad Transformers, θα προβούμε σε ένα παράδειγμα που θα μας εισάγει στην κεντρική ιδέα που πραγματεύονται. Παρακάτω μπορούμε να δούμε τη δημιουργία ενός τύπου, ονόματι *StateT*, για την αναπαράσταση της συγχώνευσης του State monad με ένα οποιοδήποτε αυθαίρετο monad m:

```
newtype StateT s m a = s → m ( a, s )
```

[2] Το ίδιο ισχύει για τη σύνθεση του N με το M

Από το συμπερασμό τύπων της Haskell, προκύπτει πως το m είναι ένας type constructor. Έτσι αν το m είναι ένα monad, τότε και το ”StateT s m” θα είναι επίσης monad, ενώ το ”StateT s” αποτελεί έναν monad Transformer. Έτσι, αν αντικαταστήσουμε το m, με κάποιο συγκεκριμένο monad θα προκύψει ένα νέο monad που θα εξυπηρετεί ταυτόχρονα τους σκοπούς των δύο monad.

Στη συνέχεια, μπορούμε να δούμε τον ορισμό ενός monad για τη διαχείριση σφαλμάτων (error handling):

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

Το Error monad βοηθάει στην περιγραφή των σφαλμάτων σε περίπτωση που κάτι πάει λάθος. Με αντικατάσταση του στη θέση m του StateT transformer προκύπτει το παρακάτω monad, το οποίο διαχειρίζεται παράλληλα τα impure στοιχεία των δύο monads:

```
StateT s Error a = s → Error (a,s)
```

Υπάρχουν δύο βασικά πλεονεκτήματα στη χρήση των monad Transformers. Αφενός, προσθέτουν επιπλέον λειτουργίες στα monads που εφαρμόζονται, καθώς βοηθούν στην εισαγωγή νέων χαρακτηριστικών στο εσωτερικό τους. Αφετέρου, οι monad Transformers είναι εύκολο να συνδιαστούν μεταξύ τους, μέσω της σύνθεσης, η οποία πλέον αποτελεί μία καθόλα νόμιμη πράξη. Παρακάτω μπορούμε να δούμε μία τέτοια περίπτωση:

```
StateT t (StateT s Error) a = t → (StateT s Error) (t,a)
                            = t → s →  Error (s,(t,a))
```

Όπως φαίνεται, το νέο monad μπορεί πλεόν να διαχειριστεί ταυτόχρονα δύο καταστάσεις και να δώσει μηνύματα σφάλματος σε περίπτωση αποτυχίας. Ωστόσο, αυτό μπορεί να μην είναι τόσο λειτουργικό καθώς δημιουργούνται προβλήματα στον τρόπο με τον οποίο μπορείς να αναφερθείς (να διαβάσεις ή να τροποποιήσεις) σε κάθε κατάσταση. Το εν λόγω ζήτημα θα εξεταστεί λεπτομερώς στη συνέχεια, καθώς αποτελεί ένα γενικευμένο μειονέκτημα των monad transformers και σχετίζεται με τον τρόπο που προστίθενται χαρακτηριστικά σε ένα monad. Προς το παρόν πάμε να δούμε τον τυπικό ορισμό των monad transformers:

> Ορίζουμε ως monad transformer οποιονδήποτε κατασκευαστή τύπων t, τέτοιον ώστε αν m είναι ένα monad (που υπακούει στους monadic laws που περιγράφησαν στο πρώτο κεφάλαιο), τότε και το tm αποτελέι επίσης monad. Το παραπάνω μπορεί να εκφραστεί προγραμματιστικά, με χρήση της κλάσης δύο παραμέτρων MonadT:

```
class (Monad m, Monad (t m)) ⇒ MonadT t m where
    lift :: m a → t m a
```

Η $lift$ αποτελεί τη συνάρτηση που ενσωματώνει τα χαρακτηριστικά του monad m, στο ”t m” monad. Εντούτοις, υπάρχει η απαίτηση το monad m, να μην αλλοιώνει με κανένα τρόπο τη φύση του παραγόμενου ”t m”. Η συγκεκριμένη ιδιότητα περιγράφεται σαφώς, μέσω των νόμων που τίθενται για τους monad transformers:

$$lift \circ unit_{\mathrm{m}} = unit_{\mathrm{t\,m}}$$
$$lift\ (m\ \ 'bind_{\mathrm{m}}'\ \ k) = lift\ m\ \ 'bind_{\mathrm{t\,m}}'\ \ (lift\ \ k)$$

Αναλυτικά, ο πρώτος νόμος αναφέρει πως το να κάνεις lift έναν null υπολογισμό, θα επιστρέψει έναν null υπολογισμό. Ο δεύτερος νόμος αφορά τη σειριοποίηση των υπολογισμών. Έτσι, αν υπάρχουν δύο υπολογισμοί m, k που συνδέονται με το bind στο monad m, μπορούμε αρχικά να κάνουμε lift κάθε υπολογισμό ξεχωριστά και έπειτα να τους συνδιάσουμε στο ”t m” monad.

4.2.2 State Monad Transformer

Στην παρούσα ενότητα, θα ασχοληθούμε λεπτομερώς με έναν αντιπροσωπευτικό monad transformer, τον StateT transformer που ορίσαμε προηγουμένως. Όπως προείπαμε, τό "StateT s m" είναι monad, επομένως πρέπει να ορίσουμε αντίστοιχο instance της κλάσης Monad.

```
instance Monad m ⇒ Monad( StateT s m) where
    return a = \s → (a,s)
    (m >>= k) = \s0 → m s0 >>= \(a,s1) →
                    k a s1
```

Εδώ, πρέπει να σημειωθεί πως οι παραπάνω ορισμοί δεν είναι αναδρομικοί. Ο compiler της Haskell συμπεραίνει αυτόματα πως ο bind τελεστής που εφμανίζεται στο δεξί μέλος, αφορά το monad m.

Με την ίδια λογική πρέπει να ορίσουμε instance της κλάσης MonadT για τον "StateT s" transformer.

```
instance (Monad m, Monad( StateT s m)) ⇒
                        MonadT (StateT s) m where
    lift m = \s → m >>= \x →return (x,s)
```

Παρατηρούμε, ότι η $lift$ τρέχει τον υπολογισμό στο καινούργιο πλαίσιο (context), ενώ διατηρεί αναλλοίωτη την κατάσταση s.

Μέχρι εδώ, έχουμε ορίσει πλήρως τη λειτουργία του StateT transformer, ωστόσο, μένει να διευθετηθεί ένα τελευταίο ζήτημα. Αυτό σχετίζεται με το γεγονός, ότι κάθε monad που διαχειρίζεται καποια κατάσταση, υποχρεούται να υποστηρίζει τη λειτουργία του update της κατάστασης. Ο εν λόγω περιορισμός αναπαριστάται στη Haskell, μέσω της κλάσης StateMonad, ως ακολούθως[3]:

```
class Monad m ⇒ StateMonad s m where
    update :: (s → s) → m s
```

Όπως είναι εμφανές, θα πρέπει να οριστεί κατάλληλο instance της δεδομένης κλάσης για το monad "StateT s m":

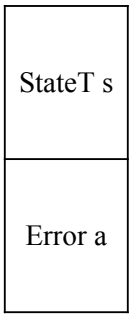
```
instance Monad m ⇒ StateMonad s (StateT s m) where
    update f = \s → return (s, f s)
```

Ο παραπάνω κώδικας μας πληροφορεί πως ο "StateT s" transformer, μετατρέπει κάθε monad σε state monad, όπου το *update f* δηλώνει την ανανέωση της τρέχουσας κατάστασης με την εφαρμογή της συνάρτησης f πάνω στην s. Ως αποτέλεσμα του υπολογισμού επιστρέφεται η κατάσταση s, πριν τροποποιηθεί.

4.2.3 Η συνάρτηση *lift*

Μέχρι στιγμής έχουμε εξετάσει διεξοδικά την ικανότητα των monad transformers να προσθέσουν χαρακτηριστικά σε ένα monad, αλλά δεν έχουμε αναφερθεί καθόλου στον τρόπο με τον οποίο επιτυγχάνουν να "μεταφέρουν" (lift) λειτουργίες μεταξύ των εμπλεκόμενων monad. Για τους σκοπούς αυτής της ανάλυσης, μπορούμε να σκεφτούμε πως οι monad transformers στοιχειοθετούν μία στοίβα (stack) από monadic επίπεδα (layers). Για παράδειγμα το monad "StateT s Error", μπορούμε να το συσχετίσουμε με μία στοίβα της μορφής του επόμενου σχήματος:

[3] Εδώ χρησιμοποιούμε, για λόγους απλότητας, την αναπαράσταση της κλάσηςS tateMonad όπως απαντάται στο άρθρο "Monad Transformers and Modular Interpreters" . Πλέον η κλάση StateMonad απαιτεί τον ορισμό των συναρτήσεων put, get και state.



Σχήμα 4.1: Αναπαράσταση στοίβας για ”StateT s Error” monad

Οι error actions ενυπάρχουν στο τελευταίο επίπεδο της λίστας, οπότε για να γίνουν perform από το πρώτο επίπεδο πρέπει να γίνουν lift. Σε περίπτωση που η λίστα είχε περισσότερα του ενός επίπεδα, θα μπορούσαμε να κάνουμε πολλαπλά lift - αρκεί να γνωρίζουμε το επίπεδο του κάθε monad. Έτσι, αν είχαμε ένα ”MaybeT (StateT s Error)” monad η στοίβα μας θα είχε 3 επίπεδα, επομένως για να καλέσουμε μια συνάρτηση, όπως η *throwError*, του Error monad που βρίσκεται στην τελευταία θέση της στοίβας θα έπρεπε να καλέσουμε δύο φορές τη $lift$ με $lift \$ lift \$ throwError$, προκειμένου να ανέβουμε δύο επίπεδα. Η διαδικασία της διαχείρισης της συνάρτησης $lift$ από τους προγραμματιστές κατά αυτόν τον τρόπο, ονομάζεται explicit (κατηγορηματικό) lift[Schr11] και αποτελεί μία ιδιαίτερα κοπιαστική εργασία.

Για να αποφευχθούν οι ενδεχόμενες δυσκολίες της πολλαπλής εφαρμογής της $lift$ ανάλογα με το επίπεδο του monad στη στοίβα, οι δημιουργοί της mtl (monad transformer library) επινόησαν την έννοια του implicit (έμμεσο) lift. Για τον σκοπό αυτόν, δημιούργησαν instances όλων των monad Transformers σε όλες τις κλάσεις MonadX[4] - n^2 instances. [Grab06]

```
instance (MonadState s m) ⇒
           MonadState s (MaybeT m) where
  update = lift update
```

Παραπάνω μπορούμε να δούμε ένα ενδεικτικό instance, με την βοήθεια του οποίου ένας MaybeT transformer μπορεί να καλέσει τη συνάρτηση update του state monad χωρίς να κάνει lift.

Το μοναδικό πρόβλημα είναι πως για τα IO actions δεν υπάρχει κλάση MonadIO που να ορίζει τις λειτουργίες της, ώστε να μπορούμε να φτιάξουμε instances της. Αυτό συμαίνει πως για να χρησιμοποιήσουμε κάποια IO συνάρτηση θα πρέπει να ξέρουμε το επίπεδο που βρίσκεται στη στοίβα και να την κάνουμε lift αντίστοιχες φορές. Οι δημιουργοί της mtl, έφτιαξαν μιά κλάση MonadIO με σκοπό την αποφυγή της παραπάνω επίπονης διαδικασίας. [Grab06]

```
class (Monad m) ⇒ MonadIO m where
  liftIO :: IO a → m a

instance MonadIO IO where
  liftIO = id

instance (MonadIO m) ⇒
           MonadIO (StateT s m) where
  liftIO = lift . liftIO

  ..............
```

[4] MonadX ονομάζουμε τις κλάσεις που επιβάλλουν την υλοποίηση συγκεκριμένων συναρτήσεων για την διαχείριση του προστακτικού χαρακτηριστικού που αφορούν. Κάποιες από αυτές είναι η MonadState, η MonadError, η MonadReader κ.τ.λ

Με τον τρόπο αυτό θα μπορούμε να καλούμε την liftIO μία φορά, χωρίς να μας ενδιαφέρει το επίπεδο στο οποίο βρίσκεται το IO monad. Για παράδειγμα:

```
debugState :: (MonadIO m) ⇒ String → m a
debugState input = do
                    liftIO $ print "Input:"++ input
```

4.2.4 Αδυναμίες των Monad Transformers

Παρ'ότι οι monad Transformers έδωσαν μία αρκετά πιστική λύση στο ζήτημα της σύνθεσης impure γνωρισμάτων, έχουν κάποιες βασικές αδυναμίες τις οποίες θα επιχειρήσουμε να περιγράψουμε. Όπως αναφέραμε προηγούμένως, οι λειτουργία των monad transformers μπορεί να προσιδιαστεί με μία στοίβα, αποτελούμενη από τα monads που περιέχονται στον υπολογισμό διατεταγμένα με ορισμένη σειρά. Αυτή η διάταξη είναι στατική και αποτελεί τον κυριότερο περιορισμό που εργαλείου, καθώς δημιουργεί ένα περίπλοκο και πολλές φορές ασαφές πλαίσιο πάνω στο οποίο ο προγραμματιστής καλείται να ανταπεξέλθει. Στη συνέχεια, θα δούμε ένα πρακτικό παράδειγμα που αναδεικνύει τις ως άνω δυσκολίες.

Ας θεωρήσουμε μία συνάρτηση $ex1$ η οποία λειτουργεί για ένα οποιοδήποτε monad m, το οποίο όμως μπορεί να διαχειριστεί εξαιρέσεις[5], με τον ακόλουθο τρόπο: αν το υποφαινόμενο monad περικλύει κάποια τιμή η οποία είναι μεγαλύτερη από 5, πραγματοποιείται μία εξαίρεση ($TooBig$), διαφορετικά επιστρέφεται ο αριθμός ως έχει. Σημασιολογικά αυτό αποσκοπεί στην περιγραφή μίας συνάρτησης που θα τερματίζει υπολογισμούς κατά την εμφάνιση μιάς εξαίρεσης.

```
newtype TooBig = TooBig Int deriving (Show)

ex1 :: MonadError TooBig m ⇒ m Int → m Int
ex1 m = do
  v ← m
  if v > 5 then throwError (TooBig v)
     else return v
```

Ας σκεφτούμε, από την άλλη, μία συνάρτηση που υποδεικνύει έναν μη ντετερμινιστικό υπολογισμό, στον οποίο επιλέγεται ένας ακέραιος από μία λίστα ακεραίων[6]:

```
choose :: MonadPlus m ⇒ [a] → m a
choose = msum ∘ map return
```

Έτσι λοιπόν, αν εφαρμόσουμε την $ex1$ πάνω στην $choose[5, 7, 1]$, θα προκύψει ένας υπολογισμός, ο οποίος θα πρέπει να ικανοποιεί τους δύο περιορισμούς των contexts της $ex1$ και $choose$. Συνεπώς το monad που συνοδεύει τη συνάρτηση $ex1(choose[5, 7, 1])$ θα πρέπει να είναι instance τόσο της κλάσης MonadError όσο και της κλάσης MonadPlus και κατά συνέπεια θα είναι σε θέση να διαχειριστεί συνάμα εξαίρεση και μη ντετερμινισμό. Η mtl βιβλιοθήκη μας δίνει αυτή τη δυνατότητα μέσω της σύνθεσης των αντίστοιχων monad transformers, ErrorT και ListT με το Identity monad[7].

Παραμένει, ωστόσο, το ερώτημα της διάταξης που θα έχουν τα monad στη στοίβα που θα δημιουργηθεί, καθώς υπάρχουν δύο τρόποι για να γίνει η σύνθεση τους. Η μία σειρά σύνθεσης δίνει στη συνάρτηση τον τύπο :*(ErrorT TooBig (ListT Identity)) a*. Επομένως, αν κάνουμε run τον υπολογισμό με την σειρά που αντιστοιχεί στον παραπάνω τύπο:

[5] Ο περιορισμός MonadError στο context της συνάρτησης $ex1$, επιβάλλει το monad m να μπορεί να διαχειριστεί εξαιρέσεις.

[6] η κλάση MonadPlus περιορίζει το monad m, ώστε να επιτελεί έναν μη ντετερμινιστικό ρόλο.

[7] Το identity monad υποδεικνύει έναν υπολογισμό που δεν περιέχει προστακτικά χαρακτηριστικά

```
ex1_1 = runIdentity ∘ runListT ∘ runErrorT $ ex1 (choose [5,7,1])
```

,το αποτέλεσμα που θα πάρουμε θα έχει τύπο [*Either TooBig a*] που εννοιολογικά επισημαίνει έναν μη ντετερμινιστικό υπολογισμό, στον οποίο κάθε επιλογή μπορεί είτε να πετύχει παράγοντας μία τιμή a, είτε να αποτύχει δίνοντας μία TooBig εξαίρεση. Η εξαίρεση, λοιπόν, είναι εμφωλευμένη μέσα σε μία μη ντετερμινιστική απόφαση.

Αντιθέτως, όπως προείπαμε, το ζητούμενο στην περίπτωση μας είναι μία εξαίρεση να οδηγήσει στον τερματισμό του προγράμματος. Συνεπώς, θα πρέπει να χρησιμοποιήσουμε την αντίθετη σειρά στα επίπεδα της στοίβας, ώστε η $ex1(choose[5,7,1])$ να έχει τύπο *ListT (ErrorT TooBig Identity) a*. Με κατάλληλα run:

```
ex1_2 = runIdentity ∘ runErrorT ∘ runListT $ ex1 (choose [5,7,1])

-- Left (TooBig 7)
```

καταλήγουμε στον τύπο *Either TooBig [a]* που υποδηλώνει το επιθυμητό, δηλαδή είτε την παραγωγή κάποιας εξαίρεσης TooBig, είτε την επιστροφή μίας λίστας, για μία μη ντετερμινιστική επιλογή. Στα σχόλια του κώδικα, φαίνεται το αποτέλεσμα της παραπάνω εκτέλεσης, που είναι φυσικά μία εξαίρεση καθώς εντός της λίστας που δίνεται ως όρισμα, υπάρχει αριθμός μεγαλύτερος του 5.

Μέχρι στιγμής μπορούμε να κατανοήσουμε τη σύγχυση που πιθανώς να δημιουργηθεί, ακόμα και για μία απλή περίπτωση, σύνθεσης δύο monad transformers. Στο άρθρο τους "Extensible Effects An Alternative to Monad Transformers" [Kise13], οι O. Kiselyov, A. Sabry και C. Swords επεκτείνουν το παραπάνω παράδειγμα. Αυτή τη φορά, το ζητούμενο είναι η "ανάκαμψη" από την εξαίρεση σε περίπτωση που ο αριθμός που πραγματεύεται η συνάρτηση $ex1$ δεν είναι πολύ μεγαλύτερος του 5. Έτσι, αν ο αριθμός είναι μικρότερος ή ίσος του 7, θέλουμε το πρόγραμμα να συνεχίζει κανονικά την εκτέλεση του, ενώ σε αντίθετη περίπτωση να προκύπτει σφάλμα. Για το σκοπό αυτό ορίζεται η συνάρτηση $exRec$ που χρησιμοποιεί την $catchError$ της κλάσης MonadError, ως εξής:

```
exRec :: MonadError TooBig m ⇒ m Int → m Int
exRec m = catchError m handler
    where handler (TooBig n) | n ≤ 7 = return n
          handler e = throwError e
```

Θα περιμέναμε η εκτέλεση του παρακάτω κώδικα να επιλύει το καινούργιο πρόβλημα:

```
runIdentity ∘ runErrorT ∘ runListT $ exRec (ex1 (choose [5,7,1]))
-- Right [7]
```

ωστόσο, η απάντηση που δίνεται δεν ειναι σωστή. Ο αρχικός υπολογισμός έχει τρεις μη ντετερμινιστικές επιλογές 5,7,1, εκ των οποίων οι δύο δεν πρέπει να προξενούν error. Η επιλογή 7, θα έπρεπε μεν να προκαλεί κάποια εξαίρεση, η οποία όμως θα διευθετούνταν κατάλληλα ωστέ να μήν προκαλεί ούτε εκείνη σφάλμα. Το αναμενόμενο αποτελέσμα λοιπόν θα έπρεπε να αποτελείται από τρεις μη ντετερμινιστικές επιλογές, κάτι που ικανοποιείται από την πρώτη αλληλουχία των transformers που αναφέραμε (*(ErrorT TooBig (ListT Identity)) a*).

Εφόσον για το πρώτο κομμάτι τη συνάρτησης, δηλαδή τη δημιουργία (ολικής) εξαίρεσης αν κάποιος αριθμός είναι μεγαλύτερος από 5, είναι απαραίτητη η (*ListT (ErrorT TooBig Identity) a*) σειρά των transformers , ενώ για το κομμάτι της ολικής ανάκαμψης από την εξαίρεση χρειάζεται η αντίστροφη σειρά, καταλήγουμε πως ο υπολογισμός μας θα πρέπει να έχει δύο διαφορετικά επίπεδα ErrorT στη στοίβα. Επομένως, απαιτείται η δημιουργία ενός αντικειμένου με τον εξής, ιδιόμορφο τύπο: *ErrorT TooBig (ListT (ErrorT TooBig Identity))*. Αυτό δημιουργεί πρόβλημα στη διαχείριση

των επιπέδων και επιβάλλει την επιστροφή στην κουραστική και αναποτελεσματική διαδικασία του explicit lift[8]. Τελικά χρειαζάζεται να τροποποιήσουμε την $ex1$ και να δημιουργήσουμε μία ακόμα συνάρτηση $runErrorRelay$, ωστέ να δομήσουμε μία επιτυχή, αλλά επίπονη υλοποίηση:

```
ex1' :: MonadError TooBig m ⇒ m Int → m Int
ex1' m = do
  v ←lift m
  if v > 5 then throwError (TooBig v)
     else return v


runErrorRelay :: MonadError e m ⇒ ErrorT e m a → m a
runErrorRelay m = runErrorT m >>= check
                  where check (Right x) = return x
                        check (Left e) = throwError e

ex1'_2 = runIdentity ◦ runErrorT ◦ runListT ◦ runErrorRelay $
exRec (ex1' (choose [5,7,1]))
-- Right [5,7,1]
```

Αν το παραπάνω πρόβλημα περιπλέκει ακόμα παραπάνω τα πράγματα, στο επικείμενο άρθρο, οι συγγραφείς επισημαίνουν ένα επιπλέον παράδειγμα που αποδεικνύει πως υπάρχουν προβλήματα που δεν μπορούν καν να επιλυθούν με τους monad transformers, λόγω της στατικής διατάξης της στοίβας τους.

4.3 Extensible Effects

Μία εναλλακτική στους monad Transformers είναι το μοντέλο που ονομάζεται Extensible Effects και βασίζεται στην ιδέα που παρουσίασαν οι R. Cartwright και M. Felleisen στο άρθρο τους "Extensible Denotational Language Specifications" (EDLS)[Cart94]. Στην παρούσα ενότητα, θα επιχειρήσουμε να περιγράψουμε τη νέα αυτή μέθοδο σύνθεσης προστακτικών χαρακτηριστικών, όπως παρουσιάστηκε στη δημοσίευση με τίτλο "Extensible Effects An Alternative to Monad Transformers".[Kise13]Να σημειωθεί πως ο κύριος στόχος των ερευνητών ήταν τα Extensible Effects να ξεπεράσουν τα εμπόδια που τίθενται από τους Monad Transfomers και σχετίζονται με το γεγονός ότι η διάταξη των impure γνωρισμάτων είναι στατική.

4.3.1 Κεντρική ιδέα

Στο νέο σύστημα, μπορούμε αφαιρετικά να προσομοιάσουμε το κάθε effect με μία επικοινωνία (interaction): ένα τμήμα του προγράμματος που επιθυμεί να μεταβάλλει μία κατάσταση, να δώσει μία εξαίρεση ή να γράψει σε κάποιο αρχείο στέλνει ένα αίτημα (request) σε μία Αρχή (authority). Το request περιγράφει το action που πρέπει να γίνει perform και περιέχει επιπλέον μία διεύθυνση επιστροφής (return address) για τη συνέχεια της εκτέλεσης του προγράμματος. Η Αρχή από την άλλη, διαχειρίζεται όλους τους πόρους του συστήματος (αρχεία, μνήμη,κ.τ.λ) και είναι υπεύθυνη για την εκτέλεση ή απόρριψη των αιτημάτων. Στο EDLS πρότυπο, η Αρχή ήταν καθορισμένη και ανεξάρτητη του προγράμματος - με τον ίδιο τρόπο που στο λειτουργικό σύστημα ο kernel δεν είναι μέρος των διεργασιών του χρήστη.

Αντίθετως, τα Extensible Effects σχεδιάστηκαν ώστε η Αρχή να είναι διαμοιρασμένη και αυτομάτως επικτάσιμη, ως "γραφειοκρατία" που είναι μέρος του προγράμματος του χρήστη. Μπορούμε

[8] Για μιά πιο εκτενή περιγρφή των μειονεκτημάτων των explicit και implicit lift, καλούμε τους αναγνώστες να ανατρέξουν στο άρθρο Monads, zippers and views: virtualizing the monad stack[Schr11]

να σκεφτούμε κάθε ”υποαρχή” (που διαχειρίζεται κάποιους πόρους και ερμηνεύει κάποια αιτήματα) ως έναν handler (χειριστή). Όπως σε ένα σύστημα επικοινωνίας peer-to-peer, κάθε handler είναι ταυτόχρονα ένας πελάτης (client) που στέλνει requests και μία Αρχή (σαν server) που διευθετεί requests. Αν ένας handler λάβει κάποιο request που δεν εμπίπτει στη δικαιοδοσία του, το μεταβιβάζει στους αρμόδιους handlers.

Ένα δεύτερο σημείο που αξίζει να σημειωθεί είναι πως το μοντέλο των Extensible Effects επιτυγχάνει την καταγραφή των ενεργών effect στον τρέχων υπολογισμό, μέσω ενός εκφραστικού type-and-effect συστήματος που αναπτύχθηκε. Το σύστημα διατηρεί ένα open union το οποίο περιέχει ένα μη διατεταγμένο σύνολο των ενεργών effects. Κάθε action που γίνεται perform από έναν handler αποτυπώνεται, αφαιρώντας από τον τύπο τα effects που έλαβαν χώρα. Με τον τρόπο αυτό, το σύστημα εγγυάται πως ένα πρόγραμμα δεν περιέχει ”αχρείαστα” effects.

4.3.2 Περιήγηση στο Framework των Extensible Effects

Ήρθε η στιγμή να εξετάσουμε τον τρόπο με τον οποίο τα Extensible Effects μπορούν να εφαρμοστούν στην πράξη. Προς διευκόλυνση των προγραμματιστών οι δημιουργοί του framework επίλεξαν να αναπτύξουν μία βιβλιοθήκη (Σχήμα 4.2) που να θυμίζει σε μεγάλο βαθμό το δημοφιλές μοντέλο των monad Transformers. Τα δύο κυριότερα σημεία ενδιαφέροντος είναι κατά πρώτον πως όλοι οι υπολογισμοί αναπαριστώνται από ένα monad *Eff r* και κατά δεύτερον πως η παράμετρος r στοιχειοθετεί ένα open union από μεμονωμένα effects, των οποίων τα συστατικά πρέπει να είναι Typeable. Διαισθητικά μπορούμε να αντιληφθούμε το r, ως το σύνολο των effects που ο υπολογισμός μπορεί να κάνει perform. Το r προκύπτει από τα requests, καθώς όπως είπαμε τα effects αναπαριστώνται από αιτήματα.

Υπάρχουν δύο βασικοί τρόποι που μπορούν να εκφράσουν ότι ένα effect m ανήκει στο open union r. Ο πρώτος είναι μέσω ενός περιορισμού *(Member m r)* στο context του τύπου. Ο δεύτερος είναι μέ χρήση του τελεστή :>, σε τύπους της μορφής *(m :> r')*. Με αυτόν τον τρόπο το open union r διαχωρίζεται σε δύο τμήματα: στο effect m και αυτό που απομένει από το σύνολο r, δηλαδή το σύνολο r'. Παρατηρούμε ότι υπάρχει μία αντιστοιχία με το συμβολισμό $\{m\} \cup r'$ της συνολοθεωρίας.Έτσι, μπορούμε να σκεφτούμε πως ο τύπος Void ισοδυναμεί με το κενό σύνολο $\emptyset$, συνεπώς ένας τύπος *Eff Void a* υποδεικνύει έναν αγνό υπολογισμό, δηλαδή μια τιμή με τύπο a.

Στη συνέχεια θα πάμε να εξετάσουμε κάποια παραδείγματα χρήσης των Extensible Effects:

```
t1 :: Member (State Int) r ⇒ Eff r Int
t1 = do v ← get
        return (v + 1:: Int )
```

Η συνάρτηση $t1$ δίνει σαν αποτέλεσμα μία τιμή τύπου Int, η οποία αναπαριστά την υπάρχουσα κατάσταση, αυξημένη κατά 1. Το γεγονός πως το r περιέχει τουλάχιστον το State effect, εκφράζεται εντός του context από τον περιορισμό Member (State Int) r. Μπορούμε να κάνουμε perform το action του υπολογισμού, δίνοντας μία αρχική κατάσταση στη συνάρτηση $runState$ (π.χ *runState t1 (1::Int)*). Ο τύπος που συμπεραίνεται για τη δεδομένη έκφραση θα είναι ο *Eff r Int*, καθώς ο περιορισμός για το State effect αφαιρείται από το open union, όταν διευθετηθεί το action του. Στην περίπτωση που δεν υπάρχει άλλο effect στο σύνολο r, μπορούμε πλέον να τρέξουμε τον υπολογισμό, ως ακολούθως:

```
t1r = run $ runState t1 (10:: Int )
-- 11
```

Αξίζει να παρατηρήσουμε πως ο κώδικας για την $t1$ ακολουθεί τον τρόπο με τον οποίο θα γράφαμε την συνάρτηση με monad Transformers, με τη διαφορά ότι θα είχαμε τον περιορισμό MonadState στη θέση του Member (State Int). Στην πραγματικότητα το *Eff r* αποτελεί μία γενίκευση του StateT καθώς, το r μπορεί να περιέχει μία σειρά από επιπλέον effects. Στο σημείο αυτό θα προβούμε σε ένα

```
newtype TooBig = TooBig Int deriving (Show)
instance Monad (Eff r)

-- Pure computations
data Void
run :: Eff Void w → w

-- Reader (or environment) effect
type Reader e
ask :: (Typeable e, Member (Reader e) r) ⇒ Eff r e
local :: (Typeable e, Member (Reader e) r) ⇒
         (e → e) → Eff r w → Eff r w
runReader :: Typeable e ⇒
             Eff (Reader e :> r) w → e → Eff r w

-- Exceptions
type Exc e
throwError :: (Typeable e, Member (Exc e) r) ⇒ e → Eff r a
catchError :: (Typeable e, Member (Exc e) r) ⇒
              Eff r w → (e → Eff r w) → Eff r w
runError :: Typeable e ⇒
            Eff (Exc e :> r) w → Eff r (Either e w)

-- State
type State s
get :: (Typeable s, Member (State s) r) ⇒ Eff r s
put :: (Typeable s, Member (State s) r) ⇒ s → Eff r ()
runState :: Typeable s ⇒
            Eff (State s :> r) w → s → Eff r (w,s)

-- Non-determinism
type Choose
choose :: Member Choose r ⇒ [w] → Eff r w
makeChoice :: Eff (Choose :> r) w → Eff r [w]

-- Tracing
type Trace
trace :: Member Trace r ⇒ String → Eff r ()
runTrace :: Eff (Trace :> Void) w → IO w

-- Build-in effects (e.g., IO)
type Lift m
lift :: (Typeable1 m, MemberU2 Lift (Lift m) r) ⇒
        m w → Eff r w
runLift :: (Monad m, Typeable1 m) ⇒
           Eff (Lift m :> Void) w → m w
```

Σχήμα 4.2: Το interface της βιβλιοθήκης των Extensible Effects

πιο σύνθετο παράδειγμα που μπλέκει δύο καταστάσεις, εκ των οποίων η μία είναι τύπου Int και η άλλη τύπου Float:

```
t2 :: (Member (State Int) r, Member (State Float) r) ⇒
      Eff r Float
t2 = do
       v1 ← get
       v2 ← get
       return $ fromIntegral (v1 + (1:: Int )) + (v2 + (2:: Float))
```

Κάθε μία από τις v1,v2 μεταβλητές παίρνει την τιμή από το effect με το οποίο σχετίζεται, επομένως η v1 αφορά την Int κατάσταση και η v2 την Float. Ένα τέτοιο πρόγραμμα δεν θα μπορούσε να γραφεί με τον ίδιο τρόπο με τους monad Transformers. Θα έπρεπε να βάλουμε στη στοίβα δύο επίπεδα του StateT και να ορίσουμε τη διάταξη τους, ώστε μετέπειτα να μπορέσουμε να κάνουμε explicit lift. Από την άλλη πλευρά, η συνάρτηση t2 δεν υποδεικνύει με κανέναν τρόπο κάποια σειρα για τα effects. Η σειρά ορίζεται μονάχα όταν επιδιώξουμε να τα κάνουμε perform, αλλά δίνει και για τις δύο επιλογές το ίδιο αποτέλεσμα:

```
t2run1 = run $ runState (20:: Float ) (runState (10:: Int ) t2 )
-- 33.0

t2run2 = run $ runState (10:: Int ) (runState (20:: Float ) t2 )
-- 33.0
```

Παρ' όλα αυτά, υπάρχουν περιπτώσεις που αν και το σύνολο r είναι μη διατεταγμένο, η σειρά που τρέχουμε τους υπολογισμούς επηρεάζει το αποτέλεσμα. Ας θυμηθούμε το παράδειγμα της ενότητας 4.2.4. Παρακάτω θα δούμε πως μπορεί να επιλυθεί με χρήση των extensible effects:

```
newtype TooBig = TooBig Int deriving (Show, Typeable)

ex2 :: Member (Exc TooBig) r ⇒ Eff r Int → Eff r Int
ex2 m = do
          v ← m
          if v > 5 then throwError (TooBig v)
             else return v
```

Το πρώτο και πιο απλοϊκό ζητούμενο που είναι η δημιουργία εξαίρεσης σε περίπτωση που έστω και ένας αριθμός είναι μεγαλύτερος από 5, μπορεί να υλοποιηθεί εύκολα ως εξής:

```
runErrBig :: Eff (Exc TooBig :> r) a → Eff r (Either TooBig a)
runErrBig m = runError m

ex2r = run ∘ runErrBig ∘ makeChoice $ ex2 (choose [5,7,1])
-- Left (TooBig 7)
```

Η διαφοροποίηση των δύο μεθόδων γίνεται στο δεύτερο ερώτημα του προβλήματος. Εδώ, τα extensible effects αποδίδουν καλύτερα καθώς δίνουν τη δυνατότητα να εκφραστεί μία πιο κομψή και ευκολονόητη λύση, με τη συνάρτηση $exRec$ που, αυτή τη φορά, δίνει το σωστό αποτέλεσμα:

```
exRec :: Member (Exc TooBig) r ⇒ Eff r Int → Eff r Int
exRec m = catchError m handler
          where handler (TooBig n) | n ≤ 7 = return n
                handler e = throwError e

ex2'r = run ∘ runErrBig ∘ makeChoice $ exRec (ex2 (choose [5,7,1]))
-- Right [5,7,1]

ex2'r = run ∘ runErrBig ∘ makeChoice $ exRec (ex2 (choose [5,7,11,1]))
-- Left (TooBig 11)
```

Όπως μπορούμε να δούμε, το γεγονός ότι δεν υπάρχει στατική διάταξη των monads, δίνει την ελευθερία στον χρήστη να τα χρησιμοποιήσει κατά βούληση, χωρίς προηγούμενες αποφάσεις σχεδιασμού να επηρεάζουν μετέπειτα κινήσεις. Επομένως, δεν χρειάζεται να τροποποιήσουμε την ex2 για να κάνουμε lift κάποια συνάρτηση. Αυτό αποτελεί μία γενική κατάσταση, με μία εξαίρεση που θα δούμε παρακάτω, για τα Extensible Effects: το σύνολο r δεν είναι διατεταγμένο, άρα δεν υπάρχει η έννοια του lifting.

4.3.3 Lift handler

Στην προηγούμενη παράγραφο είπαμε πως δεν υπάρχει ανάγκη να γίνονται lift οι συναρτήσεις των αναμιγνυομενων υπολογισμών. Ωστόσο, στη βιβλιοθήκη 4.2 βλέπουμε πως υπάρχει συνάρτηση $lift$. Η χρησιμότητα της, στα πλαίσια του *Eff r* monad είναι να μπορεί να εισάγει αυθαίρετα effect. Έτσι το effect ενός τυχαίου monad m σημειώνεται ως $(Lift\ m)$. Να σημειωθεί, πως εφόσον δεν ορίζεται η σύνθεση αυθαίρετων monad, μπορούμε να έχουμε το πολύ ένα Lift effect σε κάθε υπολογισμό. Το γεγονός αυτό υπογραμμίζεται απο τον περιορισμό (MemberU2 Lift (Lift m) r) στο context της συνάρτησης $lift$. Για την υποστήριξη της λειτουργίας *lift m* το framework των Extensible Effet έχει έναν ειδικό handler, τον Lift handler, στον οποίο αποστέλεται το προς εκτέλεση action m. Αυτό το γεγονός διαχωρίζει τη διαδικασία των lifting στα Extensible Effects και τους monad Transformers.

Η δυνατότητα του lifting επιτρέπει το εργαλείο των Extensible Effects να μπορεί να χρησιμοποιηθεί με αξιοποίηση κάποιας ήδη υπάρχουσας monadic βιβλιοθήκης (που μπορεί να έχει οριστεί ακόμη και από τον χρήστη). Το πιο αντιπροσωπευτικό παράδειγμα είναι η εισαγωγή IO στοιχείων, που μπορούμε να δούμε και ακολούθως:

```
tl1 :: (MemberU2 Lift (Lift IO) r , Member (State Int) r) ⇒
       Eff r ()
tl1 = get >>= \x → lift ∘ print $ (x+1:: Int)
```

Εδώ η συνάρτηση $tl1$ συνδιάζει τα State και IO effects. Αξίζει να παρατηρήσουμε πως σε αντίθεση με ότι ίσχυε μέχρι τώρα, η $tl1$ ορίζει μία διατεταγμένη σειρά για τα monads, η οποία διαπιστώνεται και από τον τύπο της.

4.3.4 Σύγκριση με Monad Transformers

Τα Extensible Effects απότελουν ένα εργαλείο για τη σύνθεση impure χαρακτηριστικών, το οποίο μπορεί να εκφράσει ένα υπερσύνολο των προγραμμάτων που μπορούν να εκφραστούν με τους monad Transformers. Διατηρώντας το "πνεύμα" των monad Transformers, κάθε πρόγραμμα γραμμένο με τη βοήθεια των monad Transformers μπορεί να γραφτεί με ίδια ή και περισσότερη απλότητα και κομψότητα. Αντίθετα, υπάρχουν προβλήματα που δεν μπορεί να περιγράψει το μοντέλο των transformers, ενώ είναι εύκολο να αναπαρασταθούν με τα Extensible Effects.

Ένα σημείο που αξίζει προσοχής είναι πως τα Extensible Effects μπορούν να αποδειχθούν αποδοτικότερα από τους monad Transformers. Αυτό εξηγείται αν σκεφτεί κανείς πως κάθε επίπεδο στη

στοίβα των transformers προσθέτει επιπλέον κόστος εκτέλεσης (overhead). Έτσι, υπάρχει μία αναλογία ανάμεσα στο πλήθος των επιπέδων και στο παραγόμενο overhead. Επίσης, το κάθε επίπεδο μπορεί να αξιοποιηθεί από ένα μικρό μέρος του υπολογισμού, ενώ επιβαρύνει ολόκληρο το monad. Από την άλλη μεριά, το overhead στα Extensible Effects σχετίζεται με το πλήθος των ενδιάμεσων handler, μεταξύ ενός requester και ενός τελικού (υπό την έννοια οτι είναι ο ίδιο υπεύθυνος για το αίτημα) handler. Επόμένως, αν και στη χειρότερη περίπτωση η ύπαρξη πολλαπλών handlers αυξάνει το overhead, στην καλύτερη περίπτωση δεν το επηρεάζει.

Οι υποστηρικτές των Extensible Effects διατείνονται τέλος, πως είναι πολύ πιο εύκολο να επιχειρηματολογήσει κανείς για την ορθότητα ενός προγράμματος που χρησιμοποιεί το *Eff r* monad απ'ότι για το αντίστοιχο γραμμένο με monad Transformers. Δεν είναι στο σκοπό αυτής της εργασίας να αναφερθεί περεταίρω στο εν λόγω ζήτημα. Για μία εκτενή ανάλυση των προβλημάτων που είναι πιθανό να προκύψουν στις αποδείξεις προγραμμάτων με monad Transformers καλούμε τους αναγνώστες να εξετάσουν το άρθρο "Deriving backtracking monad transformers" [Hinz00]. Από την άλλη, στο άρθρο "Extensible Effects An Alternative to Monad Transformers" [Kise13] περιγράφεται ενδελεχώς μία συγκριτική μελέτη των αποδεικτικών τεχνικών, που αναδεικνύει τα προτερήματα των Extensible Effects.

Κεφάλαιο 5

Υλοποίηση του Μετασχηματισμού

5.1 Εισαγωγή

Στο παρόν κεφάλαιο θα επιχειρήσουμε να περιγράψουμε την υλοποίηση της εργασίας. Αρχικά θα αναφερθούμε στις σχεδιατικές αποφάσεις που πήραμε και αποτέλεσαν το μεταβατικό βήμα από το θεωρητικό στο πρακτικό πλαίσιο της εργασίας. Στη συνέχεια, θα αποδώσουμε με αφηρημένο τρόπο τη ροή εκτέλεσης του εργαλείου που αναπτύξαμε. Στις μετέπειτα ενότητες θα περιγράψουμε με λεπτομέρειες τα συστατικά της υλοποίησης, ξεκινώντας από την κατασκευή της γλώσσας RouReg και συνεχίζοντας με τον Lexer και τον Parser που δημιουργήσαμε για να μπορούμε να την αξιοποιήσουμε. Το επόμενο κομμάτι που θα αναλυθεί, είναι η υλοποίηση του ίδιου του transformation, ενώ στο τέλος θα δώσουμε κάποια παραδείγματα προς απόδειξη της χρησιμοτητας του.

5.1.1 Σχεδιαστικές αποφάσεις

Για την υλοποίηση του επικείμενου μετασχηματισμού, επιλέξαμε να χρησιμοποιήσουμε το εργαλείο των Extensible Effects. Το κύριο επιχείρημα ήταν πως είναι επιθυμητή η σύνθεση των effects, χωρίς την ύπαρξη λειτουργιών lifting. Θεωρήσαμε πως το μοντέλο των monad Transformers θα ήταν δύσκολο να χρησιμοποιηθεί, εξαιτίας των, πολλές φορές αναπόφευκτων, explicit lift. Για έναν περίπλοκο μετασχηματισμό θα ήταν πρακτικά εξόντωτικο να υπάρχει επίγνωση του επιπέδου που βρίσκεται το κάθε monad, ώστε να πραγματοποιηθούν αντίστοιχα lift.

Από την άλλη, τα Extensible Effects επιτρέπουν τη μείξη των προστακτικών γνωρισμάτων, εντός μιάς μη διατεταγμένης δομής open union. Το να εισάγεις και να εξάγεις effects στη δομή, μπορεί να υλοποιηθεί με απλό προγραμματιστικό τρόπο, ενώ όπως είδαμε στην παράγραφο 4.3.3 υπάρχει το πολύ ένα επιπεδο το οποίο μπορεί να χρειαστεί lifting. Τα παραπάνω καταδεικνύουν πως το *Eff r* monad είναι ιδανικό για την υλοποίηση ενός αυτοματοποιημένου σχεδιασμού σαν αυτόν που επιδιώκουμε στη συγκεκριμένη εργασία.

Ένα δεύτερο σημείο που αφορά τον σχεδιασμό της υλοποίησης είναι η σημασιολογία του όρου monadic form. Όπως αναφέραμε και στην εισαγωγή, επιθυμούμε μέσω του μετασχηματισμού, να μεταβούμε σε ένα πρόγραμμα που όλες οι συναρτήσεις θα επιστρέφουν κάτι monadic. Όμως, όπως έχει ήδη συζητηθεί, το curring στη Haskell κάνει τις συναρτήσεις να αποτελούν συναρτήσεις ενός μόνο ορίσματος. Επομένως, μία συνάρτηση, έστω f, με τύπο $a \rightarrow\ b \rightarrow\ c$ είναι μία συνάρτηση που παίρνει ένα αντικείμενο με τύπο a και επιστρέφει μία συνάρτηση, έστω $f1$, με τύπο $b \rightarrow\ c$. Για να είναι το αποτέλεσμα της f monadic θα πρέπει να περικλύσουμε την $f1$ σε ένα monad και, με τον ίδιο τρόπο, για να είναι το αποτέλεσμα της $f1$ monadic θα πρέπει να περικλύσουμε το αντικείμενο με τύπο c επίσης σε monad. Συνεπώς, η συνάρτηση που θα περιμέναμε να ικανοποιεί τις απαιτήσεις μας, θα είχε τύπο $a \rightarrow\ m\ (b \rightarrow\ m\ c)$.

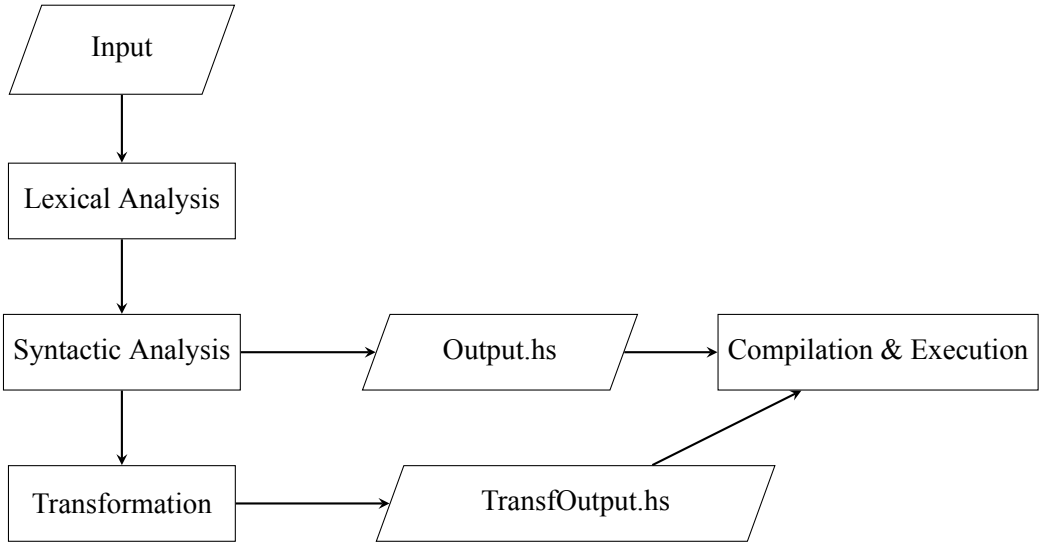
Εντούτοις, με τον συγκεκριμένο τύπο θα ήταν πιθανό να προκύψουν αντιφάσεις. Αυτό προκύπτει από το γεγονός ότι υπάρχει η επιπλέον απαίτηση, όλες οι συναρτήσεις να αναπαριστώνται από μία καθολική μορφή τύπου. Αν παρατηρήσουμε, όμως την f και την $f1$, που αναφέραμε προηγουμένως, αντιλαμβανόμαστε διαφορές στους τυπους τους. Η $f1$ (και στην πραγματικότητα κάθε παραγόμενη συνάρτηση), θα εσωκλείεται ολοκληρωτικά μέσα σε monad, $m\ (a \rightarrow\ m\ b)$. Αντιθέτως η f (και κάθε συνάρτηση που δεν προκύπτει ως αποτέλεσμα κάποιας άλλης συνάρτησης), θα έχει μερικώς monadic

τύπο, $a \rightarrow m\ (b \rightarrow m\ c)$. Τελικά, η ομοιομορφία επέρχεται στον μετασχηματισμό μας, επιλέγοντας και ο τύπος της f να αντιστοιχεί σε $m\ (a \rightarrow m\ (b \rightarrow m\ c))$.

5.1.2 Περιγραφή ροής εκτέλεσης

Η υλοποίηση του μετασχηματισμού έγινε σε Haskell με χρήση του εργαλείου stack και μπορεί να χωριστεί σε δύο βασικούς άξονες. Ο πρώτος αφορά τη δημιουργία της γλώσσας RouReg, η οποία αποτελεί υποσύνολο της Haskell, και είναι η γλώσσα στην οποία γράφονται τα προγράμματα προς επεξεργασία. Ο δεύτερος άξονας, αφορά τον μετασχηματισμό των προγραμμάτων σε μοναδιαία μορφή (monadic form). Προς ευκολία ελέγχου και επαλήθευσης των αποτελεσμάτων, δημιουργήσαμε κατάλληλο interface, αντίστοιχο με το ghci, στο οποίο με την εντολή :l name, επιτυγχάνεται ολόκληρος ο κύκλος εκτέλεσης της εφαρμογής μας, με τα αποτελέσματα να απεικονίζονται στο τερματικό του υπολογιστή μας. Η ροή εκτέλεσης αποτυπώνεται στο Διάγραμμα 5.1 και συνοψίζεται στα εξής στάδια:

1. Λεκτική ανάλυση του κώδικα Roureg που περιέχεται στο αρχείο προς φόρτωση
2. Συνατκτική ανάλυση του κώδικα του πρώτου σταδίου
3. Δημιουργία αρχείου haskell, ονόματι nameOutput.hs, που περιέχει τον ανεπεξέργαστο κώδικα
4. Μετασχηματισμός του κώδικα RouReg σε monadic form
5. Δημιουργία αρχείου haskell, ονόματι nameTransfOutput.hs, που περιέχει τον επεξεργασμένο κώδικα
6. Μεταγλώττιση και εκτέλεση των προγραμμάτων nameOutput.hs και nameTransfOutput.hs



Σχήμα 5.1: Διάγραμμα Ροής για το interface.

5.2 Η Γλώσσα RouReg

Για την ανάγκη της υλοποίηση μας επιλέξαμε να πειραματιστούμε με ένα υποσύνολο της Haskell, το οποίο ονομάσαμε RouReg και παρουσιάζεται, αφαιρετικά, στον πίνακα 5.1 . Στόχος μας είναι η απόδειξη της λογικής (proof-of-concept) του μετασχηματισμού οποιασδήποτε συνάρτησης της Haskell

σε monadic form, οπότε φροντίσαμε η γλώσσα μας να μπορεί να αποτυπώσει ένα μεγάλο εύρος προγραμμάτων που μπορούν να γραφτούν σε Haskell, ακόμα και αν η εκφραστηκότητα της είναι σχετικά χαμηλή - δηλαδή μπορεί να χρειαστεί να γράψουμε αρκετές γραμμές κώδικα για κάτι που στη Haskell θα έπαιρνε μόνο λίγες, καθώς χρησιμοποιούμε θεμελιώδεις τύπους και ένα αρκετά μικρό σύνολο predefined συναρτήσεων και τελεστών. Πιο αναλυτικά το υποσύνολο που επιλέξαμε περιέχει:

Για τον ορισμό των συναρτήσεων:

- Κυριολεκτικά (Literals) αλφαριθμητικών (strings), ακεραίων αριθμών και λογικών τιμών (True, Flase)
- Ζευγάρια κυριολεκτικών (Tuples) - όχι εκφράσεων
- Λίστες εκφράσεων.
- Εκφράσεις: Αναγνωριστικά μεταβλητών, let εκφράσεις, εκφράσεις εφαρμογής (apply) συνάρτησης, λ-εκφράσεις, case-of εκφράσεις, bind ($>>=$) εκφράσεις.
- infix τελεστές - ο τελεστής ανάμεσα στα τελούμενα - για αριθμητικούς υπολογισμούς (+ , - , *, div), και για χειρισμό λιστών (: cons operator)

Για τα ορίσματα (arguments) των συναρτήσεων:

- Κυριολεκτικά (Literals)
- Αναγνωριστικά μεταβλητών
- Λίστες, είτε εντός square Brackets (π.χ [1,2,x]), είτε με τον τελεστή cons (π.χ (x:xs))
- Κατασκευαστές Τιμών (Data Constructors)

Για τον ορισμό των τύπων:

- Κυριολεκτικά αλφαριθμητικών
- Τύπους Λιστών
- Τύπους για Tuples
- Τύπους Συναρτήσεων (ορίζονται όπως και στη Haskell με τον infix τελεστή $\rightarrow$)
- Containers (όπως διάφορμα monads)
- Ειδικό Container για το Eff monad, ονομαι EffUnionContainer
- Τύπος Void

5.3 Λεκτική και Συντακτική Ανάλυση

Ο λεκτικός αναλυτής που χρησιμοποιήσαμε, δημιουργήθηκε με τη βοήθεια του εργαλείου Alex, εντός του αρχείου Lexer.x. Το Alex είναι ένα πρόγραμμα, αντίστοιχο με το lex της C, που δέχεται σαν είσοδο την περιγραφή των tokens μίας δεδομένης γλώσσας, σε μορφή regular expressions και δημιουργεί αυτόματα ένα module σε Haskell που περιέχει κώδικα για την αποδοτική ανάγνωση των προγραμμάτων της γλώσσας αυτής και τη διαίρεση τους σε λεκτικές μονάδες (tokens), δηλαδή σε συμβολοσειρές που αποτελούν τα μικρότερα αδιαίρετα κομμάτια της σύνταξης του προγράμματος .[CDor03] Στον πίνακα 5.2 μπορούμε να δούμε τα tokens με τις αντίστοιχες εκφράσεις που χρησιμοποιήσαμε για τη λεκτική ανάλυση της γλώσσας RouReg. Η συνάρτηση που διεκπεραιώνει την

```
AllDclrs : AllDclr AllDclrs
         | AllDclr

AllDclr : TSign ; Dclrs

Dclrs : Dclr ; Dclrs
      | Dclr

Dclr : VAR Apats = Expr

Expr : let AllDclrs in Expr
     | case Expr of Cases
     | \Apats → Expr
     | Expr : Expr
     | Expr »= Expr
     | Cons Expr Expr
     | Form

Form : Form + Form
     | Form - Form
     | Form * Form
     | Fact

Fact : Fact Atom
     | Atom
```

```
Atom : ( Expr )
     | Tuple
     | List
     | ”Str”
     | VAR
     | Values

TSign : VAR :: TScp
      | VAR :: Contexts => TScp

TScp : Type
     | forall Names . ( Type)

Type : Contr → Type
     | Contr

Contr : VAR ListT
      | SimpleT

ListT : SimpleT ListT
      | SimpleT

SimpleT : (Type)
        | (TupleT)
        | [Type]
        | VAR
        | ()
```

Πίνακας 5.1: Η γραμματική της RouReg.

διαδικασία της λεκτικής ανάλυσης, ονόματι scanTokens, έχει υλοποιηθεί εντός του Except monad, για να δίνει σφάλμα σε περίπτωση που χρησιμοποιηθεί κάποιο token το οποίο δεν υπάρχει στη γλώσσα.

Τη λεκτική ανάλυση της εισόδου ακολουθεί η συντακτική ανάλυση, με σκοπό την εξαγωγή του AST (Abstract Syntax Tree - Αφηρημένου Συντακτικού Δέντρου). Ο συντακτικός αναλυτής (Parser) της γλώσσας μας δημιουργήθηκε με χρήση του εργαλείου Happy, το οποίο είναι όμοιο με το yacc για τη C. Το Happy παίρνει σαν είσοδο ένα αρχείο (εν προκειμένω το Parser.y) που περιέχει την περιγραφή της γραμματικής της γλώσσας μας και παράγει ένα Haskell module που δίνει τον parser που επιθυμούμε. Το Happy επιλέχθηκε γιατί είναι εύκολο στη χρήση και εγγυάται την δημιουργία ενός συντακτικού αναλυτή που θα είναι κατά κανόνα γρηγορότερος και αποδοτικότερος από αντίστοιχους που θα μπορούσαμε να δημιουργήσουμε με τη χρήση parser combinators ή άλλων εργαλείων. Επιπλέον κατά το εγχειρίδιο χρήσης (Documentation) τού, οι μελλοντικές βελτιώσεις του εργαλείου θα μπορούν να βελτιώσουν μία υπάρχουσα γραμματική, χωρίς να χρειάζεται να γραφτεί ξανά.[Marl01]

Η περιγραφή της γραμματικής που αναφέραμε, θα πρέπει να είναι σε BNF μορφή (Backus Normal Form), δηλαδή να περιέχει:

- Ένα σύνολο από λεκτικές μονάδες (tokens).
- Ένα σύνολο από μη τερματικά σύμβολα (non-terminals)

\n	{\s → TokenNewLine }
: :	{\s → TokenHasType }
⇒	{\s → TokenContext }
forall	{\s → TokenForAll}
let	{\s → TokenLet }
in	{\s → TokenIn }
True	{\s → TokenTrue }
False	{\s → TokenFalse }
$digit+	{\s → TokenNum (read s) }
→	{\s → TokenArrow }
=	{\s → TokenEq }
\	{\s → TokenLambda }
+	{\s → TokenAdd }
-	{\s → TokenSub }
*	{\s → TokenMul }
(	{\s → TokenLParen }
)	{\s → TokenRParen }
$ alpha [$ alpha $ digit _ \']*	{\s → TokenSym s }
;	{\s → TokenSemicolon }
{	{\s → TokenLBracket }
}	{\s → TokenRBracket }
[	{\s → TokenLListOp }
]	{\s → TokenRListOp }
,	{\s → TokenComma }
:	{\s → TokenCons }
»=	{\s → TokenBind }
.	{\s → TokenDot }

Πίνακας 5.2: Tokens και Regular Expressions για τον Lexer της RouReg.

- Το αρχικό σύμβολο (start symbol), δηλαδή το μη τερματικό σύμβολο που αποτελεί τη ρίζα του συντακτικού δένδρου για κάθε αποδεκτό από τη γλώσσα πρόγραμμα.
- Ένα σύνολο από κανόνες παραγωγής (production rules), όπου κάθε κανόνας αποτελείται από ένα μη τερματικό σύμβολο στα αριστερά του, ακολουθούμενο από ένα colon (:) και μία ή περισσότερες επεκτάσεις κανόνα στα δεξιά, χωρισμένες με |. Κάθε επέκταση κανόνα σχετίζεται με κάποιο κομμάτι κώδικα Haskell το οποίο είναι κλεισμένο σε αγκύλες.

Θα μπορούσαμε να σκεφτούμε τη λειτουργία ενός Parser ως εξής: Κάθε σύμβολο έχει μία τιμή "value", η τιμή των tokens είναι ορισμένη, και η τιμή των μη τερματικών συμβόλων ορίζεται από τη γραμματική ως συσχέτιση με άλλα συμβολα (tokens ή μη-τερματικά). Σε έναν κανόνα παραγωγής, όπως ο παρακάτω:

```
n  :  t_1 ... t_n        { E }
```

όποτε ο parser βρει τα σύμβολα t_1 ... t_n στην ροή των λεκτικών μονάδων (token stream) κατασκευάζει το σύμβολο n και του δίνει την τιμή E, που μπορεί να αναφέρεται στην τιμή των t_1 ... t_n, χρησιμοποιώντας τα σύμβολα $1...$n.

Ο parser καταναλώνει την είσοδο χρησιμοποιώντας τους κανόνες της γραμματικής, εώς ότου απομείνει ένα μόνο σύμβολο: το entry symbol. Η τιμή αυτού του συμβόλου είναι και η τιμή επιστροφής

```
AllDclrs :   AllDclr AllDclrs                    { $1 : $3 }
           | AllDclr                             { [$1] }

AllDclr :    TypeSignature ';' Dclrs             { WithSign $1 $3 }

Dclrs :   Dclr ';' Dclrs                         { $1 : $3 }
        | Dclr                                   { [$1] }

Dclr :   VAR Apats '=' Expr                      { Assign $1 $2 $4 }
```

Πίνακας 5.3: Παραδείγματα κανόνων παραγωγής

του parser.[Marl01]

Στην περίπτωση μας, οι τιμές E, αποτελούν τους τύπους δεδομένων (data types) της γλώσσας RouReg που έχουμε ορίσει στο αρχείο Syntax.hs . Το αρχείο αυτό είναι ο θεμέλιος λύθος της γλώσσας μας και γίνεται import σε όλα τα επί μέρους αρχείου του προγράμματος μας. Στον πίνακα 5.3 μπορούμε να δούμε ως παραδείγματα κάποιους από τους κανόνες παραγωγής που έχουμε θέσει.

Επεξηγηματικά, ο πρώτος κανόνας αποτελεί το entry point της συντακτικής ανάλυσης, επομένως θα δώσει και την τιμή επιστροφής του parser, που θα είναι μία λίστα από κόμβους AllDclr. Αυτό φαίνεται από το γεγονός ότι αφενός η τιμή που κατασκευάζεται για το δεδομένο μη τερματικό σύμβολο είναι λίστα, αφετέρου από το γεγονός ότι το τερματικό σύμβολο που χρησιμοποιείται στο δεξί μέλος του colon, δηλαδή το AllDclr, παράγει κόμβους του τύπου AllDclr, καθώς το WithSign δεν αποτελεί τίποτα άλλο πάρα κατασκευαστή δεδομένων (data Constructor) τύπου AllDclr. Αντιστοίχως, ο τρίτος κανόνας δίνει μία λίστα, το περιεχόμενο της οποίας συμπερένεται από την τιμή που δίνει ο τέταρτος κανόνας. Εφόσον το Assign αποτελεί data Constructor για κόμβους Dclr, καταλήγουμε πως το τερματικό σύμβολο Dclrs παράγει κόμβους τύπου [Dclr].

Όσον αφορά, τις προϋποθέσεις που τίθενται για το αρχείο εισόδου, από τους κανονες του πίνακα 5.3 , μπορούμε να δούμε πως ένα πρόγραμμα γραμμένο σε RouReg, αποτελείται από μία σειρά δηλώσεων (AllDclr). Κάθε δήλωση συνάρτησης απαιτείται να περιέχει τον τύπου της (TypeSignature), ο οποίος ακολουθείται από ένα semicolon (;)(δεύτερος κανόνας). Αντίστοιχα κάθε ορισμός της συνάρτησης, πλην του τελευταίου, ακολουθείται από ένα semicolon. Ο τέταρτος κανόνας μας πληροφορεί για τη δομή των ορισμών των συναρτήσεων.

Σε αυτό το σημείο, προκειμένου να προβούμε σε ένα ολοκληρωμένο παράδειγμα αρχειου εισόδου (όπως αυτό του σχήματος 5.2), είναι σκόπιμο να υπογραμμίσουμε τον τελευταίο κανόνα που πρέπει να ακολουθείται, ο οποίος δεν τίθεται από τους περιορισμούς του Parser, αλλά από τη σύμβαση που κάναμε για την υλοποίηση του transformation. Ο κανόνας αυτός αφορά την υποχρεωτική ύπαρξη μίας συνάρτησης με όνομα result, η οποία θα έχει μη συναρτησιακό τύπο, δηλαδή δεν θα έχει βέλη $\rightarrow$ στην υπογραφή τύπου (Type Signature), και θα έχει ρόλο αντίστοιχο με τη main των προγραμμάτων Haskell.

Πρέπει να σημειωθεί, τέλος, πως σκοπίμως μέσω της συντακτικής ανάλυσης περιορίσαμε τις δυνατότητες της γλώσσας RouReg που θα δυσκόλευαν τον επικείμενο μετασχηματισμό, ώστε να καταφέρουμε το στόχο μας, που όπως προαναφέρθηκε είναι η απόδειξη της λογικής του εγχειρήματος. Έτσι, ενώ για παράδειγμα η γλώσσα επιτρέπει την ύπραξη δηλώσης συναρτήσεων χωρίς type signature και είναι κάτι που χρησιμοποιείται εντός του transformation, το αρχείο εισόδου επιβάλλεται να έχει type signature για όλες του τις συναρτήσεις, προς διευκόλυνση του μετασχηματισμού.

```
divExc :: Integer → Integer → Except String Integer;
divExc i j=
  case j of
    0 → throwError "Divide with zero" ;
    x → return (i `div` x)


addStateToValue :: Integer →State Integer Integer ;
addStateToValue a = get >>= (\s→ return (a+s))


totalTest ::Monad m ⇒Integer → Integer → m Integer ;
totalTest i j = (addStateToValue j) >>= (\x→divExc i x)



res ::Monad m ⇒ m Integer;
res = (fst (runState (totalTest 5 0) 0))


result :: Either String Integer ;
result = runExcept res
```

Σχήμα 5.2: Παράδειγμα αρχείου εισόδου

5.4 Το Transformation

Τη διαδικασία της λεκτικής και συντακτικής ανάλυσης ακολουθεί το transformation του αφηρημένου συντακτικού δέντρου ($AST_{initial}$), όπως φαίνεται και στο σχήμα 5.1. Αυτό που επιδιώκεται στο στάδιο αυτό, είναι ο μετασχηματισμός των κόμβων του $AST_{initial}$ για τη δημιουργία του μετασχηματισμένου AST_{transf}. Με την ύπαρξη των ASTs είναι αρκετά εύκολο να μεταβούμε από κώδικα RouReg σε κώδικα Haskell, καθώς έχουμε δημιουργήσει module Print, το οποίο δοθέντος ένος AST σε RouReg, τυπώνει το αντίστοιχο πρόγραμμα σε Haskell.

Στο κεφάλαιο αυτό, θα εξηγηθεί με λεπτομέρειες η υλοποίηση του μετασχηματισμός που ακολούθησε δύο βασικά μέρη. Το πρώτο αφορούσε το μετασχηματισμό του τύπου των συναρτήσεων και το δεύτερο το μετασχηματισμό της έκφρασης σε monadic form. Ωστόσο, τα δύο αυτά κομμάτια δεν θα μπορούσαν να είναι αναξάρτητα για λόγους που θα αναλυθούν εκτενώς παρακάτω. Ολόκληρος ο κώδικας του μετασχηματισμού, βρίσκεται στο αρχείο Transformation.hs, το οποίο χρησιμοποιείται ως module από το αρχείο Main.hs που είναι και το βασικό αρχείο που καλεί το stack.

5.4.1 Μετασχηματισμός TypeSignature

Το ζητούμενο σε αυτή τη φάση της υλοποίησης είναι η μετατροπή οποιουδήποτε τύπου συνάρτησης με τους κανόνες του πίνακα 5.4.

Αν μία συνάρτηση έχει type signature κάποιον απλό τύπο (χωρίς βέλη), τότε θα μετατρέπεται με τον πρώτο κανόνα μετασχηματισμού τύπου, ενώ αν είναι monadic θα μετατρέπεται με τον δεύτερο κανόνα. Διαφορετικά αν αποτελεί σύνθετο τύπο (τύπο που περιέχει βέλη →, ορίζοντας κάποια συναρτήση) θα μετασχηματίζεται με τον τρίτο κανόνα. Για τον δεύτερο και τρίτο κανόνα, αν τα a,b αποτελούν σύνθετους τύπους, θα πρέπει να μετασχηματιστούν επαγωγικά.

a	$\xrightarrow{Transformation}$	*Eff r a*	*,where a is a pure & non-functional type*
m a	$\xrightarrow{Transformation}$	*Eff r a*	$, \forall a, m,$ *where m is a Monad*
$a \rightarrow b$	$\xrightarrow{Transformation}$	*Eff r (a → Eff r b)*	$, \forall a, b$

Πίνακας 5.4: Κανόνες Μετασχηματισμού Τύπου

Το πιο ενδιαφέρον σημείο στον μετασχηματισμό του type signature είναι το Context του παραγόμενου τύπου, καθώς θα πρέπει να εμπλουτίζεται με τα effects που θα αποτελούν το σύνολο r, του *Eff r* monad. Έτσι, τα -μη αόριστα- monads (π.χ *State s, Except e*) που ορίζονται στον αρχικό τύπο αφενός θα αντικαθίστανται από το *Eff r* monad, όπως ορίζει ο δεύτερος κανόνας του πίνακα 5.4, αφετέρου θα δημιουργούν ένα επιπλέον Constraint της μορφής *Member t r*, όπου t είναι το monad που αντικαταστάθηκε. Παρακάτω μπορούμε να δούμε κάποια παραδείγματα μετασχηματισμών τύπου:

Παράδειγμα τύπων Συναρτήσεων

```
plus :: Integer → Integer → Integer

map :: [a] → (a→b) → [b]

addState :: (Num a) ⇒ a → State s a

ifState :: State Integer a → Except e a → m a
```

Παράδειγμα Μετασχηματισμένων Τύπων Συναρτήσεων

```
plus :: Eff r (Integer → Eff r (Integer → Eff r Integer))

map :: Eff r ([a] → Eff r (Eff r (a → Eff r b) → Eff r [b]))

addState :: (Member (State s) r, Num a) ⇒ a → Eff r a

ifState :: (Member (State Integer) r, Member (Exc e)) ⇒
Eff r (Eff r a → (Eff r (Eff r a → Eff r a)))
```

Ως εδώ, τα πράγματα στον μετασχηματισμό τύπων φαίνονται αρκετά απλά. Όμως πειραματικά προέκυψαν δύο σημαντικά προβλήματα. Το πρώτο ήταν πως όταν μία συνάρτηση *f* χρησιμοποιούσε στο σώμα της κάποια άλλη συνάρτηση *g* , η οποία περιείχε στο Context του μετασχηματισμένου τύπου της κάποιο/α effects που δεν περιέχονταν στο Context της *f*, το πρόγραμμα μας δεν περνούσε το type check. Συνεπώς, προέκυψε η ανάγκη να γνωρίζουμε της συναρτήσεις που χρησιμοποιεί η προς εξέταση συνάρτηση στο σώμα της. Το δεδομένο αυτό καταφέραμε να το κατοχυρώσουμε από τον μετασχηματισμό της έκφρασης, που εξετάζεται παρακάτω, και με το lazy evaluation της Haskell, ήταν πλέον εύκολο να συνάγουμε το επιθυμητό Context.

Το δεύτερο ζήτημα αφορούσε το γεγονός ότι κάποιο effect θα μπορούσε να μην σχετίζεται με το αποτέλεσμα, σε περίπτωση που υπάρξει κάποιο run effect (π.χ runState), αλλά πίθανώς για κάποιο arguments θα έπρεπε ακόμα να διαφαίνεται ο συσχετισμός του με το εν λόγω effect. Έτσι, ενώ διαγράφαμε το effect από τη λίστα των Constraints, χρειάστηκε να χρησιμοποιήσουμε τη δυνατότητα που μας δίνει το πακέτο των extensible effects, για εισαγωγή του Constraint, στη δήλωση του τύπου ως εξής:

Eff (x ': r) a , όπου x είναι το effect που συζητάμε.

5.4.2 Μετασχηματισμός Εκφράσεων

Ο μετασχηματισμό του σώματος των συνάρτησεων, ακολούθησε και αυτός δύο επίπεδα, για συναρτήσεις τάξεως[1] ≥ 2, που αποτέλεσαν και τα πιο σύνθετα προβλήματα. Για μία συνάρτηση *f*, το πρώτο αφορούσε τον μετασχηματισμό της σε μία ενδιάμεση συνάρτηση *f'*, η οποία ουσιαστικά θα μετέτρεπε το αποτέλεσμα της *f* σε monadic form. Έτσι, αν για παράδειγμα η *f* είχε έναν αρχικό τύπο:

$$a \rightarrow b \rightarrow c$$

, τότε η *f'*, θα ήταν τέτοια ώστε να αντιστοιχεί στον τύπο

$$a \rightarrow b \rightarrow Eff\ r\ c$$

. Ο ως άνω μετασχηματισμός, αποτέλεσε και το μεγαλύτερο κομμάτι ενδιαφέροντος της υλοποίησης της διπλωματικής και θα εξεταστεί αναλυτικά παρακάτω. Για την επεξεργασία της *f'* στην τελική, επιθυμητή, μοναδιαία μορφή χρειάστηκε να ορίσουμε μία οικογένεια συναρτήσεων, η οποία εξάγεται από τη Main.hs, και ενυπάρχει στο module InputNameMConvert. Οι συναρτήσεις αυτές, με μερική εφαρμογή (partial application) πάνω στην *f'*, μας δίνουν τους -σχεδόν- τελικούς τύπους της *f*. Για το προηγούμενο παράδειγμα, η *(mConvert2 f')* θα έχει τύπο:

$$a \rightarrow Eff\ r\ (b \rightarrow Eff\ r\ c)$$

Όπως φαίνεται, με ένα απλό *return* παίρνουμε τον τελικό τύπο:

$$Eff\ r\ (a \rightarrow Eff\ r\ (b \rightarrow Eff\ r\ c))$$

Στον πίνακα 5.5 μπορούμε να δούμε της συναρτήσεις mConvert, τάξεως ≤ 3.

```
mConvert0 :: a → Eff r a
mConvert0 = return

mConvert1 :: (t → a) → (t → Eff r a)
mConvert1 f x = return(f x)

mConvert2 :: (t2 → t1 → a) → t2 → Eff r (t1 → Eff r a)
mConvert2 f x = return $ mConvert1 $ f x

mConvert3 :: (t → t2 → t1 → a) → t → Eff r (t2 → Eff r (t1 →
Eff r a))
mConvert3 f x =return $ mConvert2 $ f x
```

Πίνακας 5.5: Συναρτήσεις της οικογένειας MConvert, τάξεως ≤ 3

Στο σημείο αυτό, ήρθε η στιγμή να εξηγήσουμε τον μετασχηματισμό των συναρτήσεων στην ενδιάμεση μορφή *f'*. Για την υλοποίηση μας, αρχικά υποθέσαμε πως υπάρχουν μόνο pure τιμες στον κώδικα που είναι να μετασχηματιστεί. Έτσι, συντάξαμε τους αρχικούς κανόνες μετασχηματισμού των κόμβων, όπως αποτυπώνονται στον πίνακα 5.6. Σημειώνεται, ότι ο συμβολισμός με απόστροφο (’),

[1] Με τον όρο ”τάξη”, αναφερόμαστε στο πλήθος των arguments μιας συνάρτησης *f*

Intial Node	Initial Expr	Transformated Expr
Apat a	a	**return** a
Str str	”str”	**return** ”str”
List exprs	[e_1, ...,e_n]	**sequence** [e_1’, ...,e_n’]
Cons e es	e:es	**cons’** e’ es’
Tuple as	(a_1, a_2)	**return** (a_1, a_2)
App e_1 e_2	g (x)	x’ »= \$x_i \rightarrow$ (g’ »= \$g_i \rightarrow$ g_i (x_i))
Let dclrs e	**let** dclrs **in** e	**let** dclrs’ **in** e’
Op binop e_1 e_2	e_1 **bop** e_2	**bop’** e_1’ e_2’
Case e cs	**case** e **of** cs	e’ »= \$e_i \rightarrow$ **case** e_i **of** cs’

Πίνακας 5.6: Πίνακας Μετασχηματισμού Εκφράσεων

χρησιμποιείται για τους μετασχηματισμένους όρους, δηλαδή η e’ προκύπτει από τον μετασχηματισμό της e.

Ο κυριότερος κανόνας, αφορά την εφαρμοργή συνάρτησης (App), καθώς εκεί είναι που εισάγονται τα monads στον επεξεργασμένο κώδικα. Όπως μπορούμε να δούμε, η εφαρμογή μίας συνάρτησης g πάνω σε μία έκφραση x, δίνεται από την εφαρμοφή της g’ στην x’, με τη διαφορά πως εφόσον οι μετασχηματισμένοι όροι (g’, x’) είναι σε μοναδιαία μορφή, για να χρησιμοποιηθούν και να δώσουν σωστό αποτέλεσμα, πρέπει να βγούν εκτός monad (unwrapping), κάτι που γίνεται με τη βοήθεια του τελεστή bind (>>=).

Monads εισάγωνται και στην περίπτωση των Case expression, όπου χρειάζεται να μετατραπεί πρώτα η έκφραση, η οποία θα διαχωρίσει τις περιπτώσεις cs’. Ομοίως με πριν, εφόσον η έκφραση e’ θα είναι σε μοναδιαία μορφή, πρέπει να χρησιμοποιήσουμε τον τελεστή bind για να την κάνουμε unwrapped. Τα cs’, θα δωθούν από τον μετατροπή καθενός εκ τών κλάδων cs, ο μετασχηματισμός των οποίων, χρειάζεται την ύπαρξη impure ορισμάτων που θα δούμε παρακάτω.

Το transformation για τα declarations (dclrs’) ενός Let expression, ακολουθεί ακριβώς την ίδια διαδικασία με αυτή του μετασχηματισμού των top level declarations, με τη διαφορά ότι στην περίπτωση αυτή δημιουργούνται κόμβοι χωρίς type signatyre.

Για τους κανόνες που αφορούν τη χρήση δυαδικών τελεστών (Op, Cons), ο συμβολισμός bop’ και cons’ αφορά τις συναρτήσεις που είναι predifend στο αρχείο που δημιουργείται nameTransfOutput.hs, και αποτελούν την μετασχηματισμένη ερμηνεία (σε monadic form) των τελεστών +, -, *, /, :. Ο κώδικας που αντιστοιχεί στις δεδομένες συναρτήσεις (plus, minus, multiply, divide, cons) βρίσκεται στο Παράρτημα Α.

Μέχρι εδώ, ο μετασχηματισμός των pure εκφράσεων ολοκληρώθηκε με επιτυχία - πλην, της εκφρασης Case of. Πάμε τώρα να εξετάσουμε τον μετασχηματισμό εκφράσεων τόσο pure όσο και impure όρων. Αρχικά θα πρέπει να τροποποιηθούν οι κανόνες που σχετίζονται με τους κόμβους Apat και App, λαμβάνοντας υπόψην τις δύο περιπτώσεις - pure και impure. Στον πίνακα 5.7 μπορούμε να δούμε τους κανόνες που χρήζουν αλλαγής. Σημειώνεται, ότι ο συμβολισμός {e} δηλώνει πως ο όρος e είναι monadic.

Επεξηγηματικά, σχετικά με τους κόμβους Apat, που σχετίζονται με μεταβλητές (και άλλους απλούς όρους όπως literals, που εν προκειμένω δεν μας απασχολούν), στην περίπτωση που εκφράζουν κάτι που είναι pure θα πρέπει να το επιστρέφουν εντός monad, με *return*, ενώ αν είναι ήδη monadic, να το επιστρέφουν ατόφυο. Για τον κανόνα της εφαρμογής συνάρτησης, μας ενδιαφέρει αν το σημείο εφαρμογής της συνάρτησης στην αρχική έκφραση είναι pure ή όχι. Στην περίπτωση που δεν είναι, σημαίνει πως η εν λόγω συνάρτηση, έστω g, έχει τύπο:

$$g :: m\ a \rightarrow \ ...$$

Intial Node	Initial Expr	Transformated Expr
Apat a	a	**return** a
	{a}	{a}
App e_1 e_2	g (x)	x' »= \$x_i \rightarrow$ (g' »= \$g_i \rightarrow$ g_i (x_i))
	g ({x})	g' »= \$g_i \rightarrow$ g_i ({x}')

Πίνακας 5.7: Πίνακας Μετασχηματισμού Εκφράσεων για Pure και Impure Όρους

Έτσι, παρότι χρειάζεται να μετατρέψουμε τόσο την g, όσο και την x[2], σε monadic form, η x' δεν πρέπει να γίνει unwrapped, καθώς η g', ομοίως με τη g, περιμένει κάποιο monad και όχι κάποια αγνή τιμή.

Για την υλοποίηση της παραπάνω συλλογιστικής, χρειάστηκε να χρησιμοποιήσουμε τον αρχικό τύπο της συνάρτησης που θέλουμε να εφαρμόσουμε. Με βάση αυτόν, ήταν πλέον εύκολο να αντιληφθούμε σε ποιά περίπτωση βρισκόμαστε και να εφαρμόσουμε τον αντίστοιχο κανόνα μετασχηματισμού.

5.5 Παραδείγμα Transformation

Ήρθε η στιγμή να δούμε κάποια παραδείγματα, που αναδεικνύουν τη χρησιμότητα του εργαλείου που κατασκευάσαμε. Θα ξεκινήσουμε με το αρχείου εισόδου του σχήματος 5.2, αναλύοντας αρχικά τη σημασιολογία του. Η συνάρτηση $divExc$ αποσκοπεί στη διαίρεση δύο ακεραίων, εξετάζοντας την πιθανότητα ο διαιρέτης να είναι μηδέν. Αν συμβεί αυτό, τυπώνει ένα κατάλληλο μήνυμα σφάλματος, ενώ σε αντίθετη περίπτωση επιστρέφει το αποτέλεσμα της διαίρεσης των δύο αριθμών. Η $addStateToValue$ είναι η συνάρτηση που παίρνει έναν ακέραιο και του προσθέτει την τιμή ενός μετρητή (state), ενώ η $totalTest$ συνδέει τις $divExc$ και $addStateToValue$, ώστε να παράξει τη διαίρεση ενός ακεραίου με τον αριθμό που προκύπτει από το άθροισμα ενός άλλου ακεραίου με τον μετρητή. Η res δίνει ένα παράδειγμα εκτέλεσης της $totalTest$ με αρχική τιμή μετρητή το μηδέν και αριθμούς το 5 (διαιρετέος) και 0 (το άθροισμα του με τον μετρητή δίνει τον διαιρέτη).

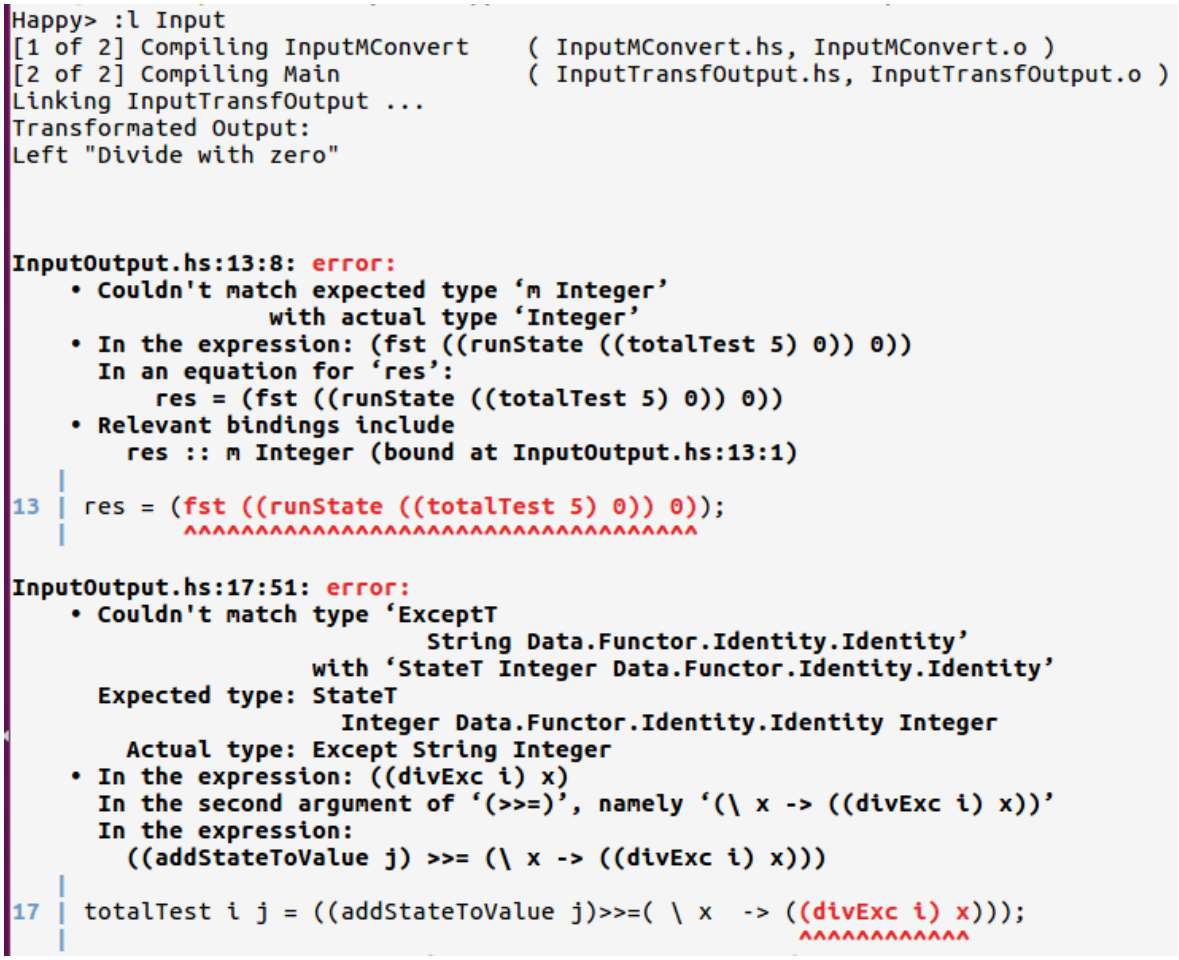
Παρακάτω μπορούμε να δούμε τα αποτελέσματα της εφαρμογής:

Όπως φαίνεται το αρχείο InputTransfOutput.hs τρέχει κανονικά για τον μετασχηματισμένο κώδικα, παράγοντας την επιθυμητή εξαίρεση (Left "Divide with zero"). Αντιθέτως, το ανεπεξέργαστο πρόγραμμα Haskell με όνομα InputOutput.hs, δίνει δύο σφάλματα. Το κυριότερο από αυτά είναι το σφάλμα της γραμμής 17, το οποίο, όπως θα περιμέναμε, δηλώνει πως δεν υπάρχει συμβατότητα των δύο συναρτήσεων που χρησιμοποιούνται καθώς η μία σχετίζεται με το State ενώ άλλη με το Exception monad. Το σφάλμα της γραμμής 13, είναι απόρροια του σφάλματος στη γραμμή 17.

Ένας διαφορετικός τρόπος να γράψουμε το ίδιο (σημασιολογικά) πρόγραμμα σε RouReg φαίνεται στο σχήμα 5.4. Στο παράδειγμα αυτό μπορούμε να παρατηρήσουμε πως πέρα από τη σύνθεση διαφορετικών impure χαρακτηριστικών, το framework που αναπτύξαμε επιτρέπει τη χρήση monadic στοιχείων, εκεί που μέχρι τώρα επιτρέπονταν μόνο pure τιμές. Έτσι η συνάρτηση get με τύπο State s s, μπορεί να χρησιμοποιηθεί χωρίς τη χρήση bind (δηλαδή εκτός do block) στη συνάρτηση αθροίσματος "+". Το ίδιο συμβαίνει και για τη συνάρτηση $addStateToValue$ η οποία χρησιμοποιείται ως το δεύτερο όρισμα της $divExc$, που όπως φαίνεται από τον τύπο της δέχεται ως ορίσματα ακεραίους (pure τιμές).

Προς απόδειξη της παραπάνω θεώρησης στο σχήμα 5.5 θα δείξουμε τα αποτελέσματα από την εκτέλεση του κώδικα του σχήματος 5.4. Όπως βλέπουμε το αρχείο InputTransfOutput.hs τρέχει κανονικά, ενώ ο compiler παραπονιέται και πάλι για το InputOutput.hs. Αυτή τη φορά μάλιστα, προβάλλει και ως σφάλματα (γρ. 13 και 17) τη χρήση υπολογισμών, σε θέσεις που αναμένονταν Integers.

[2] Το ότι το x είναι monadic, δεν σημαίνει πως είναι σε monadic form. Για παράδειγμα, *x:: m (a → b)*



Σχήμα 5.3: Εκτέλεση αρχείου εισόδου 5.2

Στο σημείο αυτό θα προβούμε στην ανάλυση ενός πιό σύνθετου παραδείγματος, προς ενίσχυση των αποτελεσμάτων μας. Έστω ότι έχουμε μία λίστα ακεραίων που θέλουμε να μην περιέχει μηδενικά στοιχεία. Αν το παραπάνω κριτήριο ικανοποιείται, θέλουμε ακόμα να προκύπτει εξαίρεση σε περίπτωση που το συνολικό άθροισμα των στοιχείων της λίστας είναι ακριβώς δέκα. Το πρόγραμμα του σχήματος 5.6 αποτυπώνει μία λύση για το παραπάνω πρόβλημα.

Για τη λίστα [3,3,4] που δίνεται ως είσοδο, η εκτέλεση μας δίνει ως αποτέλεσμα το: (Left ”Total sum is exactly ten”), καθώς το άθροισμα των στοιχείων της λίστας είναι ακριβώς ίσο με 10. Για μία λίστα όπως η [1,5,0,0] θα παίρναμε ως αποτελεσμα το: (Left ”The list contains at least one zero element”), ενώ την ίδια απάντηση θα παίρναμε και για μία λίστα όπως η [0,5,5], καθώς το πρωταρχικό κριτήριο είναι η λίστα να μην περιέχει μηδενικά. Τέλος για μία λίστα σαν την [2,7] θα παίρναμε ως αποτέλεσμα το (Right 9), εφόσον θα πληρούνταν ταυτοχρόνως τα δύο κριτήρια που αναφέραμε.

Αξίζει να επισημάνουμε, εδώ, πως συναρτήσεις όπως η put:: s -> State s (), που δίνουν μεν monadic αποτέλεσμα, αλλά έχουν τιμή επιστροφής Void χρειάζεται να βρίσκονται εντός ενός do block ή να χρησιμοποιούν τον bind operator. Διαφορετικά θα μπορούσαν να δεσμεύσουν κάποια ”εικονική” μεταβλητή που δεν θα χρησιμοποιηθεί ποτέ. Θα υπάρχει μονάχα για να αποτυπώνει το action που μένει να εκτελεστεί μετά το transformation. Αυτό συμβαίνει γιατί στη Haskell δεν υπάρχει σύνταξη,πέρα από το bind (και do), για εκφράσεις που αντιπροσωπεύουν μονάχα actions.

```
divExc:: Integer → Integer → Except String Integer;
divExc i j=
  case j of
    0 → throwError "Divide with zero" ;
    x → return (i `div` x)

addStateToValue:: Integer →State Integer Integer ;
addStateToValue a = return (a+get)

totalTest::Monad m⇒Integer → Integer → m Integer ;
totalTest i j = divExc i (addStateToValue j)

res::Monad m⇒ m Integer;
res = (fst (runState (totalTest 5 0) 0))

result:: Either String Integer ;
result = runExcept res
```

Σχήμα 5.4: Τροποποιημένος κώδικας για το παράδειγμα αρχείου εισόδου του σχήματος 5.2

```
Happy> :l Input
[1 of 2] Compiling InputMConvert    ( InputMConvert.hs, InputMConvert.o )
[2 of 2] Compiling Main             ( InputTransfOutput.hs, InputTransfOutput.o )
Linking InputTransfOutput ...
Transformated Output:
Left "Divide with zero"



InputOutput.hs:9:32: error:
    • Couldn't match expected type ‘Integer’ with actual type ‘m0 s0’
    • In the second argument of ‘(+)’, namely ‘get’
      In the first argument of ‘return’, namely ‘(a + get)’
      In the expression: (return (a + get))
  |
9 | addStateToValue a = (return (a+get));
  |                                ^^^

InputOutput.hs:13:8: error:
    • Couldn't match expected type ‘m Integer’
                  with actual type ‘Integer’
    • In the expression: (fst ((runState ((totalTest 5) 0)) 0))
      In an equation for ‘res’:
          res = (fst ((runState ((totalTest 5) 0)) 0))
    • Relevant bindings include
        res :: m Integer (bound at InputOutput.hs:13:1)
  |
13 | res = (fst ((runState ((totalTest 5) 0)) 0));
   |        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

InputOutput.hs:17:30: error:
    • Couldn't match type ‘StateT
                             Integer Data.Functor.Identity.Identity Integer’
                      with ‘Integer’
      Expected type: Integer
        Actual type: State Integer Integer
    • In the second argument of ‘divExc’, namely ‘(addStateToValue j)’
      In the expression: ((divExc i) (addStateToValue j))
      In an equation for ‘totalTest’:
          totalTest i j = ((divExc i) (addStateToValue j))
  |
17 | totalTest i j = ((divExc i) (addStateToValue j));
   |                              ^^^^^^^^^^^^^^^^^
```

Σχήμα 5.5: Εκτέλεση αρχείου εισόδου 5.4

```
map' :: (a → b) → [a] → [b] ;
map' f [] = [] ;
map' f (x:xs) = (f x) : (map' f xs)

checkNum:: Integer → Except String Integer;
checkNum num =
     case num of
      0 → throwError "The list contains at least one zero element" ;
      x → x

checkSum:: Integer → Except String Integer;
checkSum sum =
    case sum of
     10 → throwError "Total sum is exactly ten" ;
     x  → x

sumOfAList:: [Integer]↦ State Integer Integer;
sumOfAList [] = get;
sumOfAList (x:xs) =  put (x+get)>>= (\_ → (sumOfAList xs))


res2 ::Monad m⇒ m Integer;
res2 = fst (runState (checkSum (sumOfAList
       (map' checkNum [3,3,4]))) 0)


result:: Either String Integer ;
result = runExcept res2
```

Σχήμα 5.6: Σύνθετο παράδειγμα αρχείου εισόδου

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

Σε αυτήν την διπλωματική εργασία καταφέραμε να επιτύχουμε τον αρχικό μας στόχο, δηλαδή τη δημιουργία ενός εργαλείου που θα διευκολύνει τη χρήση των monads. Ο μετασχηματισμός που επιδιώξαμε συνεισέφερε σε δύο βασικά σημεία.

Το πρώτο σχετίζεται με τη σύνθεση impure χαρακτηριστικών, που αποτελεί μία ιστορική δυσκολία στον κόσμο της Haskell. Με χρήση των Extensible Effects καταφέραμε να εκφράσουμε προγράμματα που περιλαμβάνουν διαφορετικής φύσεως προστακτικά γνωρίσματα. Το *Eff r* monad μας έδωσε τη δυνατότητα να προσθέτουμε όλα τα επιθυμητά effects σε μία μη διατεταγμένη δομή, που ονομάζεται open union. Έτσι καταφέραμε να αποδώσουμε με απλό και κομψό προγραμματιστικό τρόπο την ερμηνεία πολύπλοκων προβλημάτων που ήταν δύσκολο ή και αδύνατο να αναπαρασταθούν με τα άλλα μοντέλα που εξετάστηκαν.

Το δεύτερο σημείο στο οποίο καταφέραμε να επιτύχουμε πρόοδο είναι ο ίδιο ο τρόπος χρήσης των monad, ο οποίος αποτελεί αγκάθι για πολλούς προγραμματιστές. Συγκεκριμένα, ο μετασχηματισμένος κώδικας που θα προκύπτει από την επεξεργασία της εφαρμογής που κατασκευάσαμε θα είναι μία μετατροπή του αρχικού προγράμματος σε monadic form. Με τη μορφή αυτή επιτυγχάνουμε τα αποτελέσματα όλων των μετασχηματισμένων συναρτήσεων να είναι monadic. Θεωρώντας πως το αποτέλεσμα κάθε συνάρτησης είναι ένα do block, μπορούμε να χρησιμοποιούμε εντός του τα values που - υπό άλλες συνθήκες - θα περικλείονταν σε κάποιον impure υπολογισμό (και θα έπρεπε να κάνουμε unwrap με τη βοήθεια του bind), ως pure τιμές.

6.2 Σχετικές Εργασίες - Μελλοντική Έρευνα

Η βιβλιογραφική αναζήτηση που κάναμε δεν μας έδωσε κάποιο στοιχείο για παρακείμενες προσπάθειες, ως προς την επιδίωξη ενός πιο εύχρηστου πλαισίου που να διευκολύνει πρακτικά τη χρήση των monads. Ωστόσο, παρατηρήσαμε να υπάρχει μεγάλο ενδιαφέρον της επιστημονικής κοινότητας, για τη δημιουργία όλο και πιο πλούσιων και αποδοτικών μοντέλων που απασκοπούν στην ταυτόχρονη αλληλεπίδραση με πολλαπλά προστακτικά χαρακτηριστικά. Ενδεικτικά, αναφέρουμε την ιδέα για σύνθεση των monads με χρήση των coproducts,[Lüt02] η οποία πυροδότησε την κατασκευή των free monads. [Swie08] Μετέπειτα έρευνες, παρουσίασαν έναν συνδιασμό των open union με τα free monads για την κατασκευή του αλγεβρικού τύπου των freer monads, άροντας ακόμη και καθολικούς, εώς τότε, περιορισμούς της monad Class (π.χ τον constraint Functor). Περισσότερη ελευθερία στα monads, θεωρήθηκε πως βελτίωσε την αποδοτικότητα των Extensible Effects.[Kise15]

Θεωρούμε πως τα παραπάνω μοντέλα θα ήταν πιθανό να χρησιμοποιηθούν προς όφελος ενός πιο αποδοτικού και γενικευμένου μετασχηματισμού. Επίσης, θα είχε μεγάλο ενδιαφέρον η, κάθε αυτή, διεύρυνση του μετασχηματισμού για ολόκληρη τη γλώσσα Haskell, ώστε να μπορέσει να αποτελέσει ένα πρακτικό εργαλείο για τους προγραμματιστές. Σημαντικό βήμα θα ήταν, τέλος, ο μετασχηματισμός να ήταν επεκτάσιμος (δηλαδή, να αποτελούσε μέρος του προγράμματος χρήστη), ώστε να εφαρμόζεται και σε monad που δεν είναι εξαρχής ορισμένα, αλλά κατασκευάζονται από τον ίδιο τον χρήστη.

Βιβλιογραφία

[Back78] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, vol. 21, 1978.

[Bird95] Richard Bird and Philip Wadler, *Introduction to Functional Programming*, Prentice Hall, 1995.

[Cart94] Robert Cartwright and Matthias Felleisen, "Extensible Denotational Language Specifications", in *SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SOFTWARE, NUMBER 789 IN LNCS*, pp. 244–272, Springer-Verlag, 1994.

[CDor03] I.Jones C. Dornan and S. Marlow, *Alex User Guide*, 2003.

[Coli93] David Wakeling Colin Runciman, "Heap Profiling of Lazy Functional Programs", *Journal of Functional Programming*, vol. 3, pp. 217–245, 1993.

[Cunn14] H. Conrad Cunningham, "Notes on Functional Programming with Haskell". unpubliched, 2014.

[Davi92] Antony J. T. Davie, *Introduction to Functional Programming Systems Using Haskell*, Cambridge University Press, 1992.

[Grab06] Martin Grabmüller, "Monad transformers step by step", Technical report, 2006.

[Hinz00] Ralf Hinze, "Deriving Backtracking Monad Transformers", in *In The International Conference on Functional Programming (ICFP*, pp. 186–197, ACM Press, 2000.

[Hugh89] John Hughes, "Why Functional Programming Matters", *The Computer Journal*, vol. 23, pp. 98—107, 1989.

[Jone93] Mark P. Jones and Luc Duponcheel, "Composing Monads", Technical report, Yale University, New Haven, Connecticut, USA, 1993.

[Jone01] Simon Peyton Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell", in *Engineering theories of software construction*, pp. 47–96, Press, 2001.

[King93] David J King and Philip Wadler, "Combining Monads", in John Launchbury and Patrick Sansom, editors, *Workshops in Computing*, 1993.

[Kise13] Oleg Kiselyov, Amr Sabry and Cameron Swords, "Extensible Effects An Alternative to Monad Transformers", in *Haskell '13 Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pp. 59–70, ACM New York, NY, USA, 2013.

[Kise15] Oleg Kiselyov and Hiromi Ishii, "Freer monads, more extensible effects", in *Haskell '15 Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pp. 94–105, 2015.

[Laun92] John Launchbury and Patrick Sansom, "Functional Programming, Glasgow 1992", in *Workshops in Computing*, 1992.

[Lüt02] Christoph Lüth and Neil Ghani, "Composing Monads Using Coproducts", in *ICFP '02 Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pp. 133–144, 2002.

[Marl01] S. Marlow and A. Gill, *Happy User Guide*, 2001.

[Nick00] John Hughes Nick Benton and Eugenio Moggi, "Monads and Effects", in *Proceeding Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, Advanced Lectures*, pp. 42–122, 2000.

[Peyt93] Simon Peyton and Jones Philip Wadler, "Imperative Functional Programming", 1993.

[Schr11] Tom Schrijvers and Bruno C. d. S. Oliveira, "Monads, Zippers and Views: Virtualizing the Monad Stack", *ACM SIGPLAN Notices - ICFP '11*, vol. 46, pp. 32–44, 2011.

[Seba11] Oleg Kiselyov Sebastian Fischer and Chung chieh Shan, "Purely Functional Lazy Non-deterministic Programming", *Journal of Functional Programming*, vol. 21, pp. 413–465, 2011.

[Swie08] Wouter Swierstra, "Data types à la carte", *Journal of Functional Programming*, vol. 18, pp. 423–436, 2008.

[Thom11] Simon Thompson, *Haskell: The Craft of Functional Programming*, Addison-Wesley Professional, 2011.

[Wadl92] Philip Wadler, "The Essence of Functional Programming", in *POPL '92 Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 1–14, Prentice Hall, 1992.

[Κα] Γιάννης Κασσιός, "Εισαγωγή στο Συναρτησιακό Προγραμματισμό". unpublished.

[Στ15] Παναγιώτης Σταματόπουλος, *Λογικός και Συναρτησιακός Προγραμματισμός*, Εκδόσεις Κάλλιπος, 2015.

Παράρτημα Α

Predefined Συναρτήσεις & Δομή του Μετασχηματισμένου Κώδικα

```
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE MonoLocalBinds #-}
{-#LANGUAGE FlexibleContexts , TypeOperators , DataKinds #-}

import MConvert
import Control.Eff
import Control.Monad
import Control.Eff.State.Lazy
import Control.Eff.Exception
import Data.Tuple.Sequence


plus::(Num a)=> Eff r (a->Eff r (a-> Eff r a))
plus =
 let
   plus' x y = return (x+y)
 in
   return (mConvert1 plus')

sub::(Num a)=> Eff r (a->Eff r (a-> Eff r a))
sub =
 let
   sub' x y = return (x-y)
 in
   return (mConvert1 sub')

multiply::(Num a)=> Eff r (a->Eff r (a-> Eff r a))
multiply =
 let
   multiply' x y = return (x*y)
 in
   return (mConvert1 multiply')

divide:: (Integral a, Num a) => Eff r (a->Eff r (a-> Eff r a))
divide =
 let
   divide' x y = return (x `div` y)
 in
   return (mConvert1 divide')
```

```
cons:: Eff r (a→ Eff r ([a]→Eff r [a]))
cons =
  let
    cons' x xs = return (x:xs)
  in
    return (mConvert1 cons')

main::IO ()
main= putStrLn $show $run $result
```