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

Ο Μετασχηματιστής Μονάδων Επανόδου και η υλοποίησή του σε JavaScript

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΣΑΚΚΑΣ

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

Αθήνα, Ιούλιος 2018

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

Ο Μετασχηματιστής Μονάδων Επανόδου και η υλοποίησή του σε JavaScript

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΣΑΚΚΑΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 6η Ιουλίου 2018.

Νικόλαος Παπασπύρου	Γεώργιος Γκούμας	Αριστείδης Παγουρτζής
Αναπληρωτής Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2018

Γεώργιος Σακκάς

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

Abstract

Resumptions are a valuable tool in the analysis and design of semantic models for *concurrent programming languages*, in which computations consist of sequences of atomic steps that may be interleaved. Briefly, a resumption is either a computed value of some domain or an atomic computation that produces a new resumption. In an appropriate category of semantic domains we define a monad transformer R which, given an arbitrary monad M that represents the atomic computations, constructs a monad $\mathsf{R}(M)$ for interleaved computations. What is more, monad transformers allow monads to be constructed in a "layered" fashion. This allowed us in our work, if, for example, D is the state monad transformer, to implement the direct semantics approach, where $\mathsf{R}(M)(D)$ will be a monad aware of a state and resumptions. Therefore, if M is chosen to be the power-domain monad P (this can be the list monad in Haskell in order to accumulate results) we can define the semantics of concurrent programming languages where the interleaving of computations is allowed.

Furthermore, we use our introduced *Resumption Monad Transformer* (RMT) to define the *denotational semantics* of a simple imperative language featuring non-determinism and concurrency. This language is being implemented on *JavaScript* and is actually a subset of it. We then define our own version of monads and monad transformers in *JavaScript* in order to implement resumption monad transformers efficiently. Additionally, we can now define new operators that let us extend this simple language and insert non-determinism and interleaving of computations into it. Our goal is to evaluate the results of RMTs on a single-threaded language, that doesn't support concurrency natively, and show that they can be a low-overhead and very expressive approach for the semantics of concurrency. We did tests in *JavaScript* to prove these points, where programs are written on plain *JavaScript* and in our language that uses RMTs. We also compare their performance against the JavaScript *Promises*, whose semantics resemble those of our resumption monad transformer.

Key words

Programming Languages, Concurrent Programming, Denotational Semantics, Functional Programming, Monads, Resumptions, Monad Transformers, JavaScript.

Περίληψη

Οι *επάνοδοι* (resumptions) είναι ένα πολύτιμο εργαλείο για την ανάλυση και το σχεδιασμό σημασιολογικών μοντέλων για *ταυτόχρονες γλώσσες προγραμματισμού*, στους οποίους οι υπολογισμοί αποτελούνται από αλληλουχίες ατομικών βημάτων (atomic steps) που μπορεί να παρεμβληθούν μεταξύ τους. Εν συντομία, μία επάνοδος είναι είτε μια υπολογισμένη τιμή κάποιου πεδίου (domain) είτε ένας ατομικός υπολογισμός (atomic computation) που παράγει μια νέα επάνοδο. Σε μια κατάλληλη κατηγορία σημασιολογικών πεδίων ορίζουμε έναν μετασχηματιστή μονάδων (monad transformer) R, ο οποίος, δεδομένου μιας αυθαίρετης μονάδας (monad) M που αντιπροσωπεύει τους ατομικούς υπολογισμούς, κατασκευάζει μία μονάδα $\mathsf{R}(M)$ για διαφυλλώμενους υπολογισμούς (interleaved computations). Επιπλέον, οι μετασχηματιστές μονάδων επιτρέπουν στις μονάδες να κατασκευάζονται με "στρώσεις". Αυτό μας επέτρεψε στη δουλειά μας, αν, για παράδειγμα, το D είναι ο μετασχηματιστής μονάδων κατάστασης (state monad transformer), να εφαρμόσουμε την προσέγγιση της άμεσης σημασιολογίας (direct semantics approach), όπου $\mathsf{R}(M)(D)$ θα είναι μία μονάδα που θα λαμβάνει υπόψη την κατάσταση του προγράμματος (state) αλλά και των επανόδων. Επομένως, αν το M επιλέξει είναι η μονάδα δυναμοπεδίου P (αυτή μπορεί να είναι η μονάδα λίστας (list monad) στη Haskell για να συγκεντρώνει όλα τα αποτελέσματα) μπορούμε να ορίσουμε τη σημασιολογία των παράλληλων γλωσσών προγραμματισμού στις οποίες επιτρέπεται η παρεμβολή των διάφορων υπολογισμών.

Επιπλέον, χρησιμοποιούμε τον δικό μας ορισμένο μετασχηματιστή μονάδων για να ορίσουμε τη *δηλωτική σημασιολογία* (denotational semantics) μιας απλής προστακτικής γλώσσας που χαρακτηρίζεται από μη-ντετερμινισμό και ταυτοχρονισμό. Αυτή η γλώσσα υλοποιήθηκε στη *JavaScript* και είναι στην πραγματικότητα ένα υποσύνολο της. Στη συνέχεια, καθορίζουμε τη δική μας εκδοχή των μονάδων και των μετασχηματιστών μονάδων στη *JavaScript* προκειμένου να υλοποιήσουμε αποτελεσματικά τους μετασχηματιστές μονάδων επανόδου. Επιπλέον, μπορούμε τώρα να ορίσουμε νέους τελεστές (operators) που μας επιτρέπουν να επεκτείνουμε αυτή την απλή γλώσσα και να εισάγουμε μη-ντετερμινισμό και διαφυλλισμό των υπολογισμών (interleaving of computation) σε αυτή. Ο στόχος μας είναι να αξιολογήσουμε τα αποτελέσματα των μετασχηματιστών μονάδων σε μία γλώσσα ενός νήματος (single-threaded language), η οποία δεν υποστηρίζει ταυτοχρονισμό εγγενώς, και να δείξουμε ότι μπορεί να είναι μια χαμηλού κόστους και πολύ εκφραστική προσέγγιση για τη σημασιολογία του ταυτοχρονισμού. Κάναμε πειράματα σε *JavaScript* για να δείξουμε αυτά τα σημεία, όπου τα προγράμματα γράφονται σε απλή *JavaScript* και στη δική μας γλώσσα που χρησιμοποιεί το *μετασχηματιστή μονάδων επανόδου* (RMTs). Επίσης, συγκρίνουμε την απόδοσή τους με τα *Promises* της JavaScript, η σημασιολογία των οποίων μοιάζει με αυτή των μετασχηματιστών μονάδων επανόδου.

Λέξεις κλειδιά

Γλώσσες Προγραμματισμού, Ταυτόχρονος Προγραμματισμός, Δηλωτική Σημασιολογία, Συναρτησιακός Προγραμματισμός, Μονάδες, Επάνοδοι, Μετασχηματιστές Μονάδων, JavaScript.

Ευχαριστίες

Πρώτα απ ’όλα, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Νικόλαο Παπασπύρου για την καθοδήγησή του κατά τη διεξαγωγή της διπλωματικής μου. Ο κ. Παπασπύρου υπήρξε πραγματική έμπνευση για μένα για να συνεχίσω τα όνειρά μου στην έρευνα.

Επιπρόσθετα, θα ήθελα να ευχαριστήσω τον κ. Γεώργιο Γκούμα και τον κ. Άρη Παγουρτζή για την συμμετοχή τους στην επιτροπή εξέτασης της διπλωματικής μου καθώς και για τα εμπνευσμένα μαθήματα που μας πρόσφεραν στο Εθνικό Μετσόβιο Πολυτεχνείο (ΕΜΠ).

Θα ήθελα επίσης να εκφράσω την εκτίμησή μου και την ευγνωμοσύνη μου για τους γονείς μου που με στήριξαν τόσο διανοητικά όσο και οικονομικά κατά την διάρκεια του ακαδημαϊκού μου ταξιδιού. Χωρίς τη βοήθειά τους και την απεριόριστη υποστήριξη τους, αυτό το ταξίδι δεν θα ήταν δυνατό.

Τέλος, θα ήθελα να ευχαριστήσω όλους τους φίλους μου για το χρόνο που περάσαμε μαζί και την ενθάρρυνση που μου παρείχαν όλα αυτά τα χρόνια για να συνεχίσω και να επιτύχω τους στόχους μου.

Γεώργιος Σακκάς,

Αθήνα, 6η Ιουλίου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-18, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2018.

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

Περιεχόμενα

Κατάλογος πινάκων

Κατάλογος σχημάτων

Κατάλογος αλγορίθμων

Κεφάλαιο 1

Εισαγωγή

1.1 Ταυτοχρονισμός και ταυτόχρονες γλώσσες προγραμματισμού

1.1.1 Ταυτοχρονισμός και ταυτόχρονα συστήματα

Οι παράλληλοι υπολογιστές είναι τα κυρίαρχα συστήματα σήμερα. Τα συστήματα υπολογιστών που παρέχουν μόνο μία και μόνο ροή υπολογισμών, όπως οι επεξεργαστές ενός πυρήνα, έχουν πλέον γίνει λιγοστά. Τα παράλληλα συστήματα είναι το πιο προφανές παράδειγμα των ταυτόχρονων συστημάτων (concurrent systems). Τα ταυτόχρονα συστήματα είναι εκείνα που επιτρέπουν πολλές διαφορετικές εφαρμογές ή δραστηριότητες, που ονομάζονται *νήματα* (threads) στο πεδίο της ταυτοχρονισμού, να εκτελούνται ταυτόχρονα. Αυτά τα νήματα συνήθως πρέπει να επικοινωνούν μεταξύ τους για να ολοκληρώσουν σωστά το συνολικό έργο. Η θεωρία του *ταυτοχρονισμού* (concurrency) περιλαμβάνει τη μελέτη τέτοιων συστημάτων επικοινωνίας και τα μοντέλα που χρησιμοποιούνται για το λόγο αυτό.

Επομένως, τα ταυτόχρονα συστήματα είναι πιο περίπλοκα από τα σειριακά (sequential) αντίστοιχά τους να τα εξεταστούν και να μελετηθούν. Σε μια σειριακή εκτέλεση προγράμματος, κάθε υπολογισμός - δηλ. μια βασική λειτουργία - θα γίνει ”μία τη φορά”, ενώ σε μια παράλληλη εκτέλεση πολλοί υπολογισμοί μπορεί να συμβούν ”μαζί”. Αυτό σημαίνει ότι κάθε εκτέλεση του προγράμματος έχει την προσωπική του *κατάσταση* (state) - δηλαδή τις μεταβλητές και τις τιμές τους, ενώ σε μια παράλληλη εκτέλεση κάθε νήμα μπορεί να έχει (περισσότερο ή λιγότερο) μία ανεξάρτητη κατάσταση. Αλλά αυτές οι καταστάσεις των νημάτων πρέπει απαραίτητα να επικοινωνούν μεταξύ τους όπως προαναφέρθηκε. Συνεπώς, μπορεί να προκύψουν διάφοροι συνδυασμοί αυτών. Ο αριθμός των συνδυασμών όλων των καταστάσεων αυξάνεται εκθετικά με τον αριθμό των νημάτων σε σύγκριση με τη γραμμική ανάπτυξη των σειριακών. Τέτοια προβλήματα περιγράφονται στο [Bust90].

Τα ταυτόχρονα συστήματα μπορεί επίσης να εμφανίζουν *μη-ντετερμινισμό* (nondeterminism), πράγμα που συμβαίνει στην περίπτωση της ταυτόχρονης εκτέλεσης του ίδιου κώδικα με την ίδια είσοδο αλλά με αποτέλεσμα διαφορετικές εξόδους. Επομένως, τα μη-ντερερμινιστικά συστήματα είναι κατ ’αρχήν ασταθή και μόνο με αυστηρά μαθηματική λογική και συλλογιστική μπορούμε να προβλέψουμε τις ντετερμινιστικές ιδιότητες τέτοιων συστημάτων.

Ένα ταυτόχρονο σύστημα μπορεί επίσης να είναι σε *αδιέξοδο* (deadlocked), πράγμα που συμβαίνει με την ταυτόχρονη εκτέλεση ενός προγράμματος που δεν οδηγεί σε κανένα νήμα να κάνει κάποια πρόοδο λόγω της αναμονής για επικοινωνία μεταξύ τους, αλλά τίποτα προς αυτό δεν συμβαίνει. Το αδιέξοδο είναι ένα από τα βασικά προβλήματα κατά την αντιμετώπιση των ταυτόχρονων συστημάτων.

Μια άλλη κακή συμπεριφορά που μπορεί να παρουσιάσει ένα παράλληλο σύστημα είναι *livelock*. Εξωτερικά μπορεί να φαίνεται σαν αδιέξοδο αφού δεν σημειώνεται πραγματική πρόοδος, αλλά το livelock είναι βασικά μια απεριόριστη αδιάσπαστη ακολουθία εσωτερικών υπολογισμών.

Η κατανόηση τέτοιων προβλημάτων στα ταυτόχρονα συστήματα είναι απαραίτητη για την ανάλυση του ταυτοχρονισμού και την εξεύρεση τυπικών μεθόδων για αυτό. Δεδομένου ότι τα μαθηματικά μοντέλα και οι τεχνικές τεχνολογίας λογισμικού που έχουν σχεδιαστεί για σειριακά συστήματα είναι

συνήθως ανεπαρκείς για τη μοντελοποίηση των λεπτομερειών του ταυτοχρονισμού, πρέπει να αναπτυχθούν διαφορετικές για το σκοπό του ταυτοχρονισμού.

1.1.2 Ταυτόχρονες γλώσσες προγραμματισμού και η σημασιολογία τους

Οι γλώσσες προγραμματισμού που επιτρέπουν στο χρήστη να χρησιμοποιεί συστήματα όπως τα προαναφερθέντα ταυτόχρονα, μπορούν να οριστούν ως *ταυτόχρονες γλώσσες προγραμματισμού* (concurrent programming languages). Αυτές οι γλώσσες έχουν συνήθως ενσωματωμένες λειτουργίες που επιτρέπουν στους χρήστες να γράφουν αβίαστα παράλληλα προγράμματα, όπως στο Cilk (όπου ο παράλληλος προγραμματισμός μπορεί να θεωρηθεί ως υποσύνολο του ταυτόχρονου προγραμματισμού) [Blum96] ή την Erlang [Arms93]. Αυτές οι ταυτόχρονες λειτουργίες μπορεί να περιλαμβάνουν πολλαπλά νήματα, πέρασμα μηνυμάτων, κοινόχρηστους πόρους (συμπεριλαμβανομένης της μνήμης κοινής χρήσης) ή συμβόλαια μελλοντικής εκπλήρωσης και υποσχέσεις. Πολλές τεχνικές έχουν αξιοποιηθεί για να επισημοποιήσουν μαθηματικά αυτές τις λειτουργίες και να ορίσουν τη σημασιολογία των ταυτόχρονων γλωσσών προγραμματισμού [dBak96], [Rosc97]. Ένα από αυτά τα επίσημα σημασιολογικά μοντέλα τα οποία έχουν μακρά μελετηθεί [Mogg90], [dBak96], [Papa00], [Papa01], [Harr04] ως μοντέλο διαφυλλώμενων υπολογισμών για τη σημασιολογία των παράλληλων γλωσσών προγραμματισμού είναι οι *επάνοδοι* (resumptions).

Σε αυτή την εργασία καθορίσαμε ένα *Μετασχηματιστής Μονάδων Επανόδου* (Resumption Monad Transformer - RMT) και τον υλοποιήσαμε στην *JavaScript* παράλληλα με μια απλή προστακτική γλώσσα για να καθορίσουμε την απόδοσή του. Τα RMTs εισήχθησαν από τον [Mogg90] ως μέρος μιας ενοποιημένης προσέγγισης της δηλωτικής σημασιολογίας των γλωσσών προγραμματισμού που βασίζονται σε μονάδες (monads).

1.1.3 Ταυτοχρονισμός σε JavaScript

Η *JavaScript* σε σχεδόν όλες τις υλοποιήσεις είναι μονονηματική (single-threaded) και δεν υποστηρίζει ταυτοχρονισμό εγγενώς. Κάποια περιβάλλοντα ωστόσο επιτρέπουν στη JavaScript να εκμεταλλευτεί τον ταυτοχρονισμό όπως αναφέρεται και στο [Nami15].

Μια από τις πιο συνηθισμένες περιπτώσεις σήμερα στον ασύγχρονο υπολογισμό για JavaScript είναι οι *Web Workers*. Ένας Web Worker είναι βασικά ένα νήμα που τρέχει σε JavaScript στο παρασκήνιο, φέρνοντας το actor style [Bake77] στο διαδίκτυο. Το νήμα του worker μπορεί να εκτελέσει εργασίες χωρίς να παρεμβαίνει στη διεπαφή χρήστη. Μόλις δημιουργηθεί, ένας worker μπορεί να στείλει μηνύματα στον κώδικα JavaScript που το δημιούργησε, τοποθετώντας μηνύματα σε ένα χειριστή συμβάντων που καθορίζεται από αυτόν τον κώδικα (και αντίστροφα). Αυτή η ανταλλαγή μηνυμάτων δανείζεται από τους actors.

Ενώ οι Web Workers επιτυγχάνουν το σχεδιαστικό τους στόχο να εκφορτώσουν μακροχρόνιους υπολογισμούς σε νήματα υποβάθρου, δεν είναι κατάλληλοι για την ανάπτυξη κλιμακωτών φόρτων εργασίας λόγω του υψηλού κόστους επικοινωνίας και χαμηλού επιπέδου αφαίρεσης.

Ένα άλλο περιβάλλον που εισάγει τον ταυτοχρονισμό στην JavaScript είναι το River Tail των Intel Labs [Labs16], το οποίο επιτρέπει τον παραλληλισμό των δεδομένων σε εφαρμογές ιστού. Το River Trail επεκτείνει απαλά τη JavaScript με απλά ντετερμινιστικά κατασκευάσματα παραλληλισμού δεδομένων που μεταφράζονται κατά τη διάρκεια εκτέλεσης σε ένα στρώμα αφαίρεσης υλικού χαμηλού επιπέδου. Το κεντρικό στοιχείο του River Trail είναι ο τύπος *ParallelArray*. Τα αντικείμενα ParallelArray είναι ουσιαστικά παραταγμένες συλλογές κλιμακωτών τιμών που μπορούν να αντιπροσωπεύουν πολυδιάστατες συλλογές δεδομένων. Όλα τα αντικείμενα ParallelArray έχουν σχήμα που περιγράφει συνοπτικά τη διάσταση και το μέγεθος του αντικειμένου. Τα ParallelArrays είναι αμετάβλητες μόλις δημιουργηθούν και χειραγωγούνται με την επίκληση μεθόδων πάνω τους, οι οποίες παράγουν και

επιστρέφουν νέα αντικείμενα ParallelArray. Τέλος, τα ParallelArrays παρέχουν ένα API για χρήση λειτουργιών όπως το *map, reduce, filter* κτλ πάνω τους.

Το River Tail εισάγει βασικά το γνωστό παράδειγμα *map-reduce* στη JavaScript με τα ParallelArrays. Ενώ αυτό το παράδειγμα είναι καθιερωμένο, η σημασιολογία του παραμένει κάπως ασαφής και δύσκολη για τον μέσο προγραμματιστή να κατανοήσει και να ενσωματωθεί στις εφαρμογές του, προκειμένου να αξιοποιήσει το πραγματικό τους δυναμικό.

Ενώ τα παραπάνω πλαίσια παρέχουν κάποιο ταυτοχρονισμό στις εφαρμογές JavaScript, τα ενσωματωμένα στη JavaScript *Promises* παρέχουν ένα καλύτερο και πιο προσιτό πλαίσιο. Τα promises είναι αντικείμενα που είναι εγγενή στις περισσότερες υλοποιήσεις της JavaScript σήμερα [Prom15]. Ένα promise χρησιμοποιείται ως σύμβολο κράτησης θέσης για τα τελικά αποτελέσματα ενός αναβαλλόμενου (και πιθανώς ασύγχρονου) υπολογισμού [Lori17]. Κάθε αντικείμενο Promise βρίσκεται σε μία από τις τρεις αμοιβαία αποκλειόμενες καταστάσεις: εκπληρωμένη (fullfilled), απορριφθείσα (rejected) και εκκρεμής (pending). Ένα promise λέγεται ότι διευθετείται αν δεν εκκρεμεί, δηλαδή εάν εκπληρωθεί ή απορριφθεί. Ένα promise επιλύεται εάν έχει διευθετηθεί ή έχει "κλειδωθεί" για να ταιριάζει με την κατάσταση ενός άλλου promise. Η προσπάθεια επίλυσης ή απόρριψης ενός επιλυθέντος promise δεν έχει καμία επίδραση. Ένα ανεπίλυτο promise βρίσκεται πάντα στην εκκρεμούσα κατάσταση. Ένα αποφασισμένο promise ενδέχεται να εκκρεμεί, να εκπληρωθεί ή να απορριφθεί.

Αν το e είναι αντικείμενο Promise, τότε ορίζονται οι ακόλουθες λειτουργίες για τα promises:

- Το $Promise()$ δημιουργεί ένα νέο αντικείμενο Promise e.
- Το $Promise.resolve(e_2)$ επιλύει ένα Promise e_1 στην τιμή του e_2.
- Το $e_1.then(e_2)$ προγραμματίζει την υποσχέση e_2 που θα εκτελεστεί μετά την επίλυση του Promise e_1.
- Το $Promise.all([e_i])$, όπου το $[e_i]$ είναι ένας πίνακας των Promises e_i, δημιουργεί ένα νέο αντικείμενο Promise e το οποίο επιλύεται όταν επιλυθούν όλες τα promises του πίνακα.

Χρησιμοποιώντας τις παραπάνω λειτουργίες μπορούν να δημιουργηθούν promises και να είναι αλυσίδες το ένα μετά το άλλο προκειμένου να χρησιμοποιηθούν για ασύγχρονες εφαρμογές. Εάν χρησιμοποιούνται σωστά, τα promises μπορούν να μοντελοποιήσουν τον ταυτόχρονο προγραμματισμό και να δώσουν έναν τρόπο διαφυλλώμενων υπολογισμών. Η σημασιολογία και η αφαίρεση τους συνδέονται στενά με αυτό το έργο (RMTs). Κάνουμε μια σύγκριση και των δύο αργότερα σε αυτή τη διπλωματική.

1.2 Στόχοι και συνεισφορά της διπλωματικής

Οι στόχοι αυτής της διπλωματικής καθοδηγούνται από την υλοποίηση ενός μοντέλου σημασιολογίας του ταυτοχρονισμού για χρήση σε πραγματικές γλώσσες προγραμματισμού, όπως την *JavaScript*. Απαιτούνται νέα εργαλεία για τον ορισμό της σημασιολογίας των ταυτόχρονων γλωσσών προγραμματισμού, οι οποίες επιτρέπουν στα μέρη ενός προγράμματος που εκτελούνται ταυτόχρονα να αλληλεπιδρούν μεταξύ τους, χρησιμοποιώντας συνήθως τις ίδιες μεταβλητές μνήμης.

Οι επάνοδοι έχουν προταθεί από καιρό ως ένα μοντέλο μεταγλωττισμένων υπολογισμών στη σημασιολογία των παράλληλων γλωσσών προγραμματισμού. Εν συντομία, μια επάνοδος είναι είτε μια υπολογισμένη τιμή κάποιας περιοχής D είτε ένας ατομικός υπολογισμός που παράγει μια νέα επάνοδο.

Σε αυτή την εργασία, επεκτείνουμε μια δομημένη γενίκευση αυτής της τεχνικής που παρουσιάζεται στο [Papa01]. Επιτρέπουμε στα ατομικά βήματα να εκτελούν κάθε τύπο υπολογισμού, που αντιπροσωπεύεται από μία αυθαίρετη μονάδα M σε μια κατάλληλη κατηγορία σημασιολογικών πεδίων. Έτσι,

ορίζουμε τον *Μετασχηματιστή Μονάδων Επανόδου* R, ο οποίος μετατρέπει τη μονάδα M σε μία νέα μονάδα $R(M)$ που αντιπροσωπεύει τους διαφυλλώμενους υπολογισμούς. Ο μετασχηματιστής μονάδων επανόδου χρησιμοποιείται σε αυτό τη δουλειά για να εκφράσει τη σημασιολογία μιας απλής ντετερμινιστικής γλώσσας.

Η συνεισφορά αυτής της εργασίας είναι τριπλή:

1. Παρουσιάζουμε **τη θεωρία και το απαραίτητο μαθηματικό υπόβαθρο** ώστε ο αναγνώστης να μπορεί να κατανοήσει την δηλωτική σημασιολογία των ταυτόχρονων γλωσσών προγραμματισμού. Επιπλέον, παρουσιάζεται εδώ ένα θεωρητικό πλαίσιο για τον *Μετασχηματιστή Μονάδων Επανόδου*, με βάση το προαναφερθέν υπόβαθρο.

2. Εξετάζουμε **την απόδοση του Μετασχηματιστή Μονάδων Επανόδου** σε μονονηματικές γλώσσες προγραμματισμού που δεν υποστηρίζουν εγγενώς τον ταυτοχρονισμό. Για το σκοπό αυτό υλοποιήσαμε τον Μετασχηματιστή Μονάδων Επανόδου στη *JavaScript* και τον δοκιμάσαμε σε επτά διαφορετικά τεστ αξιολόγησης. Αυτά τα τεστ αποτελούνται από αλγόριθμους διαφορετικής χρονικής πολυπλοκότητας (δηλ. $O(n)$, $O(n\ logn)$, $O(n^2)$, $O(2^n)$). Για να κατανοήσουμε καλύτερα τα αποτελέσματά μας, τα συγκρίνουμε με τα ***Promises* της JavaScript's**, τα οποία μπορούν να μοντελοποιήσουν ένα μοντέλο ταυτοχρονισμού εάν χρησιμοποιηθούν με τον κατάλληλο τρόπο.

3. Παρουσιάζουμε τελικά **μία απλή προστακτική και ντετερμινιστική γλώσσα προγραμματισμού** (ένα μικρό αλλά αποτελεσματικό υποσύνολο της JavaScript) του οποίου η δηλωτική σημασιολογία υλοποιείται χρησιμοποιώντας το RMT μας. Αυτή η γλώσσα προγραμματισμού είναι επίσης **επεκτεταμένη με τελεστές** που παρέχουν ένα πλαίσιο χαμηλού κόστους και εύκολο στη χρήση για τη διεξαγωγή υπολογισμών με διαφυλλώμενο τρόπο.

1.3 Οργάνωση διπλωματικής

Το υπόλοιπο της παρούσας εργασίας οργανώνεται ως εξής. Το κεφάλαιο 2 περιέχει το βασικό μαθηματικό υπόβαθρο ώστε ο αναγνώστης να κατανοήσει τη σημασιολογία του ταυτοχρονισμού και της υπόλοιπης διπλωματικής. Περιλαμβάνει επίσης έναν μαθηματικό ορισμό των μονάδων, των μετασχηματιστών μονάδων και τη θεωρία πεδίων. Το κεφάλαιο 3 παρουσιάζει τους ορισμούς των μετασχηματιστών μονάδων επανόδου και την απαραίτητη θεωρία πίσω από αυτά. Το κεφάλαιο 4 περιγράφει την υλοποίηση των μετασχηματιστών μονάδων επανόδου στη *JavaScript* και τον τρόπο με τον οποίο χρησιμοποιούνται για τον ορισμό της δηλωτικής σημασιολογίας μιας απλής προστακτικής γλώσσας. Στο Κεφάλαιο 5 ορίζονται οι αλγόριθμοι αξιολόγησης μας και παρουσιάζουμε τα συγκριτικά αποτελέσματα της υλοποίησής μας, μαζί με μια σύγκριση επιδόσεων με τα promises. Τέλος, στο κεφάλαιο 6 δίνουμε τα συμπεράσματά μας σχετικά με τα RMTs και τις πιθανές μελλοντικές κατευθύνσεις τους.

Σε αυτό το σημείο θα ήταν χρήσιμο να αναφερθεί ότι το παρόν κείμενο αποτελεί **μετάφραση** του αγγλικού, οπότε είναι πιθανό να υπάρχουν αρκετές ασάφειες. Επίσης κάποια τμήματα παρέμειναν στα αγγλικά για την ευκολότερη κατανόηση τους.

Κεφάλαιο 2

Μαθηματικό υπόβαθρο

Στο κεφάλαιο αυτό συνοψίζουμε σαφώς το μαθηματικό υπόβαθρο που είναι απαραίτητο για τον αναγνώστη να κατανοήσει την υπόλοιπη διπλωματική. Για μια πιο ενημερωτική εισαγωγή και τις αποδείξεις των θεωρημάτων, ο αναγνώστης παραπέμπεται στη σχετική βιβλιογραφία που αναφέρεται παρακάτω.

2.1 Θεωρία κατηγοριών

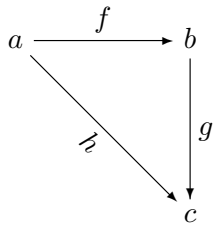
Η θεωρία των κατηγοριών (category theory) αναπτύχθηκε σε μια προσπάθεια ενοποίησης των απλών αφηρημένων εννοιών που εφαρμόζονταν σε πολλούς κλάδους των μαθηματικών. Άριστες εισαγωγές στη θεωρία κατηγοριών και τις εφαρμογές της στην Πληροφορική μπορούν να βρεθούν στα [Pier90, Gogu91, Pier91, Aspe91, Barr96].

Ορισμός 2.1.1 *Μια* κατηγορία *(category)* C *είναι μια συλλογή από* αντικείμενα *(objects) και μια συλλογή από* βέλη *(arrows) που ικανοποιούν τις ακόλουθες ιδιότητες:*

- *Για κάθε βέλος f υπάρχει ένα αντικείμενο* πεδίου *(domain object)* $\mathbf{dom}(f)$ *και ένα αντικείμενο* πεδίου τιμών *(codomain)* $\mathbf{codom}(f)$*, και γράφοντας $f : x \to y$ δηλώνεται ότι $x = \mathbf{dom}(f)$ και $y = \mathbf{codom}(f)$.*
- *Για κάθε ζεύγος βέλων $f : x \to y$ και $g : y \to z$ υπάρχει ένα βέλος* composite $g \circ f : x \to z$.
- *Η σύνθεση των βέλων είναι προσεταιριστική, δηλαδή για όλα τα βέλη $f : x \to y$, $g : y \to z$ και $h : z \to w$ it is $h \circ (g \circ f) = (h \circ g) \circ f$.*
- *Για κάθε αντικείμενο x υπάρχει ένα* ταυτοτικό *(identity) βέλος $\mathbf{id}_x : x \to x$.*
- *Τα βέλη ταυτότητας είναι μονάδες για τη σύνθεση βελών, δηλαδή για όλα τα βέλη $f : x \to y$ είναι $f \circ \mathbf{id}_x = \mathbf{id}_y \circ f = f$.*

Ορισμός 2.1.2 *Δύο αντικείμενα x και y της κατηγορίας* C *είναι* ισομορφικά *(isomorphic) εάν υπάρχουν βέλη $f : x \to y$ και $g : y \to x$ τέτοια ώστε $f \circ g = \mathbf{id}_y$ και $g \circ f = \mathbf{id}_x$. Το ζεύγος βέλων f και g ονομάζεται* ισομορφισμός *(isomorphism).*

Οι ιδιότητες των κατηγοριών παρουσιάζονται συνήθως χρησιμοποιώντας τα *commuting diagrams*. Ένα διάγραμμα είναι ένα γράφημα των οποίων οι κόμβοι είναι αντικείμενα και των οποίων τα άκρα είναι βέλη. Ένα διάγραμμα *επικοινωνεί* (commute) αν για κάθε ζεύγος κόμβων και για κάθε ζεύγος διαδρομών που συνδέουν αυτούς τους δύο κόμβους η σύνθεση βελών κατά μήκος της πρώτης διαδρομής είναι ίση με τη σύνθεση των βελών κατά μήκος του δεύτερου. Ένα παράδειγμα ενός commuting diagram, που υποδηλώνει ότι το $g \circ f = h$, φαίνεται παρακάτω.



Ορισμός 2.1.3 *Ένα* functor F *από την κατηγορία* C *στην κατηγορία* D*, γραμμένο ως* $F : \mathsf{C} \to \mathsf{D}$*, είναι ένα ζεύγος απεικονίσεων. Κάθε αντικείμενο* x *στο* C *απεικονίζεται σε ένα αντικείμενο* $F(x)$ *στο* D *και κάθε βέλος* $f : x \to y$ *στο* C *αντιστοιχεί σε ένα βέλος* $F(f) : F(x) \to F(y)$ *στο* D*. Επιπλέον, πρέπει να πληρούνται οι ακόλουθες ιδιότητες:*

- $F(\mathbf{id}_x) = \mathbf{id}_{F(x)}$ *για όλα τα αντικείμενα* x *in* C*.*
- $F(g \circ f) = F(g) \circ F(f)$ *για όλα τα βέλη* $f : x \to y$ *and* $g : y \to z$ *in* C*.*

Ορισμός 2.1.4 *Ένα* endofunctor *στην κατηγορία* C *είναι ένας functor* $F : \mathsf{C} \to \mathsf{C}$*.*

Ορισμός 2.1.5 *Αν τα* $F : \mathsf{C} \to \mathsf{D}$ *και* $G : \mathsf{D} \to \mathsf{E}$ *είναι functors, τότε η* σύνθεση *τους (composition) είναι ένας functor* $G \circ F : \mathsf{C} \to \mathsf{E}$*. Ορίζεται παίρνοντας* $(G \circ F)(x) = G(F(x))$ *και* $(G \circ F)(f) = G(F(f))$*.*

Ορισμός 2.1.6 *Για κάθε κατηγορία* C*, μπορεί να οριστεί ένας* functor ταυτότητας *(identity functor)* $\mathbf{Id}_{\mathsf{C}} : \mathsf{C} \to \mathsf{C}$ *λαμβάνοντας τα* $\mathbf{Id}_{\mathsf{C}}(x) = x$ *και* $\mathbf{Id}_{\mathsf{C}}(f) = f$*.*

Ας σημειωθεί ότι αν $F : \mathsf{C} \to \mathsf{C}$ είναι endofunctor και n είναι ένας θετικός φυσικός αριθμός, τότε ο συμβολισμός $F^n : \mathsf{C} \to \mathsf{C}$ μπορεί να χρησιμοποιηθεί για τη σύνθεση του F με τον ευατό του n φορές. Ο συμβολισμός μπορεί να επεκταθεί έτσι ώστε $F^0 = \mathbf{Id}_{\mathsf{C}}$.

Θεώρημα 2.1.1 *Οι functors διατηρούν τους ισομορφισμούς.*

Θεώρημα 2.1.2 *Οι functors ταυτότητας είναι οι μονάδες για τη σύνθεση των functor, δηλαδή εάν το* $F : \mathsf{C} \to \mathsf{D}$ *είναι, συνεπώς, functor* $F \circ \mathbf{Id}_{\mathsf{C}} = \mathbf{Id}_{\mathsf{D}} \circ F = F$

Ορισμός 2.1.7 *Αν τα* $F : \mathsf{C} \to \mathsf{D}$ *και* $G : \mathsf{C} \to \mathsf{D}$ *είναι functors, τότε ένας* φυσικός μετασχηματισμός *(natural transformation)* η *μεταξύ* F *και* G*, γραμμένο ως* $\eta : F \dot{\to} G$ *είναι μια οικογένεια βέλων στο* D*. Σε αυτήν την οικογένεια, ένα βέλος* $\eta_x : F(x) \to G(x)$ *στο* D *ορίζεται για κάθε αντικείμενο* x *στο* C*. Επιπλέον, το παρακάτω διάγραμμα πρέπει να επικοινωνεί:*

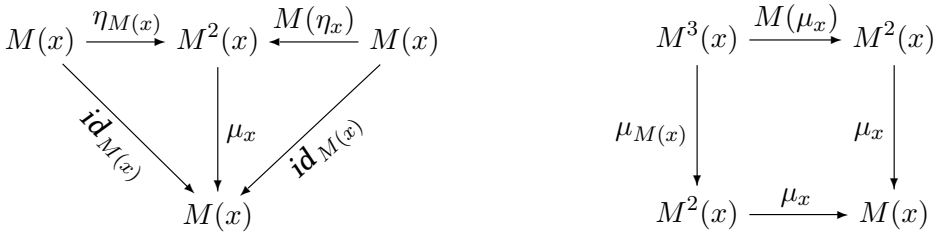
$$\begin{array}{ccc} x & & F(x) \xrightarrow{\eta_x} G(x) \\ \downarrow f & & \downarrow F(f) \qquad\qquad \downarrow G(f) \\ y & & F(y) \xrightarrow{\eta_y} G(y) \end{array}$$

2.2 Μονάδες και μετασχηματιστές μονάδων

Η έννοια της *μονάδας* (monad), που ονομάζεται επίσης τριάδα (triple), δεν είναι καινούργια στο πλαίσιο της θεωρίας των κατηγοριών. Στην επιστήμη των υπολογιστών, τα monads έγιναν πολύ δημοφιλή στη δεκαετία του 1990. Οι κατηγορικές ιδιότητες των μονάδων συζητούνται στα περισσότερα βιβλία

της θεωρίας των κατηγοριών, π.χ. σε [Barr96]. Για μια ολοκληρωμένη εισαγωγή των monads και τη χρήση τους στην δηλωτική σημασιολογία ο χρήστης αναφέρεται στο [Mogg90]. Μια κάπως διαφορετική προσέγγιση στον ορισμό των monads βρίσκεται στο [Wadl92], το οποίο εκφράζει την τρέχουσα πρακτική των monads στον συναρτησιακό προγραμματισμό. Οι δύο προσεγγίσεις είναι ισοδύναμες. Σε αυτή τη διπλωματική, η κατηγορική προσέγγιση (που παρουσιάζεται εδώ) χρησιμοποιείται για τον ορισμό των monads, αφού είναι πολύ πιο κομψή και η συναρτησιακή προσέγγιση (που παρουσιάζεται στην ενότητα 2.4) χρησιμοποιείται για την περιγραφή της σημασιολογίας των γλωσσών προγραμματισμού.

Ορισμός 2.2.1 *Ένα* monad *σε μια κατηγορία* C *είναι μία τριάδα* $\langle M, \eta, \mu \rangle$*, όπου* $M : \mathsf{C} \to \mathsf{C}$ *είναι ένα endofunctor,* $\eta : \mathbf{Id}_{\mathsf{C}} \dot{\to} M$ *και* $\mu : M^2 \dot{\to} M$ *είναι φυσικοί μετασχηματισμοί. Για όλα τα αντικείμενα* x *στο* C*, τα παρακάτω διαγράμματα πρέπει να επικοινωνούν.*



Ο μετασχηματισμός η *ονομάζεται* unit *του monad, ενώ ο μετασχηματισμός* μ *ονομάζεται* multiplication *ή* join*.*

Η μεταβατικότητα των παρακάτω δύο διαγραμμάτων είναι ισοδύναμη με τις ακόλουθες τρεις εξισώσεις, κοινώς ονομαζόμενες ως οι τρεις *νόμοι των μονάδων* (monad laws):

$$\begin{array}{lcll} \mu_x \circ \eta_{M(x)} & = & \mathbf{id}_{M(x)} & \textit{(1ος Νόμος Μονάδων)} \\ \mu_x \circ M(\eta_x) & = & \mathbf{id}_{M(x)} & \textit{(2ος Νόμος Μονάδων)} \\ \mu_x \circ M(\mu_x) & = & \mu_x \circ \mu_{M(x)} & \textit{(3ος Νόμος Μονάδων)} \end{array}$$

Ορισμός 2.2.2 *Αν το* C *είναι μια κατηγορία, ένας* μετασχηματιστής μονάδων *(monad transformer) στο* C *είναι μια αντιστοίχιση μεταξύ monads στο* C*.* [1]

2.3 Θεωρία πεδίων

Η θεωρία των πεδίων (domain theory) διατυπώθηκε από τους Scott και Strachey, προκειμένου να παρασχεθούν κατάλληλοι μαθηματικοί χώροι πάνω στους οποίους θα καθοριστεί η δηλωτική σημασιολογία των γλωσσών προγραμματισμού. Εισαγωγές στη θεωρία πεδίων, διαφόρων μεγεθών και επιπέδων, μπορείτε να βρείτε στα [Scot71, Scot82, Gunt90, Gunt92]. Διάφορα είδη πεδίων χρησιμοποιούνται συνήθως στην δηλωτική σημασιολογία, η πλειοψηφία των οποίων βασίζεται σε πλήρεις μερικές διατάξεις (cpo's). Η παραλλαγή που χρησιμοποιείται εδώ είναι μία από τις πολλές πιθανές επιλογές.

[1] *Πολλές επιλογές για τον ορισμό των μετασχηματιστών μονάδων έχουν προταθεί στη βιβλιογραφία. Δεδομένης της κατηγορίας* C*, οι monads στα* C *και* monad morphisms *(που δεν έχουν οριστεί σε αυτή τη διπλωματική) αποτελούν μια κατηγορία* Mon(C)*. Οι μετασχηματιστές μονάδων μπορούν να οριστούν ως αντιστοιχίσεις μεταξύ αντικειμένων στο* Mon(C)*, ως endofunctor στο* Mon(C)*, ως premonads σε* Mon(C) *(δηλαδή endofunctors με μια μονάδα), και ως monads στο* Mon(C)*. Σε αυτή τη διπλωματική επιλέξαμε την πρώτη επιλογή.*

Ορισμός 2.3.1 *Μία* μερική διάταξη *(partial order), ή* poset*, είναι ένα σύνολο D μαζί με μια δυαδική σχέση διάταξης $\sqsubseteq$ που είναι αντανακλαστική, αντισυμμετρική και μεταβατική.*

Ορισμός 2.3.2 *Ένα υποσύνολο $P \subseteq D$ ενός poset D είναι* φραγμένο *(bounded) αν υπάρχει $x \in D$ τέτοιο ώστε $y \sqsubseteq x$ για όλα τα $y \in P$. Σε αυτή την περίπτωση, το x είναι ένα* άνω όριο *(upper bound) του P.*

Ορισμός 2.3.3 *Το* least upper bound *ενός υποσυνόλου $P \subseteq D$, γραμμένο ως $\bigsqcup P$, είναι ένα άνω όριο P έτσι ώστε $\bigsqcup P \sqsubseteq x$ για όλα τα ανώτερα όρια x του P.* [2]

Ορισμός 2.3.4 *Ένα υποσύνολο $P \subseteq D$ ενός poset D είναι* directed *αν κάθε πεπερασμένο υποσύνολο $F \subseteq P$ έχει ένα άνω όριο $x \in P$.*

Ορισμός 2.3.5 *Ένα poset D είναι* πλήρες *(complete) αν κάθε κατευθυνόμενο υποσύνολο $P \subseteq D$ έχει ένα ελάχιστο ανώτερο όριο. Μια πλήρης μερική διάταξη ονομάζεται επίσης* cpo*.*

Ορισμός 2.3.6 *Ένα* domain *είναι ένα cpo D με ένα στοιχείο κάτω, γραμμένο ως $\bot$. Για όλα τα στοιχεία $x \in D$, πρέπει να είναι $\bot \sqsubseteq x$.*

Ορισμός 2.3.7 *Κάθε σύνολο S ορίζει ένα* flat domain *S^{o}, του οποίου το υποκείμενο σύνολο είναι $S \cup \{ \bot \}$ και στο οποίο $x \sqsubseteq y$ iff $x = y$ ή $x = \bot$.*

Σε αυτό το σημείο μπορούν να οριστούν ορισμένα χρήσιμα πεδία. Το απλό πεδίο **O** είναι το επίπεδο πεδίο που αντιστοιχεί στο κενό σύνολο. περιέχει ένα μόνο στοιχείο $\bot$. Ένα χρήσιμο πεδίο με ένα απλό κοινό στοιχείο είναι $\mathbf{U} = \{ \mathbf{u} \}^{\mathrm{o}}$. Οι φυσικοί αριθμοί κάτω από τη συνήθη παραγγελία τους $\leq$ σχηματίζουν ένα poset ω που δεν είναι cpo, αφού κατευθύνεται και δεν έχει ελάχιστο ανώτερο όριο.

Ορισμός 2.3.8 *Εάν D είναι poset, ένα ω-*chain *$(x_n)_{n \in \omega}$ στο D είναι ένα σύνολο στοιχείων $x_n \in D$ έτσι ώστε $n \leq m$ να υποδηλώνει $x_n \sqsubseteq x_m$.*

Ορισμός 2.3.9 *Μια συνάρτηση $f : D \to E$ μεταξύ των θέσεων D και E είναι* monotone *αν $x \sqsubseteq y$ υπονοεί $f(x) \sqsubseteq f(y)$.*

Ορισμός 2.3.10 *Μια συνάρτηση $f : D \to E$ μεταξύ των θέσεων D και E είναι* continuous *αν είναι μονότονη και $f(\bigsqcup P) = \bigsqcup \{ f(x) \mid x \in P \}$ για όλα τα κατευθυνόμενα $P \subseteq D$.*

Ορισμός 2.3.11 *Μια συνάρτηση $f : D \to E$ μεταξύ των πεδίων D και E είναι* strict *αν $f(\bot) = \bot$.*

Ορισμός 2.3.12 *Μια σχέση $\sqsubseteq$ μπορεί να οριστεί για τις συναρτήσεις μεταξύ πεδίων D και E ως εξής. Εάν $f, g : D \to E$, τότε $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ για όλα τα $x \in D$.*

Θεώρημα 2.3.1 *Το σύνολο συνεχών συναρτήσεων μεταξύ D και E υπό τη σχέση του ορισμού 2.3.12 είναι ένα πεδίο. Το πεδίο αυτό υποδηλώνεται με $D \to E$.*

Ορισμός 2.3.13 *Ένα στοιχείο $x \in D$ είναι ένα* fixed point *μιας συνάρτησης $f : D \to D$ αν $x = f(x)$.*

Θεώρημα 2.3.2 *Εάν D είναι πεδίο και $f : D \to D$ είναι συνεχής, τότε f έχει* least fixed point *$\mathbf{fix}(f) \in D$, δηλαδή $\mathbf{fix}(f) = f(\mathbf{fix}(f))$ και $\mathbf{fix}(f) \sqsubseteq x$ για όλα τα x τέτοια ώστε $x = f(x)$. Επί πλέον, $\mathbf{fix}(f) = \bigsqcup_{n \in \omega} f^n(\bot)$.*

Θεώρημα 2.3.3 *Για όλα τα $n \in \omega$, έστω $f_n : A \to B$. Έστω $x \in A$. Στη συνέχεια, εάν υπάρχει το ελάχιστο ανώτερο όριο στην αριστερή πλευρά, είναι $\left(\bigsqcup_{n \in \omega} f_n\right)(x) = \bigsqcup_{n \in \omega} f_n(x)$.*

[2] *Ο συμβολισμός $a \sqcup b$ χρησιμοποιείται ως συντομογραφία $\bigsqcup \{ a, b \}$.*

Θεώρημα 2.3.4 *Τα πεδία και οι συνεχείς συναρτήσεις αποτελούν μια κατηγορία* Dom.

Για την απλοποίηση της παρουσίασης, στην κατηγορία Dom παραλείπουμε συχνά τις παρενθέσεις που περιβάλλουν το όρισμα μιας συνάρτησης, δηλαδή γράφουμε $f\ x$ αντί για $f(x)$.

Ορισμός 2.3.14 *Ένας functor* $F : \mathsf{Dom} \to \mathsf{Dom}$ *είναι* locally monotone *αν* $f \sqsubseteq g$ *υπονοεί* $F(f) \sqsubseteq F(g)$*, για όλα τα πεδία* A *και* B *και για όλες τις συναρτήσεις* $f, g : A \to B$*.*

Ορισμός 2.3.15 *Ένας functor* $F : \mathsf{Dom} \to \mathsf{Dom}$ *είναι* locally continuous *εάν είναι τοπικά μονότονος και* $F(\bigsqcup P) = \bigsqcup \{ F(f) \mid f \in P \}$ *για όλους τα πεδία* A *και* B *και για όλα τα κατευθυνόμενα* $P \subseteq A \to B$*.*

Ορισμός 2.3.16 *Εάν τα* D *και* E *είναι πεδία, τότε το* product $D \times E$ *είναι ένα πεδίο. Τα στοιχεία του* $D \times E$ *είναι τα ζεύγη* $\langle x, y \rangle$ *με* $x \in D$ *και* $y \in E$ *και η σχέση παραγγελίας ορίζεται ως* $\langle x_1, y_1 \rangle \sqsubseteq \langle x_2, y_2 \rangle \Leftrightarrow x_1 \sqsubseteq_D x_2 \wedge y_1 \sqsubseteq_E y_2$*.*

Ορισμός 2.3.17 *Εάν το* $D \times E$ *είναι πεδίο προϊόντος, δύο συνεχείς συναρτήσεις προβολής* $\mathbf{fst} : D \times E \to D$ *και* $\mathbf{snd} : D \times E \to E$ *μπορούν να οριστούν λαμβάνοντας τα* $\mathbf{fst}\ \langle x, y \rangle = x$ *και* $\mathbf{snd}\ \langle x, y \rangle = y$*.*

Ορισμός 2.3.18 *Εάν τα* D *και* E *είναι πεδία, τότε το* separated sum $D + E$ *είναι ένα πεδίο. Το σύνολο των στοιχείων του* $D + E$ *είναι:*

$$\{ \langle x, 0 \rangle \mid x \in D \} \cup \{ \langle y, 1 \rangle \mid y \in E \} \cup \{ \bot_{D+E} \}$$

Η σχέση παραγγελίας ορίζεται ξεχωριστά για κάθε ένα από τα τρία διαφορετικά υποσύνολα του $D + E$*, δηλαδή* $\langle x_1, 0 \rangle \sqsubseteq \langle x_2, 0 \rangle \Leftrightarrow x_1 \sqsubseteq_D x_2$ *και* $\langle y_1, 1 \rangle \sqsubseteq \langle y_2, 1 \rangle \Leftrightarrow y_1 \sqsubseteq_E y_2$*. Επιπλέον,* $\bot_{D+E} \sqsubseteq z$ *για όλα τα* $z \in D + E$*.*

Ορισμός 2.3.19 *Εάν το* $D + E$ *είναι ένα πεδίο αθροίσματος, μπορούν να οριστούν δύο συνεχείς συναρτήσεις ψεκασμού* $\mathbf{inl} : D \to D + E$ *και* $\mathbf{inr} : E \to D + E$ *λαμβάνοντας τα* $\mathbf{inl}\ x = \langle x, 0 \rangle$ *και* $\mathbf{inr}\ y = \langle y, 1 \rangle$*.*

Ορισμός 2.3.20 *Εάν τα* D*,* E *και* F *είναι πεδία και* $f_1 : D \to F$ *και* $f_2 : E \to F$ *είναι συνεχείς συναρτήσεις, μια αυστηρή συνεχής συνάρτηση* $[\, f_1, f_2 \,] : D + E \to F$ *μπορεί να οριστεί ως:*

$$[\, f_1, f_2 \,](z) \quad = \quad \begin{cases} \bot_F & , \textit{ if } z = \bot_{D+E} \\ f_1(x) & , \textit{ if } z = \langle x, 0 \rangle \\ f_2(y) & , \textit{ if } z = \langle y, 1 \rangle \end{cases}$$

Θεώρημα 2.3.5 *Έστω τα* A*,* B *και* C *είναι πεδία,* $f : A \to C$ *και* $g : B \to C$ *συνεχείς συναρτήσεις. Επειτα* $[\, f, g \,] \circ \mathbf{inl} = f$ *και* $[\, f, g \,] \circ \mathbf{inr} = g$*.*

Θεώρημα 2.3.6 *Έστω τα* A *και* B *είναι πεδία. Επειτα* $[\, \mathbf{inl}, \mathbf{inr} \,] = \mathbf{id}_{A+B}$*.*

Θεώρημα 2.3.7 *Έστω τα* A_1*,* B_1*,* C_1*,* A_2*,* B_2 *και* C_2 *είναι πεδία. Έστω οι* $f_1 : B_1 \to C_1$*,* $f_2 : A_1 \to B_1$*,* $g_1 : B_2 \to C_2$ *και* $g_2 : A_2 \to B_2$ *είναι συνεχείς συναρτήσεις. Επειτα* $[\, f_1 \circ f_2, g_1 \circ g_2 \,] = [\, f_1, g_1 \,] \circ [\, \mathbf{inl} \circ f_2, \mathbf{inr} \circ g_2 \,]$*.*

Θεώρημα 2.3.8 *Έστω τα* A*,* B*,* C *και* D *είναι πεδία,* $f : C \to D$*,* $g_1 : A \to C$ *και* $g_2 : B \to C$ *συνεχείς συναρτήσεις. Εάν το* f *είναι αυστηρό, τότε* $f \circ [\, g_1, g_2 \,] = [\, f \circ g_1, f \circ g_2 \,]$*.*

Τα δυναμοπεδία είναι το πεδίο-θεωρητικό ισοδύναμο των δυναμοσυνόλων. Έχουν εισαχθεί σαν ένα εργαλείο για τη μοντελοποίηση της σημασιολογίας μη-ντετερμινιστικών προγραμμάτων και έχουν χρησιμοποιηθεί ευρέως για τη σημασιολογία του ταυτόχρονισμού. Σε αυτή τη διπλωματική αποφεύγουμε έναν πλήρη ορισμό των πεδίων της εξουσίας. ο αναγνώστης αναφέρεται στο [Gunt92] για έναν λεπτομερή ορισμό και μια μελέτη για τις κατηγορικές και τις θεωρητικές ιδιότητες των πεδίων. [3]

Ορισμός 2.3.21 *Έστω το D είναι ένα πεδίο. Γράφουμε $D^\natural$ για το (κυρτό)* power-domain *του D.*

Ορισμός 2.3.22 *Έστω τα D και E είναι πεδία και $f : D \to E$ μια συνεχής συνάρτηση. Μπορούμε να ορίσουμε μια συνεχή συνάρτηση $f^\natural : D^\natural \to E^\natural$.*

Θεώρημα 2.3.9 *Λαμβάνοντας τα $P(D) = D^\natural$ και $P(f) = f^\natural$ μπορούμε να ορίσουμε ένα endfunctor $P : \mathsf{Dom} \to \mathsf{Dom}$, το οποίο ονομάζεται* power-domain functor.

Ορισμός 2.3.23 *Έστω το D είναι ένας πεδίο. Μπορούμε να ορίσουμε μια συνεχή συνάρτηση $\{\!|\cdot|\!\} : D \to D^\natural$, η οποία ονομάζεται συνάρτηση* power-domain singleton.

Ορισμός 2.3.24 *Έστω το D είναι ένα πεδίο. Μπορούμε να ορίσουμε μια συνεχή δυαδική συνάρτηση $\uplus^\natural : D^\natural \times D^\natural \to D^\natural$, η οποία ονομάζεται* power-domain union. *Επιπλέον, αυτή η δυαδική συνάρτηση είναι συσχετιστική, μεταβλητή και idempotent.*

Ορισμός 2.3.25 *Έστω το D είναι ένα πεδίο. Μπορούμε να ορίσουμε μια συνεχή συνάρτηση $\bigcup^\natural : D^{\natural\natural} \to D^\natural$, η οποία ονομάζεται συνάρτηση* power-domain big union.

Θεώρημα 2.3.10 *Το singleton δυναμοπεδίο είναι φυσικός μετασχηματισμός μεταξύ του functor της ταυτότητας $\mathbf{Id}_{\mathsf{Dom}}$ και του functor domain power-domain P.*

Θεώρημα 2.3.11 *Η μεγάλη ένωση του δυναμοπεδίου είναι μια φυσική μεταμόρφωση μεταξύ των functors P^2 και P.*

Θεώρημα 2.3.12 *Ο functor δυναμοπεδίο-P με τον δυναμοπεδίο-singleton ως μονάδα και η μεγάλη ένωση του δυναμοπεδίου ορίζουν ένα monad P, το οποίο ονομάζεται* power-domain monad.

2.4 Οι μονάδες στο συναρτησιακό προγραμματισμό

Μια εναλλακτική προσέγγιση στον ορισμό των μονάδων έχει γίνει πολύ δημοφιλής στη συναρτησιακή κοινότητα προγραμματισμού. Σύμφωνα με αυτή, ένα monad στην κατηγορία Dom ορίζεται ως τριάδα $\langle M, \boldsymbol{unit}_{\mathsf{M}}, *_{\mathsf{M}}\rangle$. Σε αυτή την τριάδα, το M είναι ένας κατασκευαστής πεδίων, $\boldsymbol{unit}_{\mathsf{M}} : D \to M(D)$ είναι μια συνεχής συνάρτηση και $*_{\mathsf{M}} : M(A) \times (A \to M(B)) \to M(B)$ είναι μια δυαδική συνάρτηση. Αυτές οι τριάδες $\langle M, \boldsymbol{unit}_{\mathsf{M}}, *_{\mathsf{M}}\rangle$ αναφέρονται συχνά ως $\langle M, \mathbf{return}_M, \mathbf{bind}_M\rangle$ σε συναρτησιακές γλώσσες προγραμματισμού όπως τη *Haskell*.

Στη σημασιολογία των γλωσσών προγραμματισμού, τα πεδία που κατασκευάζονται από το monad M τυπικά υποδηλώνουν *υπολογισμούς* (computations), π.χ. το πεδίο $M(D)$ υποδηλώνει υπολογισμούς που επιστρέφουν τιμές του πεδίου D. Το αποτέλεσμα του $\boldsymbol{unit}_{\mathsf{M}}\ v$ είναι απλά ένας υπολογισμός που επιστρέφει την τιμή v και το αποτέλεσμα του $m\ *_{\mathsf{M}}\ f$ είναι ο συνδυασμένος υπολογισμός m,

[3] Αγνοούμε επίσης το γεγονός ότι ολόκληρη η κατηγορία Dom των περιοχών Scott δεν είναι κατάλληλη για τον ορισμό πεδίων ισχύος. Μια από τις κατηγορίες SFP (των ακολουθιών των πεπερασμένων posets) ή Bif (πεδίων bifinite), οι οποίες είναι κλειστές υποκατηγορίες Dom, θα πρέπει να χρησιμοποιηθούν αντ 'αυτού. Ο αναγνώστης αναφέρεται και πάλι στο [Gunt92]. Παρατηρήστε, ωστόσο, ότι τα αποτελέσματα που παρουσιάζονται στο Κεφάλαιο 3 ισχύουν εξίσου για όλες τις κλειστές υποκατηγορίες του Dom, συμπεριλαμβανομένου του ίδιου του Dom. Χρησιμοποιούμε μόνο δυναμοπεδία στην ενότητα 4.4.

που επιστρέφει v, ακολουθούμενος από τον υπολογισμό $f(v)$. Οι μετασχηματιστές μονάδων είναι χρήσιμοι για τη μετατροπή μεταξύ διαφορετικών τύπων υπολογισμών [Lian95].

Οι ακόλουθες εξισώσεις συνδέουν ένα monad $\langle M, \mathbf{unit}_{\mathsf{M}}, *_{\mathsf{M}} \rangle$ που ορίζεται χρησιμοποιώντας τη συναρτησιακή προσέγγιση με τη μονάδα $\langle M, \eta, \mu \rangle$ που ορίζεται χρησιμοποιώντας την προσέγγιση των κατηγοριών.

$$\begin{array}{lcl} \mathbf{unit}_{\mathsf{M}} & = & \eta \\ m \ *_{\mathsf{M}} \ f & = & (\mu \circ M(f))\ m \end{array} \qquad \begin{array}{lcl} \eta & = & \mathbf{unit}_{\mathsf{M}} \\ \mu & = & \lambda m.\ m \ *_{\mathsf{M}} \ \mathbf{id} \\ M(f) & = & \lambda m.\ m \ *_{\mathsf{M}} \ (\mathbf{unit}_{\mathsf{M}} \circ f) \end{array}$$

Στη συναρτησιακή προσέγγιση, οι τρεις νόμοι μοναδα μπορούν να διατυπωθούν ως εξής.

$$\begin{array}{lcl} m \ *_{\mathsf{M}} \ \mathbf{unit}_{\mathsf{M}} & = & m \\ (\mathbf{unit}_{\mathsf{M}}\ v) \ *_{\mathsf{M}} \ f & = & f\ v \\ m \ *_{\mathsf{M}} \ (\lambda v.\ (f\ v) \ *_{\mathsf{M}} \ g) & = & (m \ *_{\mathsf{M}} \ f) \ *_{\mathsf{M}} \ g \end{array}$$

Μια ενδιαφέρουσα παρατήρηση είναι ότι αυτοί οι τρεις νόμοι είναι αρκετοί για να αποδείξουν ότι το ισοδύναμο $\langle M, \eta, \mu \rangle$, όπως ορίστηκε παραπάνω, είναι πράγματι ένα monad, δηλαδή ότι το M είναι ένας functor (διατηρεί ταυτότητες συναρτήσεων και σύνθεση) και ότι η και μ είναι φυσικοί μετασχηματισμοί.

Σε αυτή τη ρύθμιση, είναι χρήσιμο να ορίσουμε δύο ειδικές τάξεις monads, εξοπλισμένες με πρόσθετες συναρτήσεις που είναι χρήσιμες για τη μοντελοποίηση της σημασιολογίας της ταυτόχρονσης στις γλώσσες προγραμματισμού.

Ορισμός 2.4.1 *Το* multi-monad *είναι ένα monad M με μια δυαδική συνάρτηση $\|_{\mathsf{M}} : M(D) \times M(D) \to M(D)$, όπου D είναι ένα πεδίο.*

Ορισμός 2.4.2 *Το* strong monad *είναι ένα monad M με μια δυαδική συνάρτηση $\bowtie_{\mathsf{M}} : M(A) \times M(B) \to M(A \times B)$, όπου τα A και B είναι πεδία.*

Η δυαδική συνάρτηση $\|$ ενός multi-monad χρησιμοποιείται για να εκφράσει την αποσύνδεση των υπολογισμών. Με άλλα λόγια, αν το M είναι ένα πολυ-μονοδιάστατο, το D είναι ένα πεδίο και το $m_1, m_2 \in M(D)$ είναι δύο υπολογισμοί, ο υπολογισμός $m_1 \ \| \ m_2$ υποδεικνύει μια (πιθανώς μη αιτιοκρατική) επιλογή μεταξύ m_1 και m_2. Επιπλέον, η δυαδική συνάρτηση $\bowtie$ ενός ισχυρού monad χρησιμοποιείται για να εκφράσει τη σχέση σε υπολογισμούς. Έστω το M είναι ένα ισχυρό monad, έστω A και B είναι πεδία. Εάν οι $m_1 \in M(A)$ και $m_2 \in M(B)$ είναι δύο υπολογισμοί, ο υπολογισμός $m_1 \bowtie m_2$ υποδεικνύει ότι και τα δύο m_1 και m_2 θα εκτελεστούν και τα αποτελέσματά τους θα συνδυαστούν. Η επιλογή εδώ σχετίζεται με τη σειρά, εάν υπάρχει, στην οποία θα εκτελεστούν οι δύο υπολογισμοί.

Κεφάλαιο 3

Οι μετασχηματιστές μονάδων επανόδου

Η έννοια της *διαφυλλισμένης εκτέλεσης* (interleaving) είναι πολύ γνωστή στη θεωρία του ταυτοχρονισμού. Σε αυτό το πλαίσιο, οι υπολογισμοί θεωρούνται αλληλουχίες *ατομικών βημάτων* (atomic steps), η φύση των οποίων εξαρτάται από την έννοια του υπολογισμού. Μεμονωμένα, αυτά τα ατομικά βήματα εκτελούνται το ένα μετά το άλλο μέχρις ότου ολοκληρωθεί ο υπολογισμός τους. Δεδομένων δύο υπολογισμών A και B, ένας διαφυλλισμένος υπολογισμός των A και B αποτελείται από μια αυθαίρετη συγχώνευση των ατομικών βημάτων που συνιστούν A και B. Η παρεμβολή εύκολα εκτείνεται σε περισσότερους από δύο υπολογισμούς. Τα ατομικά βήματα οποιουδήποτε υπολογισμού πρέπει να εκτελούνται με τη σωστή σειρά, αλλά αυτή η διαδικασία μπορεί να διακοπεί με την εκτέλεση ατομικών βημάτων που ανήκουν σε άλλους υπολογισμούς.

Ο πρωταρχικός στόχος μας είναι να ορίσουμε έναν μετασχηματιστή μονάδων R ικανό να μοντελοποιεί διαφυλλώμενους υπολογισμούς. Με αυτόν τον τρόπο, αν μας δοθεί ένα monad M που διαμορφώνει τους υπολογισμούς που λαμβάνουν χώρα στα ατομικά βήματα, μπορούμε να αποκτήσουμε ένα monad $\mathsf{R}(M)$ που διαμορφώνει διαφυλλώμενους υπολογισμούς τέτοιων ατομικών βημάτων. Μια πιθανή λύση στο πρόβλημα αυτό είναι η χρήση της προτεινόμενης τεχνικής των *επανόδων* (resumptions), που απεικονίζεται στο [dBak96, Schm86] για συγκεκριμένες περιπτώσεις M.

Γενικεύοντας αυτή την τεχνική, το πεδίο $\mathsf{R}(M)(D)$ των επανόδων πρέπει να ικανοποιεί τον ακόλουθο ισομορφισμό:

$$\mathsf{R}(M)(D) \quad \simeq \quad D + M(\mathsf{R}(M)(D))$$

Στο πεδίο αυτό, τα ατομικά βήματα είναι αυθαίρετοι υπολογισμοί που ορίζονται από το M. Το αριστερό μέρος του αθροίσματος αντιπροσωπεύει ένα ήδη εκτιμημένο αποτέλεσμα, δηλ. Έναν υπολογισμό που αποτελείται από μηδενικά ατομικά βήματα. Το δεξιό μέρος αντιπροσωπεύει έναν υπολογισμό που απαιτεί τουλάχιστον ένα ατομικό βήμα. Το αποτέλεσμα αυτού του ατομικού βήματος είναι ένα νέο στοιχείο του πεδίου επανόδου.

Σε αυτό το κεφάλαιο ορίζουμε επίσημα τον μετασχηματιστή μονάδων επανόδου όπως παρουσιάζεται στο [Papa01], όπου μπορούν να βρεθούν όλες οι αποδείξεις για τα επόμενα θεωρήματα και τα λήμματα. Ξεκινάμε εξετάζοντας μια αυθαίρετη τοπικά μονάδα M στο Dom. Το υπόλοιπο κεφάλαιο οργανώνεται ως εξής. Στην ενότητα 3.1 ορίζουμε ένα endfunctor $\mathbf{R}_M : \mathsf{Dom} \to \mathsf{Dom}$. Στην ενότητα 3.2 ορίζουμε δύο φυσικούς μετασχηματισμούς $\mathit{unit} : \mathit{Id} \dot{\to} \mathbf{R}_M$ και $\mathit{join} : \mathbf{R}_M^2 \dot{\to} \mathbf{R}_M$ και στην ενότητα 3.3 αποδεικνύουμε ότι το $\langle \mathbf{R}_M, \mathit{unit}, \mathit{join} \rangle$ ικανοποιεί τους τρεις νόμους των μονάδων. Με αυτό τον τρόπο ορίζουμε τον μετασχηματιστή μονάδων R. Στη συνέχεια, στην ενότητα 3.4 αποδεικνύουμε ότι $\mathsf{R}(M)(D)$ ικανοποιεί τον προαναφερθέντα ισομορφισμό κατασκευάζοντας τα δύο συστατικά h^e και h^p του ισομορφισμού. Τέλος, στην ενότητα 3.5 ορίζουμε μερικές επιπλέον συναρτήσεις στα πεδία που κατασκευάζονται από το $\mathsf{R}(M)$, το οποίο θα είναι χρήσιμο για τον προσδιορισμό της σημασιολογίας των ταυτόχρονων γλωσσών προγραμματισμού.

3.1 Functor $\mathbf{R}_M$

Ξεκινάμε καθορίζοντας για κάθε πεδίο D ένα endofunctor $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ και μερικές βοηθητικές συναρτήσεις. Το πεδίο $\mathsf{R}(M)(D)$ που προσπαθούμε να ορίσουμε είναι ένα σταθερό σημείο $\mathbf{F}_{M,D}$.

Ορισμός 3.1.1 *Έστω τα D, A και B είναι πεδία και $f : A \to B$ μια συνεχής συνάρτηση. Ορίζουμε τις ακόλουθες αντιστοιχίσεις:*

$$\begin{array}{lcl} \mathbf{F}_{M,D}(X) & = & D + M(X) \\ \mathbf{F}_{M,D}(f) & = & [\,\mathbf{inl}, \mathbf{inr} \circ M(f)\,] \end{array}$$

Λήμμα 3.1.1 $\mathbf{F}_{M,D}(f) \circ \mathbf{inr} = \mathbf{inr} \circ M(f)$

Θεώρημα 3.1.1 *Το $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ είναι ένας functor.*

Δεν είναι δύσκολο να αποδείξουμε ότι ο functor $\mathbf{F}_{M,D}$ είναι τοπικά μονότονος και τοπικά συνεχής. Αυτό το αποτέλεσμα έρχεται εύκολα, αφού το monad M έχει αυτές τις δύο ιδιότητες και το $\mathbf{F}_{M,D}$ ορίζεται από την άποψη του M, χρησιμοποιώντας μόνο τις βασικές συναρτήσεις πεδίου που διατηρούν τη μονοτονικότητα και τη συνέχεια.

Λήμμα 3.1.2 *Ο functor $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ είναι τοπικά μονότονος.*

Λήμμα 3.1.3 *Ο functor $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ είναι τοπικά συνεχής.*

Οι δύο συναρτήσεις ι^e και ι^p είναι χρήσιμες στον ορισμό του $\mathbf{R}_M(D)$. Ορίζουν μια ενσωμάτωση και μια προβολή μεταξύ των πεδίων $\mathbf{O}$ και $\mathbf{F}_{M,D}(\mathbf{O})$.

Ορισμός 3.1.2 *Έστω το D είναι ένα πεδίο. Ορίζουμε το ζεύγος συνεχών συναρτήσεων $\iota^e : \mathbf{O} \to \mathbf{F}_{M,D}(\mathbf{O})$ και $\iota^p : \mathbf{F}_{M,D}(\mathbf{O}) \to \mathbf{O}$ είναι ίσο με $\bot$.*

Ως εκ τούτου, μπορεί να αποδειχθεί ότι:

Λήμμα 3.1.4 $\iota^p \circ \iota^e = \mathbf{id}_{\mathbf{O}}$

Λήμμα 3.1.5 $\iota^e \circ \iota^p \sqsubseteq \mathbf{id}_{\mathbf{F}_{M,D}(\mathbf{O})}$

Προχωρούμε καθορίζοντας μια χαρτογράφηση αντικειμένων και μια χαρτογράφηση των συναρτήσεων, η οποία θα καθορίσει τον endfunctor $\mathbf{R}_M : \mathsf{Dom} \to \mathsf{Dom}$ στο τέλος αυτής της ενότητας. Αυτός είναι ο βασικός ορισμός του [Papa01] που χρησιμοποιούμε στην υπόλοιπη εργασία.

Ορισμός 3.1.3 *Έστω το D είναι ένα πεδίο. Το πεδίο $\mathbf{R}_M(D)$ είναι το σύνολο*

$$\mathbf{R}_M(D) \;=\; \{\, (x_n)_{n\in\omega} \;\mid\; \forall\, n \in \omega.\; x_n \in \mathbf{F}^n_{M,D}(\mathbf{O}) \;\wedge\; x_n = \mathbf{F}^n_{M,D}(\iota^p)(x_{n+1}) \,\}$$

με τα στοιχεία του να διατάσσονται κατά σημείο:

$$(x_n)_{n\in\omega} \sqsubseteq_{\mathbf{R}_M(D)} (y_n)_{n\in\omega} \;\Leftrightarrow\; \forall\, n \in \omega.\; x_n \sqsubseteq_{\mathbf{F}^n_{M,D}(\mathbf{O})} y_n$$

Τα στοιχεία του πεδίου $\mathbf{R}_M(D)$ είναι άπειρες ακολουθίες, με ευρετηρίαση από το σύνολο των φυσικών αριθμών ω. Το στοιχείο n της ακολουθίας είναι ένα στοιχείο του πεδίου $\mathbf{F}^n_{M,D}(\mathbf{O})$. Τέτοια στοιχεία αντιπροσωπεύουν το *finite approximations* των υπολογισμών επανόδου: εάν ένας δεδομένος υπολογισμός επαναφοράς τερματιστεί σε στάδια μικρότερα από n, η προσέγγισή του n είναι ικανή να υπολογίσει με ακρίβεια το αποτέλεσμα. διαφορετικά παράγει $\perp$. Η συνθήκη $x_n = \mathbf{F}^n_{M,D}(\iota^p)(x_{n+1})$ δηλώνει ότι τα στοιχεία της άπειρης αλληλουχίας πρέπει να είναι πράγματι προσεγγίσεις: το αποτέλεσμα της προβολής της προσέγγισης $(n+1)$ (στοιχείο $\mathbf{F}^{n+1}_{M,D}(\mathbf{O})$) σε ένα στοιχείο του $\mathbf{F}^n_{M,D}(\mathbf{O})$ πρέπει να είναι ίσο με την προσέγγιση n.

Προτού μπορέσουμε να ορίσουμε τη χαρτογράφηση των συναρτήσεων που αντιστοιχούν στο $\mathbf{R}_M$, είναι απαραίτητο να ορίσουμε μερικές οικογένειες βοηθητικών συναρτήσεων. Η πρώτη είναι η οικογένεια των συναρτήσεων $f^D_{m,n}$ που αντιστοιχούν σε διαφορετικές προσεγγίσεις ενός υπολογισμού εκ νέου.

Ορισμός 3.1.4 *Έστω το D είναι ένα πεδίο. Για όλα τα $m, n \in \omega$, ορίζουμε μια συνάρτηση $f^D_{m,n} : \mathbf{F}^m_{M,D}(\mathbf{O}) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ με:*

$$\begin{array}{lcll} f^D_{m,n} & = & \mathit{id}_{\mathbf{F}^n_{M,D}(\mathbf{O})} & \text{, if } m = n \\ f^D_{m,n+1} & = & f^D_{m,n} \circ \mathbf{F}^n_{M,D}(\iota^p) & \text{, if } m \leq n \\ f^D_{m+1,n} & = & \mathbf{F}^m_{M,D}(\iota^e) \circ f^D_{m,n} & \text{, if } m \geq n \end{array}$$

Έτσι, τα ακόλουθα λήμματα ισχύουν:

Λήμμα 3.1.6 *Για όλα τα $m, n \in \omega$, $f^D_{m,n} \circ \mathbf{F}^n_{M,D}(\iota^p) \sqsubseteq f^D_{m,n+1}$.*

Λήμμα 3.1.7 *Για όλα τα $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$ και για όλα τα $m, n \in \omega$, $f^D_{m,n}\, x_n \sqsubseteq x_m$.*

Οι οικογένειες των συναρτήσεων μ^e_n και μ^p_n ορίζουν επίσης αντιστοιχίσεις μεταξύ υπολογισμών επανόδου και των προσεγγίσεών τους. Ο πρώτος ενσωματώνει μια προσέγγιση που απαιτεί βήματα μικρότερα από n σε ένα στοιχείο του πεδίου $\mathbf{R}_M(D)$, ενώ το τελευταίο προβάλλει ένα στοιχείο του πεδίου $\mathbf{R}_M(D)$ στην προσέγγισή του n.

Ορισμός 3.1.5 *Έστω το D είναι ένα πεδίο, $n \in \omega$, $z \in \mathbf{F}^n_{M,D}(\mathbf{O})$ και $(x_m)_{m\in\omega} \in \mathbf{R}_M(D)$. Ορίζουμε το ζεύγος συναρτήσεων $\mu^e_n : \mathbf{F}^n_{M,D}(\mathbf{O}) \to \mathbf{R}_M(D)$ και $\mu^p_n : \mathbf{R}_M(D) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ ως εξής:*

$$\begin{array}{lcl} \mu^e_n\, z & = & (f^D_{m,n}\, z)_{m\in\omega} \\ \mu^p_n\, (x_m)_{m\in\omega} & = & x_n \end{array}$$

Τώρα είμαστε έτοιμοι να καθορίσουμε τη χαρτογράφηση των συναρτήσεων που απαιτούνται από τον functor $\mathbf{R}_M$. Αντί να ορίσουμε αυτήν την χαρτογράφηση απευθείας από την άποψη των στοιχείων του πεδίου επανόδου $\mathbf{R}_M(D)$, χρησιμοποιούμε την οικογένεια των συναρτήσεων $\zeta^{A,B}_n$ και την ορίζουμε με όρους πεπερασμένων προσεγγίσεων.

Ορισμός 3.1.6 *Έστω τα A και B είναι πεδία και έστω το $f : A \to B$ είναι συνεχής συνάρτηση. Για όλα τα $n \in \omega$ ορίζουμε μια αυστηρή συνεχή συνάρτηση $\zeta^{A,B}_n\ f : \mathbf{F}^n_{M,A}(\mathbf{O}) \to \mathbf{F}^n_{M,B}(\mathbf{O})$ με:*

$$\begin{array}{lcl} \zeta^{A,B}_0\ f & = & \perp \\ \zeta^{A,B}_{n+1}\ f & = & [\,\mathit{inl} \circ f, \mathit{inr} \circ M(\zeta^{A,B}_n\ f)\,] \end{array}$$

Ορισμός 3.1.7 *Έστω τα A και B είναι πεδία και έστω το $f : A \to B$ είναι συνεχής συνάρτηση. Ορίζουμε μια συνεχή συνάρτηση $\mathbf{R}_M(f) : \mathbf{R}_M(A) \to \mathbf{R}_M(B)$ με:*

$$\mathbf{R}_M(f)\ (x_n)_{n\in\omega} \quad = \quad (\zeta_n^{A,B}\ f\ x_n)_{n\in\omega}$$

Το κεντρικό αποτέλεσμα αυτής της ενότητας είναι το θεώρημα 3.1.2 στο οποίο αποδεικνύουμε ότι το $\mathbf{R}_M$ είναι functor. Για το σκοπό αυτό, χρησιμοποιούμε τα ακόλουθα λήμματα, των οποίων τα αποδεικτικά μπορούν να βρεθούν ξανά στο [Papa01].

Λήμμα 3.1.8 *Για όλα τα $n \in \omega$, $\mu_n^e \circ \mathbf{F}_{M,D}^n(\iota^p) \sqsubseteq \mu_{n+1}^e$.*

Λήμμα 3.1.9 *Έστω τα A και B είναι πεδία, $f : A \to B$ μια συνεχής συνάρτηση. Στη συνέχεια για όλα τα $n \in \omega$,*
$\zeta_{n+1}^{A,B}\ f \circ \mathbf{inr} = \mathbf{inr} \circ M(\zeta_n^{A,B}\ f)$

Λήμμα 3.1.10 *Για όλα τα $x \in \mathbf{R}_M(D)$, $(\mu_n^p\ x)_{n\in\omega} = x$.*

Λήμμα 3.1.11 *Για όλα τα $m, n \in \omega$, $\mu_m^p \circ \mu_n^e = f_{m,n}^D$.*

Λήμμα 3.1.12 *Για όλα τα $n \in \omega$, $\mu_n^p \circ \mu_n^e = \mathbf{id}_{\mathbf{F}_{M,D}^n(\mathbf{O})}$.*

Λήμμα 3.1.13 *Για όλα τα $n \in \omega$, $\mu_m^e \circ \mu_n^p \sqsubseteq \mathbf{id}_{\mathbf{R}_M(D)}$.*

Λήμμα 3.1.14 *Έστω τα A και B είναι πεδία, $f : A \to B$ μια συνεχής συνάρτηση. Στη συνέχεια για όλα τα $n \in \omega$,*
$\mu_n^p \circ \mathbf{R}_M(f) = \zeta_n^{A,B}\ f \circ \mu_n^p$

Λήμμα 3.1.15 *Έστω το A είναι ένα πεδίο. Στη συνέχεια για όλα τα $n \in \omega$, $\zeta_n^{A,A}\ \mathbf{id}_A = \mathbf{id}_{\mathbf{F}_{M,A}^n(\mathbf{O})}$.*

Λήμμα 3.1.16 *Έστω τα A, B και C είναι πεδία, $f : A \to B$ και $g : B \to C$ συνεχείς συναρτήσεις. Στη συνέχεια για όλα τα $n \in \omega$, $\zeta_n^{A,C}\ (g \circ f) = \zeta_n^{B,C}\ g \circ \zeta_n^{A,B}\ f$.*

Μπορούμε τώρα να προχωρήσουμε με την απόδειξη του θεωρήματος 3.1.2.

Θεώρημα 3.1.2 *Το $\mathbf{R}_M : \mathsf{Dom} \to \mathsf{Dom}$ είναι ένας functor.*
Απόδειξη Πρέπει να αποδείξουμε ότι το $\mathbf{R}_M$ διατηρεί ταυτότητες και τη σύνθεση συνεχών συναρτήσεων.

1. Έστω ότι το X είναι ένα πεδίο και $(x_n)_{n\in\omega} \in \mathbf{R}_M(X)$.

   $\mathbf{R}_M(\mathbf{id}_X)\ (x_n)_{n\in\omega}$
   = ⟨ *Ορισμός του $\mathbf{R}_M$ (3.1.7)* ⟩
   $(\zeta_n^{X,X}\ \mathbf{id}_X\ x_n)_{n\in\omega}$
   = ⟨ *Λήμμα 3.1.15* ⟩
   $(\mathbf{id}_{\mathbf{F}_{M,X}^n(\mathbf{O})}\ x_n)_{n\in\omega}$
   = ⟨ *Ταυτοτική συνάρτηση* ⟩
   $(x_n)_{n\in\omega}$
   = ⟨ *Ταυτοτική συνάρτηση* ⟩
   $\mathbf{id}_{\mathbf{R}_M(X)}\ (x_n)_{n\in\omega}$

2. Έστω ότι τα A και B είναι πεδία, $f : A \to B$ και $g : B \to C$ είναι συνεχείς συναρτήσεις και $(x_n)_{n\in\omega} \in \mathbf{R}_M(X)$.

   $\mathbf{R}_M(g \circ f)\ (x_n)_{n\in\omega}$
   = ⟨ *Ορισμός του $\mathbf{R}_M$ (3.1.7)* ⟩
   $(\zeta_n^{A,C}\ (g \circ f)\ x_n)_{n\in\omega}$

$= \langle$ *Λήμμα 3.1.16* $\rangle$
$((\zeta_n^{B,C}\ g \circ \zeta_n^{A,B}\ f)\ x_n)_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$(\zeta_n^{B,C}\ g\ (\zeta_n^{A,B}\ f\ x_n))_{n\in\omega}$
$= \langle$ *Ορισμός του* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\mathbf{R}_M(g)\ (\zeta_n^{A,B}\ f\ x_n)_{n\in\omega}$
$= \langle$ *Ορισμός του* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\mathbf{R}_M(g)\ (\mathbf{R}_M(f)\ (x_n)_{n\in\omega})$
$= \langle$ *Σύνθεση* $\rangle$
$(\mathbf{R}_M(g) \circ \mathbf{R}_M(f))\ (x_n)_{n\in\omega}$ □

3.2 Unit και join

Έχοντας ορίσει το $\mathbf{R}_M$ ως functor, ορίζουμε τώρα τις δύο συναρτήσεις των μονάδων unit και join. Για καθένα από αυτά, αποδεικνύεται στο [Papa01] ότι πρόκειται για φυσικό μετασχηματισμό.

Η συνάρτηση unit χαρτογραφεί ένα στοιχείο $d \in D$ σε έναν υπολογισμό επανόδου, χρησιμοποιώντας την οικογένεια βοηθητικών συναρτήσεων η. Όλες οι προσεγγίσεις στον υπολογισμό επανόδου είναι ίσες με $\mathit{inl}\ d$ (εκτός από την ασήμαντη προσέγγιση των μηδενικών βημάτων).

Ορισμός 3.2.1 *Έστω το D είναι ένα πεδίο. Για όλα τα $n \in \omega$ ορίζουμε μια συνεχή συνάρτηση $\eta_n^D : D \to \mathbf{F}_{M,D}^n(\mathbf{O})$ με:*

$$\begin{array}{lcl} \eta_0^D & = & \bot \\ \eta_{n+1}^D & = & \mathit{inl} \end{array}$$

Ορισμός 3.2.2 *Έστω το D είναι ένα πεδίο και $d \in D$. Ορίζουμε τη συνάρτηση $\mathbf{unit}_D : D \to \mathbf{R}_M(D)$ με:*

$$\mathbf{unit}_D\ d \quad = \quad (\eta_n^D\ d)_{n\in\omega}$$

Το επόμενο λήμμα είναι χρήσιμο για να αποδείξει ότι το unit είναι φυσικός μετασχηματισμός.

Λήμμα 3.2.1 *Έστω τα A και B είναι πεδία, $f : A \to B$ μια συνεχής συνάρτηση. Στη συνέχεια, για όλα τα $n \in \omega$,*
$\zeta_n^{A,B}\ f \circ \eta_n^A = \eta_n^B \circ f$

το οποίο μπορεί να χρησιμοποιηθεί για να αποδείξει τα ακόλουθα:

Θεώρημα 3.2.1 *Το $\mathbf{unit} : \mathbf{Id} \dot{\to} \mathbf{R}_M$ είναι ένας φυσικός μετασχηματισμός.*
Απόδειξη Έστω τα A και B είναι πεδία και $f : A \to B$ μια συνεχής συνάρτηση. Πρέπει να δείξουμε ότι $\mathbf{unit}_B \circ f = \mathbf{R}_M(f) \circ \mathbf{unit}_A$. Έστω $a \in A$.

$(\mathbf{unit}_B \circ f)\ a$
$= \langle$ *Σύνθεση* $\rangle$
$\mathbf{unit}_B\ (f\ a)$
$= \langle$ *Ορισμός του* $\mathbf{unit}$ *(3.2.2)* $\rangle$
$(\eta_n^D\ (f\ a))_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$((\eta_n^D \circ f)\ a)_{n\in\omega}$

$= \langle$ *Λήμμα 3.2.1* $\rangle$
$\quad ((\zeta_n^{A,B} \circ \eta_n^A)\ a)_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad (\zeta_n^{A,B}\ (\eta_n^A\ a))_{n\in\omega}$
$= \langle$ *Ορισμός του* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\quad \mathbf{R}_M(f)\ (\eta_n^A\ a)_{n\in\omega}$
$= \langle$ *Ορισμός του* $\mathbf{unit}$ *(3.2.2)* $\rangle$
$\quad \mathbf{R}_M(f)\ (\mathbf{unit}_A\ a)$
$= \langle$ *Σύνθεση* $\rangle$
$\quad (\mathbf{R}_M(f) \circ \mathbf{unit}_A)\ a$ □

Ο ορισμός της συνάρτησης $\mathbf{join}$ απαιτεί την οικογένεια των συναρτήσεων ξ που συσχετίζουν τις αντίστοιχες προσεγγίσεις στα πεδία $\mathbf{R}_M(D)$ και D.

Ορισμός 3.2.3 *Έστω το D είναι ένα πεδίο. Για όλα τα $n \in \omega$ ορίζουμε μια αυστηρή συνεχή συνάρτηση $\xi_n^D : \mathbf{F}^n_{M,\mathbf{R}_M(D)}(\mathbf{O}) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ με:*

$$\begin{array}{lcl} \xi_0^D & = & \bot \\ \xi_{n+1}^D f & = & [\,\mu^p_{n+1}, \mathbf{inr} \circ M(\xi_n^D)\,] \end{array}$$

Ορισμός 3.2.4 *Έστω το D είναι domain και $(x_n)_{n\in\omega} \in \mathbf{R}^2_M(D)$. Ορίζουμε τη συνάρτηση $\mathbf{join}_D : \mathbf{R}^2_M(D) \to \mathbf{R}_M(D)$ με:*

$$\mathbf{join}_D\ (x_n)_{n\in\omega} \quad = \quad (\xi_n^D\ x_n)_{n\in\omega}$$

Τα ακόλουθα λήμματα είναι απαραίτητα για να αποδείξουμε ότι το $\mathbf{join}$ είναι ένας φυσικός μετασχηματισμός.

Λήμμα 3.2.2 *Έστω το D είναι ένα πεδίο. Στη συνέχεια για όλα τα $n \in \omega$,*
$\xi_{n+1}^D \circ \mathbf{inr} = \mathbf{inr} \circ M(\xi_n^D)$

Λήμμα 3.2.3 *Έστω τα A και B είναι πεδία, $f : A \to B$ μια συνεχής συνάρτηση. Στη συνέχεια για όλα τα $n \in \omega$,*
$\xi_n^B \circ \zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f)) = \zeta_n^{A,B}\ f \circ \xi_n^A$

Και τέλος μπορεί να αποδειχθεί ότι:

Θεώρημα 3.2.2 *Το $\mathbf{join} : \mathbf{R}^2_M \dot{\to} \mathbf{R}_M$ είναι ένας φυσικός μετασχηματισμός.*
Απόδειξη Έστω τα A και B είναι πεδία και $f : A \to B$ μια συνεχής συνάρτηση. Πρέπει να το δείξουμε αυτό$\mathbf{join}_B \circ \mathbf{R}_M(\mathbf{R}_M(f)) = \mathbf{R}_M(f) \circ \mathbf{join}_A$. Let $(x_n)_{n\in\omega} \in \mathbf{R}^2_M(A)$.

$\quad (\mathbf{join}_B \circ \mathbf{R}_M(\mathbf{R}_M(f)))\ (x_n)_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad \mathbf{join}_B\ (\mathbf{R}_M(\mathbf{R}_M(f))\ (x_n)_{n\in\omega})$
$= \langle$ *Ορισμός του* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\quad \mathbf{join}_B\ (\zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f))\ x_n)_{n\in\omega}$
$= \langle$ *Ορισμός του* $\mathbf{join}$ *(3.2.4)* $\rangle$
$\quad (\xi_n^B\ (\zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f))\ x_n))_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad ((\xi_n^B \circ \zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f)))\ x_n)_{n\in\omega}$
$= \langle$ *Λήμμα 3.2.3* $\rangle$
$\quad ((\zeta_n^{A,B}\ f \circ \xi_n^A)\ x_n)_{n\in\omega}$

$= \langle$ *Σύνθεση* $\rangle$
$\quad (\zeta_n^{A,B}\ f\ (\xi_n^A\ x_n))_{n\in\omega}$
$= \langle$ *Ορισμός του* $\mathbf{R}_M(f)$ *(3.1.7)* $\rangle$
$\quad \mathbf{R}_M(f)\ (\xi_n^A\ x_n)_{n\in\omega}$
$= \langle$ *Ορισμός του* join *(3.2.4)* $\rangle$
$\quad \mathbf{R}_M(f)\ (\mathit{join}_A\ (x_n)_{n\in\omega})$
$= \langle$ *Σύνθεση* $\rangle$
$\quad (\mathbf{R}_M(f) \circ \mathit{join}_A)\ (x_n)_{n\in\omega}$ □

3.3 Η μονάδα $\mathsf{R}(M)$

Σε αυτή την ενότητα δείχνουμε ότι ο functor $\mathbf{R}_M$ μαζί με τους φυσικούς μετασχηματισμούς unit και join ορίζουν μια υπολογιστική monad, κατά την έννοια του [Mogg89]. Τα τρία πρώτα θεωρήματα αυτού του τμήματος επαληθεύουν τους τρεις νόμους των μονάδων και τα ακόλουθα λήμματα είναι απαραίτητα για την απόδειξη τους. Το τέταρτο θεώρημα αποδεικνύει ότι η καθορισμένη μονοάδα ικανοποιεί την μοναδική απαίτηση. Έστω το D είναι ένα πεδίο.

Λήμμα 3.3.1 *Για όλα τα* $n \in \omega$, $\xi_n^D \circ \eta_n^{\mathbf{R}_M(D)} = \mu_n^p$.

Λήμμα 3.3.2 *Για όλα τα* $n \in \omega$, $\mu_{n+1}^p \circ \mathit{unit}_D = \mathit{inl}$.

Λήμμα 3.3.3 *Για όλα τα* $n \in \omega$, $\xi_n^D \circ \zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D = \mathit{id}_{\mathbf{F}_{M,D}^n(\mathbf{O})}$.

Λήμμα 3.3.4 *Για όλα τα* $n \in \omega$, $\mu_n^p \circ \mathit{join}_D = \xi_n^D \circ \mu_n^p$.

Λήμμα 3.3.5 *Για όλα τα* $d \in \omega$, $\xi_n^D \circ \zeta_n^{\mathbf{R}_M^2(D),\mathbf{R}_M(D)}\ \mathit{join}_D = \xi_n^D \circ \xi_n^{\mathbf{R}_M(D)}$.

Τώρα μπορούμε να προχωρήσουμε αποδεικνύοντας τους τρεις νόμους των μονάδων.

Θεώρημα 3.3.1 (1ος Νόμος Μονάδων) $\mathit{join}_D \circ \mathit{unit}_{\mathbf{R}_M(D)} = \mathit{id}_{\mathbf{R}_M(D)}$
Απόδειξη Έστω $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$. Επειτα

$\quad (\mathit{join}_D \circ \mathit{unit}_{\mathbf{R}_M(D)})\ (x_n)_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad \mathit{join}_D\ (\mathit{unit}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega})$
$= \langle$ *Ορισμός του* unit *(3.2.2)* $\rangle$
$\quad \mathit{join}_D\ (\eta_n^{\mathbf{R}_M(D)}\ (x_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Ορισμός του* join *(3.2.4)* $\rangle$
$\quad (\xi_n^D\ (\eta_n^{\mathbf{R}_M(D)}\ (x_m)_{m\in\omega}))_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad ((\xi_n^D \circ \eta_n^{\mathbf{R}_M(D)})\ (x_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Λήμμα 3.3.1* $\rangle$
$\quad (\mu_n^p\ (x_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Ορισμός του* μ_n^p *(3.1.5)* $\rangle$
$\quad (x_n)_{n\in\omega}$
$= \langle$ *Ταυτότητα* $\rangle$
$\quad \mathit{id}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}$ □

Θεώρημα 3.3.2 (2ος Νόμος Μονάδων) $\mathit{join}_D \circ \mathbf{R}_M(\mathit{unit}_D) = \mathit{id}_{\mathbf{R}_M(D)}$

Απόδειξη Έστω $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$. Επειτα

$$
\begin{aligned}
&(\mathit{join}_D \circ \mathbf{R}_M(\mathit{unit}_D))\ (x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Σύνθεση}\ \rangle\\
&\mathit{join}_D\ (\mathbf{R}_M(\mathit{unit}_D)\ (x_n)_{n\in\omega})\\
=\ &\langle\ \textit{Ορισμός του } \mathbf{R}_M \textit{ (3.1.7)}\ \rangle\\
&\mathit{join}_D\ (\zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D\ x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Ορισμός του } \mathit{join} \textit{ (3.2.4)}\ \rangle\\
&(\xi_n^D\ (\zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D\ x_n))_{n\in\omega}\\
=\ &\langle\ \textit{Σύνθεση}\ \rangle\\
&((\xi_n^D \circ \zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D)\ x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Λήμμα 3.3.3}\ \rangle\\
&(\mathit{id}_{\mathbf{F}^n_{M,D}(\mathbf{O})}\ x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Ταυτότητα}\ \rangle\\
&(x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Ταυτότητα}\ \rangle\\
&\mathit{id}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}
\end{aligned}
$$

□

Θεώρημα 3.3.3 (3ος Νόμος Μονάδων) $\mathit{join}_D \circ \mathbf{R}_M(\mathit{join}_D) = \mathit{join}_D \circ \mathit{join}_{\mathbf{R}_M(D)}$

Απόδειξη Έστω $(x_n)_{n\in\omega} \in \mathbf{R}^3_M(D)$. Επειτα

$$
\begin{aligned}
&(\mathit{join}_D \circ \mathbf{R}_M(\mathit{join}_D))\ (x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Σύνθεση}\ \rangle\\
&\mathit{join}_D\ (\mathbf{R}_M(\mathit{join}_D)\ (x_n)_{n\in\omega})\\
=\ &\langle\ \textit{Ορισμός του } \mathbf{R}_M \textit{ (3.1.7)}\ \rangle\\
&\mathit{join}_D\ (\zeta_n^{\mathbf{R}^2_M(D),\mathbf{R}_M(D)}\ \mathit{join}_D\ x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Ορισμός του } \mathit{join} \textit{ (3.2.4)}\ \rangle\\
&(\xi_n^D\ (\zeta_n^{\mathbf{R}^2_M(D),\mathbf{R}_M(D)}\ \mathit{join}_D\ x_n))_{n\in\omega}\\
=\ &\langle\ \textit{Σύνθεση}\ \rangle\\
&((\xi_n^D \circ \zeta_n^{\mathbf{R}^2_M(D),\mathbf{R}_M(D)}\ \mathit{join}_D)\ x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Λήμμα 3.3.5}\ \rangle\\
&((\xi_n^D \circ \xi_n^{\mathbf{R}_M(D)})\ x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Σύνθεση}\ \rangle\\
&(\xi_n^D\ (\xi_n^{\mathbf{R}_M(D)}\ x_n))_{n\in\omega}\\
=\ &\langle\ \textit{Ορισμός του } \mathit{join} \textit{ (3.2.4)}\ \rangle\\
&\mathit{join}_D\ (\xi_n^{\mathbf{R}_M(D)}\ x_n)_{n\in\omega}\\
=\ &\langle\ \textit{Ορισμός του } \mathit{join} \textit{ (3.2.4)}\ \rangle\\
&\mathit{join}_D\ (\mathit{join}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega})\\
=\ &\langle\ \textit{Σύνθεση}\ \rangle\\
&(\mathit{join}_D \circ \mathit{join}_{\mathbf{R}_M(D)})\ (x_n)_{n\in\omega}
\end{aligned}
$$

□

Έχοντας διαπιστώσει ότι το $\mathbf{R}_M$ ικανοποιεί τους τρεις νόμους των μονάδων, μπορούμε τώρα να ολοκληρώσουμε τον ορισμό του μετασχηματιστή μονάδων επανόδου R.

Ορισμός 3.3.1 *Το* resumption monad transformer R *ορίζεται από την απεικόνιση* $\mathsf{R}(M) = \mathbf{R}_M$.

Τώρα αποδεικνύουμε ότι το $\mathsf{R}(M)$ είναι ένα υπολογιστικό monad, όπως ορίζεται στο [Mogg89], και επομένως χρήσιμο ως ένα εξισωτικό μοντέλο υπολογισμών στη σημασιολογία των γλωσσών προγραμματισμού.

Θεώρημα 3.3.4 *Το* $\mathsf{R}(M)$ *είναι μια υπολογιστική μονάδα, δηλαδή ικανοποιεί την απαίτηση μόνο (mono requirement).*

Απόδειξη Έστω το D είναι ένα πεδίο. Πρέπει να δείξουμε ότι το $\mathbf{unit}_D$ είναι ένας μονομορφισμός, δηλαδή πρέπει να δείξουμε ότι για όλα τα πεδία E και για όλες τις συνεχείς συναρτήσεις $f, g : E \to D$, $\mathbf{unit}_D \circ f = \mathbf{unit}_D \circ g$ συνεπάγεται $f = g$. Εξετάστε ένα αυθαίρετο $x \in E$. Τότε, έχουμε

$$
\begin{array}{ll}
& \mathbf{unit}_D\ (f\ x) = \mathbf{unit}_D\ (g\ x) \\
\Leftrightarrow & \langle\ \textit{Ορισμός του } \mathbf{unit}\ \textit{(3.2.2)}\ \rangle \\
& (\eta_n^D\ (f\ x))_{n\in\omega} = (\eta_n^D\ (g\ x))_{n\in\omega} \\
\Leftrightarrow & \langle\ \textit{Η ισότητα άπειρων ακολουθιών ορίζεται σημείο προς σημείο}\ \rangle \\
& \forall\, n \in \omega.\ \eta_n^D\ (f\ x) = \eta_n^D\ (g\ x) \\
\Leftrightarrow & \langle\ \textit{Ορισμός του } \eta\ \textit{(3.2.1), the case } n = 0\ \textit{is trivial}\ \rangle \\
& \mathbf{inl}\ (f\ x) = \mathbf{inl}\ (g\ x) \\
\Leftrightarrow & \langle\ \mathbf{inl}\ \textit{είναι μία εισαγωγή}\ \rangle \\
& f\ x = g\ x
\end{array}
$$

□

3.4 Ισομορφισμός

Έστω το D είναι ένα πεδίο. Σε αυτή την ενότητα, ορίζουμε το ζεύγος συναρτήσεων h^e και h^p που καθορίζουν τον ισομορφισμό μεταξύ των πεδίων $\mathbf{R}_M(D)$ και $D + M(\mathbf{R}_M(D))$. Χρησιμοποιώντας αυτές τις συναρτήσεις, είναι δυνατόν να ορίσετε μια συνάρτηση σε έναν από αυτά τα δύο πεδία και να αποκτήσετε την αντίστοιχη συνάρτηση στο άλλο πεδίο εφαρμόζοντας κατάλληλα τα h^e και h^p.

Ο ορισμός της συνάρτησης ενσωμάτωσης h^e είναι απλός. Χρησιμοποιούμε μια οικογένεια βοηθητικών συναρτήσεων θ, οι οποίες κατασκευάζουν τις απαραίτητες προσεγγίσεις.

Ορισμός 3.4.1 *Για όλα τα* $n \in \omega$ *ορίζουμε μια αυστηρή συνεχή συνάρτηση* $\theta_n^D : \mathbf{F}_{M,D}(\mathbf{R}_M(D)) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ *με:*

$$
\begin{array}{lcl}
\theta_0^D & = & \bot \\
\theta_{n+1}^D & = & [\,\mathbf{inl}, \mathbf{inr} \circ M(\mu_n^p)\,]
\end{array}
$$

Ορισμός 3.4.2 *Έστω* $z \in \mathbf{F}_{M,D}(\mathbf{R}_M(D))$. *Ορίζουμε μια συνεχή συνάρτηση* $h^e : \mathbf{F}_{M,D}(\mathbf{R}_M(D)) \to \mathbf{R}_M(D)$ *με:*

$$
h^e\ z \ = \ (\theta_n^D\ z)_{n\in\omega}
$$

Από την άλλη πλευρά, ο ορισμός της συνάρτησης προβολής h^p είναι πιο περίπλοκος. Απαιτεί πρώτα τον ορισμό ενός πρόσθετου πεδίου $\mathbf{Q}_M(D)$ του οποίου τα στοιχεία είναι άπειρες ακολουθίες υπολογισμών που δίνουν προσεγγίσεις (εμείς θα τις ονομάσουμε *approximate computations* για συντομία). Θεωρούμε επίσης χρήσιμο να ορίσουμε μια οικογένεια βοηθητικών συναρτήσεων σ για τη συσχέτιση στοιχείων $\mathbf{Q}_M(D)$ με προσεγγίσεις στο $\mathbf{R}_M(D)$.

Ορισμός 3.4.3 *Το πεδίο* $\mathbf{Q}_M(D)$ *είναι το σύνολο*

$$
\begin{array}{lll}
\mathbf{Q}_M(D) & = & \{\,(z_n)_{n\in\omega} \mid \forall\, n \in \omega.\ z_n \in M(\mathbf{F}^n_{M,D}(\mathbf{O})) \\
& & \qquad\qquad \wedge\ z_n = M(\mathbf{F}^n_{M,D}(\iota^p))(z_{n+1})\,\}
\end{array}
$$

με τα στοιχεία του να διατάσσονται κατά σημείο:

$$
(z_n)_{n\in\omega} \sqsubseteq_{\mathbf{Q}_M(D)} (w_n)_{n\in\omega} \ \Leftrightarrow\ \forall\, n \in \omega.\ z_n \sqsubseteq_{M(\mathbf{F}^n_{M,D}(\mathbf{O}))} w_n
$$

Ορισμός 3.4.4 *Έστω $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$. Για όλα τα $n \in \omega$, ορίζουμε μια συνεχή συνάρτηση $\sigma_n^D : \mathbf{Q}_M(D) \to \mathbf{F}_{M,D}^n(\mathbf{O})$ από:*

$$\begin{array}{lcl} \sigma_0^D\ (z_m)_{m\in\omega} & = & \bot \\ \sigma_{n+1}^D\ (z_m)_{m\in\omega} & = & \mathbf{inr}\ z_n \end{array}$$

Επιπλέον, ο ορισμός του h^p απαιτεί την απόδειξη του λήμματος 3.4.1, που δηλώνει ότι τα στοιχεία του $\mathbf{R}_M(D)$ έρχονται σε τρεις διακριτές μορφές. Η απόδειξη μπορεί να βρεθεί ξανά στο [Papa01]. Αυτό το λεμμα είναι κρίσιμο για τον ορισμό του h^p και για τις αποδείξεις αρκετών θεωρημάτων που ακολουθούν.

Λήμμα 3.4.1 *Έστω $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$. Στη συνέχεια, ισχύει ένα από τα παρακάτω:*

1. *Για όλα τα $n \in \omega$, $x_n = \bot$.*
2. *Υπάρχει ένα $t \in D$ έτσι ώστε για όλες τις $n \in \omega$, $x_n = \eta_n^D\ t$.*
3. *Υπάρχει ένα $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$ έτσι ώστε για όλες τις $n \in \omega$, $x_n = \sigma_n^D\ (z_m)_{m\in\omega}$.*

Τώρα μπορούμε να προχωρήσουμε στον ορισμό του h^p, με βάση τις τρεις περιπτώσεις του λήμματος 3.4.1. Για τις δύο πρώτες περιπτώσεις, ο ορισμός είναι εύκολος. Στην τρίτη περίπτωση, κάθε κατά προσέγγιση υπολογισμός z_n χαρτογραφείται σε έναν υπολογισμό $M(\mu_n^e)\ z_n \in M(\mathbf{R}_M(D))$ και λαμβάνεται το ελάχιστο ανώτερο όριο αυτής της άπειρης σειράς υπολογισμών.

Ορισμός 3.4.5 *Ορίζουμε την συνάρτηση $h^p : \mathbf{R}_M(D) \to \mathbf{F}_{M,D}(\mathbf{R}_M(D))$ από την περίπτωση ανάλυση στο όρισμα $(x_n)_{n\in\omega}$ με βάση το λήμμα 3.4.1:*

1. *Αν για όλες τις $n \in \omega$, $x_n = \bot$, τότε*
   $h^p\ (x_n)_{n\in\omega} = \bot$
2. *Αν υπάρχει $t \in D$ τέτοια ώστε για όλα τα $n \in \omega$, $x_n = \eta_n^D\ t$, τότε*
   $h^p\ (x_n)_{n\in\omega} = \mathbf{inl}\ t$
3. *Αν υπάρχει $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$ τέτοια ώστε για όλα τα $n \in \omega$, $x_n = \sigma_n^D\ (z_m)_{m\in\omega}$, τότε*
   $$h^p\ (x_n)_{n\in\omega} = \mathbf{inr}\left(\bigsqcup_{n\in\omega} M(\mu_n^e)\ z_n\right)$$

Προκειμένου να διασφαλίσουμε ότι υπάρχει το ελάχιστο ανώτερο όριο στην τρίτη περίπτωση του προηγούμενου ορισμού, χρειαζόμαστε το λήμμα 3.4.2 που δηλώνει ότι το $M(\mu_n^e)\ z_n$ σχηματίζει μια αλυσίδα ω.

Λήμμα 3.4.2 *Έστω $(z_n)_{n\in\omega} \in \mathbf{Q}_M(D)$. Για όλα τα $n \in \omega$,*
$M(\mu_n^e)\ z_n \sqsubseteq M(\mu_{n+1}^e)\ z_{n+1}$

Τα ακόλουθα λήμματα είναι απαραίτητα για να αποδείξουν τα κεντρικά θεωρήματα αυτού του τμήματος.

Λήμμα 3.4.3 *Για όλα τα $t \in D$, για όλα τα $n \in \omega$, $\theta_n^D\ (\mathbf{inl}\ t) = \eta_n^D\ t$.*

Λήμμα 3.4.4 *Για όλα τα $w \in M(\mathbf{R}_M(D))$, για όλα τα $n \in \omega$,*
$\theta_n^D\ (\mathbf{inr}\ w) = \sigma_n^D\ (M(\mu_m^p)\ w)_{m\in\omega}$

Λήμμα 3.4.5 $\bigsqcup_{n\in\omega} \mu_n^e \circ \mu_n^p = \mathbf{id}_{\mathbf{R}_M(D)}$

Λήμμα 3.4.6 *Για όλα τα* $w \in M(\mathbf{R}_M(D))$, $\bigsqcup_{n\in\omega} M(\mu^e_n \circ \mu^p_n)\ w = w$.

Λήμμα 3.4.7 *Έστω* $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$. *Για όλα τα* $m \in \omega$, $\bigsqcup_{n\in\omega} M(f^D_{m,n})\ z_n = z_m$.

Σε αυτό το σημείο, μπορούμε να προχωρήσουμε στο θεώρημα 3.4.1 και το 3.4.2 Αυτά τα δύο θεωρήματα καταλήγουν στο συμπέρασμα ότι οι συναρτήσεις h^e και h^p ορίζουν πράγματι έναν ισομορφισμό μεταξύ των πεδίων $\mathbf{R}_M(D)$ και $D + M(\mathbf{R}_M(D))$.

Θεώρημα 3.4.1 $h^p \circ h^e = id_{\mathbf{F}_{M,D}(\mathbf{R}_M(D))}$
Απόδειξη Έστω $z \in \mathbf{F}_{M,D}(\mathbf{R}_M(D)) = D + M(\mathbf{R}_M(D))$. Με ανάλυση των περιπτώσεων στο z.

1. Περίπτωση $z = \bot$. Τότε

   $(h^p \circ h^e)\ z$
   $= \langle$ *Υπόθεση* $\rangle$
   $(h^p \circ h^e)\ \bot$
   $= \langle$ *Σύνθεση* $\rangle$
   $h^p\ (h^e\ \bot)$
   $= \langle$ *Ορισμός του* h^e *(3.4.2)* $\rangle$
   $h^p\ (\bot)_{n\in\omega}$
   $= \langle$ *Ορισμός του* h^p *(3.4.5)* $\rangle$
   $\bot$
   $= \langle$ *Υπόθεση* $\rangle$
   z

2. Περίπτωση $z = \mathbf{inl}\ t$ for some $t \in D$. Τότε

   $(h^p \circ h^e)\ z$
   $= \langle$ *Υπόθεση* $\rangle$
   $(h^p \circ h^e)\ (\mathbf{inl}\ t)$
   $= \langle$ *Σύνθεση* $\rangle$
   $h^p\ (h^e\ (\mathbf{inl}\ t))$
   $= \langle$ *Ορισμός του* h^e *(3.4.2)* $\rangle$
   $h^p\ (\theta^D_n\ (\mathbf{inl}\ t))_{n\in\omega}$
   $= \langle$ *Λήμμα 3.4.3* $\rangle$
   $h^p\ (\eta^D_n\ t)_{n\in\omega}$
   $= \langle$ *Ορισμός του* h^p *(3.4.5)* $\rangle$
   $\mathbf{inl}\ t$
   $= \langle$ *Υπόθεση* $\rangle$
   z

3. Περίπτωση $z = \mathbf{inr}\ w$ for some $w \in M(\mathbf{R}_M(D))$. Τότε

   $(h^p \circ h^e)\ z$
   $= \langle$ *Υπόθεση* $\rangle$
   $(h^p \circ h^e)\ (\mathbf{inr}\ w)$
   $= \langle$ *Σύνθεση* $\rangle$
   $h^p\ (h^e\ (\mathbf{inr}\ w))$
   $= \langle$ *Ορισμός του* h^e *(3.4.2)* $\rangle$
   $h^p\ (\theta^D_n\ (\mathbf{inr}\ w))_{n\in\omega}$
   $= \langle$ *Λήμμα 3.4.4* $\rangle$
   $h^p\ (\sigma^D_n\ (M(\mu^p_m)\ w)_{m\in\omega})_{n\in\omega}$

$= \langle$ *Ορισμός του* h^p *(3.4.5)* $\rangle$
$\quad \mathbf{inr} \left(\bigsqcup_{n \in \omega} M(\mu_n^e)\ (M(\mu_n^p)\ w) \right)$
$= \langle$ *Σύνθεση* $\rangle$
$\quad \mathbf{inr} \left(\bigsqcup_{n \in \omega} (M(\mu_n^e) \circ M(\mu_n^p))\ w \right)$
$= \langle$ M *είναι ένα functor* $\rangle$
$\quad \mathbf{inr} \left(\bigsqcup_{n \in \omega} M(\mu_n^e \circ \mu_n^p)\ w \right)$
$= \langle$ *Λήμμα 3.4.6* $\rangle$
$\quad \mathbf{inr}\ w$
$= \langle$ *Υπόθεση* $\rangle$
$\quad z$ □

Θεώρημα 3.4.2 $h^e \circ h^p = \mathbf{id}_{\mathbf{R}_M(D)}$
Απόδειξη Έστω $(x_n)_{n \in \omega} \in \mathbf{R}_M(D)$. Με ανάλυση περιπτώσεων σε $(x_n)_{n \in \omega}$ με βάση το λήμμα 3.4.1:

1. Αν για όλες τις $n \in \omega$, $x_n = \bot$, τότε
$\quad (h^e \circ h^p)\ (x_n)_{n \in \omega}$
$= \langle$ *Υπόθεση* $\rangle$
$\quad (h^e \circ h^p)\ (\bot)_{n \in \omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad h^e\ (h^p\ (\bot)_{n \in \omega})$
$= \langle$ *Ορισμός του* h^p *(3.4.5)* $\rangle$
$\quad h^e\ \bot$
$= \langle$ *Ορισμός του* h^e *(3.4.2)* $\rangle$
$\quad (\theta_n^D\ \bot)_{n \in \omega}$
$= \langle$ *Ορισμός του* θ *(3.4.1)* $\rangle$
$\quad (\bot)_{n \in \omega}$
$= \langle$ *Υπόθεση* $\rangle$
$\quad (x_n)_{n \in \omega}$
$= \langle$ *Ταυτότητα* $\rangle$
$\quad \mathbf{id}_{\mathbf{R}_M(D)}\ (x_n)_{n \in \omega}$

2. Αν υπάρχει $t \in D$ τέτοια ώστε για όλα τα $n \in \omega$, $x_n = \eta_n^D\ t$, τότε
$\quad (h^e \circ h^p)\ (x_n)_{n \in \omega}$
$= \langle$ *Υπόθεση* $\rangle$
$\quad (h^e \circ h^p)\ (\eta_n^D\ t)_{n \in \omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad h^e\ (h^p\ (\eta_n^D\ t)_{n \in \omega})$
$= \langle$ *Ορισμός του* h^p *(3.4.5)* $\rangle$
$\quad h^e\ (\mathbf{inl}\ t)$
$= \langle$ *Ορισμός του* h^e *(3.4.2)* $\rangle$
$\quad (\theta_n^D\ (\mathbf{inl}\ t))_{n \in \omega}$
$= \langle$ *Λήμμα 3.4.3* $\rangle$
$\quad (\eta_n^D\ t)_{n \in \omega}$
$= \langle$ *Υπόθεση* $\rangle$
$\quad (x_n)_{n \in \omega}$

$= \langle$ *Ταυτότητα* $\rangle$
$\quad \mathbf{id}_{\mathbf{R}_{M(D)}}\ (x_n)_{n\in\omega}$

3. Αν υπάρχει $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$ τέτοια ώστε για όλα τα $n \in \omega$, $x_n = \sigma_n^D\ (z_m)_{m\in\omega}$, τότε
$\quad (h^e \circ h^p)\ (x_n)_{n\in\omega}$
$= \langle$ *Υπόθεση* $\rangle$
$\quad (h^e \circ h^p)\ (\sigma_n^D\ (z_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad h^e\ (h^p\ (\sigma_n^D\ (z_m)_{m\in\omega})_{n\in\omega})$
$= \langle$ *Ορισμός του* h^p *(3.4.5)* $\rangle$
$\quad h^e\ \left(\mathbf{inr}\ \left(\bigsqcup_{n\in\omega} M(\mu_n^e)\ z_n\right)\right)$
$= \langle$ *Ορισμός του* h^e *(3.4.2)* $\rangle$
$\quad \left(\theta_n^D\ \left(\mathbf{inr}\ \left(\bigsqcup_{n\in\omega} M(\mu_n^e)\ z_n\right)\right)\right)_{n\in\omega}$
$= \langle$ *Λήμμα 3.4.4* $\rangle$
$\quad \left(\sigma_n^D\ \left(M(\mu_m^p)\ \left(\bigsqcup_{n'\in\omega} M(\mu_{n'}^e)\ z_{n'}\right)\right)_{m\in\omega}\right)_{n\in\omega}$
$= \langle\ M(\mu_m^p)$ *είναι συνεχής* $\rangle$
$\quad \left(\sigma_n^D\ \left(\bigsqcup_{n'\in\omega} M(\mu_m^p)\ (M(\mu_{n'}^e)\ z_{n'})\right)_{m\in\omega}\right)_{n\in\omega}$
$= \langle$ *Σύνθεση* $\rangle$
$\quad \left(\sigma_n^D\ \left(\bigsqcup_{n'\in\omega} (M(\mu_m^p) \circ M(\mu_{n'}^e))\ z_{n'}\right)_{m\in\omega}\right)_{n\in\omega}$
$= \langle\ M$ *είναι ένα functor* $\rangle$
$\quad \left(\sigma_n^D\ \left(\bigsqcup_{n'\in\omega} M(\mu_m^p \circ \mu_{n'}^e)\ z_{n'}\right)_{m\in\omega}\right)_{n\in\omega}$
$= \langle$ *Λήμμα 3.1.11* $\rangle$
$\quad \left(\sigma_n^D\ \left(\bigsqcup_{n'\in\omega} M(f_{m,n'}^D)\ z_{n'}\right)_{m\in\omega}\right)_{n\in\omega}$
$= \langle$ *Λήμμα 3.4.7* $\rangle$
$\quad (\sigma_n^D\ (z_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Υπόθεση* $\rangle$
$\quad (x_n)_{n\in\omega}$
$= \langle$ *Ταυτότητα* $\rangle$
$\quad \mathbf{id}_{\mathbf{R}_{M(D)}}\ (x_n)_{n\in\omega}$ □

3.5 Πρόσθετες λειτουργίες

Σε αυτή την ενότητα ορίζουμε δύο συναρτήσεις, $\mathbf{step}$ και $\mathbf{run}$, οι οποίες μετατρέπουν έναν μη παρεμβαλλόμενο υπολογισμό του τύπου $M(A)$ σε έναν interleaved υπολογισμό του τύπου $\mathsf{R}(M)(A)$ και αντίστροφα. Τα ονόματα αυτών των λειτουργιών υποδεικνύουν τη συμπεριφορά τους. Ο πρώτος μετατρέπει έναν ολόκληρο υπολογισμό σε ένα μόνο ατομικό *step* σε έναν μετασχηματισμένο υπολογισμό. Το δεύτερο *runs* ολόκληρη η ακολουθία ατομικών βημάτων ενός διεμπλεκόμενου υπολογισμού

χωρίς να επιτρέπεται η επέμβαση άλλων υπολογισμών. Και οι δύο συναρτήσεις είναι πολύ χρήσιμες για τον προσδιορισμό της σημασιολογίας των παράλληλων γλωσσών προγραμματισμού.

Στην υπόλοιπη ενότητα αυτή υποθέτουμε ότι το $\langle M, \eta, \mu \rangle$ είναι monad και το D είναι domain.

Ορισμός 3.5.1 *$\mathbf{step}_D : M(D) \to \mathsf{R}(M)(D)$ είναι η συνεχής συνάρτηση που ορίζεται από:*

$$\mathbf{step}_D \;=\; h^e \circ \mathbf{inr} \circ M(h^e \circ \mathbf{inl})$$

Ορισμός 3.5.2 *$\mathbf{run}_D : \mathsf{R}(M)(D) \to M(D)$ είναι η συνεχής συνάρτηση που ορίζεται από:*

$$\mathbf{run}_D \;=\; \mathbf{fix}\,(\lambda g.\,[\,\eta_D, \mu_D \circ M(g)\,] \circ h^p)$$

Το ακόλουθο θεώρημα δηλώνει ότι η σύνθεση των $\mathbf{run}$ και $\mathbf{step}$, με αυτή τη σειρά, αποδίδει ταυτότητα. Η αντίστροφη σύνθεση δεν αποδίδει ταυτότητα, δεδομένου ότι υποχρεώνει έναν μεταγλωττισμένο υπολογισμό να εκτελεστεί σε ένα ατομικό βήμα (θα χρησιμοποιηθεί στην ενότητα 4.4 για αυτόν τον ακριβή σκοπό).

Θεώρημα 3.5.1 *$\mathbf{run}_D \circ \mathbf{step}_D = \mathbf{id}_{M(D)}$*
Απόδειξη

$\mathbf{run}_D \circ \mathbf{step}_D$
= ⟨ *Ξεδιπλώνοντας το* **fix** *στον ορισμό του* $\mathbf{run}_D$ *(3.5.2)* ⟩
$[\,\eta_D, \mu_D \circ M(\mathbf{run}_D)\,] \circ h^p \circ \mathbf{step}_D$
= ⟨ *Ορισμός του* $\mathbf{step}_D$ *(3.5.1)* ⟩
$[\,\eta_D, \mu_D \circ M(\mathbf{run}_D)\,] \circ h^p \circ h^e \circ \mathbf{inr} \circ M(h^e \circ \mathbf{inl})$
= ⟨ *Θεώρημα 3.4.1* ⟩
$[\,\eta_D, \mu_D \circ M(\mathbf{run}_D)\,] \circ \mathbf{id}_{\mathbf{F}_{M,D}(\mathbf{R}_M(D))} \circ \mathbf{inr} \circ M(h^e \circ \mathbf{inl})$
= ⟨ *Σύνθεση με την ταυτότητα* ⟩
$[\,\eta_D, \mu_D \circ M(\mathbf{run}_D)\,] \circ \mathbf{inr} \circ M(h^e \circ \mathbf{inl})$
= ⟨ *Θεώρημα 2.3.5* ⟩
$\mu_D \circ M(\mathbf{run}_D) \circ M(h^e \circ \mathbf{inl})$
= ⟨ M *είναι ένα functor* ⟩
$\mu_D \circ M(\mathbf{run}_D \circ h^e \circ \mathbf{inl})$
= ⟨ *Ξεδιπλώνοντας το* **fix** *στον ορισμό του* $\mathbf{run}_D$ *(3.5.2)* ⟩
$\mu_D \circ M([\,\eta_D, \mu_D \circ M(\mathbf{run}_D)\,] \circ h^p \circ h^e \circ \mathbf{inl})$
= ⟨ *Θεώρημα 3.4.1* ⟩
$\mu_D \circ M([\,\eta_D, \mu_D \circ M(\mathbf{run}_D)\,] \circ \mathbf{id}_{\mathbf{F}_{M,D}(\mathbf{R}_M(D))} \circ \mathbf{inl})$
= ⟨ *Σύνθεση με την ταυτότητα* ⟩
$\mu_D \circ M([\,\eta_D, \mu_D \circ M(\mathbf{run}_D)\,] \circ \mathbf{inl})$
= ⟨ *Θεώρημα 2.3.5* ⟩
$\mu_D \circ M(\eta_D)$
= ⟨ M *είναι ένα monad, 2ος νόμος των μονάδων* ⟩
$\mathbf{id}_{M(D)}$ □

Η συνάρτηση $\mathbf{prom}$, η οποία ανυψώνει τον υπολογισμό του τύπου $\mathsf{R}(M)(D)$ σε έναν υπολογισμό του τύπου $M(\mathsf{R}(M)(D))$, είναι χρήσιμος στο υπόλοιπο αυτής της ενότητας όπου διαπιστώνουμε ότι το $\mathsf{R}(M)(D)$ μπορεί να οριστεί ως ένα multi-monad και ένα ισχυρό monad. Αυτές οι δύο ιδιότητες του $\mathsf{R}(M)(D)$ θα χρησιμοποιηθούν επίσης στην ενότητα 4.4.

Ορισμός 3.5.3 *Το $\mathbf{prom}_D : \mathsf{R}(M)(D) \to M(\mathsf{R}(M)(D))$ είναι η συνεχής συνάρτηση που ορίζεται από:*

$$\mathbf{prom}_D \;=\; [\,\eta_{\mathbf{R}_M(D)} \circ \mathbf{inl}, \mathbf{id}_{M(\mathbf{R}_M(D))}\,] \circ h^p$$

Ας υποθέσουμε τώρα ότι το M είναι ένα multi-monad και το $\|_{\mathsf{M}}$ είναι ένας χειριστής *non-deterministic option* για υπολογισμούς που αντιπροσωπεύονται από monad M. Είναι εύκολο να επεκτείνετε αυτή τη συμπεριφορά στο monad $\mathsf{R}(M)$.

Ορισμός 3.5.4 *Έστω το M είναι ένα multi-monad. Έστω το D είναι ένα πεδίο. Ορίζουμε τη δυαδική συνάρτηση $\|_{\mathsf{R(M)}} : \mathsf{R}(M)(D) \times \mathsf{R}(M)(D) \to \mathsf{R}(M)(D)$ από:*

$$x \;\|_{\mathsf{R(M)}}\; y \;=\; h^e\;(\mathbf{inr}\;(\mathbf{prom}\;x\;\|_{\mathsf{M}}\;\mathbf{prom}\;y))$$

Το Monad $\mathsf{R}(M)$ με το $\|_{\mathsf{R(M)}}$ είναι ένα multi-monad.

Επιπλέον, μπορούμε να εισαγάγουμε έναν τρόπο δημιουργίας ενός νέου διεμπλεγμένου υπολογισμού του τύπου $\mathsf{R}(M)(A \times B)$ δίνοντας δύο υπάρχοντες υπολογισμούς των τύπων $\mathsf{R}(M)(A)$ και $\mathsf{R}(M)(B)$. Εδώ προτιμούμε να χρησιμοποιούμε τις μονάδες M και $\mathsf{R}(M)$ με το συναρτησιακό τρόπο. Εάν ένας από τους δύο υπολογισμούς δεν απαιτεί την εκτέλεση οποιουδήποτε ατομικού βήματος, δηλαδή εάν ένας από τους δύο υπολογισμούς έχει ήδη ολοκληρωθεί, ο άλλος υπολογισμός εκτελείται και τα δύο αποτελέσματα συνδυάζονται. Διαφορετικά, αν και οι δύο υπολογισμοί απαιτούν τουλάχιστον ένα ατομικό βήμα, επιλέγουμε μη-ντετερμινιστικά ποιος υπολογισμός θα αρχίσει να εκτελείται.

Ορισμός 3.5.5 *Έστω το M είναι ένα multi-monad. Έστω τα A και B είναι πεδία. Ορίζουμε τη δυαδική συνάρτηση $\bowtie_{\mathsf{R(M)}} : \mathsf{R}(M)(A) \times \mathsf{R}(M)(B) \to \mathsf{R}(M)(A \times B)$ από:*

$$\begin{array}{l}
\bowtie_{\mathsf{R(M)}} \;=\; \mathbf{fix}\;(\lambda\, g.\;\lambda\,\langle x, y\rangle. \\
\quad [\,\lambda\, v_x.\; y \;*_{\mathsf{R(M)}}\; (\lambda\, v_y.\; \mathbf{unit}_{\mathsf{R(M)}}\;\langle v_x, v_y\rangle), \lambda\, m_x. \\
\quad [\,\lambda\, v_y.\; x \;*_{\mathsf{R(M)}}\; (\lambda\, v_x.\; \mathbf{unit}_{\mathsf{R(M)}}\;\langle v_x, v_y\rangle), \lambda\, m_y. \\
\qquad h^e\;(\mathbf{inr}\;(m_x \;*_{\mathsf{M}}\; (\lambda\, x'.\; \mathbf{unit}_{\mathsf{M}}\;(g\;\langle x', y\rangle)) \;\|_{\mathsf{M}} \\
\qquad\qquad\qquad m_y \;*_{\mathsf{M}}\; (\lambda\, y'.\; \mathbf{unit}_{\mathsf{M}}\;(g\;\langle x, y'\rangle))))\,]\;(h^p\;y)\,]\;(h^p\;x))
\end{array}$$

Μία μονάδα $\mathsf{R}(M)$ με $\bowtie_{\mathsf{R(M)}}$ είναι ένα strong monad.

Κεφάλαιο 4

Μια υλοποίηση του RMT

Σε αυτή την ενότητα παρουσιάζουμε την υλοποίηση του *Resumption Monad Transformer (RMT)* στη *JavaScript*. Πρώτον, στην ενότητα 4.1 θα ορίσουμε τις βασικές μονάδες που απαιτούνται για να καθορίσουμε αργότερα τη σημασιολογία μιας απλής προστακτικής γλώσσας τύπου JavaScript. Στην ενότητα 4.2 ορίζουμε τα *State Monad* και *State Monad Transformer* που θα χρειαστούν για να εισαγάγουμε παρενέργειες στην απλή προστακτική γλώσσα μας. Στη συνέχεια, στην ενότητα 4.3 ορίζουμε στη *JavaScript* το *RMT* που περιγράφεται στο Κεφάλαιο 3 μαζί με τις πρόσθετες συναρτήσεις του από την ενότητα 3.5, οι οποίες θα είναι απαραίτητες για να τρέξουμε προγράμματα στη σημασιολογία μας. Τέλος, χρησιμοποιούμε την υλοποίηση του *RMT* για να ορίσουμε τη σημασιολογία της ταυτόχρονης γλώσσας μας.

4.1 Μονάδες

4.1.1 Μονάδα ταυτότητας

Μια λογική επιλογή για τους υπολογισμούς χωρίς κατάσταση και την αριστερά προς δεξιά αποτίμηση των εκφράσεων στη σημασιολογία μας είναι το *Identity monad*.

```
class IdentityM {
    constructor(x) { this.valueId = x; }
    static unit(x) { return new IdentityM(x); }
    bind(f)        { return f(this.valueId); }
}
```

Listing 4.1: Identity monad

4.1.2 Μονάδα λίστας

Για να επιτρέψουμε την αμφισημία στην αποτίμηση των εκφράσεών μας, δεν μπορούμε να χρησιμοποιήσουμε την ταυτότητα. Πρέπει να χρησιμοποιώ αντ ’αυτού μία μονάδα που υποστηρίζει πολλαπλά αποτελέσματα. Η μονάδα του *power-domain*, που συχνά αναφέρεται ως *List monad* σε συναρτησιακές γλώσσες προγραμματισμού, είναι η προφανής επιλογή εδώ. Η μονάδα *λίστας* πρέπει να είναι μια περίπτωση της κλάσης *multi-monad* και να υποστηρίζει ένα *”union”* διαφορετικών αποτελεσμάτων υπολογισμών, το οποίο επιτυγχάνεται με τη `multi` μέθοδο που ορίζεται κατωτέρω.

```
class ListM {
    constructor(l) { this.values = l; }
    static unit(x) { return new ListM([x]); }
    bind(f) {
        return new ListM([].concat.apply([], this.values.map(x => f(x).values)));
    }
    static multi(m1, m2) { return new ListM([].concat(m1.values, m2.values)); }
```

```
}
```

Listing 4.2: List monad

4.1.3 Μονάδα συνόλου

Ως επέκταση στο List monad, ορίζουμε το *Set monad*. Το Set monad έχει βασικά την ίδια διεπαφή με το List monad, αλλά αντί να συνδυάζει τα αποτελέσματα, παρακολουθεί τα ίδια αποτελέσματα κρατώντας μαζί έναν μετρητή των εμφανίσεών τους. Η υλοποίηση εξαρτάται σε μεγάλο βαθμό από το *State* που χρησιμοποιήσαμε, το οποίο θα ορίσουμε στην ενότητα 4.2.1.

```
class SetM {
    constructor(t, d) { this.type = t; this.values = d; }
    static unit(x) {
        return new SetM(false, {[x]: [1, x]});
    }
    bind(f) {
        let d = [];
        for (let [key, val] of this.values) {
            for (let [k, v] of f(key).values) {
                if (d.length == 0) {
                    d.push([k, v]);
                }
                else {
                    for (var i = 0; i < d.length; i++) {
                        if (cmpStates(k.state, d[i][0].state))
                            d[i] = [d[i][0], d[i][1] + 1];
                        else
                            d.push([k, v]);
                    }
                }
            }
        }
        return new SetM(this.type, d);
    }
    static multi(m1, m2) {
        let d = m1.values;
        for (let [key, val] of m2.values) {
            if (d.length == 0) {
                d.push([key, val]);
            }
            else {
                for (var i = 0; i < d.length; i++) {
                    if (cmpStates(key.state, d[i][0].state))
                        d[i] = [d[i][0], d[i][1] + val];
                    else
                        d.push([key, val]);
                }
            }
        }
        return new SetM(m1.type, d);
    }
}
```

Listing 4.3: Set monad

4.2 Μονάδες και κατάσταση

4.2.1 Καταστάσεις προγραμμάτων

Η έννοια της κατάστασης του προγράμματος είναι πολύ σημαντική στη μελέτη των προστακτικών γλωσσών. Μια κατάσταση είναι ένα στοιχείο ενός τύπου που υποστηρίζει δύο κύριες συναρτήσεις, `load` και `store`, για την ανάκτηση και την ενημέρωση των περιεχομένων μιας μεταβλητής στη μνήμη. Ένα διακεκριμένο στοιχείο αυτού του τύπου είναι η `initial` κατάσταση, τυπικά μια κατάσταση με όλες τις μεταβλητές απροσδιόριστες. Αρχικά ορίζουμε ένα απλό stack-based `State` και στη συνέχεια παρουσιάζουμε δύο διαφορετικές εκδοχές του που αποδείχτηκαν να έχουν καλύτερες επιδόσεις στις δοκιμές μας.

```
class State {
    store(x, value) {
        return new StateUpdated(x, value, this);
    }
}

class StateInitial extends State {
    constructor() {
        super();
        this.values = {};
    }
    load(x) {
        show("Undefined variable", x);
        return undefined;
    }
}

class StateUpdated extends State {
    constructor(x, value, parent) {
        super();
        this.x = x;
        this.value = value;
        this.parent = parent;
        // Only for SetM
        this.values = parent.values;
        this.values[x] = value;
    }
    load(x) {
        return x === this.x ? this.value : this.parent.load(x);
    }
}
```

Listing 4.4: A simple stack-based State

```
class ArrayState {
    constructor() {
        this.values = {};
    }
    store(x, value) {
        this.values[x] = value;
        return this;
    }
    load(x) {
        if (x in this.values) {
            return this.values[x];
        } else {
            show("Undefined variable", x);
            return undefined;
        }
```

```
    }
}
```

Listing 4.5: A simple State using an array

```
class MapState {
    constructor() {
        this.values = new Map();
    }
    store(x, value) {
        this.values.set(x, value);
        return this;
    }
    load(x) {
        if (this.values.has(x)) {
            return this.values.get(x);
        } else {
            show("Undefined variable", x);
            return undefined;
        }
    }
}
```

Listing 4.6: A simple State using JavaScript's Map

4.2.2 Μονάδα κατάστασης

Η κατηγορία των μονάδων που γνωρίζουν τη κατάσταση του προγράμματος είναι προφανώς απαραίτητη για τις υλοποιήσεις μας. Η κλάση `StateM` υποστηρίζει δύο συναρτήσεις ως διεπαφή μεταξύ υπολογισμών και της κατάστασης. Η μέθοδος `modify` ενημερώνει την κατάσταση εφαρμόζοντας την συνάρτηση όρισμα της και επιστρέφει έναν υπολογισμό της παλιάς κατάστασης. Η μέθοδος `get` απλά επιστρέφει έναν υπολογισμό της τρέχουσας κατάστασης.

```
class StateM {
    constructor(fun)  { this.runState = fun; }
    static pair(x, s) { return {value: x, state: s}; }
    static unit(x)    { return new StateM(s => pair(x, s)); }
    bind(f) {
        return new StateM(s => {
            let p = this.runState(s);
            return f(p.value).runState(p.state)});
    }
    static get()     { return new StateM(s => (pair(s, s))); }
    static modify(f) { return new StateM(s => (pair(s, f(s)))); }
}
```

Listing 4.7: State monad

4.2.3 Μετασχηματιστής μονάδας κατάστασης

Είναι επίσης χρήσιμο να ορίσουμε έναν μετασχηματιστή μονάδας που εφαρμόζει την προσέγγιση της άμεσης σημασιολογίας (direct semantics approach). Για κάθε τύπο κατάστασης `s` που δίνεται ως παράμετρος, το *State monad transformer* μπορεί να οριστεί ως εξής. Η παράμετρος `M` καθορίζει το monad που αντιπροσωπεύει τους υπολογισμούς χωρίς κατάσταση.

```
function StateT(M) {
    return class SM {
        constructor(fun)  { this.runState = fun; }
        static pair(x, s) { return {value: x, state: s}; }
```

```
        static unit(x)    { return new SM(s => M.unit(this.pair(x, s))); }
        bind(f) {
            return new SM(s =>
                this.runState(s).bind(p => f(p.value).runState(p.state)));
        }
        static get()     { return new SM(s => M.unit(this.pair(s, s))); }
        static modify(f) { return new SM(s => M.unit(this.pair(s, f(s)))); }
        static multi(m1, m2) {
            return new SM(s => M.multi(m1.runState(s), m2.runState(s)));
        }
    }
}
```

Listing 4.8: State monad transformer

Μονάδες που κατασκευάζονται χρησιμοποιώντας το `StateT` γνωρίζουν τη κατάσταση του προγράμματος και ως εκ τούτου το monad `StateT(M)` είναι ένα στιγμιότυπο του *State monad* για έναν τύπο κατάστασης s. Παρατηρούμε εδώ ότι αν στο `StateT(M)` χρησιμοποιούμε το *Identity monad*, παίρνουμε το απλό `StateM` που ορίζεται στο Υποκεφάλαιο 4.2.2.

4.3 Τα RMT στη JavaScript

Σε αυτή την ενότητα ορίζουμε το *resumption monad transformer* που υλοποιεί τη σημασιολογία τύπου δέντρου που ορίσαμε στο Κεφάλαιο 3. Οι επάνοδοι είναι κατασκευές που χωρίζουν έναν υπολογισμό σε ένα μόνο ατομικό βήμα (για εκτέλεση πρώτα) και ένα επαναλαμβανόμενο μέρος, το οποίο αντιστοιχεί στο υπόλοιπο του υπολογισμού. Συνεπώς, η επάνοδος μπορεί να είναι είτε *υπολογισμένη* (Computed), είτε πρέπει να *συνεχιστεί* (Resumed).

Η αρχική μας υλοποίηση είναι η ακόλουθη. Αυτή η υλοποίηση είναι μια ακριβής μετάφραση των ορισμών που δώσαμε στο Κεφάλαιο 3 σε JavaScript. Ωστόσο, λόγω των υπερβολικά πολλών δημιουργιών αντικειμένων και των πάρα πολλών πληροφοριών που διατηρούνται σε κάθε αντικείμενο, διαπιστώσαμε ότι η βελτιστοποιημένη έκδοση που παρουσιάζουμε αργότερα σε αυτήν την ενότητα λειτουργεί πολύ καλύτερα. Επιπλέον, η αρχική έκδοση παίρνει στην πραγματικότητα ένα monad `M` ως όρισμα, αλλά καθώς αυτό είναι πάντα το *State Monad* στις δοκιμές μας, το ενσωματώνουμε στη βελτιστοποιημένη έκδοση μας για να αξιοποιήσουμε καλύτερες μεθόδους υλοποίησης.

```
function ResumptionT(M) {
    return class RM {
        constructor(computed, Mnd, a) {
            // true –> ”Computed”, false –> ”Resume”
            this.status = computed;
            this.Mnd = Mnd;
            this.value = a;
        }
        static unit(x) { return new RM(true, undefined, x); }
        bind(f) {
            if (this.status) {
                return f(this.value);
            }
            else {
                return new RM(false, this.Mnd.bind(r => M.unit(r.bind(f))), undefined);
            }
        }
        static get()     { return RM.stepR(M.get()); }
        static modify(f) { return RM.stepR(M.modify(f)); }

        static runR(R) {
            if (R.status) return M.unit(R.value);
```

```
            else          return R.Mnd.bind(RM.runR);
        }

        static stepR(Mnd) {
            return new RM(false, Mnd.bind(x => M.unit(RM.unit(x))), undefined);
        }

        static multi(m1, m2) {
            return new RM(false, M.multi(M.unit(m1), M.unit(m2)), undefined);
        }
    }
}
```

Listing 4.9: Original resumption monad transformer

4.3.1 Μετασχηματιστής μονάδων επανόδου

Ορίζουμε εδώ την υπερκλάση που υλοποιεί τη βελτιστοποιημένη έκδοση του RMT μας. Αυτή η κλάση θα επεκταθεί από τις κλάσεις των *Computed* και *Resumed* RMTs. Όπως ορίσαμε στην Υποενότητα 3.5, οι RMTs έχουν δύο επιπλέον συναρτήσεις προκειμένου να εκτελέσουν τη σημασιολογία μας. Το `stepR` η μέθοδος παράγει έναν υπολογισμό μόνο ενός ατομικού βήματος, επομένως ορίζεται εδώ ως στατική μέθοδος. Οι μέθοδοι `runR` και `bind` είναι διαφορετικές για κάθε κατηγορία, επομένως αργότερα ορίζονται σε κάθε μία από αυτές. Υλοποιούμε επίσης το `get` και το `modify` του `StateM`, οι οποίες εδώ ανυψώνουν τους υπολογισμούς RMT σε υπολογισμούς που λαμβάνουν υπόψη την κατάσταση του προγράμματος, παρέχοντας έτσι ένα περιβάλλον διεπαφής μεταξύ επανόδων και State monad. Τέλος, οι επάνοδοι πρέπει να είναι μια περίπτωση των multi-monads, επομένως ορίζουμε το `multi` που εφαρμόζει την απαραίτητη συνάρτηση $\|_{R(M)}$.

```
class ResumM {
    static unit(x)    { return new Computed(x); }
    static get() {
        return new Resume(
            new StateM(s => ({value: (new Computed(s)), state: s})));
    }
    static modify(f)  {
        return new Resume(
            new StateM(s => ({value: (new Computed(s)), state: f(s)})));
    }
    static stepR(Mnd) {
        return new Resume(new StateM(s => {
            let p = Mnd.runState(s);
            return {value: new Computed(p.value), state: p.state} }));
    }
    static multi(m1, m2) {
        return new RM(false, StateM.multi(StateM.unit(m1), StateM.unit(m2)), undefined);
    }
}
```

Listing 4.10: Resumption monad transformer

4.3.2 Περίπτωση *Computed*

Η υλοποίηση των επανόδων *Computed* παρουσιάζεται εδώ. Μια υπολογισμένη επάνοδος αποθηκεύει την υπολογιζσμένη τιμή της και η `bind` μέθοδος εφαρμόζει τη συνάρτηση παράμετρο σε αυτήν την τιμή. Η `runR` μέθοδος δεν έχει ατομικά βήματα για να αποτιμήσει σε μια υπολογιζσμένη επάνοδο, επομένως απλώς επιστρέφει μια ένα state monad με την υπολογιζσμένη τιμή της τρέχουσας επανόδου.

```
class Computed extends ResumM {
    constructor(a) { super(); this.value = a; }
    bind(f)        { return f(this.value); }
    runR()         { return StateM.unit(this.value); }
}
```

Listing 4.11: Computed RMT

4.3.3 Περίπτωση *Resume*

Η υλοποίηση των επανόδων *Resumed* παρουσιάζεται εδώ. Μια επάνοδος που πρέπει να συνεχιστεί, αποθηκεύει έναν υπολογισμό μέσα στην παράμετρο monad του και η `bind` μέθοδος εφαρμόζει τη συνάρτηση παράμετρο σε αυτόν τον υπολογισμό. Η `runR` μέθοδος αποτιμά πλήρως την επάνοδο πραγματοποιώντας όλα τα ατομικά βήματα της δεδομένης επανέναρξης.

```
class Resume extends ResumM {
    constructor(Mnd) { super(); this.Mnd = Mnd; }
    bind(f) {
        return new Resume(new StateM(s => {
            let p = this.Mnd.runState(s);
            return {value: p.value.bind(f), state: p.state}}));
    }
    runR() {
        return new StateM(s => {
            let p = this.Mnd.runState(s);
            return p.value.runR().runState(p.state)});
    }
}
```

Listing 4.12: Resumed RMT

4.3.4 Οι επάνοδοι ως ισχυρές μονάδες

Δεδομένου ότι η προαναφερθείσα υλοποίηση των επανόδων πρέπει επίσης να είναι μια περίπτωση των strong monads, πρέπει να τις εξοπλίσουμε με τη συνάρτηση $\bowtie_{R(M)}$ που αναφέρεται στην ενότητα 3.5. Η ακόλουθη υλοποίηση είναι και πάλι μια βελτιστοποιημένη έκδοση που χρησιμοποιείται για τα τεστ αξιολόγησης που επιλέγουν τυχαία μια διαδρομή όλων των πιθανών interleavings.

```
function InterleaveTensor(m1, m2) {
    if (m1.Mnd && m2.Mnd) {
        if (Math.random() <= 0.5)
            return new Resume(
                new StateM(s => {
                    let p = m1.Mnd.runState(s);
                    return {value: InterleaveTensor(p.value, m2), state: p.state}
                }));
        else
            return new Resume(
                new StateM(s => {
                    let p = m1.Mnd.runState(s);
                    return {value: InterleaveTensor(m2, p.value), state: p.state}
                }));
    } else if (m2.Mnd) {
        return new Resume(new StateM(s => {
            let p = m2.Mnd.runState(s);
            return {value: p.value.bind(v2 => new Computed([m1.value, v2])), state: p.state}}))
    } else {
        return new Resume(new StateM(s => {
            let p = m1.Mnd.runState(s);
```

```
            return {value: p.value.bind(v1 => new Computed([v1, m2.value])), state: p.state}}))
    }
}
```

Listing 4.13: Interleaving with non-deterministic choice but executing one path

Για τις εκτελέσεις αναφοράς των τεστ μας που χρησιμοποιούν μόνο το State Monad, ακολουθήσαμε μια απλή αποτίμησης των εκφράσεων από αριστερά προς δεξιά, την οποία επιβάλλει ο απλός ακόλουθος κώδικας.

```
function LeftRightTensor(m1, m2) {
    return m1.bind(v1 =>
           m2.bind(v2 =>
           StateM.unit([v1, v2])));
}
```

Listing 4.14: Simple left-to-right evaluation

4.4 Μια αρθρωτή σημασιολογία του ταυτοχρονισμού

4.4.1 Μία γλώσσα για παράδειγμα

Εξετάζουμε την απλή σειριακή προστακτική γλώσσα της οποίας η αφηρημένη σύνταξη δίνεται παρακάτω.

$$s ::= \textbf{skip} \mid x \textbf{ := } e \mid s \textbf{ ; } s \mid \textbf{if } e \textbf{ then } s \textbf{ else } s \mid \textbf{while } e \textbf{ do } s$$

Διαθέτει μια κενή δήλωση, ανάθεση, διαδοχική σύνθεση δηλώσεων, μια δομή για συνθήκες και μία άλλη για βρόχους *while*. Το σύμβολο $x \in \textbf{Var}$ αντιπροσωπεύει μια μεταβλητή.

Η γλώσσα των εκφράσεων e είναι η ακόλουθη.

$$e ::= n \mid x \mid e + e \mid e * e \mid e \,/\, e \mid \ldots \mid x{+}{+} \mid \ldots \mid e < e \mid e == e \mid \ldots$$

Η γλώσσα των εκφράσεων e αποτελείται από όλους τους αριθμούς και τις μεταβλητές, τις απλές πράξεις όπως την πρόσθεση, αφαίρεση κλπ., πράξεις αύξησης και μείωσης και όλες τις βασικές λογικές εκφράσεις όπως $<$, $<=$, $==$, κλπ. Εδώ παραλείψαμε να ορίσουμε μερικές από τη στιγμή που η σημασιολογία τους θα οριστεί με τον ίδιο τρόπο στην υλοποίηση μας. Το σύμβολο $n \in \textbf{Num}$ αντιπροσωπεύει έναν αριθμό, ο οποίος μπορεί να είναι είτε ένας ακέραιος είτε ένας πραγματικός αριθμός. Το σύμβολο $x \in \textbf{Var}$ αντιπροσωπεύει και πάλι μια μεταβλητή και εδώ χρησιμοποιείται για την πρόσβαση στην τιμή μιας μεταβλητής στη γλώσσα μας.

4.4.2 Η σημασιολογία της γλώσσας μας

Ορίζουμε την δηλωτική σημασιολογία αυτής της γλώσσας, υποθέτοντας ότι οι τιμές των εκφράσεων είναι στοιχεία του σημασιολογικού πεδίου **V**. Το *program state*, που αντιστοιχεί τις μεταβλητές στις τρέχουσες τιμές τους, είναι ένα στοιχείο του πεδίου $\textbf{S} = \textbf{Var} \to \textbf{V}$.

Ως πρόβλεψη για το τι θα ακολουθήσει, ορίζουμε έναν μετασχηματιστή μονάδων D που εφαρμόζει την προσέγγιση *direct semantics*. [1] Αν M είναι monad, ορίζουμε το monad $\mathsf{D}(M)$ ως:

[1] Αυτός είναι ο μετασχηματιστής κατάστασης monad, όπως ορίζεται στο [Lian95, Lian98].

$$
\begin{array}{lcl}
\mathsf{D}(M)(D) & = & \mathbf{S} \to M(D \times \mathbf{S}) \\
\mathit{unit}_{\mathsf{D(M)}}\ v & = & \lambda\sigma.\ \mathit{unit}_{\mathsf{M}}\ \langle v, \sigma\rangle \\
m\ *_{\mathsf{D(M)}}\ f & = & \lambda\sigma.\ m\ \sigma\ *_{\mathsf{M}}\ (\lambda\langle v, \sigma'\rangle.\ f\ v\ \sigma')
\end{array}
$$

Οι υπολογισμοί που δημιουργούνται από τον direct semantics monad transformer είναι συναρτήσεις (στοιχεία του $\mathsf{D}(M)(D)$) που παίρνουν την αρχική κατάσταση προγράμματος (ένα στοιχείο του $\mathbf{S}$) και επιστρέφουν έναν υπολογισμό που αποδίδει την υπολογισμένη τιμή (στοιχείο του D) και την τελική κατάσταση του προγράμματος (ένα στοιχείο του $\mathbf{S}$). Οι υλοποιήσεις του $\mathit{unit}_{\mathsf{D(M)}}$ και του $*_{\mathsf{D(M)}}$ πραγματοποιούν τη διάδοση της κατάστασης του προγράμματος.

Ορίζουμε επίσης μια συνάρτηση για την εκχώρηση τιμών σε μεταβλητές. [2]

$$
\begin{array}{lcl}
\mathit{store}_{\mathsf{D}} & : & \mathbf{Var} \to \mathbf{V} \to \mathsf{D}(M)(\mathbf{U}) \\
\mathit{store}_{\mathsf{D}}\ x\ v & = & \lambda\sigma.\ \mathit{unit}_{\mathsf{M}}\ \langle \mathbf{u}, \sigma\{x \mapsto v\}\rangle
\end{array}
$$

Λαμβάνοντας το *identity monad* Id ως όρισμα του D, λαμβάνουμε το monad M που μοντελοποιεί την απλή μας έννοια του υπολογισμού (ordinary direct semantics).

$$
\mathsf{M} \ = \ \mathsf{D}(\mathsf{Id})
$$

Η έννοια μιας δήλωσης s είναι ένας υπολογισμός $[\![s]\!]$ του τύπου $\mathsf{M}(\mathbf{U})$. Ο μη-τερματισμός αντιπροσωπεύεται από το bottom στοιχείο. Εμείς υποθέτουμε επίσης ότι η έννοια μιας έκφρασης e είναι ένας υπολογισμός $[\![e]\!]$ του τύπου $\mathsf{M}(\mathbf{V})$. Η σημασιολογική συνάρτηση των δηλώσεων της απλής προστακτικής γλώσσας μας είναι εντελώς προφανής.

$$
\begin{array}{l}
[\![\textbf{skip}]\!] \ = \ \mathit{unit}\ \mathbf{u} \\
[\![x := e]\!] \ = \ [\![e]\!] \ * \ (\mathit{store}\ x) \\
[\![s_1 \ ; s_2]\!] \ = \ [\![s_1]\!] \ * \ (\lambda u.\ [\![s_2]\!]) \\
[\![\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2]\!] \ = \ [\![e]\!] \ * \ (\lambda b.\ \text{if } b \text{ then } [\![s_1]\!] \text{ else } [\![s_2]\!]) \\
[\![\textbf{while}\ e\ \textbf{do}\ s]\!] \ = \ \mathit{fix}\ (\lambda g.\ [\![e]\!] \ * \ (\lambda b. \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{if } b \text{ then } [\![s]\!] \ * \ (\lambda u.\ g) \text{ else } \mathit{unit}\ \mathbf{u}))
\end{array}
$$

Η συνάρτηση σημασιολογίας για τις εκφράσεις είναι επίσης αρκετά απλή.

$$
\begin{array}{l}
[\![n]\!] \ = \ \mathit{unit}\ n \\
[\![x]\!] \ = \ \mathit{load}\ x \\
[\![e_1 + e_2]\!] \ = \ [\![e_1]\!] \ * \ (\lambda u_1.\ [\![e_2]\!] \ * \ (\lambda u_2.\ \mathit{unit}\ u_1 + u_2)) \\
[\![e_1 * e_2]\!] \ = \ [\![e_1]\!] \ * \ (\lambda u_1.\ [\![e_2]\!] \ * \ (\lambda u_2.\ \mathit{unit}\ u_1 * u_2)) \\
\ldots \\
[\![x{+}{+}]\!] \ = \ [\![x]\!] \ * \ (\lambda u.\ \mathit{store}\ x\ (u+1)) \\
\ldots \\
[\![e_1 < e_2]\!] \ = \ [\![e_1]\!] \ * \ (\lambda u_1.\ [\![e_2]\!] \ * \ (\lambda u_2.\ \mathit{unit}\ u_1 < u_2)) \\
[\![e_1 == e_2]\!] \ = \ [\![e_1]\!] \ * \ (\lambda u_1.\ [\![e_2]\!] \ * \ (\lambda u_2.\ \mathit{unit}\ u_1 == u_2)) \\
\ldots
\end{array}
$$

Στην υλοποίηση μας στο JavaScript, ορίζουμε μια κλάση για κάθε στοιχείο των αφηρημένων συντάξεων των δηλώσεων s και των εκφράσεων e. Αυτές οι κλάσεις έχουν μια μέθοδο κατασκευαστή για τη διατήρηση των παραμέτρων τους (π.χ. στη δήλωση $x := e$ διατηρούμε τη συμβολοσειρά της μεταβλητής x και την έκφραση e) και τη μέθοδο `sem` που υλοποιεί τη σημασιολογία κάθε στοιχείου.

Για παράδειγμα, η σημασιολογία της δήλωσης *if* υλοποιείται ως εξής:

[2] Αν τα A και B είναι πεδία, $f : A \to B$,$a \in A$ and $b \in B$, χρησιμοποιούμε το συμβολισμό $f\{a \mapsto b\}$ για να δηλώσουμε μια συνάρτηση $f' : A \to B$ έτσι ώστε $f'(a) = b$ και, για όλα τα $x \neq a$, $f'(x) = f(x)$.

```
class StmtIf {
    constructor(e, s1, s2) { this.cond = e; this.then = s1; this.else = s2; }
    sem(M) {
        return this.cond.sem(M).bind(c =>
            c ? this.then.sem(M) : this.else.sem(M));
    }
}
```

Η υλοποίηση της δήλωσης *while* είναι η ακόλουθη:

```
class StmtWhile {
    constructor(e, s) { this.cond = e; this.body = s; }
    sem(M) {
        return fix(g => this.cond.sem(M).bind(c => c ? this.body.sem(M).bind(g) : M.unit()));
    }
}
```

Η δήλωση *while* χρησιμοποιεί το *fixed-point combinator*, το οποίο ορίζουμε ως τον ακόλουθο κώδικα στην υλοποίηση μας.

```
let fix = fun => fun(() => fix(fun));
```

4.4.3 Ταυτοχρονισμός στη γλώσσα μας

Ας εισαγάγουμε τώρα μη-ντετερμινισμό και ταυτοχρονισμό στη γλώσσα μας, επεκτείνοντας τη με τρεις νέες κατασκευές.

$$s \ ::= \ \ldots \mid s \oplus s \mid s \parallel s \mid \langle s \rangle$$

Ο τελεστής $\oplus$ εκτελεί ακριβώς μία από τις δηλώσεις που παίρνει ως ορίσματα. Η επιλογή είναι μη-ντετερμινιστική. Στο άλλο χέρι, ο χειριστής $\parallel$ εκτελεί και τις δύο δηλώσεις που παίρνει ως ορίσματα με ένα παρεμβαλλόμενο τρόπο. Τέλος, το κατασκεύασμα $\langle s \rangle$ τρέχει τη δήλωση s σε ένα μόνο ατομικό βήμα, χωρίς να επιτρέπεται παρεμβολή κατά τη διάρκεια της εκτέλεσης.

Προτού προχωρήσουμε στη σημασιολογία της εκτεταμένης γλώσσας μας, πρέπει να τροποποιήσουμε τον ορισμό του M. Επιλέγοντας το power-domain monad P ως όρισμα του D, αποκτάμε ένα multi-monad που μπορεί να υποστηρίξει μη-ντετερμινισμό.

$$\mathsf{M} \ = \ \mathsf{D}(\mathsf{P})$$

Ο τελεστής επιλογής $\|_{\mathsf{M}}$ ορίζεται ως εξής:

$$m_1 \ \|_{\mathsf{M}} \ m_2 \ = \ \lambda \sigma. \ (m_1 \ \sigma) \ \uplus^{\natural} \ (m_2 \ \sigma)$$

όπου το $\uplus^{\natural}$ είναι η πράξη της ένωσης στα power-domains.

Στη σημασιολογία της εκτεταμένης γλώσσας, χρησιμοποιούμε το monad $\mathsf{R}(\mathsf{M})$ για να μοντελοποιήσουμε διαφυλλώμενους υπολογισμούς. Σύμφωνα με τον ορισμό 3.5.4, το $\mathsf{R}(\mathsf{M})$ είναι ένα multi-monad εξοπλισμένο με μη-ντετερμινιστικούς επιλογή $\|_{\mathsf{R}(\mathsf{M})}$. Επίσης, σύμφωνα το 3.5.5, το $\mathsf{R}(\mathsf{M})$ είναι ένα ισχυρό monad και ο τελεστής $\bowtie_{\mathsf{R}(\mathsf{M})}$ μπορεί να χρησιμοποιηθεί για να μοντελοποιήσει το διαφυλλισμό των υπολογισμών. Επιπλέον, η $\boldsymbol{store}$ μπορεί εύκολα να ανυψωθεί στο νέο πεδίο των υπολογισμών.

$$\begin{array}{lcl} \boldsymbol{store}_{\mathsf{R}} & : & \mathbf{Var} \to \mathbf{V} \to \mathsf{R}(\mathsf{D}(M))(\mathbf{U}) \\ \boldsymbol{store}_{\mathsf{R}} \ x \ v & = & \boldsymbol{step} \ (\boldsymbol{store}_{\mathsf{D}} \ x \ v) \end{array}$$

Οι εξισώσεις που καθορίζουν την έννοια των υφιστάμενων γλωσσικών κατασκευών δεν απαιτούν αλλαγές, εκτός από την αλλαγή ότι οι έννοιες των δηλώσεων και των εκφράσεων αποτελούν πλέον στοιχεία των σημασιολογικών πεδίων $\mathsf{R}(\mathsf{M})(\mathbf{U})$ και $\mathsf{R}(\mathsf{M})(\mathbf{V})$ αντίστοιχα. Από την άλλη πλευρά, η σημασιολογία των πρόσθετων κατασκευασματών μπορούν εύκολα να εκφραστούν με όρους των συναρτήσεων του $\mathsf{R}(\mathsf{M})$.

$$
\begin{aligned}
[\![s_1 \oplus s_2]\!] &= [\![s_1]\!] \parallel_{\mathsf{R}(\mathsf{M})} [\![s_2]\!] \\
[\![s_1 \parallel s_2]\!] &= ([\![s_1]\!] \bowtie_{\mathsf{R}(\mathsf{M})} [\![s_2]\!]) * (\lambda p.\ \mathit{unit}\ u) \\
[\![\langle s \rangle]\!] &= \mathit{step}\ (\mathit{run}\ [\![s]\!])
\end{aligned}
$$

Οι παραπάνω τελεστές $\oplus$, $\parallel$ και η κατασκευή $\langle s \rangle$ της εκτεταμένης γλώσσας υλοποιούνται απλά ως εξής:

```
class ExprChoice {
    constructor(e1, e2) { this.left = e1; this.right = e2; }
    sem(M) { return M.multi(this.left.sem(M), this.right.sem(M)); }
}

class ExprInterleave {
    constructor(e1, e2) { this.left = e1; this.right = e2; }
    sem(M) { return M.tensor(this.left.sem(M), this.right.sem(M)); }
}

class ExprUnit {
    constructor(e) { this.body = e; }
    sem(M) { return M.stepR(this.body.sem(M).runR()); }
}
```

Εφαρμόσαμε επίσης μια εναλλακτική λύση για τον χειριστή $\parallel$, ο οποίος έχει την ίδια σημασιολογία αλλά λαμβάνει μια σειρά δηλώσεων αντί για δύο. Η χρήση αυτού του χειριστή θα είναι πιο ξεκάθαρη στα τεστ αξιολόγησης του 5.1.6 εάν δημιουργήσουμε περισσότερα από δύο νήματα.

```
class ExprInterleaveMany {
    constructor(arr) { this.array = arr; }
    sem(M) {
        return M.get().bind(s =>
                  M.tensorMany(s.load(this.array).map(x => x.sem(M)), 0));
    }
}
```

Αυτός ο τελεστής χρησιμοποιεί επίσης μια τροποποιημένη έκδοση του `tensor` των RMT που εκτελεί τη δεδομένη συστοιχία νημάτων με κυκλικό τρόπο. Η υλοποίηση του είναι η ακόλουθη:

```
function InterleaveTensorMany(threads, id) {
    let not_computed = threads.reduce(((x, y, id) => x + (y.Mnd ? 1 : 0)), 0);
    let num_of_threads = threads.length;
    let i = id == num_of_threads ? 0 : id;
    while (threads[i].Mnd === undefined) {
        i += 1;
        if (i == num_of_threads) i = 0;
    }
    if (not_computed > 1) {
        return new Resume(
            new StateM(s => {
                let p = threads[i].Mnd.runState(s);
                threads[i] = p.value;
                return {value: InterleaveTensorMany(threads, i + 1), state: p.state}
            }));
    } else {
        return new Resume(new StateM(s => {
            let p = threads[i].Mnd.runState(s);
```

```
            return {value: p.value.bind(v => {
                let results = threads.map(x => x.value);
                results[i] = v;
                return new Computed(results)
            }), state: p.state}}))
    }
}
```

Η προαναφερθείσα δηλωτική σημασιολογία της παραδειγματικής γλώσσας χρησιμοποιείται για την εκτέλεση των κριτηρίων αξιολόγησης που παρουσιάζονται στην ενότητα 5.1. Ο μετασχηματιστής μονάδων D που εφαρμόζει την προσέγγιση *direct semantics*, ο οποίος περιγράφεται στην αρχή αυτής της ενότητας, με το *identity monad* Id ως όρισμά του, χρησιμοποιείται για την αξιολόγηση των βασικών προγραμμάτων *sequential*. Στη συνέχεια, επιλέγοντας το power-domain monad P ως το όρισμα του D, αποκτάμε ένα multi-monad που μπορεί να υποστηρίξει μη-ντετερμινισμό και περνώντας αυτό ως M στο monad R(M), μπορούμε να μοντελοποιήσουμε τους διαφυλλώμενους υπολογισμούς και να το χρησιμοποιήσουμε για τα *concurrent* τεστ αξιολόγησης.

Κεφάλαιο 5

Τα αποτελέσματα απόδοσης

5.1 Τεστ αξιολόγησης

Σε αυτή την ενότητα περιγράφονται όλοι οι αλγόριθμοι που χρησιμοποιήθηκαν για τη μέτρηση της απόδοσης της υλοποίησης μας για τους μετασχηματιστές μονάδων επανόδου και των υλοποιήσεων αναφοράς με τα promises. Όλα τα τεστ αξιολόγησης γράφτηκαν σε *JavaScript* και εκτελέστηκαν στο περιβάλλον εκτέλεσης *Node.js* [Node18], χρησιμοποιώντας την έκδοση *10.3.0*. Τα τεστ αξιολόγησης είναι βασικά απλοί και συνηθισμένοι αλγόριθμοι διαφόρων χρονικών πολυπλοκοτήτων π.χ. $O(n)$, $O(n\ logn)$, $O(n^2)$, $O(2^n)$. Παρέχουμε εδώ τον ψευδοκώδικα τόσο για τις σειριακές όσο και για τις ταυτόχρονες εκδόσεις των υλοποιήσεών μας.

Για να αξιολογήσουμε την απόδοση χρόνου εκτέλεσης της υλοποίησης των RMT για τους σκοπούς αυτής της εργασίας, οι αλγόριθμοι που περιγράφονται παρακάτω γράφονται χρησιμοποιώντας τη σημασιολογία μας. Τα *Σειριακά* (Sequential) εκτελέστηκαν με τη σημασιολογία μας, περνώντας τους ως παράμετρο το State Monad Transformer που ανυψώνει το Identity Monad και για τα *Ταυτόχρονα* (Concurrent), περνώντας τους το RMT ανυψώνοντας τον ίδιο μετασχηματιστή με τους σειριακούς ομολόγους τους.

Για τα τεστ αναφοράς, χρησιμοποιήσαμε τους ίδιους αλγόριθμους τους οποίους τους υλοποιήσαμε σε απλή JavaScript για τις *σειριακές* εκδόσεις και μετασχηματίσαμε τον ίδιο κώδικα χρησιμοποιώντας τα *Promises* της JavaScript για τα *ταυτόχρονα*. Ωστόσο, η βασική υλοποίηση παραμένει η ίδια στις εκδόσεις RMT και Promises.

5.1.1 Το κόσκινο του Ερατοσθένη

Το πρώτο τεστ αξιολόγησης μας είναι ο κλασικός αλγόριθμος για το κόσκινο του Ερατοσθένη. Έχει χρονική πολυπλοκότητα $O(n\ loglogn)$ και ο βασικός βρόχος μπορεί να σπάσει σε δύο (ή περισσότερα) ξεχωριστά νήματα για την ταυτόχρονη έκδοση. Ο κοινός σειριακός αλγόριθμος παρουσιάζεται στον Αλγόριθμο 1. Στον παράλληλο αλγόριθμο - Αλγόριθμος 2 - δημιουργούνται δύο νήματα που έχουν το ίδιο σώμα με το βρόχο while στο 1, αλλά ξεκινούν με διαφορετικούς αριθμούς και έχουν βήμα ίσο με 2 σε κάθε επανάληψη. Το πρόβλημα εδώ είναι η ανάγνωση ή η εγγραφή του παγκόσμιου πίνακα $primes$, όπου διατηρούνται τα αποτελέσματα αν ο σχετικός δείκτης είναι πρώτος αριθμός ή όχι. Πρέπει να υπάρχει μια κλειδαριά για να αποφευχθούν συνθήκες αγώνα και να εξασφαλιστεί ότι θα εκτελεστούν οι ίδιες λειτουργίες όπως και στη σειριακή. Στη σημασιολογία μας, αυτό μπορεί να γίνει απλά με το τελεστή μας $unit$, ενώ στην υλοποίηση των promises πρέπει να διατηρήσουμε ολόκληρη τη λειτουργία ανάγνωσης ή γραφής σε ένα ατομικό βήμα μέσα σε ένα ξεχωριστό promise για να επιτύχουμε τα ίδια αποτελέσματα.

Αλγόριθμος 1 Σειριακό κόσκινο του Ερατοσθένη

```
1:  procedure SeqSieve(n)
2:      primes ← [true] * n
3:      j ← 4
4:      while j <= n do
5:          primes[j] ← false
6:          j ← j + 2
7:      end while
8:      i ← 3
9:      while i <= √n do
10:         if primes[i] is true then
11:             j ← i^2
12:             while j <= n do
13:                 primes[j] ← false
14:                 j ← j + i
15:             end while
16:         end if
17:         i ← i + 2
18:     end while
19:     return [i | i ∈ [1..n], primes[i] is true]
20: end procedure
```

Αλγόριθμος 2 Ταυτόχρονο κόσκινο του Ερατοσθένη

```
1:  function Fork(start)
2:      i_local ← start
3:      while i_local <= √n do
4:          if ⟨primes[i_local] is true⟩ then
5:              j_local ← i_local^2
6:              while j_local <= n do
7:                  ⟨primes[j_local] ← false⟩
8:                  j_local ← j_local + i_local
9:              end while
10:         end if
11:         i_local ← i_local + 4
12:     end while
13: end function

14: procedure ConcSieve(n)
15:     primes ← [true] * n
16:     j ← 4
17:     while j <= n do
18:         primes[j] ← false
19:         j ← j + 2
20:     end while
21:     thread_1 ← Fork(3)
22:     thread_2 ← Fork(5)
23:     run (thread_1 || thread_2)
24:     return [i | i ∈ [1..n], primes[i] is true]
25: end procedure
```

5.1.2 Προσέγγιση του Pi

Το δεύτερο τεστ αξιολόγησης μας είναι ένας απλός αλγόριθμος για την προσέγγιση του Pi. Έχει χρονική πολυπλοκότητα $O(n)$ και ο βασικός βρόχος μπορεί να χωριστεί σε δύο ξεχωριστά νήματα για την ταυτόχρονη έκδοση. Ο σειριακός αλγόριθμος παρουσιάζεται στον Αλγόριθμο 3. Στο ταυτόχρονο αλγόριθμο - αλγόριθμος 4 - δημιουργούνται δύο νήματα που έχουν το ίδιο σώμα με το βρόχο while στο 3, αλλά ξεκινούν με διαφορετικούς αριθμούς (1 και 2) με βήμα 2 σε κάθε επανάληψη. Με αυτόν τον τρόπο μπορούμε να αποφύγουμε στην ταυτόχρονη έκδοση να ελέγξουμε αν το i είναι ζυγός ή μονός. Κάθε νήμα τρέχει είτε για τους ζυγούς αριθμούς είτε για τους μονούς ξεχωριστά. Το πρόβλημα εδώ είναι το ίδιο με το κόσκινο παραπάνω 2 - διαβάζοντας ή γράφοντας την παγκόσμια μεταβλητή pi, όπου διατηρείται η προσέγγιση του pi. Στη σημασιολογία μας, απλά χρησιμοποιούμε ξανά τον τελεστή μας $unit$, ενώ στην υλοποίηση των υποσχέσεων πρέπει να διατηρήσουμε ολόκληρη τη λειτουργία ανάγνωσης ή γραφής σε ένα ατομικό βήμα μέσα σε ένα ξεχωριστό promise για να πετύχουμε τα ίδια αποτελέσματα.

Αλγόριθμος 3 Σειριακή προσέγγιση του Pi

```
1:  procedure SeqPi(loops)
2:      i ← 1
3:      pi ← 4
4:      while i < loops do
5:          temp ← 4/(i * 2 + 1)
6:          if i mod 2 == 0 then
7:              pi ← pi + temp
8:          else
9:              pi ← pi − temp
10:         end if
11:         i ← i + 1
12:     end while
13:     return pi
14: end procedure
```

Αλγόριθμος 4 Ταυτόχρονη προσέγγιση της Pi

```
1:  function Thread1
2:      i_local ← 2
3:      while i_local < loops do
4:          ⟨pi ← pi + 4/(i_local * 2 + 1)⟩
5:          i_local ← i_local + 2
6:      end while
7:  end function

8:  function Thread2
9:      i_local ← 1
10:     while i_local < loops do
11:         ⟨pi ← pi − 4/(i_local * 2 + 1)⟩
12:         i_local ← i_local + 2
13:     end while
14: end function

15: procedure RmtPi(loops)
16:     pi ← 4
17:     thread_1 ← Thread1
18:     thread_2 ← Thread2
19:     run (thread_1 || thread_2)
20:     return pi
21: end procedure
```

5.1.3 Τεστ επαλήθευσης πρώτου αριθμού

Το τρίτο τεστ αξιολόγησης μας είναι ένα κλασικό τεστ για το αν ένας δεδομένος ακέραιος αριθμός είναι πρώτος. Έχει χρονική πολυπλοκότητα $O(\sqrt{n})$ και πάλι ο βασικός βρόχος μπορεί να χωριστεί σε δύο ξεχωριστά νήματα για την ταυτόχρονη έκδοση. Αυτή η δοκιμή εκτελείται φυσικά για πολύ μεγάλους πρώτους αριθμούς. Ο σειριακός αλγόριθμος παρουσιάζεται στον Αλγόριθμο 5. Στο ταυτόχρονο αλγόριθμο - Αλγόριθμος 6 - δημιουργούνται δύο νήματα που έχουν το ίδιο σώμα με τον βασικό βρόχο στο 5, αλλά ξεκινούν με διαφορετικούς αριθμούς και κάθε νήμα κάνει το μισό του ελέγχου αποφεύγοντας ορισμένες πράξεις.

Αλγόριθμος 5 Σειριακό τεστ πρώτων αριθμών

```
1:  procedure SeqPrimalityTest(n)
2:      result ← true
3:      if n <= 16 then
4:          if n == 2 or n == 3 or n == 5 or n == 7 or n == 11 or n == 13 then
5:              result ← false
6:          end if
7:      else
8:          if ((n mod 2 == 0) or (n mod 3 == 0) or
9:              (n mod 5 == 0) or (n mod 7 == 0)) then
10:             result ← false
11:         else
12:             i ← 10
13:             while i^2 <= n do
14:                 if ((n mod (i + 1) == 0) or (n mod (i + 3) == 0) or
15:                     (n mod (i + 7) == 0) or (n mod (i + 9) == 0)) then
16:                     result ← false
17:                     i ← n + 1
18:                 end if
19:                 i ← i + 10
20:             end while
21:         end if
22:     end if
23:     return result
24: end procedure
```

Αλγόριθμος 6 Ταυτόχρονο τεστ πρώτων αριθμών

```
1:  function Fork(start)
2:      i_local ← start
3:      while i_local^2 <= n do
4:          if (n mod i_local == 0) or (n mod (i_local + 2) == 0) then
5:              result ← false
6:              i_local ← n + 1
7:              i_other ← n + 1
8:          end if
9:          i_local ← i_local + 10
10:     end while
11: end function

12: procedure RmtPrimalityTest(n)
13:     result ← true
14:     if n <= 16 then
15:         if n == 2 or n == 3 or n == 5 or n == 7 or n == 11 or n == 13 then
16:             result ← false
17:         end if
18:     else
19:         if ((n mod 2 == 0) or (n mod 3 == 0) or
20:             (n mod 5 == 0) or (n mod 7 == 0)) then
21:             result ← false
22:         else
23:             thread_1 ← Fork(11)
24:             thread_2 ← Fork(17)
25:             run (thread_1 || thread_2)
26:         end if
27:     end if
28:     return result
29: end procedure
```

5.1.4 Ταξινόμηση εισαγωγής

Το τέταρτο τεστ αξιολόγησης μας είναι η ταξινόμηση εισαγωγής. Έχει πολυπλοκότητα χρόνου $O(n^2)$, ειδικά στην περίπτωσή μας, όπου οι δοκιμές αποτελούνται από συστοιχίες αριθμών αντιστρόφως ταξινομημένων. Ο σειριακός αλγόριθμος παρουσιάζεται στον Αλγόριθμο 7. Για τις ταυτόχρονες εκδόσεις τρέχουμε τον ίδιο σειριακό αλγόριθμο, χωρίς όμως να δημιουργούμε νήματα που διαφυλλώνονται. Με αυτό τον τρόπο μπορούμε να υπολογίσουμε την επιβράδυνση που η υλοποίηση μας και τα promises έχουν σε σχέση με τους σεριακούς ομολόγους τους για την εκτέλεση του ίδιου αριθμού υπολογισμών.

Αλγόριθμος 7 Ταξινόμηση εισαγωγής

```
1:  procedure Insert(A)
2:      i ← 0
3:      while i < A.length do
4:          value ← A[i]
5:          j ← i
6:          while j > 0 and value < A[j − 1] do
7:              A[j] ← A[j − 1]
8:              j ← j − 1
9:          end while
10:         A[j] ← value
11:         i ← i + 1
12:     end while
13:     return A
14: end procedure
```

5.1.5 Μείωση πίνακα

Το πέμπτο τεστ αξιολόγησης μας είναι μια απλή μείωση ενός δεδομένου πίνακα, προκειμένου να έχουμε το άθροισμα των στοιχείων του πίνακα. Έχει φυσικά πολυπλοκότητα χρόνου $O(n)$ και, πάλι ο βασικός βρόχος μπορεί να χωριστεί σε δύο ξεχωριστά νήματα για την ταυτόχρονη έκδοση. Αυτό το τεστ, φυσικά, εκτελέστηκε για μεγάλους πίνακες. Ο σειριακός αλγόριθμος παρουσιάζεται στον Αλγόριθμο 8. Στον ταυτόχρονο αλγόριθμο - Αλγόριθμος 9 - δημιουργούνται δύο νήματα που έχουν τον ίδιο βρόχο όπως στο 8, αλλά κάθε νήμα αντιμετωπίζει ένα διαφορετικό ήμισυ του πίνακα. Το αποτέλεσμα αποθηκεύεται σε μια παγκόσμια μεταβλητή x, έτσι και πάλι πρέπει να "κλειδώσουμε" το x για να αποφύγουμε τους αγώνες δεδομένων.

Αλγόριθμος 8 Σειριακή μείωση του πίνακα

```
1:  procedure SeqReduce(A)
2:      x ← 0
3:      i ← 0
4:      while i < A.length do
5:          x ← x + A[i]
6:          i ← i + 1
7:      end while
8:      return x
9:  end procedure
```

Αλγόριθμος 9 Ταυτόχρονη

```
1:  function Fork(start, finish)
2:      i_local ← start
3:      while i_local < finish do
4:          ⟨x ← x + A[i_local]⟩
5:          i_local ← i_local + 1
6:      end while
7:  end function

8:  procedure RmtReduce(A)
9:      x ← 0
10:     n ← A.length
11:     thread_1 ← Fork(0, n/2)
12:     thread_2 ← Fork(n/2, n)
13:     run (thread_1 || thread_2)
14:     return x
15: end procedure
```

5.1.6 Πολλαπλασιασμός πίνακα-διανύσματος

Το έκτο τεστ αξιολόγησης μας είναι ένας πολλαπλασιασμός διανύσματος-φορέα ενός δεδομένου πίνακα και ενός δεδομένου διανύσματος, ο οποίος έχει πολυπλοκότητα χρόνου $O(n^2)$. Ο σειριακός αλγόριθμος παρουσιάζεται στον Αλγόριθμο 10. Στο ταυτόχρονο αλγόριθμο - Αλγόριθμος 11 - κάθε νήμα εκτελεί πολλαπλασιασμό διανύσματος με διάνυσμα. Επομένως, σε αυτό το τεστ αξιολόγησης δημιουργούμε dim σε αριθμό νήματα, όπου dim είναι ο αριθμός των γραμμών στο πίνακα εισόδου και τα αποθηκεύουμε σε ένα διάνυσμα $threads$. Όσον αφορά το τελικό αποτέλεσμα, κάθε νήμα διατηρεί το αποτέλεσμά του αποθηκευμένο σε διαφορετική θέση του παγκόσμιου πίνακα y, επομένως δεν απαιτούνται "κλειδώματα" εδώ. Πρέπει όμως να ορίσουμε έναν τρόπο για να εκτελεστούν τα πολλά νήματα του διανύσματος $threads$. Για την έκδοση των promises μπορούμε να χρησιμοποιήσουμε το $Promise.all(threads)$ αλλά στην έκδοση των RMT πρέπει να ορίσουμε έναν παρόμοιο τελεστή. Το **runMany** κάνει ακριβώς αυτό. Εκτελεί τους βασικούς υπολογισμούς των νημάτων με κυκλικό τρόπο όπως αναμένεται να κάνει το $Promise.all()$ και έχει παρόμοια σημασιολογία με το **run** αλλά για μια σειρά από νήματα, όπως φαίνεται στο σχόλιο της γραμμής *15* στον Αλγόριθμο 11.

Αλγόριθμος 10 Σειριακός πολλαπλασιασμός πίνακα-διανύσματος

```
1:  procedure SeqMatrixVectorMult(A, x)
2:      y ← [0] * A.NumOfRows
3:      for i ← 0; i < A.NumOfRows; i ← i + 1 do
4:          temp ← 0
5:          for j ← 0; j < A.NumOfColumns; j ← j + 1 do
6:              temp ← temp + A[i][j] * x[j]
7:          end for
8:          y[i] ← temp
9:      end for
10:     return y
11: end procedure
```

Αλγόριθμος 11 Ταυτόχρονος πολλαπλασιασμός πίνακα-διανύσματος

```
1:  function Fork(id)
2:      y_id ← 0
3:      A_id ← A.Row(id)
4:      for j_id ← 0; j_id < A_id.length; j_id ← j_id + 1 do
5:          y_id ← y_id + A_id[j_id] * x[j_id]
6:      end for
7:      y[id] ← y_id
8:  end function

9:  procedure RmtMatrixVectorMult(A, x)
10:     y ← [0] * A.NumOfRows
11:     y ← [null] * A.NumOfRows
12:     for i ← 0; i < A.NumOfRows; i ← i + 1 do
13:         thread_i ← Fork(i)
14:     end for
15:     runMany threads                    ▷ run foldl(λx.λy. x ‖ y, threads, skip)
16:     return y
17: end procedure
```

5.1.7 Συνδυασμοί

Το τελευταίο τεστ αξιολόγησης μας είναι να βρούμε όλους τους διαφορετικούς συνδυασμούς m από n (εύρος 1 έως n για τις δοκιμές μας). Έχει χρονική πολυπλοκότητα $O(2^n)$ αφού ο αλγόριθμος μας απαριθμεί απλώς όλους τους πιθανούς συνδυασμούς. Ο σειριακός αλγόριθμος παρουσιάζεται στον Αλγόριθμο 12. Όπως κάναμε και στον Αλγόριθμο 7, για τις ταυτόχρονες εκδόσεις τρέχουμε τον ίδιο σειριακό αλγόριθμο, χωρίς όμως να έχουμε ταυτόχρονα νήματα που διαφυλλώνονται, για να υπολογίσουμε την επιβράδυνση των υλοποιήσεών μας.

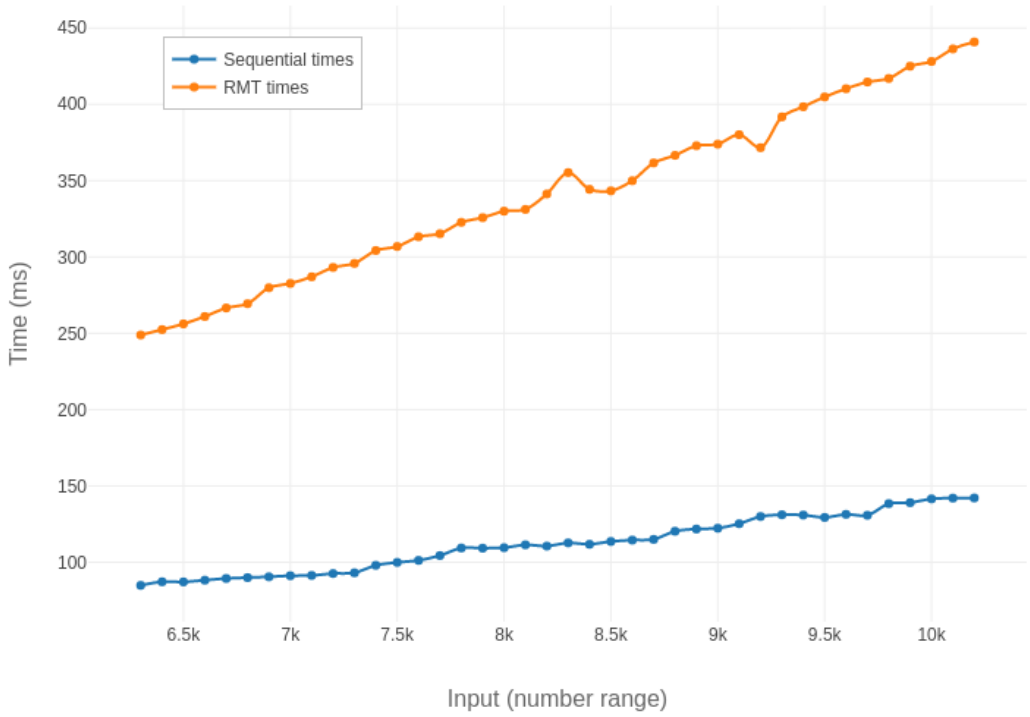
Αλγόριθμος 12 Συνδυασμοί (m ανά n)

```
1:  procedure Combinations(m, n)
2:      bits ← 1
3:      results ← []
4:      while bits <= 2^n − 1 do
5:          if num. of 1s in binary representation of bits is m then
6:              results.push(list of 1s indexes in binary representation of bits)
7:          end if
8:          bits ← bits + 1
9:      end while
10:     return x
11: end procedure
```

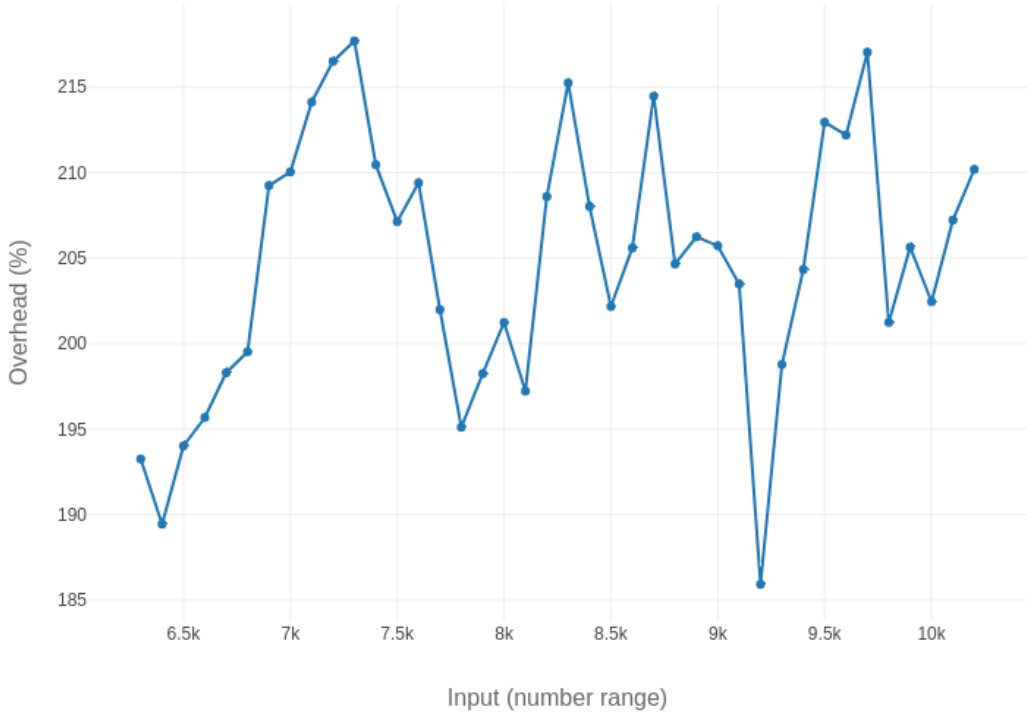
5.2 Αποτελέσματα

5.2.1 Αποτελέσματα δοκιμών απόδοσης

Κάθε τεστ αξιολόγησης που περιγράφηκε στην Ενότητα 5.1 εκτελέστηκε 100 φορές και παρουσιάζονται εδώ οι μέσοι χρόνοι εκτέλεσης. Επιπλέον, παρουσιάζουμε την επιβράδυνση που τα προγράμματα των RMT έχουν σε σχέση με τα αντίστοιχα σειριακά σε ξεχωριστό σχήμα για κάθε τεστ.

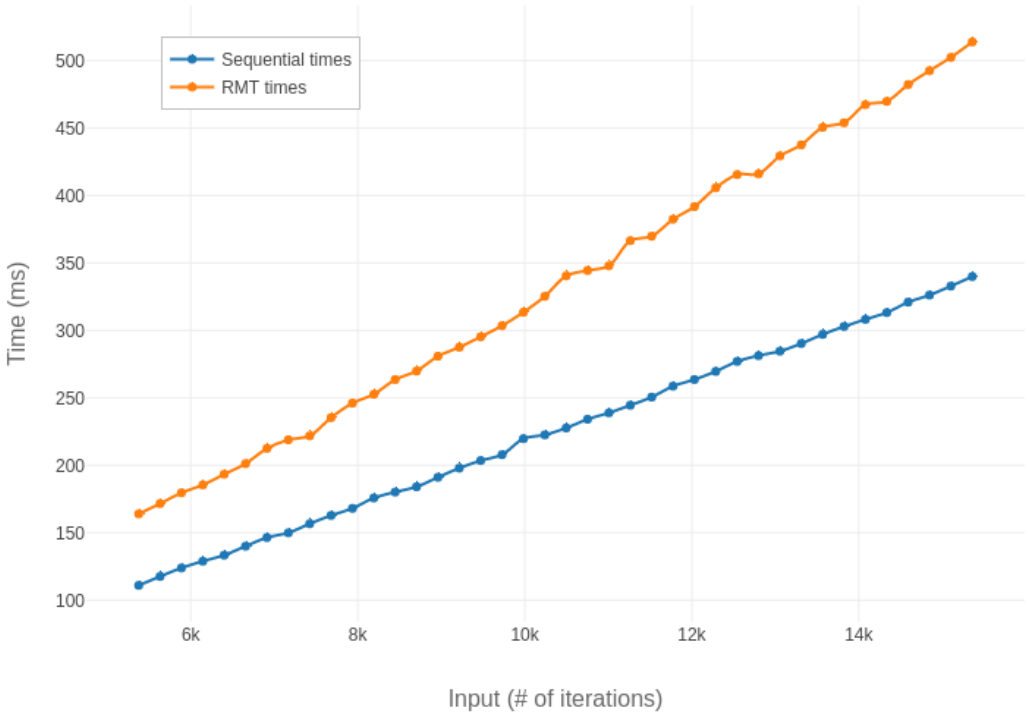


(a) Χρόνοι εκτέλεσης

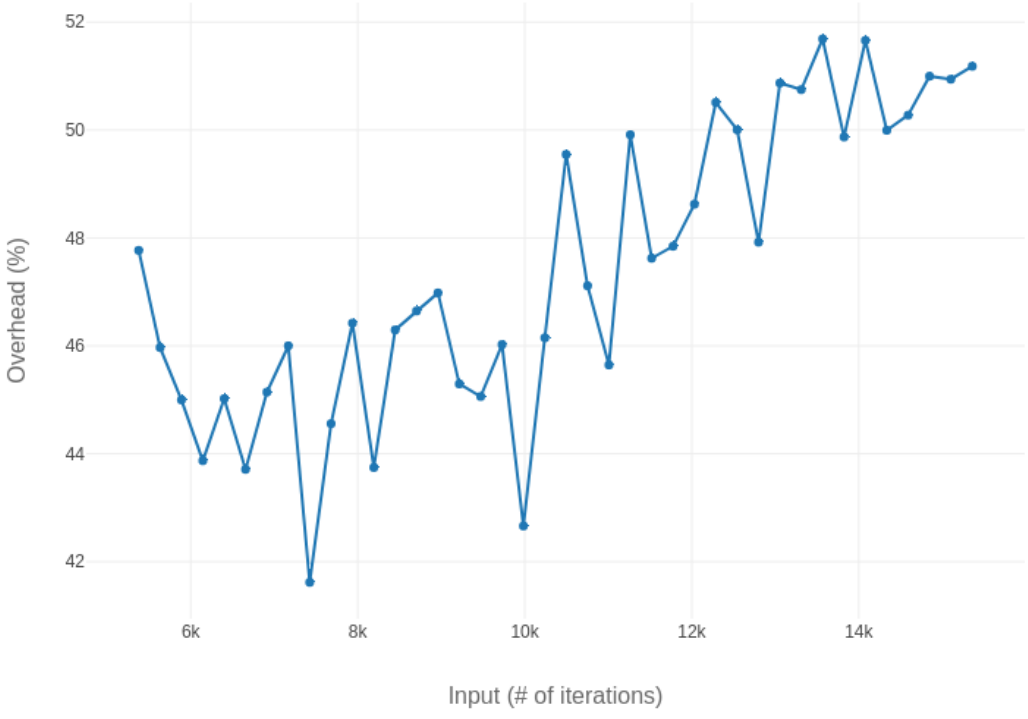


(b) Επιβράδυνση

Σχήμα 5.1: Κόσκινο του Ερατοσθένη

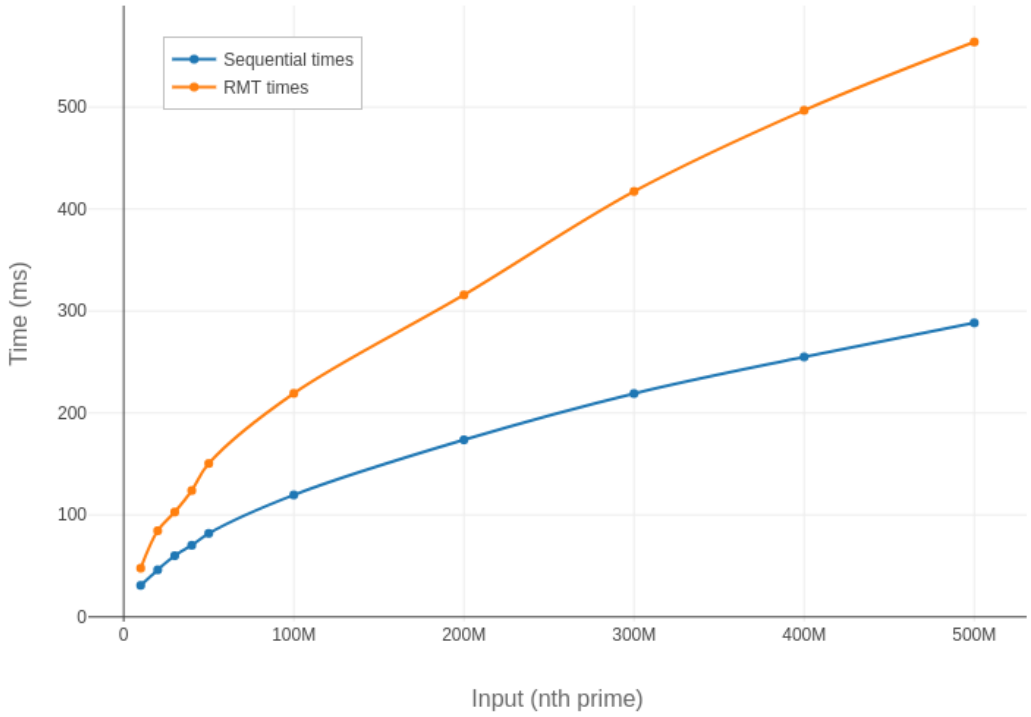


(a) Χρόνοι εκτέλεσης

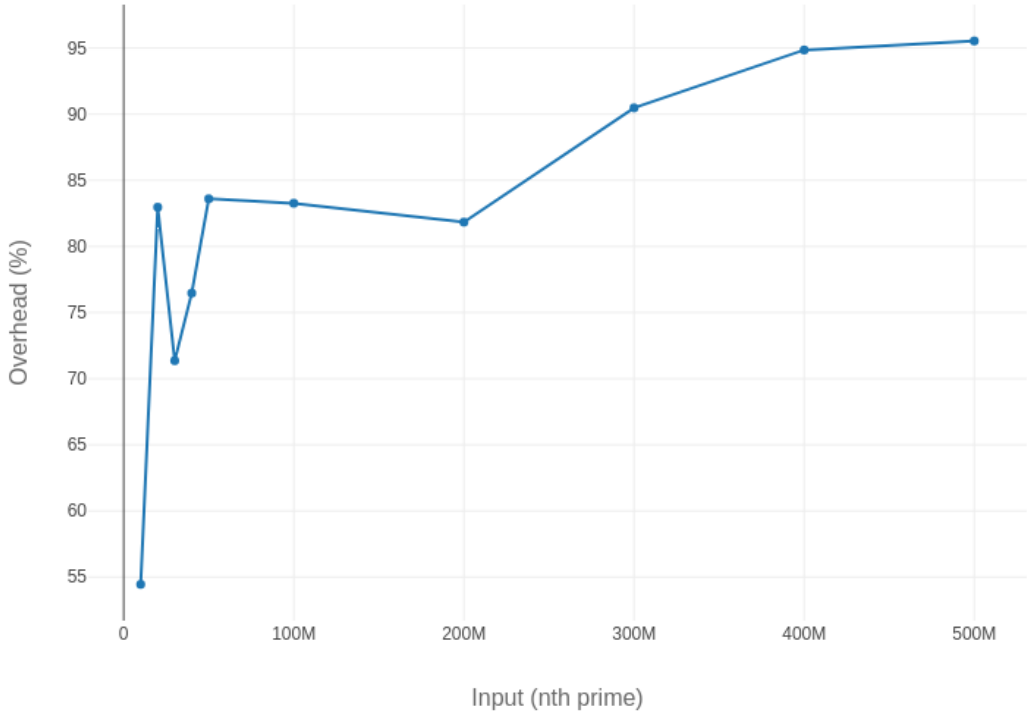


(b) Επιβράδυνση

Σχήμα 5.2: Προσέγγιση του Pi

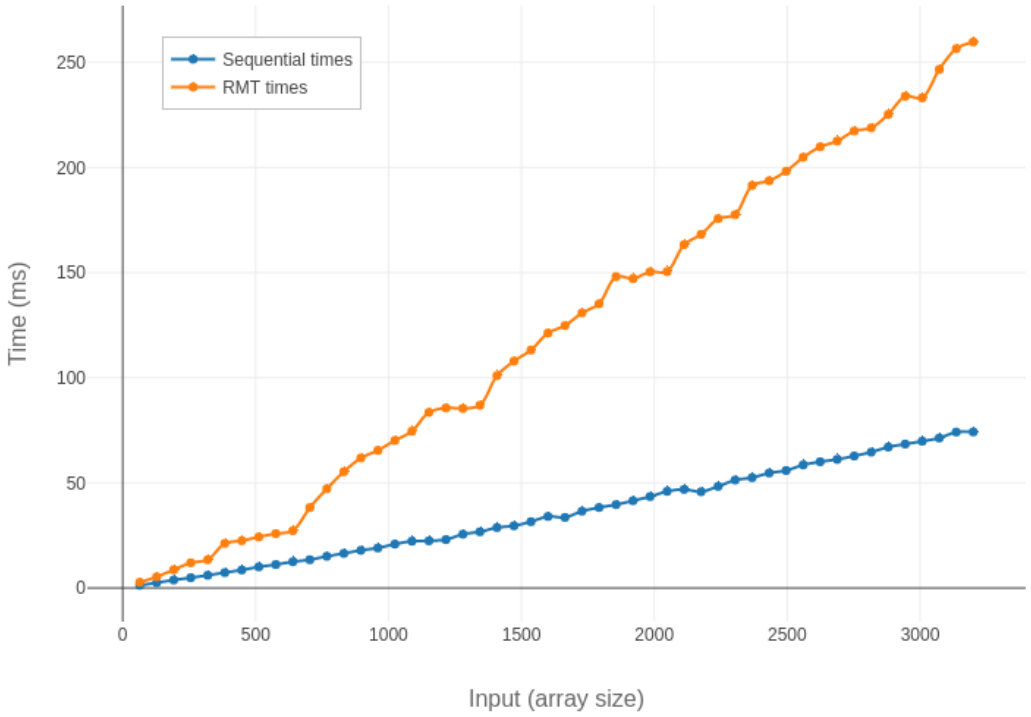


(a) Χρόνοι εκτέλεσης



(b) Πάνω από το κεφάλι

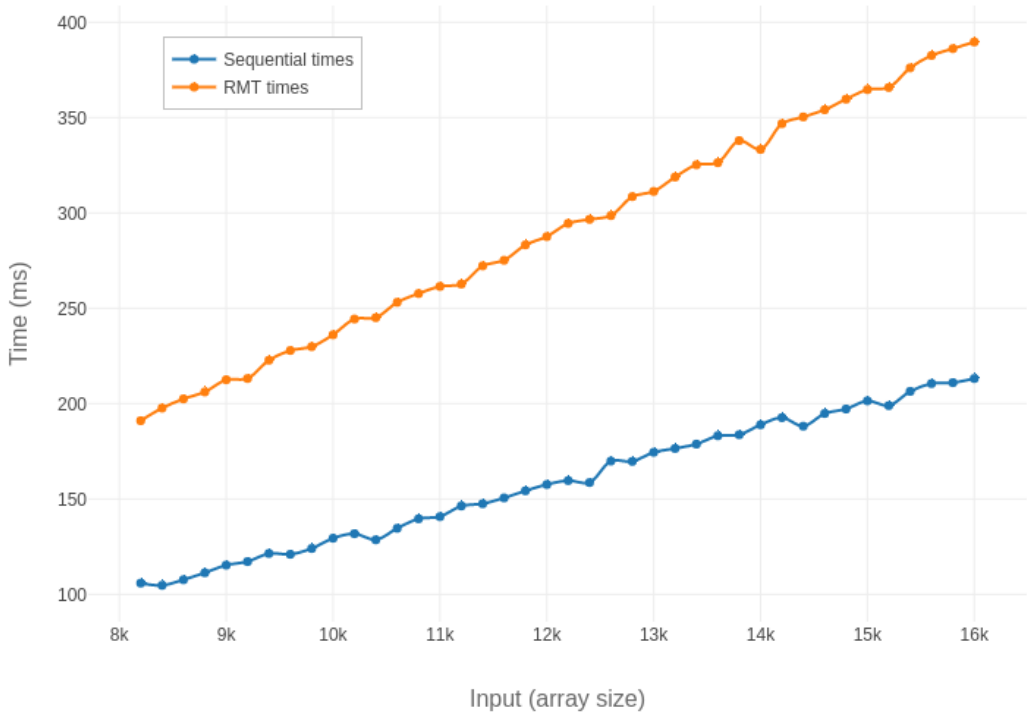
Σχήμα 5.3: Τεστ επαλήθευσης πρώτου αριθμού



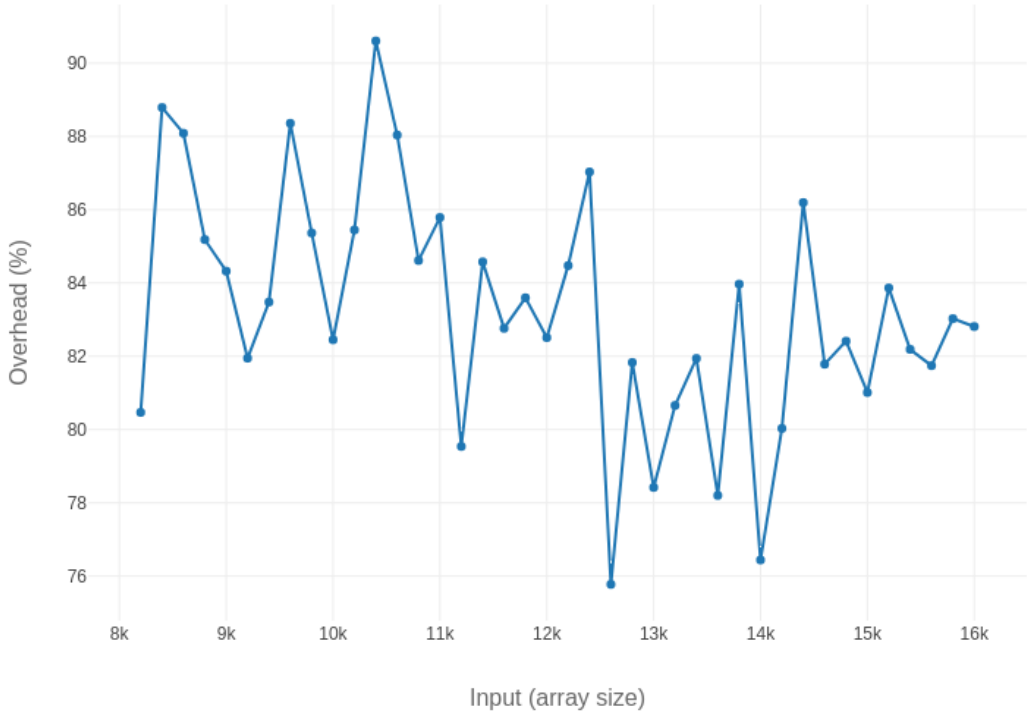
(a) Χρόνοι εκτέλεσης

(b) Επιβράδυνση

Σχήμα 5.4: Ταξινόμηση εισαγωγής

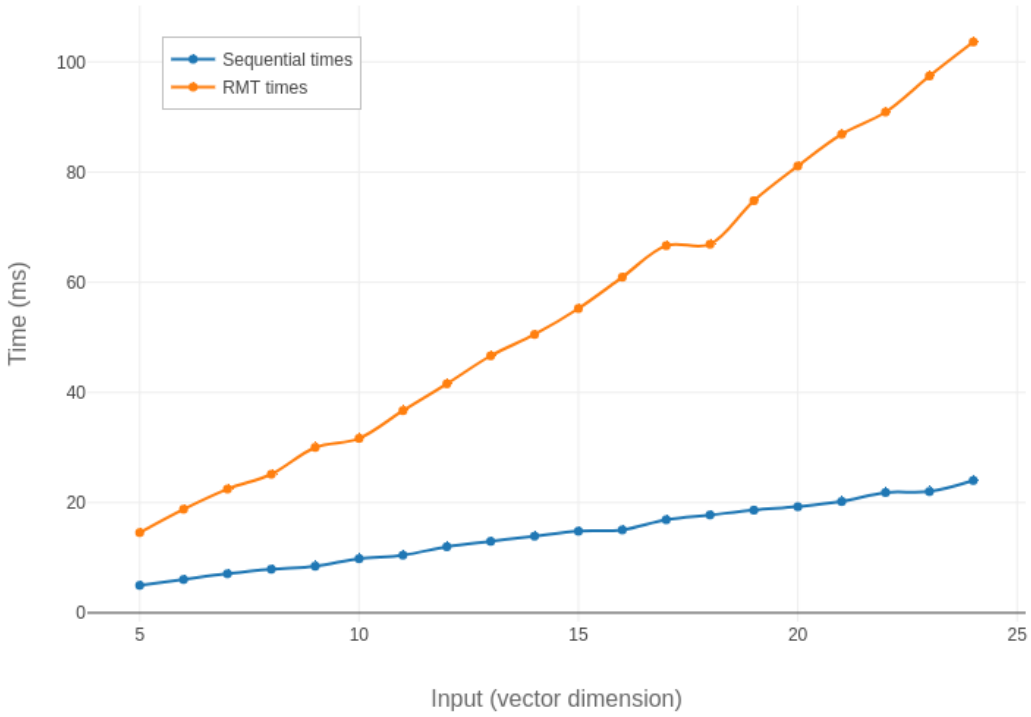


(a) Χρόνοι εκτέλεσης

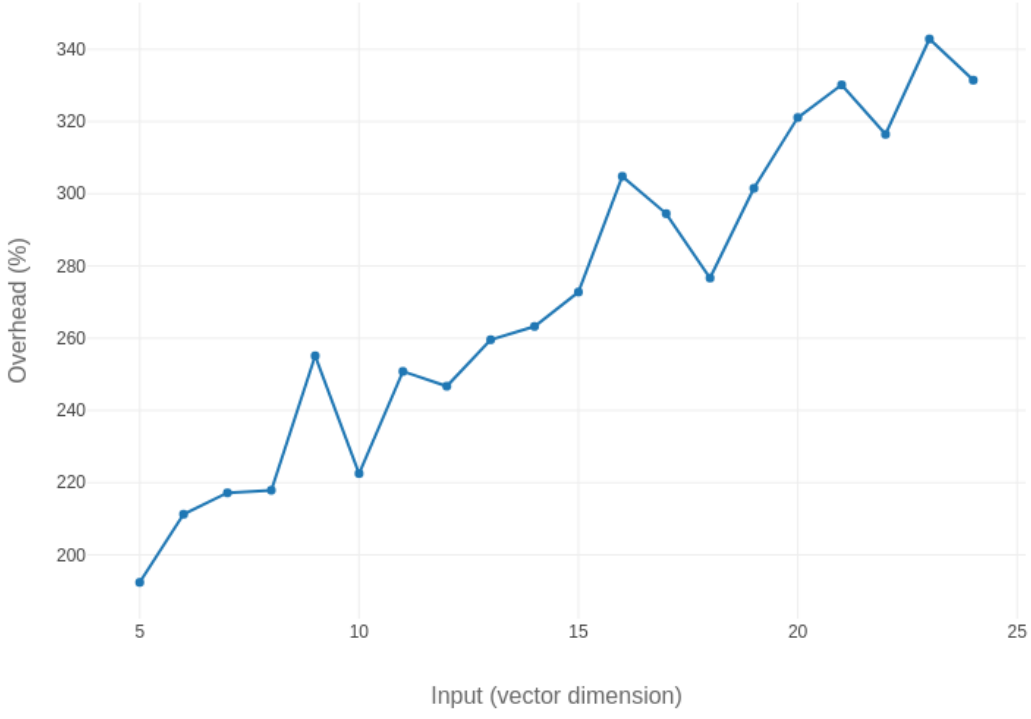


(b) Επιβράδυνση

Σχήμα 5.5: Μείωση πίνακα

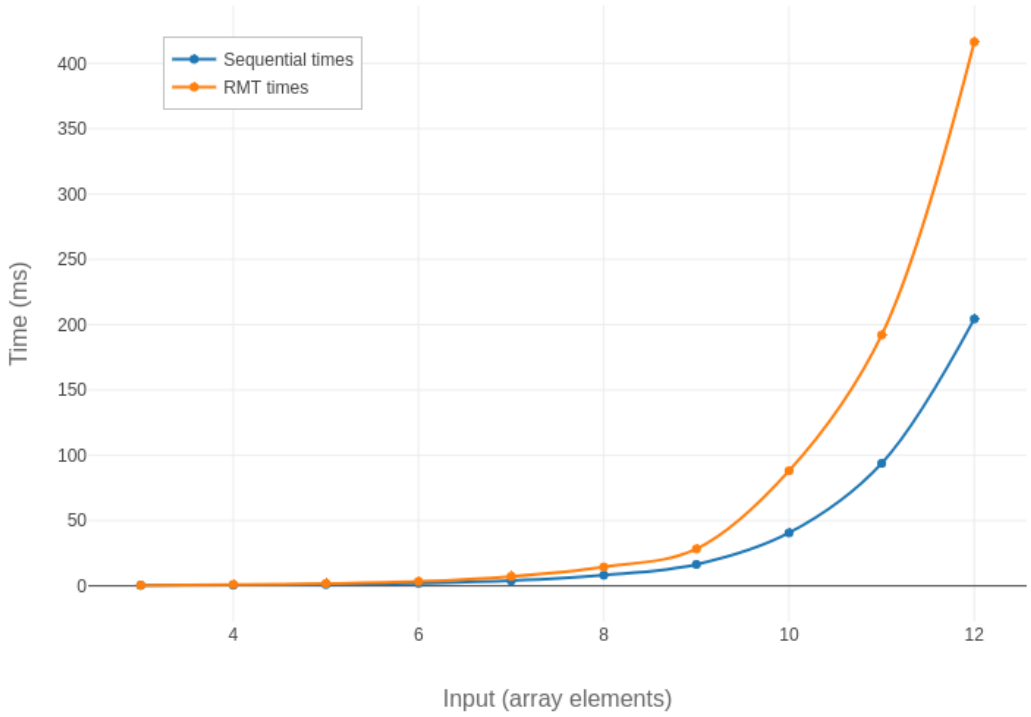


(a) Χρόνοι εκτέλεσης

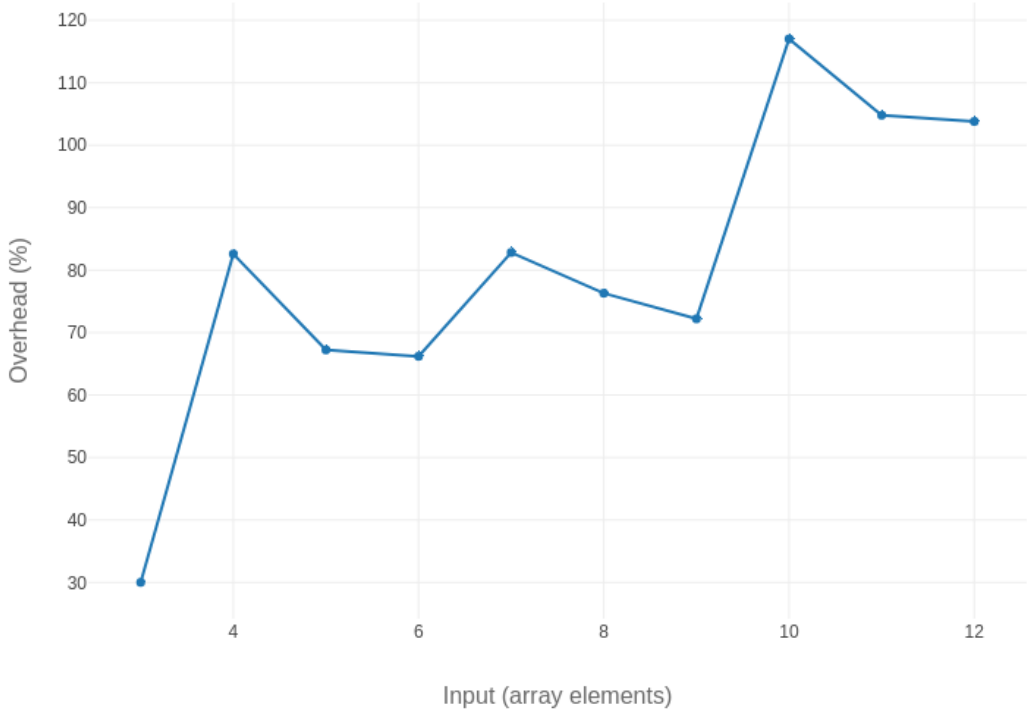


(b) Επιβράδυνση

Σχήμα 5.6: Πολλαπλασιασμός πίνακα-διανύσματος



(a) Χρόνοι εκτέλεσης

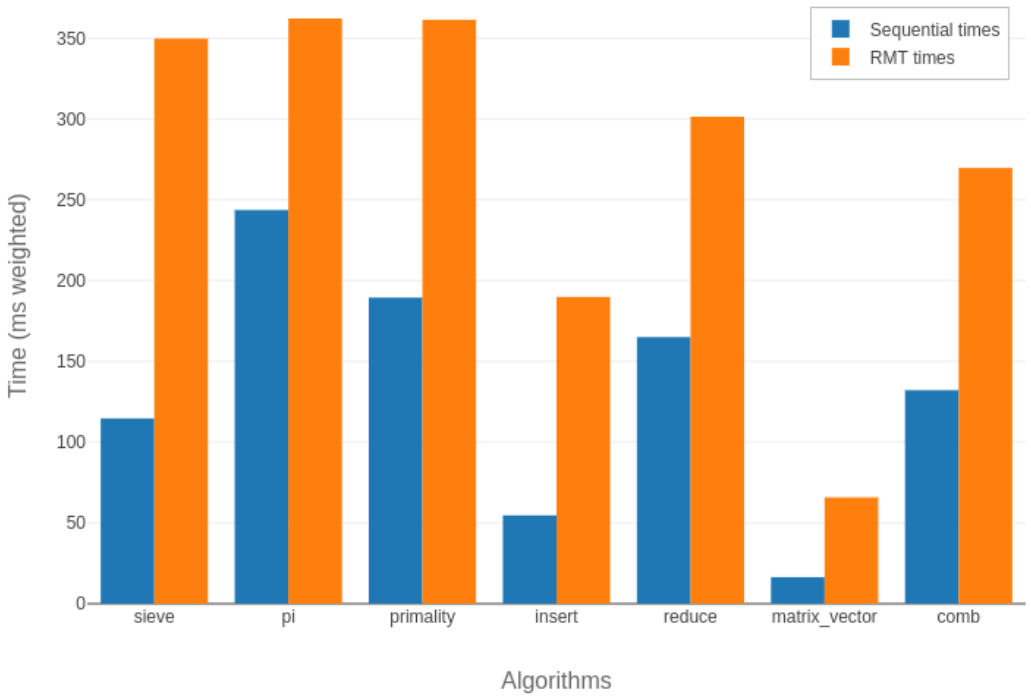


(b) Επιβράδυνση

Σχήμα 5.7: Συνδυασμοί (m ανά n)

5.2.2 Αθροιστικά αποτελέσματα

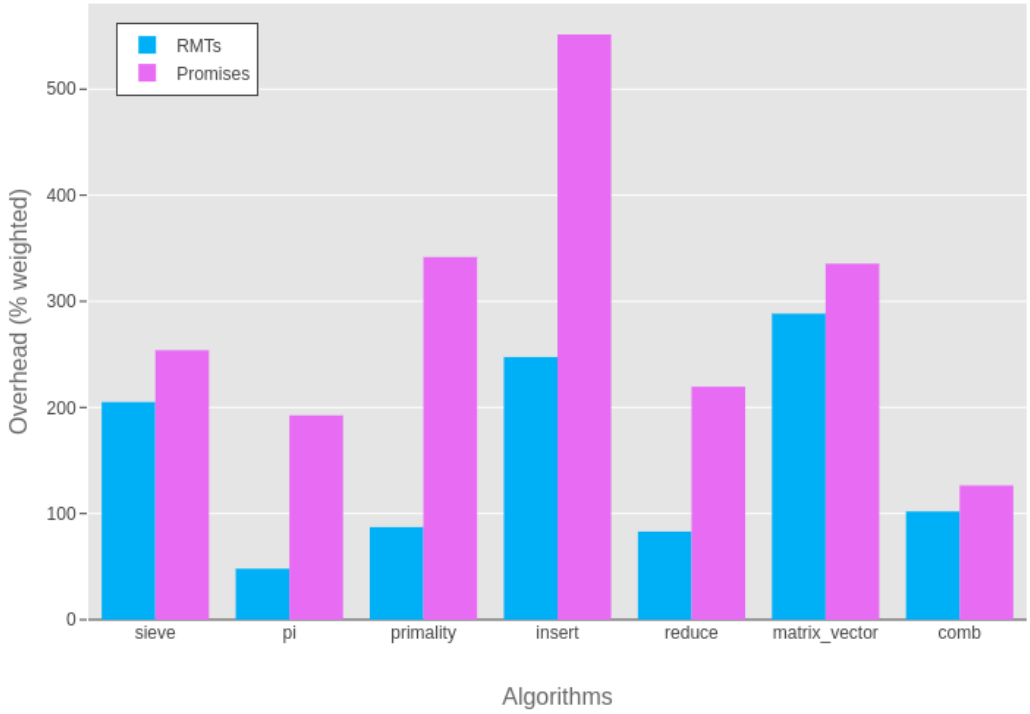
Σε αυτήν την ενότητα παρουσιάζουμε τα συσωρευμένα αποτελέσματα των υλοποιήσεών μας και τα συγκρίνουμε με τις σχετικές υλοποιήσεις σε Promises.



Σχήμα 5.8: Σταθμισμένοι μέσοι όροι χρόνου εκτέλεσης με βάση την χρονική πολυπλοκότητα του αλγόριθμου

Στο σχήμα 5.8 παρουσιάζουμε έναν σταθμισμένο μέσο χρόνο εκτέλεσης για κάθε τεστ αξιολόγησης. Τα βάρη που χρησιμοποιήσαμε για κάθε τεστ αξιολόγησης εξαρτώνται από την πολυπλοκότητα του χρόνου του αλγορίθμου. Στην Ενότητα 5.1, αναφέρθηκε η πολυπλοκότητα του χρόνου σε big-O ορολογία για κάθε αλγόριθμο. Για το βάρος κάθε εισόδου αντικαθιστούμε αυτή τη χρονική πολυπλοκότητα με τη σχετική είσοδο κάθε φορά και στη συνέχεια χρησιμοποιούμε το αποτέλεσμα ως βάρος. Τα ίδια βάρη χρησιμοποιήθηκαν επίσης στο Σχήμα 5.9, αλλά εκεί, παρουσιάζουμε επίσης τις επιβράδυνσεις που είχε η υλοποίηση των Promises.

Στο σχήμα 5.9, μπορούμε να δούμε ότι το RMT μας γενικά πέτυχε μικρότερη επιβράδυνση από τις σχετικές υλοποιήσεις με Promises. Σε ορισμένες περιπτώσεις (pi, primality, insert), όπου πολλοί υπολογισμοί είναι ενσωματωμένοι σε Promises ή RMT αντίστοιχα, παρατηρούμε ότι η υλοποίηση των promises παράγει πολύ περισσότερη επιβράδυνση, καταλήγοντας στο συμπέρασμα ότι η δημιουργία ενός μεγάλου αριθμού αντικειμένων promises, δεν κλιμακώνει εξίσου καλά όπως στην περίπτωση του RMT μας. Στην περίπτωση του πολλαπλασιασμού πίνακα-διάνυσμα, έχουμε πολύ παρόμοια αποτελέσματα λόγω του αριθμού των νημάτων που δημιουργήθηκαν σε αυτό το τεστ. Σε αυτό το τεστ χρησιμοποιούμε το $Promise.all()$ για να τρέξουμε όλα τα νήματα στην υλοποίηση των promises και το **runMany** στο RMT. Αν και εδώ, έχουμε επίσης μεγάλο αριθμό promises, το $Promise.all()$ επιτυγχάνει έναν ελαφρώς αποδοτικότερο προγραμματισμό από το **runMany**, με αποτέλεσμα να έχει επιβράδυνση πιο κοντά στα RMT από ότι αναμενόταν.



Σχήμα 5.9: Σταθμισμένοι μέσοι όροι επιβράδυνσης με βάση την χρονική πολυπλοκότητα του αλγόριθμου

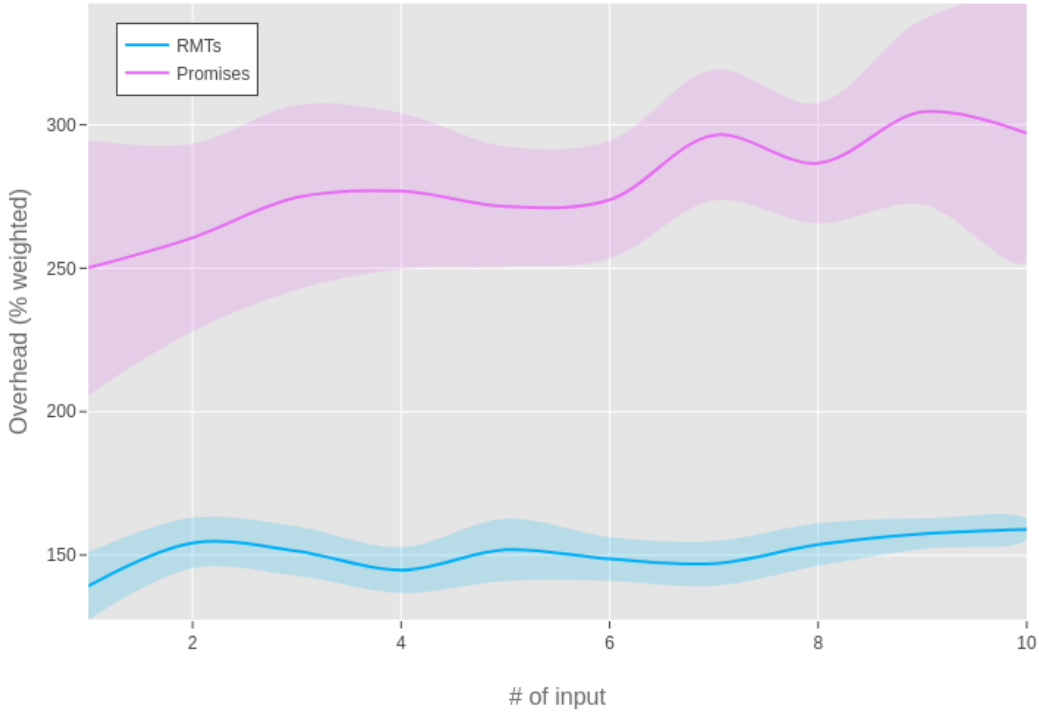
Ο πίνακας 5.1 δείχνει τα ακριβή αποτελέσματα για τα δέκα μεγαλύτερα δεδομένα εισόδου που θα μπορούσαμε να εκτελέσουμε κατά τη διάρκεια των δοκιμών μας. Μπορούμε να δούμε ότι τα RMT αποδίδουν καλύτερα από τα promises σε κάθε τεστ αξιολόγησης, εκτός ίσως από τον πολλαπλασιασμό των πινάκων με διανλυσματα. Ωστόσο, αυτές οι επιβραδύνσεις είναι κοντά και ο λόγος που τα promises είναι τόσο κοντά μπορεί να είναι ότι το **runMany** δεν είναι τόσο βελτιστοποιημένο όσο το $Promise.all()$.

Benchmark	Method	Rank of input									
		1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
sieve	RMT	**198.78**	**204.33**	**212.94**	**212.19**	**217.04**	**201.24**	**205.64**	**202.45**	**207.22**	**210.19**
	Prom.	203.82	241.00	243.75	268.17	255.58	292.53	294.35	301.67	327.87	336.59
pi	RMT	**50.87**	**50.75**	**51.69**	**49.87**	**51.66**	**50.00**	**50.28**	**51.00**	**50.94**	**51.18**
	Prom.	197.89	156.15	136.10	104.25	106.44	119.67	124.86	155.10	153.43	191.92
primality	RMT	**54.45**	**82.96**	**71.36**	**76.47**	**83.60**	**83.26**	**81.84**	**90.48**	**94.85**	**95.54**
	Prom.	313.10	327.19	322.14	350.54	341.14	345.28	351.66	332.46	351.79	341.01
insert	RMT	**249.43**	**247.53**	**246.27**	**238.66**	**235.99**	**241.69**	**233.94**	**245.79**	**245.84**	**249.87**
	Prom.	560.55	573.52	579.80	590.20	576.71	575.68	593.31	587.44	476.16	411.09
reduce	RMT	**80.03**	**86.20**	**81.78**	**82.41**	**81.01**	**83.87**	**82.19**	**81.74**	**83.03**	**82.81**
	Prom.	140.00	132.81	182.28	187.10	189.63	188.00	216.62	225.71	240.63	228.59
mat-vec	RMT	**272.79**	304.81	**294.50**	**276.67**	**301.52**	321.10	**330.15**	316.46	**342.87**	**331.45**
	Prom.	313.98	**265.17**	315.90	358.57	357.06	**298.53**	398.56	**299.30**	402.65	381.15
comb	RMT	**30.01**	**82.59**	**67.23**	**66.20**	**82.82**	**76.30**	**72.21**	**117.01**	**104.80**	**103.80**
	Prom.	63.65	129.30	158.25	87.44	90.58	94.78	105.20	117.22	134.19	131.13

Πίνακας 5.1: Η επιβράδυνση για τα δέκα μεγαλύτερα δεδομένα εισόδου για κάθε τεστ αξιολόγησης

Στο Σχήμα 5.10, παρουσιάζουμε το μέσο όρο των επιβραδύνσεων όλων των τεστ αξιολόγησης για τα δέκα μεγαλύτερα δεδομένα εισόδου που χρησιμοποιήσαμε για το καθέ τεστ, όταν τα εκτελέσαμε. Για το σκοπό αυτό, κανονικοποιούμε ξεχωριστά τις επιβραδύνσεις κάθε τεστ, πράγμα που σημαίνει ότι μετασχηματίζουμε τα δεδομένα ώστε να έχουμε μέση τιμή 0 και τυπική απόκλιση 1. Με αυτό τον τρόπο αγνοούμε προσωρινά το μέγεθος των επιβραδύνσεων και έτσι μπορούμε να βρούμε τον μέσο όρο κάθε είσοδου (π.χ. το μέσο όρο όλων των τεστ αξιολόγησης της 1ης εισόδου) λαμβάνοντας υπόψη μόνο τις πραγματικές αποστάσεις μεταξύ των δεδομένων. Στη συνέχεια, κάνουμε την αντίστροφη διαδικασία της κανονικοποίησης και μετατρέπουμε τον μέσο όρο του αποτελέσματος στο μέσο όρο όλων των τεστ και η τυπική απόκλιση είναι ίση με τον μέσο όρο όλων των τυπικών αποκλίσεων όλων των τεστ (που είναι επίσης το σφάλμα που παρουσιάζεται στο Εικόνα 5.10).

Χρησιμοποιώντας την προαναφερθείσα τεχνική, παράγουμε τις παραπάνω γραμμές οι οποίες είναι βασικά ανεξάρτητες από τα τεστ αξιολόγησης και τις τιμές εισόσου τους. Έτσι, συγκρίνουμε τις πραγματικές αποστάσεις και τις κλίσεις τους για να δούμε πώς συμπεριφέρονται γενικά οι υλοποιήσεις των RMT και των Promises. Παρατηρούμε ότι τα promises δίνουν μεγαλύτερες επιβραδύνσεις σε ολόκληρη τη γραμμή και δεν διασταυρώνονται ούτε τα σφάλματα των δύο γραμμών. Παρατηρούμε επίσης ότι η γραμμή RMT έχει μικρότερη κλίση από τη γραμμή Promises, πράγμα που σημαίνει ότι τα RMTs πραγματικά κλιμακώνουν καλύτερα από την υλοποίηση των promises, καθώς αυξάνονται τα μεγέθη των εισόδων.



Σχήμα 5.10: Μέση γενική επιβράδυνση για τα δέκα μεγαλύτερα δεδομένα εισόδου για κάθε τεστ αξιολόγησης

Κεφάλαιο 6

Επίλογος

6.1 Συμπεράσματα

Σε αυτή τη διπλωματική, ορίσαμε ένα γενικό θεωρητικό πλαίσιο για την τυποποίηση της δηλωτικής σημασιολογίας των διαφυλλώμενων υπολογισμών στον ταυτόχρονο προγραμματισμό. Παρουσιάζουμε εδώ τα συμπεράσματά μας για το *Μετασχηματιστή Μονάδων Επανόδου*. Δείξαμε ότι οι επάνοδοι μπορούν να είναι μια αρθρωτή έκφραση της σημασιολογίας του ταυτοχρονισμού. Αυτή η ευκινησία των επανόδων μας επέτρεψε να ορίσουμε εύκολα τη σημασιολογία της γλώσσας που είχαμε ως παράδειγμα και να την επεκτείνουμε με νέες λειτουργίες που εισάγουν ταυτοχρονισμό σε μια τέτοια γλώσσα. Αλλά εκτός από την εφαρμογή της στη σημασιολογία του ταυτοχρονισμού, ο μετασχηματιστής μονάδων επανόδου που προτείνεται εδώ μπορεί να χρησιμοποιηθεί στη σημασιολογία των ντετερμινιστικών γλωσσών προγραμματισμού με απροσδιόριστη σειρά αποτίμησης των εκφράσεων τους, όπως η C.

Η κύρια συμβολή όμως αυτής της εργασίας είναι τα αποτελέσματα απόδοσης των RMTs. Παρουσιάσαμε με την υλοποίηση μας των RMTs ότι οι επάνοδοι είναι κατασκευές χαμηλού κόστους και επιβράδυνσης που μπορούν να χρησιμοποιηθούν για τον προσδιορισμό της σημασιολογίας του ταυτοχρονισμού. Η σύγκρισή μας με τα Promises της JavaScript έδειξε ότι τα ξεπέρασαμε κατά 100%, κάνοντας τα RMT μια λογική επιλογή για τον ορισμό της δηλωτικής σημασιολογίας των ταυτόχρονων γλωσσών προγραμματισμού και για χρήση σε πραγματικές εφαρμογές.

6.2 Μελλοντική δουλέια

Η μελλοντική έρευνα θα πρέπει να διερευνά λεπτομερώς τις ανυψωτικές ιδιότητες του μετασχηματιστή μονάδων επανόδου, δηλαδή τον ακριβή τρόπο με τον οποίο οι πράξεις που υποστηρίζονται από τη μονάδα των ατομικών υπολογισμών M μπορούν να μεταφερθούν στη μονάδα των επανόδων $\mathsf{R}(M)$.

Επιπλέον, μια περαιτέρω επέκταση αυτής της εργασίας θα πρέπει να περιλαμβάνει ακόμα καλύτερη υλοποίηση των μετασχηματιστών μονάδων επανόδου. Στη JavaScript έχουν πλέον υλοποιηθεί στις βασικές τις λειτουργίες τα *generator*. Τα *Generator* επιτρέπουν να παράγονται αποτελέσματα με το *yield* παρά με *επιστρέφοντας αμέσως* μέσα σε μια συνάρτηση. Αυτό θα μπορούσε να είναι πολύ χρήσιμο για την εφαρμογή του στη μέθοδο *bind* στους Μετασχηματιστές Μονάδων Επανόδου.

Μια άλλη κατεύθυνση που θα ήταν ενδιαφέρουσα είναι η υλοποίηση των μετασχηματιστών μονάδων επανόδου σε μία άλλη γλώσσα. Ίσως θα μπορούσαμε να χρησιμοποιήσουμε καλύτερα εργαλεία για την υλοποίησή τους σε μια γλώσσα που μεταγλωττίζεται, όπως η C++. Η JavaScript επιτρέπει επίσης τη χρήση βιβλιοθηκών που έχουν μεταγλωττιστεί από τη C++ ως ”add-ons”, γεγονός που θα μπορούσε να δώσει ακόμα καλύτερα αποτελέσματα απόδοσης.

Βιβλιογραφία

[Arms93] J. L. Armstrong, Mike Williams, Robert Virding and Claes Wilkström, *ERLANG for Concurrent Programming*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Aspe91] A. Asperti and G. Longo, *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1991.

[Bake77] Henry Baker and Carl Hewitt, "Laws for communicating parallel processes", in *1977 IFIP Congress Proceedings*, pp. 987–992, IFIP, 1977.

[Barr96] M. Barr and C. Wells, *Category Theory for Computing Science*, Prentice-Hall International Series in Computer Science, Prentice Hall, New York, NY, 2nd edition, 1996.

[Blum96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall and Yuli Zhou, "Cilk: An Efficient Multithreaded Runtime System", *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55 – 69, 1996.

[Bust90] David W. Bustard, "Concepts of Concurrent Programming", Technical report, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, PA, 1990.

[dBak96] J.W. de Bakker and Erik P. de Vink, *Control Flow Semantics*, MIT Press, Cambridge, MA, 1996.

[Gogu91] J. A. Goguen, "A Categorical Manifesto", *Mathematical Structures in Computer Science*, vol. 1, pp. 49–68, 1991.

[Gunt90] C. A. Gunter and D. S. Scott, "Semantic Domains", in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, chapter 12, pp. 633–674, Elsevier Science Publishers B.V., 1990.

[Gunt92] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1992.

[Harr04] William L. Harrison, "Cheap (But Functional) Threads". Under consideration for publication in J. Functional Programming, 2004.

[Labs16] Intel Labs, "River Tail", `https://github.com/IntelLabs/RiverTrail`, October 2016. Accessed: 2018-06-01.

[Lian95] Sheng Liang, Paul Hudak and Mark Jones, "Monad Transformers and Modular Interpreters", in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pp. 333–343, New York, NY, USA, 1995, ACM.

[Lian98] S. Liang, *Modular Monadic Semantics and Compilation*, Ph.D. thesis, Yale University, Department of Computer Science, May 1998.

[Lori17] Matthew C. Loring, Mark Marron and Daan Leijen, "Semantics of Asynchronous JavaScript", in *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, pp. 51–62, New York, NY, USA, 2017, ACM.

[Mogg89] E. Moggi, "Computational Lambda Calculus and Monads", in *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pp. 14–23, 1989.

[Mogg90] E. Moggi, "An Abstract View of Programming Languages", Technical Report ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.

[Nami15] Dmitry Namiot and Vladimir Sukhomlin, "JavaScript Concurrency Models", *International Journal of Open Information Technologies*, vol. 3, no. 6, pp. 21–24, 2015.

[Node18] "Node.js", `https://github.com/nodejs/node`, June 2018. Accessed: 2018-06-04.

[Papa00] Nikolaos S. Papaspyrou and Dragan Maćoš, "A Study of Evaluation Order Semantics in Expressions with Side Effects", *Journal of Functional Programming*, vol. 10, no. 3, pp. 227–244, 2000.

[Papa01] Nikolaos S. Papaspyrou, "A Resumption Monad Transformer and its Applications in the Semantics of Concurrency", in *Proceedings of the 3rd Panhellenic Logic Symposium*, pp. 17–22, 2001.

[Pier90] B. C. Pierce, "A Taste of Category Theory for Computer Scientists", Technical Report CMU-CS-90-113R, Carnegie Mellon University, School of Computer Science, September 1990.

[Pier91] B. Pierce, *Basic Category Theory for Computer Scientists*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1991.

[Prom15] "ECMAScript 2015", `https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects`, June 2015. Accessed: 2018-06-01.

[Rosc97] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[Schm86] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon Newton, MA, 1986.

[Scot71] D. Scott and C. Strachey, "Towards a Mathematical Semantics for Computer Languages", in *Proceedings of the Symposium on Computers and Automata*, pp. 19–46, Brooklyn, NY, 1971, Polytechnic Press.

[Scot82] D. S. Scott, "Domains for Denotational Semantics", in *International Colloquium on Automata, Languages and Programs*, vol. 140 of *Lecture Notes in Computer Science*, pp. 577–613, Berlin, Germany, 1982, Springer Verlag.

[Wadl92] P. Wadler, "The Essence of Functional Programming", in *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL'92)*, pp. 1–14, January 1992.