National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

The Resumption Monad Transformer and its Implementation in JavaScript

DIPLOMA THESIS

GEORGIOS SAKKAS

Supervisor : Nikolaos S. Papaspyrou
Associate Professor NTUA

Athens, July 2018

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

The Resumption Monad Transformer and its Implementation in JavaScript

DIPLOMA THESIS

GEORGIOS SAKKAS

Supervisor : Nikolaos S. Papaspyrou
Associate Professor NTUA

Approved by the examining committee on the July 6, 2018.

.....	.....	.....
Nikolaos S. Papaspyrou Associate Professor NTUA	Georgios I. Goumas Assistant Professor NTUA	Aris T. Pagourtzis Associate Professor NTUA

Athens, July 2018

.....

Georgios Sakkas

Electrical and Computer Engineer

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.

Acknowledgements

First of all, I would like to thank my supervisor Prof. Nikolaos Papaspyrou for his guidance during the conduction of my thesis. Prof. Papaspyrou was a true inspiration for me in order to further pursue my dreams in research.

Additionally, I would like to thank Prof. Georgios Goumas and Prof. Aris Pagourtzis for being part of my thesis committee as well as their inspirational courses at National Technical University of Athens (NTUA).

I would also like to express my appreciation and gratitude for my parents who supported me both mentally and financially through my academic journey. Without their help and unlimited support, this journey would not be possible.

Last but not least, I would like to thank all my friends for the great time we spent together and the encouragement they provided me all these years to keep going and achieve my goals.

Georgios Sakkas,

Athens, July 6, 2018

This thesis is also available as Technical Report CSD-SW-TR-2-18, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, July 2018.

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

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Concurrency and concurrent programming languages

1.1.1 Concurrency and concurrent systems

Parallel computers are the predominant systems nowadays. Computer systems that only provide a single stream of computations, like single-core CPUs, have now become scarce. Parallel systems are the most obvious example of concurrent systems. Concurrent systems are the ones that allow many different applications or activities, that are called *threads* in the field of concurrency, to be executed simultaneously. Those threads usually need to communicate with each other in order to finish the overall task at hand correctly. The theory of *concurrency* involves the study of such communicating systems and the models used for that reason.

Concurrent systems are therefore more complicated than their sequential counterparts to examine and reason about. In a sequential program execution, each computation - i.e. a basic operation - will be done "one at a time", while in a concurrent execution many computations might happen "together". What that means is that each program execution has its individual *state* - i.e the variables and their values, whereas in a concurrent execution each thread may have its (more or less) independent state. But those thread states must necessarily communicate with each other as mentioned before. Consequently different combinations of them may arise. The number of the state combinations grows exponentially with the number of threads compared to the sequential ones' linear growth. Such problems are described in [Bust90].

Concurrent systems may also exhibit *nondeterminism*, which is the case of a concurrent execution of the same code with the same input resulting in different outputs. Nondeterministic systems are therefore in principle untestable and only with formal understanding and reasoning we can predict deterministic properties of such systems.

A concurrent system may also be *deadlocked*, which is the case of a concurrent execution of a program leading in no thread making any progress due to waiting to communicate with each other, but nothing towards that is happening. Deadlock is one of the primary concerns when dealing with concurrent systems.

Another misbehavior a concurrent system may exhibit is *livelock*. Externally it may seem like deadlock since no actual progress is made, but livelock is basically an infinite unbroken sequence of internal computations.

Understanding such misbehaviors of concurrent systems is essential for analyzing concurrency and finding formal methods for it. Since mathematical models and software engineering techniques designed for sequential systems are usually inadequate for modeling the subtleties of concurrency, different ones must be developed for the purpose of concurrency.

1.1.2 Concurrent programming languages and their semantics

Programming languages allowing the user to utilize systems such as the aforementioned concurrent ones, can be defined as *concurrent programming languages*. Those languages usually have built-in facilities that allow the users to write effortlessly concurrent programs, like Cilk (where parallel programming can be considered as subset of concurrent programming) [Blum96] or Erlang [Arms93]. These concurrent facilities may involve multi-threading, message passing, shared resources (including shared memory) or futures and promises. Many techniques have been exploited in order to mathematically formalize these facilities and define the semantics of concurrent programming languages [dBak96], [Rosc97]. One of these formal semantic models that have been long studied [Mogg90], [dBak96], [Papa00], [Papa01], [Harr04] as a model of interleaved computations for the semantics of concurrent programming languages are *resumptions*.

In this work we defined a *Resumption Monad Transformer* (RMT) and implemented it in *JavaScript* alongside with a simple imperative language to determine its performance. RMTs were fist introduced by [Mogg90] as a part of a unified approach to the denotational semantics of programming languages based on monads.

1.1.3 Concurrency in JavaScript

JavaScript in almost each of its implementations is single-threaded and does not support concurrency natively. Some frameworks though can be used to exploit concurrency in JavaScript as mentioned also here [Nami15].

One of the most common case nowadays in asynchronous computation for JavaScript are *Web Workers*. A Web Worker is basically a thread running JavaScript in the background, bringing bring actor style [Bake77] threads to the web. The worker thread can perform tasks without interfering with the user interface. Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa). This message exchange is borrowed from actors.

While Web workers achieve their design goal of offloading long-running computations to background threads, they are not suitable for the development of scalable compute-intense workloads due to high cost of communication and low level of abstraction.

Another framework introducing concurrency in JavaScript is Intel Labs' River Tail [Labs16], which enables data parallelism in web applications. River Trail gently extends JavaScript with simple deterministic data-parallel constructs that are translated at runtime into a low-level hardware abstraction layer. The central component of River Trail is the *ParallelArray* type. ParallelArray objects are essentially ordered collections of scalar values that can represent multi-dimensional collections of scalars. All ParallelArray objects have a shape that succinctly describes the dimensionality and size of the object. ParallelArrays are immutable once they are created and are manipulated by invoking methods on them, which produce and return new ParallelArray objects. Finally, ParallelArrays provide an API for using functions like *map, reduce, filter* etc on them.

River Tail basically introduces the well-known *map-reduce* paradigm into JavaScript with ParallelArrays. While this paradigm is well-established, its semantics remain somewhat obscure and difficult for the average developer to comprehend and integrate into their applications, in order to exploit its true potential.

While the above frameworks provide some concurrency in JavaScript applications, the built-in JavaScript *Promises* provide a nicer and more accessible framework. Promises are objects

that are native in most JavaScript implementations nowadays [Prom15]. A Promise is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation [Lori17]. Any Promise object is in one of three mutually exclusive states: fulfilled, rejected, and pending. A Promise is said to be settled if it is not pending, i.e. if it is either fulfilled or rejected. A Promise is resolved if it is settled or if it has been "locked in" to match the state of another Promise. Attempting to resolve or reject a resolved Promise has no effect. An unresolved Promise is always in the pending state. A resolved Promise may be pending, fulfilled or rejected.

If e is a Promise object then the following operations for working with Promises are defined:

- $Promise()$ creates a new Promise object e.
- $Promise.resolve(e_2)$ resolves a Promise e_1 to the value of e_2.
- $e_1.then(e_2)$ schedules Promise e_2 to be executed after the Promise e_1 is resolved.
- $Promise.all([e_i])$, where $[e_i]$ is an iterable of Promises e_i, creates a new Promise object e which is resolved when all of the iterable's Promises are resolved.

Utilizing the above operations Promises can be created and be chained one after another in order to be used for asynchronous applications. If they are used properly, Promises can model concurrent programming and provide a way of interleaved computations. Their semantics and abstraction are closely related to this work (RMTs). We make a comparison of both them later in this thesis.

1.2 Goals and contributions

The goals of this thesis are driven by the implementation of a semantics model for concurrency in real-life programming languages, like *JavaScript*. New tools are needed to define the semantics of concurrent programming languages, which allow the parts of a program that execute simultaneously to interact with one another, typically using the same memory variables.

Resumptions have long been suggested as a model of interleaved computation in the semantics of concurrent programming languages. In brief, a resumption is either a computed value of some domain D or an atomic computation that produces a new resumption.

In this work, we extend a structured generalization of this technique presented in [Papa01]. We allow the atomic steps to perform any type of computation, represented by an arbitrary monad M over an appropriate category of semantic domains. Thus, we define the *Resumption Monad Transformer* R, which transforms monad M to a new monad $R(M)$ representing interleaved computations. The resumption monad transformer is utilized in this work in order to express the semantics of a simple deterministic language.

The contributions of this thesis are threefold:

1. We present **the theory and the necessary technical background** for the reader to understand the denotational semantics of concurrent programming languages. Furthermore, a theoretical framework for the *Resumption Monad Transformers* (RMTs) is presented here, based on the aforementioned background.

2. We investigate **the performance of RMTs** on single-threaded programming languages that do not support concurrency natively. For this purpose we implemented RMTs in *JavaScript* and tested them in seven different benchmarks. Those benchmarks consist of algorithms of different time complexities (i.e $O(n)$, $O(n\ logn)$, $O(n^2)$, $O(2^n)$). For a better understanding of our results, we compare them to **JavaScript's built-in *Promises***, which can model a concurrent computing model if handled in a proper fashion.

3. We finally present **a simple imperative and deterministic language** (a small but effective subset of JavaScript) whose denotational semantics are implemented using our RMT. This programming language is also **extended with some operators** that provide a low-overhead and easy-to-use framework to run computations in an interleaved manner.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 contains the basic mathematical background for the reader to understand the semantics of concurrency and the rest of this thesis. It also includes a mathematical definition of Monads, Monad Transformers and Domain theory. Chapter 3 presents the definitions of RMTs and the necessary theory behind them. Chapter 4 describes our implementation of RMTs in *JavaScript* and how their used to define the denotational semantics of a simple imperative language. In Chapter 5 our benchmark algorithms are defined and we display our benchmark results of our RMT implementation, alongside with a performance comparison with Promises. Finally, in Chapter 6 we give our conclusions about RMTs and about their possible future directions.

Chapter 2

Mathematical background

In this chapter we plainly summarize the mathematical background that is necessary for the reader to comprehend the rest of the thesis. For a more informative introduction and the proofs of the theorems, the reader is referred to the related literature that is mentioned below.

2.1 Category theory

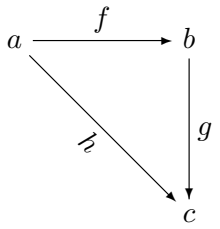
Category theory was developed in an attempt to unify simple abstract concepts that were applicable in many branches of mathematics. Excellent introductions to category theory and its applications in Computer Science can be found in [Pier90, Gogu91, Pier91, Aspe91, Barr96].

Definition 2.1.1 *A* category C *is a collection of* objects *and a collection of* arrows *satisfying the following properties:*

- *For each arrow f there is a* domain *object* $\mathbf{dom}(f)$ *and a* codomain *object* $\mathbf{codom}(f)$*, and by writing $f : x \to y$ it is indicated that $x = \mathbf{dom}(f)$ and $y = \mathbf{codom}(f)$.*
- *For every pair of arrows $f : x \to y$ and $g : y \to z$ there is a* composite *arrow $g \circ f : x \to z$.*
- *Composition of arrows is associative, i.e. for all arrows $f : x \to y$, $g : y \to z$ and $h : z \to w$ it is $h \circ (g \circ f) = (h \circ g) \circ f$.*
- *For each object x there is an* identity *arrow $\mathbf{id}_x : x \to x$.*
- *Identity arrows are identities for arrow composition, i.e. for all arrows $f : x \to y$ it is $f \circ \mathbf{id}_x = \mathbf{id}_y \circ f = f$.*

Definition 2.1.2 *Two objects x and y of category C are* isomorphic *if there are arrows $f : x \to y$ and $g : y \to x$ such that $f \circ g = \mathbf{id}_y$ and $g \circ f = \mathbf{id}_x$. The pair of arrows f and g are called an* isomorphism.

Properties of categories are commonly presented using *commuting diagrams*. A diagram is a graph whose nodes are objects and whose edges are arrows. A diagram *commutes* if for every pair of nodes and for every pair of paths connecting these two nodes the composition of arrows along the first path is equal to the composition of arrows along the second. An example of a commuting diagram, implying that $g \circ f = h$, is shown below.



Definition 2.1.3 *A* functor *F from category C to category D, written as $F : \mathsf{C} \to \mathsf{D}$, is a pair of mappings. Every object x in C is mapped to an object $F(x)$ in D and every arrow $f : x \to y$ in C is mapped to an arrow $F(f) : F(x) \to F(y)$ in D. Moreover, the following properties must be satisfied:*

- $F(\mathit{id}_x) = \mathit{id}_{F(x)}$ *for all objects x in C.*
- $F(g \circ f) = F(g) \circ F(f)$ *for all arrows $f : x \to y$ and $g : y \to z$ in C.*

Definition 2.1.4 *An* endofunctor *on category C is a functor $F : \mathsf{C} \to \mathsf{C}$.*

Definition 2.1.5 *If $F : \mathsf{C} \to \mathsf{D}$ and $G : \mathsf{D} \to \mathsf{E}$ are functors, then their* composition *is a functor $G \circ F : \mathsf{C} \to \mathsf{E}$. It is defined by taking $(G \circ F)(x) = G(F(x))$ and $(G \circ F)(f) = G(F(f))$.*

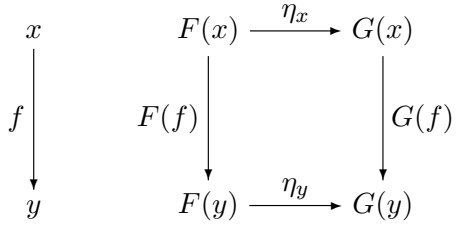
Definition 2.1.6 *For every category C, an* identity functor *$\mathit{Id}_\mathsf{C} : \mathsf{C} \to \mathsf{C}$ can be defined by taking $\mathit{Id}_\mathsf{C}(x) = x$ and $\mathit{Id}_\mathsf{C}(f) = f$.*

Note that if $F : \mathsf{C} \to \mathsf{C}$ is an endofunctor and n is a positive natural number, the notation $F^n : \mathsf{C} \to \mathsf{C}$ can be used for the composition of F with itself n times. The notation can be extended so that $F^0 = \mathit{Id}_\mathsf{C}$.

Theorem 2.1.1 *Functors preserve isomorphisms.*

Theorem 2.1.2 *Identity functors are identities for functor composition, that is, if $F : \mathsf{C} \to \mathsf{D}$ is a functor, then $F \circ \mathit{Id}_\mathsf{C} = \mathit{Id}_\mathsf{D} \circ F = F$*

Definition 2.1.7 *If $F : \mathsf{C} \to \mathsf{D}$ and $G : \mathsf{C} \to \mathsf{D}$ are functors, then a* natural transformation *η between F and G, written as $\eta : F \dot{\to} G$ is a family of arrows in D. In this family, an arrow $\eta_x : F(x) \to G(x)$ in D is defined for every object x in C. Moreover, the following diagram must commute:*

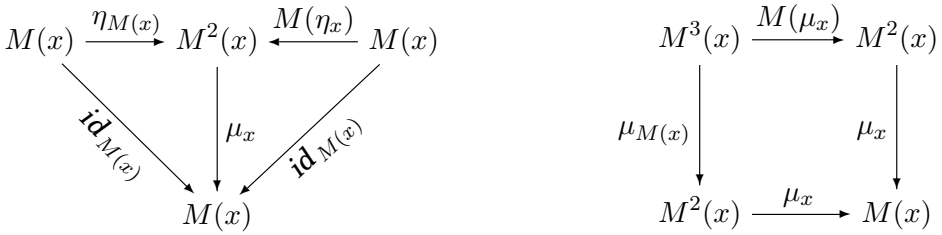


2.2 Monads and monad transformers

The notion of *monad*, also called triple, is not new in the context of category theory. In Computer Science, monads became very popular in the 1990s. The categorical properties of monads are discussed in most books on category theory, e.g. in [Barr96]. For a comprehensive introduction to monads and their use in denotational semantics the user is referred

to [Mogg90]. A somehow different approach to the definition of monads is found in [Wadl92], which expresses the current practice of monads in functional programming. The two approaches are equivalent. In this thesis, the categorical approach (presented here) is used for the definition of monads, since it is much more elegant, and the functional approach (presented in Section 2.4) is used for describing the semantics of programming languages.

Definition 2.2.1 *A* monad *on a category* C *is a triple* $\langle M, \eta, \mu \rangle$*, where* $M : \mathsf{C} \to \mathsf{C}$ *is an endofunctor,* $\eta : \mathbf{Id}_{\mathsf{C}} \dot{\to} M$ *and* $\mu : M^2 \dot{\to} M$ *are natural transformations. For all objects* x *in* C*, the following diagrams must commute.*



The transformation η *is called the* unit *of the monad, whereas the transformation* μ *is called the* multiplication *or* join*.*

The commutativity of these two diagrams is equivalent to the following three equations, commonly called the three *monad laws*:

$$\begin{array}{lcll} \mu_x \circ \eta_{M(x)} & = & \mathbf{id}_{M(x)} & \textit{(1st Monad Law)} \\ \mu_x \circ M(\eta_x) & = & \mathbf{id}_{M(x)} & \textit{(2nd Monad Law)} \\ \mu_x \circ M(\mu_x) & = & \mu_x \circ \mu_{M(x)} & \textit{(3rd Monad Law)} \end{array}$$

Definition 2.2.2 *If* C *is a category, a* monad transformer *on* C *is a mapping between monads on* C*.*[1]

2.3 Domain theory

The theory of domains was established by Scott and Strachey, in order to provide appropriate mathematical spaces on which to define the denotational semantics of programming languages. Introductions of various sizes and levels can be found in [Scot71, Scot82, Gunt90, Gunt92]. Various kinds of domains are commonly used in denotational semantics, the majority of them based on complete partial orders (cpo's). The variation used here is one of the possible options.

Definition 2.3.1 *A* partial order*, or* poset*, is a set* D *together with a binary relation* $\sqsubseteq$ *that is reflexive, anti-symmetric and transitive.*

Definition 2.3.2 *A subset* $P \subseteq D$ *of a poset* D *is* bounded *if there is a* $x \in D$ *such that* $y \sqsubseteq x$ *for all* $y \in P$*. In this case,* x *is an* upper bound *of* P*.*

[1] Many options for the definition of monad transformers have been suggested in literature. Given a category C, monads on C and *monad morphisms* (which have not been defined in this thesis) form a category $\mathsf{Mon}(\mathsf{C})$. Monad transformers can be defined as mappings between objects in $\mathsf{Mon}(\mathsf{C})$, as endofunctor on $\mathsf{Mon}(\mathsf{C})$, as premonads on $\mathsf{Mon}(\mathsf{C})$ (i.e. endofunctors with a unit), and as monads on $\mathsf{Mon}(\mathsf{C})$. In this thesis we have selected the first option.

Definition 2.3.3 *The* least upper bound *of a subset $P \subseteq D$, written as $\bigsqcup P$, is an upper bound of P such that, $\bigsqcup P \sqsubseteq x$ for all upper bounds x of P.*[2]

Definition 2.3.4 *A subset $P \subseteq D$ of a poset D is* directed *if every finite subset $F \subseteq P$ has an upper bound $x \in P$.*

Definition 2.3.5 *A poset D is* complete *if every directed subset $P \subseteq D$ has a least upper bound. A complete partial order is also called a* cpo.

Definition 2.3.6 *A* domain *is a cpo D with a bottom element, written as $\bot$. For all elements $x \in D$, it must be $\bot \sqsubseteq x$.*

Definition 2.3.7 *Every set S defines a* flat domain *S^{o}, whose underlying set is $S \cup \{ \bot \}$ and in which $x \sqsubseteq y$ iff $x = y$ or $x = \bot$.*

A number of useful domains can be defined at this point. The trivial domain **O** is the flat domain that corresponds to the empty set; it contains a single element $\bot$. A useful domain with a single ordinary element is $\mathbf{U} = \{ \boldsymbol{u} \}^{\mathrm{o}}$. The natural numbers under their usual ordering $\leq$ form a poset ω which is not a cpo, since it is directed and does not have a least upper bound.

Definition 2.3.8 *If D is a poset, an* ω-chain *$(x_n)_{n \in \omega}$ in D is a set of elements $x_n \in D$ such that $n \leq m$ implies $x_n \sqsubseteq x_m$.*

Definition 2.3.9 *A function $f : D \to E$ between posets D and E is* monotone *if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.*

Definition 2.3.10 *A function $f : D \to E$ between posets D and E is* continuous *if it is monotone and $f(\bigsqcup P) = \bigsqcup \{ f(x) \mid x \in P \}$ for all directed $P \subseteq D$.*

Definition 2.3.11 *A function $f : D \to E$ between domains D and E is* strict *if $f(\bot) = \bot$.*

Definition 2.3.12 *A relation $\sqsubseteq$ can be defined for functions between domains D and E as follows. If $f, g : D \to E$, then $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in D$.*

Theorem 2.3.1 *The set of continuous functions between D and E under the relation of Definition 2.3.12 is a domain. This domain is denoted by $D \to E$.*

Definition 2.3.13 *An element $x \in D$ is a* fixed point *of a function $f : D \to D$ if $x = f(x)$.*

Theorem 2.3.2 *If D is a domain and $f : D \to D$ is continuous, then f has a* least fixed point *$\mathbf{fix}(f) \in D$, i.e. $\mathbf{fix}(f) = f(\mathbf{fix}(f))$ and $\mathbf{fix}(f) \sqsubseteq x$ for all x such that $x = f(x)$. Furthermore, $\mathbf{fix}(f) = \bigsqcup_{n \in \omega} f^n(\bot)$.*

Theorem 2.3.3 *For all $n \in \omega$, let $f_n : A \to B$. Let $x \in A$. Then, if the least upper bound on the left hand side exists, it is $\left(\bigsqcup_{n \in \omega} f_n\right)(x) = \bigsqcup_{n \in \omega} f_n(x)$.*

Theorem 2.3.4 *Domains and continuous functions form a category* Dom.

To simplify presentation, in category Dom we often omit the parentheses surrounding a function's argument, i.e. we write $f\ x$ instead of $f(x)$.

[2] The notation $a \sqcup b$ is used as an abbreviation of $\bigsqcup\{ a, b \}$.

Definition 2.3.14 *A functor* $F : \mathsf{Dom} \to \mathsf{Dom}$ *is* locally monotone *if* $f \sqsubseteq g$ *implies* $F(f) \sqsubseteq F(g)$*, for all domains* A *and* B *and for all functions* $f, g : A \to B$*.*

Definition 2.3.15 *A functor* $F : \mathsf{Dom} \to \mathsf{Dom}$ *is* locally continuous *if it is locally monotone and* $F(\bigsqcup P) = \bigsqcup \{ F(f) \mid f \in P \}$ *for all domains* A *and* B *and for all directed* $P \subseteq A \to B$*.*

Definition 2.3.16 *If* D *and* E *are domains, then the* product $D \times E$ *is a domain. The elements of* $D \times E$ *are the pairs* $\langle x, y \rangle$ *with* $x \in D$ *and* $y \in E$*, and the ordering relation is defined as* $\langle x_1, y_1 \rangle \sqsubseteq \langle x_2, y_2 \rangle \Leftrightarrow x_1 \sqsubseteq_D x_2 \wedge y_1 \sqsubseteq_E y_2$*.*

Definition 2.3.17 *If* $D \times E$ *is a product domain, two continuous projection functions* $\mathbf{fst} : D \times E \to D$ *and* $\mathbf{snd} : D \times E \to E$ *can be defined by taking* $\mathbf{fst}\ \langle x, y \rangle = x$ *and* $\mathbf{snd}\ \langle x, y \rangle = y$*.*

Definition 2.3.18 *If* D *and* E *are domains, then the* separated sum $D + E$ *is a domain. The set of elements of* $D + E$ *is:*

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

The ordering relation is defined separately for each of the three distinct subsets of $D + E$*, i.e.* $\langle x_1, 0 \rangle \sqsubseteq \langle x_2, 0 \rangle \Leftrightarrow x_1 \sqsubseteq_D x_2$ *and* $\langle y_1, 1 \rangle \sqsubseteq \langle y_2, 1 \rangle \Leftrightarrow y_1 \sqsubseteq_E y_2$*. In addition,* $\bot_{D+E} \sqsubseteq z$ *for all* $z \in D + E$*.*

Definition 2.3.19 *If* $D + E$ *is a sum domain, two continuous injection functions* $\mathbf{inl} : D \to D + E$ *and* $\mathbf{inr} : E \to D + E$ *can be defined by taking* $\mathbf{inl}\ x = \langle x, 0 \rangle$ *and* $\mathbf{inr}\ y = \langle y, 1 \rangle$*.*

Definition 2.3.20 *If* D*,* E *and* F *are domains and* $f_1 : D \to F$ *and* $f_2 : E \to F$ *are continuous functions, a strict continuous function* $[\, f_1, f_2 \,] : D + E \to F$ *can be defined as:*

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

Theorem 2.3.5 *Let* A*,* B *and* C *be domains,* $f : A \to C$ *and* $g : B \to C$ *continuous functions. Then* $[\, f, g \,] \circ \mathbf{inl} = f$ *and* $[\, f, g \,] \circ \mathbf{inr} = g$*.*

Theorem 2.3.6 *Let* A *and* B *be domains. Then* $[\, \mathbf{inl}, \mathbf{inr} \,] = \mathbf{id}_{A+B}$*.*

Theorem 2.3.7 *Let* A_1*,* B_1*,* C_1*,* A_2*,* B_2 *and* C_2 *be domains. Let* $f_1 : B_1 \to C_1$*,* $f_2 : A_1 \to B_1$*,* $g_1 : B_2 \to C_2$ *and* $g_2 : A_2 \to B_2$ *be continuous functions. Then* $[\, f_1 \circ f_2, g_1 \circ g_2 \,] = [\, f_1, g_1 \,] \circ [\, \mathbf{inl} \circ f_2, \mathbf{inr} \circ g_2 \,]$*.*

Theorem 2.3.8 *Let* A*,* B*,* C *and* D *be domains,* $f : C \to D$*,* $g_1 : A \to C$ *and* $g_2 : B \to C$ *continuous functions. If* f *is strict, then* $f \circ [\, g_1, g_2 \,] = [\, f \circ g_1, f \circ g_2 \,]$*.*

Power-domains are the domain-theoretic equivalent of power-sets. They have been introduced as a tool for modeling the semantics of non-deterministic programs and have been widely used for the semantics of concurrency. In this thesis we avoid a full definition of power-domains; the reader is referred to [Gunt92] for a detailed definition and a study of their categoric and domain-theoretic properties.[3]

[3] We also ignore the fact that the entire category Dom of Scott domains is not appropriate for the definition of power-domains. One of the categories SFP (of sequences of finite posets) or Bif (of bifinite domains), which are closed subcategories of Dom, should be used instead. The reader is again referred to [Gunt92]. Notice, however, that the results presented in Chapter 3 equally apply to all closed subcategories of Dom, including Dom itself. We only use power-domains in Section 4.4.

Definition 2.3.21 *Let D be a domain. We write $D^\natural$ for the (convex)* power-domain *of D.*

Definition 2.3.22 *Let D and E be domains and $f : D \to E$ a continuous function. We can define a continuous function $f^\natural : D^\natural \to E^\natural$.*

Theorem 2.3.9 *By taking $P(D) = D^\natural$ and $P(f) = f^\natural$ we can define an endofunctor $P : \mathsf{Dom} \to \mathsf{Dom}$, which is called the* power-domain functor*.*

Definition 2.3.23 *Let D be a domain. We can define a continuous function $\{\!|\cdot|\!\} : D \to D^\natural$, which is called the* power-domain singleton *function.*

Definition 2.3.24 *Let D be a domain. We can define a continuous binary operation $\uplus^\natural : D^\natural \times D^\natural \to D^\natural$, which is called the* power-domain union*. Furthermore, this binary operation is associative, commutative and idempotent.*

Definition 2.3.25 *Let D be a domain. We can define a continuous function $\bigcup\!\!\!\bigcup^\natural : D^{\natural\natural} \to D^\natural$, which is called the* power-domain big union *function.*

Theorem 2.3.10 *The power-domain singleton is a natural transformation between the identity functor $\mathbf{Id}_{\mathsf{Dom}}$ and the power-domain functor P.*

Theorem 2.3.11 *The power-domain big union is a natural transformation between the functors P^2 and P.*

Theorem 2.3.12 *The power-domain functor P with the power-domain singleton as the unit and the power-domain big union as the join define a monad P, which is called the* power-domain monad*.*

2.4 Monads in functional programming

An alternative approach to the definition of monads has become very popular in the functional programming community. According to this, a monad on category Dom is defined as a triple $\langle M, \mathbf{unit}_{\mathsf{M}}, *_{\mathsf{M}}\rangle$. In this triple, M is a domain constructor, $\mathbf{unit}_{\mathsf{M}} : D \to M(D)$ is a continuous function and $*_{\mathsf{M}} : M(A) \times (A \to M(B)) \to M(B)$ is a binary operation. These triples $\langle M, \mathbf{unit}_{\mathsf{M}}, *_{\mathsf{M}}\rangle$ are often aliased as $\langle M, \mathbf{return}_M, \mathbf{bind}_M\rangle$ in functional programming languages like *Haskell*.

In the semantics of programming languages, domains constructed by monad M typically denote *computations*, e.g. the domain $M(D)$ denotes computations returning values of the domain D. The result of $\mathbf{unit}_{\mathsf{M}}\ v$ is simply a computation returning the value v and the result of $m\ *_{\mathsf{M}}\ f$ is the combined computation of m, returning v, followed by computation $f(v)$. Monad transformers are useful to transform between different types of computations [Lian95].

The following equations connect a monad $\langle M, \mathbf{unit}_{\mathsf{M}}, *_{\mathsf{M}}\rangle$ defined using the functional approach with a monad $\langle M, \eta, \mu\rangle$ defined using the categorical approach.

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

In the functional approach, the three monad laws can be formulated as follows.

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

An interesting remark is that these three laws are enough to prove that the equivalent $\langle M, \eta, \mu \rangle$, as defined above, is indeed a monad, i.e. that M is a functor (preserves function identities and composition) and that η and μ are natural transformations.

In this setting, it is useful to define two special classes of monads, equipped with additional operations that are useful for modeling the semantics of concurrency in programming languages.

Definition 2.4.1 *A* multi-monad *is a monad M with a binary operation $\|_{\mathsf{M}} : M(D) \times M(D) \to M(D)$, where D is a domain.*

Definition 2.4.2 *A* strong monad *is a monad M with a binary operation $\bowtie_{\mathsf{M}} : M(A) \times M(B) \to M(A \times B)$, where A and B are domains.*

The binary operation $\|$ of a multi-monad is used to express disjunction in computations. In other words, if M is a multi-monad, D is a domain and $m_1, m_2 \in M(D)$ are two computations, the computation $m_1 \ \| \ m_2$ indicates a (possibly non-deterministic) option between m_1 and m_2. Moreover, the binary operation $\bowtie$ of a strong monad is used to express conjunction in computations. Let M be a strong monad, let A and B be domains. If $m_1 \in M(A)$ and $m_2 \in M(B)$ are two computations, the computation $m_1 \bowtie m_2$ indicates that both m_1 and m_2 will be performed and their results will be paired. The option here relates to the order, if any, in which the two computations will be performed.

Chapter 3

Resumption monad transformers

The notion of execution *interleaving* is a well known one in the theory of concurrency. In this context, computations are considered to be sequences of *atomic steps* the nature of which depends on our notion of computation. In isolation, these atomic steps are performed one after another until the computation is complete. Given two computations A and B, an interleaved computation of A and B consists of an arbitrary merging of the atomic steps that constitute A and B. Interleaving easily extends to more than two computations. The atomic steps of any computation must still be executed in the right order, but this process can be interrupted by the execution of atomic steps belonging to other computations.

Our primary goal is to define a monad transformer R capable of modeling generic interleaved computations. In this way, if we are given a monad M which models the computations taking place at the atomic steps, we can obtain a monad $\mathsf{R}(M)$ which models interleaved computations of such atomic steps. One possible solution to this problem is to use the long suggested technique of *resumptions*, illustrated in [dBak96, Schm86] for specific instances of M.

Generalizing this technique, the domain $\mathsf{R}(M)(D)$ of resumptions must satisfy the following isomorphism:

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

In this domain, atomic steps are arbitrary computations defined by M. The left part of the sum represents an already evaluated result, i.e. a computation that consists of zero atomic steps. The right part represents a computation that requires at least one atomic step. The result of this atomic step is a new element of the resumption domain.

In this chapter we formally define the Resumption Monad Transformer as presented in [Papa01], where all the proofs for the following theorems and lemmata can be found. We start by considering an arbitrary locally continuous monad M on Dom. The rest of the chapter is organized as follows. In Section 3.1 we define an endofunctor $\mathbf{R}_M : \mathsf{Dom} \to \mathsf{Dom}$. In Section 3.2 we define two natural transformations $\mathit{unit} : \mathit{Id} \stackrel{\cdot}{\to} \mathbf{R}_M$ and $\mathit{join} : \mathbf{R}_M^2 \stackrel{\cdot}{\to} \mathbf{R}_M$ and in Section 3.3 we prove that $\langle \mathbf{R}_M, \mathit{unit}, \mathit{join} \rangle$ satisfies the three monad laws. In this way we define the monad transformer R. Next, in Section 3.4 we prove that $\mathsf{R}(M)(D)$ satisfies the aforementioned isomorphism by constructing the two components h^e and h^p of the isomorphism. Finally, in Section 3.5 we define a few additional operations on domains constructed by $\mathsf{R}(M)$, which will be handy in specifying the semantics of concurrent programming languages.

3.1 Functor $\mathbf{R}_M$

We start by defining for each domain D an endofunctor $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$, and some auxiliary functions. The domain $\mathsf{R}(M)(D)$ that we are trying to define is a fixed point of $\mathbf{F}_{M,D}$.

Definition 3.1.1 *Let D, A and B be domains and $f : A \to B$ a continuous function. We define the following mappings:*

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

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

Theorem 3.1.1 $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ *is a functor.*

It is not hard to prove that the functor $\mathbf{F}_{M,D}$ is locally monotone and locally continuous. This result comes easily, since monad M has these two properties and $\mathbf{F}_{M,D}$ is defined in terms of M, using only basic domain operations which preserve monotonicity and continuity.

Lemma 3.1.2 *Functor $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ is locally monotone.*

Lemma 3.1.3 *Functor $\mathbf{F}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ is locally continuous.*

The two functions ι^e and ι^p are useful in the definition of $\mathbf{R}_M(D)$. They define an embedding and a projection between the domains $\mathbf{O}$ and $\mathbf{F}_{M,D}(\mathbf{O})$.

Definition 3.1.2 *Let D be a domain. We define the pair of continuous functions $\iota^e : \mathbf{O} \to \mathbf{F}_{M,D}(\mathbf{O})$ and $\iota^p : \mathbf{F}_{M,D}(\mathbf{O}) \to \mathbf{O}$ to be equal to $\bot$.*

Therefore, it can be proved that:

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

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

We proceed by defining a mapping of objects and a mapping of functions, which will define the endofunctor $\mathbf{R}_M : \mathsf{Dom} \to \mathsf{Dom}$ at the end of this section. This is the key definition of [Papa01] that we use in the rest of this thesis.

Definition 3.1.3 *Let D be a domain. The domain $\mathbf{R}_M(D)$ is the set*

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

with its elements ordered pointwise:

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

The elements of the domain $\mathbf{R}_M(D)$ are infinite sequences, indexed by the set of natural numbers ω. The n-th element of the sequence is an element of the domain $\mathbf{F}^n_{M,D}(\mathbf{O})$. Such elements represent *finite approximations* of resumption computations: if a given resumption computation terminates in less than n steps, its n-th approximation is able to compute the result accurately; otherwise it produces $\bot$. The condition $x_n = \mathbf{F}^n_{M,D}(\iota^p)(x_{n+1})$ states that the elements of the infinite sequence must indeed be approximations: the result of projecting the $(n+1)$-th approximation (an element of $\mathbf{F}^{n+1}_{M,D}(\mathbf{O})$) to an element of $\mathbf{F}^n_{M,D}(\mathbf{O})$ must be equal to the n-th approximation.

Before we can define the mapping of functions that corresponds to $\mathbf{R}_M$, it is necessary to define a few families of auxiliary functions. The first is the family of functions $f^D_{m,n}$ which map between different approximations of a resumption computation.

Definition 3.1.4 *Let D be a domain. For all $m, n \in \omega$, we define a function $f^D_{m,n} : \mathbf{F}^m_{M,D}(\mathbf{O}) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ by:*

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

So, the following lemmata hold:

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

Lemma 3.1.7 *For all $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$ and for all $m, n \in \omega$, $f^D_{m,n}\ x_n \sqsubseteq x_m$.*

The families of μ^e_n and μ^p_n functions also define mappings between resumption computations and their approximations. The former embeds an approximation requiring less than n steps to an element of the domain $\mathbf{R}_M(D)$, while the latter projects an element of the domain $\mathbf{R}_M(D)$ to its n-th approximation.

Definition 3.1.5 *Let D be a domain, $n \in \omega$, $z \in \mathbf{F}^n_{M,D}(\mathbf{O})$ and $(x_m)_{m\in\omega} \in \mathbf{R}_M(D)$. We define the pair of functions $\mu^e_n : \mathbf{F}^n_{M,D}(\mathbf{O}) \to \mathbf{R}_M(D)$ and $\mu^p_n : \mathbf{R}_M(D) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ as follows:*

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

We are now ready to define the mapping of functions required by the functor $\mathbf{R}_M$. Instead of defining this mapping directly in terms of elements of the resumption domain $\mathbf{R}_M(D)$, we use the family of functions $\zeta^{A,B}_n$ and define it in terms of the finite approximations.

Definition 3.1.6 *Let A and B be domains and let $f : A \to B$ be a continuous function. For all $n \in \omega$ we define a strict continuous function $\zeta^{A,B}_n\ f : \mathbf{F}^n_{M,A}(\mathbf{O}) \to \mathbf{F}^n_{M,B}(\mathbf{O})$ by:*

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

Definition 3.1.7 *Let A and B be domains and let $f : A \to B$ be a continuous function. We define a continuous function $\mathbf{R}_M(f) : \mathbf{R}_M(A) \to \mathbf{R}_M(B)$ by:*

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

The central result of this section is Theorem 3.1.2 in which we prove that $\mathbf{R}_M$ is a functor. For doing so, we make use of the following lemmata, whose proofs can again be found in [Papa01].

Lemma 3.1.8 *For all* $n \in \omega$, $\mu^e_n \circ \mathbf{F}^n_{M,D}(\iota^p) \sqsubseteq \mu^e_{n+1}$.

Lemma 3.1.9 *Let* A *and* B *be domains,* $f : A \to B$ *a continuous function. Then for all* $n \in \omega$,
$\zeta^{A,B}_{n+1}\ f \circ \mathit{inr} = \mathit{inr} \circ M(\zeta^{A,B}_n\ f)$

Lemma 3.1.10 *For all* $x \in \mathbf{R}_M(D)$, $(\mu^p_n\ x)_{n\in\omega} = x$.

Lemma 3.1.11 *For all* $m, n \in \omega$, $\mu^p_m \circ \mu^e_n = f^D_{m,n}$.

Lemma 3.1.12 *For all* $n \in \omega$, $\mu^p_n \circ \mu^e_n = \mathit{id}_{\mathbf{F}^n_{M,D}(\mathbf{O})}$.

Lemma 3.1.13 *For all* $n \in \omega$, $\mu^e_m \circ \mu^p_n \sqsubseteq \mathit{id}_{\mathbf{R}_M(D)}$.

Lemma 3.1.14 *Let* A *and* B *be domains,* $f : A \to B$ *a continuous function. Then for all* $n \in \omega$,
$\mu^p_n \circ \mathbf{R}_M(f) = \zeta^{A,B}_n\ f \circ \mu^p_n$

Lemma 3.1.15 *Let* A *be a domain. Then for all* $n \in \omega$, $\zeta^{A,A}_n\ \mathit{id}_A = \mathit{id}_{\mathbf{F}^n_{M,A}(\mathbf{O})}$.

Lemma 3.1.16 *Let* A, B *and* C *be domains,* $f : A \to B$ *and* $g : B \to C$ *continuous functions. Then for all* $n \in \omega$, $\zeta^{A,C}_n\ (g \circ f) = \zeta^{B,C}_n\ g \circ \zeta^{A,B}_n\ f$.

We can now proceed with the proof of Theorem 3.1.2.

Theorem 3.1.2 $\mathbf{R}_M : \mathsf{Dom} \to \mathsf{Dom}$ *is a functor.*
Proof We must prove that $\mathbf{R}_M$ preserves identities and the composition of continuous functions.

1. Let X be a domain and $(x_n)_{n\in\omega} \in \mathbf{R}_M(X)$.
$\mathbf{R}_M(\mathit{id}_X)\ (x_n)_{n\in\omega}$
$= \langle$ *Definition of* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$(\zeta^{X,X}_n\ \mathit{id}_X\ x_n)_{n\in\omega}$
$= \langle$ *Lemma 3.1.15* $\rangle$
$(\mathit{id}_{\mathbf{F}^n_{M,X}(\mathbf{O})}\ x_n)_{n\in\omega}$
$= \langle$ *Identity function* $\rangle$
$(x_n)_{n\in\omega}$
$= \langle$ *Identity function* $\rangle$
$\mathit{id}_{\mathbf{R}_M(X)}\ (x_n)_{n\in\omega}$

2. Let A and B be domains, $f : A \to B$ and $g : B \to C$ continuous functions and $(x_n)_{n\in\omega} \in \mathbf{R}_M(X)$.
$\mathbf{R}_M(g \circ f)\ (x_n)_{n\in\omega}$
$= \langle$ *Definition of* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$(\zeta^{A,C}_n\ (g \circ f)\ x_n)_{n\in\omega}$
$= \langle$ *Lemma 3.1.16* $\rangle$
$((\zeta^{B,C}_n\ g \circ \zeta^{A,B}_n\ f)\ x_n)_{n\in\omega}$

$= \langle$ *Composition* $\rangle$
$(\zeta_n^{B,C}\ g\ (\zeta_n^{A,B}\ f\ x_n))_{n\in\omega}$
$= \langle$ *Definition of* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\mathbf{R}_M(g)\ (\zeta_n^{A,B}\ f\ x_n)_{n\in\omega}$
$= \langle$ *Definition of* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\mathbf{R}_M(g)\ (\mathbf{R}_M(f)\ (x_n)_{n\in\omega})$
$= \langle$ *Composition* $\rangle$
$(\mathbf{R}_M(g) \circ \mathbf{R}_M(f))\ (x_n)_{n\in\omega}$ □

3.2 Unit and join

Having defined $\mathbf{R}_M$ as a functor, we now define the two monad operations ***unit*** and ***join***. For each one, it is proved in [Papa01] that it is a natural transformation.

The ***unit*** function maps an element $d \in D$ to a resumption computation, using the family of auxiliary functions η. All approximations in the resumption computation are equal to ***inl*** d (except for the trivial approximation of zero steps).

Definition 3.2.1 *Let D be a domain. For all $n \in \omega$ we define a continuous function $\eta_n^D : D \to \mathbf{F}_{M,D}^n(\mathbf{O})$ by:*

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

Definition 3.2.2 *Let D be a domain and $d \in D$. We define the function $\textbf{\textit{unit}}_D : D \to \mathbf{R}_M(D)$ by:*

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

The following lemma is useful in proving that ***unit*** is a natural transformation.

Lemma 3.2.1 *Let A and B be domains, $f : A \to B$ a continuous function. Then, for all $n \in \omega$,*
$\zeta_n^{A,B}\ f \circ \eta_n^A = \eta_n^B \circ f$

Which can be used to prove the following:

Theorem 3.2.1 ***unit*** $: \textbf{\textit{Id}} \dot{\to} \mathbf{R}_M$ *is a natural transformation.*
Proof Let A and B be domains and $f : A \to B$ a continuous function. We must show that $\textbf{\textit{unit}}_B \circ f = \mathbf{R}_M(f) \circ \textbf{\textit{unit}}_A$. Let $a \in A$.

$(\textbf{\textit{unit}}_B \circ f)\ a$
$= \langle$ *Composition* $\rangle$
$\textbf{\textit{unit}}_B\ (f\ a)$
$= \langle$ *Definition of* ***unit*** *(3.2.2)* $\rangle$
$(\eta_n^D\ (f\ a))_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$((\eta_n^D \circ f)\ a)_{n\in\omega}$
$= \langle$ *Lemma 3.2.1* $\rangle$
$((\zeta_n^{A,B} \circ \eta_n^A)\ a)_{n\in\omega}$

$= \langle$ *Composition* $\rangle$
$(\zeta_n^{A,B}\ (\eta_n^A\ a))_{n\in\omega}$
$= \langle$ *Definition of* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\mathbf{R}_M(f)\ (\eta_n^A\ a)_{n\in\omega}$
$= \langle$ *Definition of* ***unit*** *(3.2.2)* $\rangle$
$\mathbf{R}_M(f)\ (\mathbf{unit}_A\ a)$
$= \langle$ *Composition* $\rangle$
$(\mathbf{R}_M(f) \circ \mathbf{unit}_A)\ a$ □

The definition of the ***join*** function requires the family of functions ξ which associate corresponding approximations in the domains $\mathbf{R}_M(D)$ and D.

Definition 3.2.3 *Let D be a domain. For all $n \in \omega$ we define a strict continuous function $\xi_n^D : \mathbf{F}^n_{M,\mathbf{R}_M(D)}(\mathbf{O}) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ by:*

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

Definition 3.2.4 *Let D be a domain and $(x_n)_{n\in\omega} \in \mathbf{R}_M^2(D)$. We define the function $\mathbf{join}_D : \mathbf{R}_M^2(D) \to \mathbf{R}_M(D)$ by:*

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

The following lemmata are necessary for proving that ***join*** is a natural transformation.

Lemma 3.2.2 *Let D be a domain. Then for all $n \in \omega$,*
$\xi_{n+1}^D \circ \mathbf{inr} = \mathbf{inr} \circ M(\xi_n^D)$

Lemma 3.2.3 *Let A and B be domains, $f : A \to B$ a continuous function. Then for all $n \in \omega$,*
$\xi_n^B \circ \zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f)) = \zeta_n^{A,B}\ f \circ \xi_n^A$

And finally it can be proved that:

Theorem 3.2.2 ***join*** $: \mathbf{R}_M^2 \dot{\to} \mathbf{R}_M$ *is a natural transformation.*
Proof Let A and B be domains and $f : A \to B$ a continuous function. We must show that $\mathbf{join}_B \circ \mathbf{R}_M(\mathbf{R}_M(f)) = \mathbf{R}_M(f) \circ \mathbf{join}_A$. Let $(x_n)_{n\in\omega} \in \mathbf{R}_M^2(A)$.

$(\mathbf{join}_B \circ \mathbf{R}_M(\mathbf{R}_M(f)))\ (x_n)_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$\mathbf{join}_B\ (\mathbf{R}_M(\mathbf{R}_M(f))\ (x_n)_{n\in\omega})$
$= \langle$ *Definition of* $\mathbf{R}_M$ *(3.1.7)* $\rangle$
$\mathbf{join}_B\ (\zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f))\ x_n)_{n\in\omega}$
$= \langle$ *Definition of* ***join*** *(3.2.4)* $\rangle$
$(\xi_n^B\ (\zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f))\ x_n))_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$((\xi_n^B \circ \zeta_n^{\mathbf{R}_M(A),\mathbf{R}_M(B)}\ (\mathbf{R}_M(f)))\ x_n)_{n\in\omega}$
$= \langle$ *Lemma 3.2.3* $\rangle$
$((\zeta_n^{A,B}\ f \circ \xi_n^A)\ x_n)_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$(\zeta_n^{A,B}\ f\ (\xi_n^A\ x_n))_{n\in\omega}$
$= \langle$ *Definition of* $\mathbf{R}_M(f)$ *(3.1.7)* $\rangle$

$\mathbf{R}_M(f)\ (\xi_n^A\ x_n)_{n\in\omega}$
$= \langle$ *Definition of* ***join*** *(3.2.4)* $\rangle$
$\mathbf{R}_M(f)\ (\mathit{join}_A\ (x_n)_{n\in\omega})$
$= \langle$ *Composition* $\rangle$
$(\mathbf{R}_M(f) \circ \mathit{join}_A)\ (x_n)_{n\in\omega}$ □

3.3 Monad $\mathsf{R}(M)$

In this section we show that functor $\mathbf{R}_M$ together with the natural transformations ***unit*** and ***join*** defines a computational monad, in the sense of [Mogg89]. The three first theorems of this section verify the three monad laws and the following lemmata are necessary for proving them. The fourth theorem proves that the defined monad satisfies the mono requirement. Let D be a domain.

Lemma 3.3.1 *For all* $n \in \omega$, $\xi_n^D \circ \eta_n^{\mathbf{R}_M(D)} = \mu_n^p$.

Lemma 3.3.2 *For all* $n \in \omega$, $\mu_{n+1}^p \circ \mathit{unit}_D = \mathit{inl}$.

Lemma 3.3.3 *For all* $n \in \omega$, $\xi_n^D \circ \zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D = \mathit{id}_{\mathbf{F}^n_{M,D}(\mathbf{O})}$.

Lemma 3.3.4 *For all* $n \in \omega$, $\mu_n^p \circ \mathit{join}_D = \xi_n^D \circ \mu_n^p$.

Lemma 3.3.5 *For all* $d \in \omega$, $\xi_n^D \circ \zeta_n^{\mathbf{R}_M^2(D),\mathbf{R}_M(D)}\ \mathit{join}_D = \xi_n^D \circ \xi_n^{\mathbf{R}_M(D)}$.

We can now proceed by proving the three monad laws.

Theorem 3.3.1 (1st Monad Law) $\mathit{join}_D \circ \mathit{unit}_{\mathbf{R}_M(D)} = \mathit{id}_{\mathbf{R}_M(D)}$

Proof Let $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$. Then

$(\mathit{join}_D \circ \mathit{unit}_{\mathbf{R}_M(D)})\ (x_n)_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$\mathit{join}_D\ (\mathit{unit}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega})$
$= \langle$ *Definition of* ***unit*** *(3.2.2)* $\rangle$
$\mathit{join}_D\ (\eta_n^{\mathbf{R}_M(D)}\ (x_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Definition of* ***join*** *(3.2.4)* $\rangle$
$(\xi_n^D\ (\eta_n^{\mathbf{R}_M(D)}\ (x_m)_{m\in\omega}))_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$((\xi_n^D \circ \eta_n^{\mathbf{R}_M(D)})\ (x_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Lemma 3.3.1* $\rangle$
$(\mu_n^p\ (x_m)_{m\in\omega})_{n\in\omega}$
$= \langle$ *Definition of* μ_n^p *(3.1.5)* $\rangle$
$(x_n)_{n\in\omega}$
$= \langle$ *Identity* $\rangle$
$\mathit{id}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}$ □

Theorem 3.3.2 (2nd Monad Law) $\mathit{join}_D \circ \mathbf{R}_M(\mathit{unit}_D) = \mathit{id}_{\mathbf{R}_M(D)}$
Proof Let $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$. Then

$$
\begin{array}{ll}
 & (\mathit{join}_D \circ \mathbf{R}_M(\mathit{unit}_D))\ (x_n)_{n\in\omega} \\
= & \langle\ \textit{Composition}\ \rangle \\
 & \mathit{join}_D\ (\mathbf{R}_M(\mathit{unit}_D)\ (x_n)_{n\in\omega}) \\
= & \langle\ \textit{Definition of}\ \mathbf{R}_M\ \textit{(3.1.7)}\ \rangle \\
 & \mathit{join}_D\ (\zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D\ x_n)_{n\in\omega} \\
= & \langle\ \textit{Definition of}\ \mathit{join}\ \textit{(3.2.4)}\ \rangle \\
 & (\xi_n^D\ (\zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D\ x_n))_{n\in\omega} \\
= & \langle\ \textit{Composition}\ \rangle \\
 & ((\xi_n^D \circ \zeta_n^{D,\mathbf{R}_M(D)}\ \mathit{unit}_D)\ x_n)_{n\in\omega} \\
= & \langle\ \textit{Lemma 3.3.3}\ \rangle \\
 & (\mathit{id}_{\mathbf{F}^n_{M,D}(\mathbf{O})}\ x_n)_{n\in\omega} \\
= & \langle\ \textit{Identity}\ \rangle \\
 & (x_n)_{n\in\omega} \\
= & \langle\ \textit{Identity}\ \rangle \\
 & \mathit{id}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}
\end{array}
$$

□

Theorem 3.3.3 (3rd Monad Law) $\mathit{join}_D \circ \mathbf{R}_M(\mathit{join}_D) = \mathit{join}_D \circ \mathit{join}_{\mathbf{R}_M(D)}$
Proof Let $(x_n)_{n\in\omega} \in \mathbf{R}^3_M(D)$. Then

$$
\begin{array}{ll}
 & (\mathit{join}_D \circ \mathbf{R}_M(\mathit{join}_D))\ (x_n)_{n\in\omega} \\
= & \langle\ \textit{Composition}\ \rangle \\
 & \mathit{join}_D\ (\mathbf{R}_M(\mathit{join}_D)\ (x_n)_{n\in\omega}) \\
= & \langle\ \textit{Definition of}\ \mathbf{R}_M\ \textit{(3.1.7)}\ \rangle \\
 & \mathit{join}_D\ (\zeta_n^{\mathbf{R}^2_M(D),\mathbf{R}_M(D)}\ \mathit{join}_D\ x_n)_{n\in\omega} \\
= & \langle\ \textit{Definition of}\ \mathit{join}\ \textit{(3.2.4)}\ \rangle \\
 & (\xi_n^D\ (\zeta_n^{\mathbf{R}^2_M(D),\mathbf{R}_M(D)}\ \mathit{join}_D\ x_n))_{n\in\omega} \\
= & \langle\ \textit{Composition}\ \rangle \\
 & ((\xi_n^D \circ \zeta_n^{\mathbf{R}^2_M(D),\mathbf{R}_M(D)}\ \mathit{join}_D)\ x_n)_{n\in\omega} \\
= & \langle\ \textit{Lemma 3.3.5}\ \rangle \\
 & ((\xi_n^D \circ \xi_n^{\mathbf{R}_M(D)})\ x_n)_{n\in\omega} \\
= & \langle\ \textit{Composition}\ \rangle \\
 & (\xi_n^D\ (\xi_n^{\mathbf{R}_M(D)}\ x_n))_{n\in\omega} \\
= & \langle\ \textit{Definition of}\ \mathit{join}\ \textit{(3.2.4)}\ \rangle \\
 & \mathit{join}_D\ (\xi_n^{\mathbf{R}_M(D)}\ x_n)_{n\in\omega} \\
= & \langle\ \textit{Definition of}\ \mathit{join}\ \textit{(3.2.4)}\ \rangle \\
 & \mathit{join}_D\ (\mathit{join}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}) \\
= & \langle\ \textit{Composition}\ \rangle \\
 & (\mathit{join}_D \circ \mathit{join}_{\mathbf{R}_M(D)})\ (x_n)_{n\in\omega}
\end{array}
$$

□

Having established that $\mathbf{R}_M$ satisfies the three monad laws, we can now conclude the definition of the resumption monad transformer R.

Definition 3.3.1 *The* resumption monad transformer R *is defined by the mapping* $\mathsf{R}(M) = \mathbf{R}_M$.

We now prove that $\mathsf{R}(M)$ is a computational monad, as defined in [Mogg89], and therefore useful as an equational model of computations in the semantics of programming languages.

Theorem 3.3.4 $\mathsf{R}(M)$ *is a computational monad, i.e. it satisfies the mono requirement.*

Proof Let D be a domain. We must show that $\mathbf{unit}_D$ is a monomorphism, i.e. we must show that for all domains E and for all continuous functions $f, g : E \to D$, $\mathbf{unit}_D \circ f = \mathbf{unit}_D \circ g$ implies $f = g$. Consider an arbitrary $x \in E$. Then, we have

$$
\begin{array}{ll}
& \mathbf{unit}_D\ (f\ x) = \mathbf{unit}_D\ (g\ x) \\
\Leftrightarrow & \langle\ \textit{Definition of } \mathbf{unit}\ \textit{(3.2.2)}\ \rangle \\
& (\eta^D_n\ (f\ x))_{n\in\omega} = (\eta^D_n\ (g\ x))_{n\in\omega} \\
\Leftrightarrow & \langle\ \textit{Equality of infinite sequences is defined pointwise}\ \rangle \\
& \forall\, n \in \omega.\ \ \eta^D_n\ (f\ x) = \eta^D_n\ (g\ x) \\
\Leftrightarrow & \langle\ \textit{Definition of } \eta\ \textit{(3.2.1), the case } n = 0\ \textit{is trivial}\ \rangle \\
& \mathbf{inl}\ (f\ x) = \mathbf{inl}\ (g\ x) \\
\Leftrightarrow & \langle\ \mathbf{inl}\ \textit{is an injection}\ \rangle \\
& f\ x = g\ x
\end{array}
$$

□

3.4 Isomorphism

Let D be a domain. In this section, we define the pair of functions h^e and h^p that establish the isomorphism between domains $\mathbf{R}_M(D)$ and $D + M(\mathbf{R}_M(D))$. Using these functions, it is possible to define an operation in one of these two domains and obtain the corresponding operation on the other domain by applying h^e and h^p appropriately.

The definition of the embedding function h^e is straightforward. We make use of a family of auxiliary functions θ, which construct the necessary approximations.

Definition 3.4.1 *For all $n \in \omega$ we define a strict continuous function $\theta^D_n : \mathbf{F}_{M,D}(\mathbf{R}_M(D)) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ by:*

$$
\begin{array}{lcl}
\theta^D_0 & = & \bot \\
\theta^D_{n+1} & = & [\,\mathbf{inl}, \mathbf{inr} \circ M(\mu^p_n)\,]
\end{array}
$$

Definition 3.4.2 *Let $z \in \mathbf{F}_{M,D}(\mathbf{R}_M(D))$. We define a continuous function $h^e : \mathbf{F}_{M,D}(\mathbf{R}_M(D)) \to \mathbf{R}_M(D)$ by:*

$$h^e\ z \ = \ (\theta^D_n\ z)_{n\in\omega}$$

On the other hand, the definition of the projection function h^p is more complicated. It first requires the definition of an additional domain $\mathbf{Q}_M(D)$ whose elements are infinite sequences of computations yielding approximations (we will call them *approximate computations* for short). We also find it helpful to define a family of auxiliary functions σ for associating elements of $\mathbf{Q}_M(D)$ with approximations in $\mathbf{R}_M(D)$.

Definition 3.4.3 *The domain $\mathbf{Q}_M(D)$ is the set*

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

with its elements ordered pointwise:

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

Definition 3.4.4 *Let $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$. For all $n \in \omega$, we define a continuous function $\sigma_n^D : \mathbf{Q}_M(D) \to \mathbf{F}^n_{M,D}(\mathbf{O})$ by:*

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

Furthermore, the definition of h^p requires the proof of Lemma 3.4.1, which states that elements of $\mathbf{R}_M(D)$ come in three distinct forms. Its proof can again be found at [Papa01]. This lemma is crucial in the definition of h^p and in the proofs of several theorems that follow.

Lemma 3.4.1 *Let $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$. Then exactly one of the following is true:*

1. *For all $n \in \omega$, $x_n = \bot$.*
2. *There exists a $t \in D$ such that for all $n \in \omega$, $x_n = \eta_n^D\ t$.*
3. *There exists a $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$ such that for all $n \in \omega$, $x_n = \sigma_n^D\ (z_m)_{m\in\omega}$.*

We can now proceed with the definition of h^p, based on the three cases of Lemma 3.4.1. For the first two cases, the definition is easy. In the third case, each approximate computation z_n is mapped to a computation $M(\mu_n^e)\ z_n \in M(\mathbf{R}_M(D))$ and the least upper bound of this infinite series of computations is taken.

Definition 3.4.5 *We define the function $h^p : \mathbf{R}_M(D) \to \mathbf{F}_{M,D}(\mathbf{R}_M(D))$ by case analysis on its argument $(x_n)_{n\in\omega}$ based on Lemma 3.4.1:*

1. *If for all $n \in \omega$, $x_n = \bot$, then*
   $$h^p\ (x_n)_{n\in\omega} \ = \ \bot$$
2. *If there exists a $t \in D$ such that for all $n \in \omega$, $x_n = \eta_n^D\ t$, then*
   $$h^p\ (x_n)_{n\in\omega} \ = \ \textbf{\textit{inl}}\ t$$
3. *If there exists a $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$ such that for all $n \in \omega$, $x_n = \sigma_n^D\ (z_m)_{m\in\omega}$, then*
   $$h^p\ (x_n)_{n\in\omega} \ = \ \textbf{\textit{inr}}\ \left(\bigsqcup_{n\in\omega} M(\mu_n^e)\ z_n \right)$$

In order to ensure that the least upper bound in the third case of the previous definition exists, we need Lemma 3.4.2 which states that $M(\mu_n^e)\ z_n$ form an ω-chain.

Lemma 3.4.2 *Let $(z_n)_{n\in\omega} \in \mathbf{Q}_M(D)$. For all $n \in \omega$,*
$M(\mu_n^e)\ z_n \sqsubseteq M(\mu_{n+1}^e)\ z_{n+1}$

The following lemmata are necessary for proving the central theorems of this section.

Lemma 3.4.3 *For all $t \in D$, for all $n \in \omega$, $\theta_n^D\ (\textbf{\textit{inl}}\ t) = \eta_n^D\ t$.*

Lemma 3.4.4 *For all $w \in M(\mathbf{R}_M(D))$, for all $n \in \omega$,*
$\theta_n^D\ (\textbf{\textit{inr}}\ w) = \sigma_n^D\ (M(\mu_m^p)\ w)_{m\in\omega}$

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

Lemma 3.4.6 *For all $w \in M(\mathbf{R}_M(D))$,* $\bigsqcup_{n\in\omega} M(\mu_n^e \circ \mu_n^p)\ w = w.$

Lemma 3.4.7 *Let* $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$. *For all* $m \in \omega$, $\bigsqcup_{n\in\omega} M(f^D_{m,n})\ z_n = z_m$.

At this point, we can proceed to Theorem 3.4.1 and Theorem 3.4.2, our central results in this section. These two theorems conclude that functions h^e and h^p define indeed an isomorphism between the domains $\mathbf{R}_M(D)$ and $D + M(\mathbf{R}_M(D))$.

Theorem 3.4.1 $h^p \circ h^e = \mathit{id}_{\mathbf{F}_{M,D}(\mathbf{R}_M(D))}$
Proof Let $z \in \mathbf{F}_{M,D}(\mathbf{R}_M(D)) = D + M(\mathbf{R}_M(D))$. By case analysis on z.

1. Case $z = \bot$. Then

$$\begin{aligned}
&(h^p \circ h^e)\ z\\
=&\ \langle\ \textit{Assumption}\ \rangle\\
&(h^p \circ h^e)\ \bot\\
=&\ \langle\ \textit{Composition}\ \rangle\\
&h^p\ (h^e\ \bot)\\
=&\ \langle\ \textit{Definition of } h^e\ \textit{(3.4.2)}\ \rangle\\
&h^p\ (\bot)_{n\in\omega}\\
=&\ \langle\ \textit{Definition of } h^p\ \textit{(3.4.5)}\ \rangle\\
&\bot\\
=&\ \langle\ \textit{Assumption}\ \rangle\\
&z
\end{aligned}$$

2. Case $z = \mathbf{inl}\ t$ for some $t \in D$. Then

$$\begin{aligned}
&(h^p \circ h^e)\ z\\
=&\ \langle\ \textit{Assumption}\ \rangle\\
&(h^p \circ h^e)\ (\mathbf{inl}\ t)\\
=&\ \langle\ \textit{Composition}\ \rangle\\
&h^p\ (h^e\ (\mathbf{inl}\ t))\\
=&\ \langle\ \textit{Definition of } h^e\ \textit{(3.4.2)}\ \rangle\\
&h^p\ (\theta^D_n\ (\mathbf{inl}\ t))_{n\in\omega}\\
=&\ \langle\ \textit{Lemma 3.4.3}\ \rangle\\
&h^p\ (\eta^D_n\ t)_{n\in\omega}\\
=&\ \langle\ \textit{Definition of } h^p\ \textit{(3.4.5)}\ \rangle\\
&\mathbf{inl}\ t\\
=&\ \langle\ \textit{Assumption}\ \rangle\\
&z
\end{aligned}$$

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

$$\begin{aligned}
&(h^p \circ h^e)\ z\\
=&\ \langle\ \textit{Assumption}\ \rangle\\
&(h^p \circ h^e)\ (\mathbf{inr}\ w)\\
=&\ \langle\ \textit{Composition}\ \rangle\\
&h^p\ (h^e\ (\mathbf{inr}\ w))\\
=&\ \langle\ \textit{Definition of } h^e\ \textit{(3.4.2)}\ \rangle\\
&h^p\ (\theta^D_n\ (\mathbf{inr}\ w))_{n\in\omega}\\
=&\ \langle\ \textit{Lemma 3.4.4}\ \rangle\\
&h^p\ (\sigma^D_n\ (M(\mu^p_m)\ w)_{m\in\omega})_{n\in\omega}\\
=&\ \langle\ \textit{Definition of } h^p\ \textit{(3.4.5)}\ \rangle\\
&\mathbf{inr}\ \left(\bigsqcup_{n\in\omega} M(\mu^e_n)\ (M(\mu^p_n)\ w)\right)
\end{aligned}$$

$= \langle$ *Composition* $\rangle$
$\quad \textbf{\textit{inr}} \left(\bigsqcup_{n\in\omega} (M(\mu^e_n) \circ M(\mu^p_n))\ w \right)$
$= \langle$ M *is a functor* $\rangle$
$\quad \textbf{\textit{inr}} \left(\bigsqcup_{n\in\omega} M(\mu^e_n \circ \mu^p_n)\ w \right)$
$= \langle$ *Lemma 3.4.6* $\rangle$
$\quad \textbf{\textit{inr}}\ w$
$= \langle$ *Assumption* $\rangle$
$\quad z$ □

Theorem 3.4.2 $h^e \circ h^p = \textbf{\textit{id}}_{\mathbf{R}_M(D)}$
Proof Let $(x_n)_{n\in\omega} \in \mathbf{R}_M(D)$. By case analysis on $(x_n)_{n\in\omega}$ based on Lemma 3.4.1:

1. If for all $n \in \omega$, $x_n = \bot$, then
$\quad (h^e \circ h^p)\ (x_n)_{n\in\omega}$
$= \langle$ *Assumption* $\rangle$
$\quad (h^e \circ h^p)\ (\bot)_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$\quad h^e\ (h^p\ (\bot)_{n\in\omega})$
$= \langle$ *Definition of* h^p *(3.4.5)* $\rangle$
$\quad h^e\ \bot$
$= \langle$ *Definition of* h^e *(3.4.2)* $\rangle$
$\quad (\theta^D_n\ \bot)_{n\in\omega}$
$= \langle$ *Definition of* θ *(3.4.1)* $\rangle$
$\quad (\bot)_{n\in\omega}$
$= \langle$ *Assumption* $\rangle$
$\quad (x_n)_{n\in\omega}$
$= \langle$ *Identity* $\rangle$
$\quad \textbf{\textit{id}}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}$

2. If there exists a $t \in D$ such that for all $n \in \omega$, $x_n = \eta^D_n\ t$, then
$\quad (h^e \circ h^p)\ (x_n)_{n\in\omega}$
$= \langle$ *Assumption* $\rangle$
$\quad (h^e \circ h^p)\ (\eta^D_n\ t)_{n\in\omega}$
$= \langle$ *Composition* $\rangle$
$\quad h^e\ (h^p\ (\eta^D_n\ t)_{n\in\omega})$
$= \langle$ *Definition of* h^p *(3.4.5)* $\rangle$
$\quad h^e\ (\textbf{\textit{inl}}\ t)$
$= \langle$ *Definition of* h^e *(3.4.2)* $\rangle$
$\quad (\theta^D_n\ (\textbf{\textit{inl}}\ t))_{n\in\omega}$
$= \langle$ *Lemma 3.4.3* $\rangle$
$\quad (\eta^D_n\ t)_{n\in\omega}$
$= \langle$ *Assumption* $\rangle$
$\quad (x_n)_{n\in\omega}$
$= \langle$ *Identity* $\rangle$
$\quad \textbf{\textit{id}}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}$

3. If there exists a $(z_m)_{m\in\omega} \in \mathbf{Q}_M(D)$ such that for all $n \in \omega$, $x_n = \sigma^D_n\ (z_m)_{m\in\omega}$, then

$$
\begin{aligned}
& (h^e \circ h^p)\ (x_n)_{n\in\omega} \\
= & \langle\ \textit{Assumption}\ \rangle \\
& (h^e \circ h^p)\ (\sigma^D_n\ (z_m)_{m\in\omega})_{n\in\omega} \\
= & \langle\ \textit{Composition}\ \rangle \\
& h^e\ (h^p\ (\sigma^D_n\ (z_m)_{m\in\omega})_{n\in\omega}) \\
= & \langle\ \textit{Definition of } h^p\ \textit{(3.4.5)}\ \rangle \\
& h^e\ \left(\textbf{\textit{inr}}\ \left(\bigsqcup_{n\in\omega} M(\mu^e_n)\ z_n\right)\right) \\
= & \langle\ \textit{Definition of } h^e\ \textit{(3.4.2)}\ \rangle \\
& \left(\theta^D_n\ \left(\textbf{\textit{inr}}\ \left(\bigsqcup_{n\in\omega} M(\mu^e_n)\ z_n\right)\right)\right)_{n\in\omega} \\
= & \langle\ \textit{Lemma 3.4.4}\ \rangle \\
& \left(\sigma^D_n\ \left(M(\mu^p_m)\ \left(\bigsqcup_{n'\in\omega} M(\mu^e_{n'})\ z_{n'}\right)\right)_{m\in\omega}\right)_{n\in\omega} \\
= & \langle\ M(\mu^p_m)\ \textit{is continuous}\ \rangle \\
& \left(\sigma^D_n\ \left(\bigsqcup_{n'\in\omega} M(\mu^p_m)\ (M(\mu^e_{n'})\ z_{n'})\right)_{m\in\omega}\right)_{n\in\omega} \\
= & \langle\ \textit{Composition}\ \rangle \\
& \left(\sigma^D_n\ \left(\bigsqcup_{n'\in\omega} (M(\mu^p_m) \circ M(\mu^e_{n'}))\ z_{n'}\right)_{m\in\omega}\right)_{n\in\omega} \\
= & \langle\ M\ \textit{is functor}\ \rangle \\
& \left(\sigma^D_n\ \left(\bigsqcup_{n'\in\omega} M(\mu^p_m \circ \mu^e_{n'})\ z_{n'}\right)_{m\in\omega}\right)_{n\in\omega} \\
= & \langle\ \textit{Lemma 3.1.11}\ \rangle \\
& \left(\sigma^D_n\ \left(\bigsqcup_{n'\in\omega} M(f^D_{m,n'})\ z_{n'}\right)_{m\in\omega}\right)_{n\in\omega} \\
= & \langle\ \textit{Lemma 3.4.7}\ \rangle \\
& (\sigma^D_n\ (z_m)_{m\in\omega})_{n\in\omega} \\
= & \langle\ \textit{Assumption}\ \rangle \\
& (x_n)_{n\in\omega} \\
= & \langle\ \textit{Identity}\ \rangle \\
& \textbf{\textit{id}}_{\mathbf{R}_M(D)}\ (x_n)_{n\in\omega}
\end{aligned}
$$

□

3.5 Additional operations

In this section we define two functions, ***step*** and ***run***, which convert a non interleaved computation of type $M(A)$ to an interleaved computation of type $\mathsf{R}(M)(A)$ and vice-versa. The names of these functions indicate their behavior. The first converts a whole computation to a single atomic *step* in an interleaved computation. The second *runs* the whole sequence of atomic steps of an interleaved computation without allowing other computations to intervene. Both functions are very helpful in specifying the semantics of concurrent programming languages.

In the rest of this section, we assume that $\langle M, \eta, \mu\rangle$ is a monad and that D is a domain.

Definition 3.5.1 $\mathit{step}_D : M(D) \to \mathsf{R}(M)(D)$ *is the continuous function defined by:*

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

Definition 3.5.2 $\mathit{run}_D : \mathsf{R}(M)(D) \to M(D)$ *is the continuous function defined by:*

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

The following theorem states that the composition of ***run*** and ***step***, in this order, yields identity. The reverse composition does not yield identity, since it forces an interleaved computation to be executed in one atomic step (it will be used in Section 4.4 for this exact purpose).

Theorem 3.5.1 $\mathit{run}_D \circ \mathit{step}_D = \mathit{id}_{M(D)}$
Proof

$\mathit{run}_D \circ \mathit{step}_D$
$=$ ⟨ *Unfolding* ***fix*** *in the definition of* run_D *(3.5.2)* ⟩
$[\,\eta_D, \mu_D \circ M(\mathit{run}_D)\,] \circ h^p \circ \mathit{step}_D$
$=$ ⟨ *Definition of* step_D *(3.5.1)* ⟩
$[\,\eta_D, \mu_D \circ M(\mathit{run}_D)\,] \circ h^p \circ h^e \circ \mathit{inr} \circ M(h^e \circ \mathit{inl})$
$=$ ⟨ *Theorem 3.4.1* ⟩
$[\,\eta_D, \mu_D \circ M(\mathit{run}_D)\,] \circ \mathit{id}_{\mathbf{F}_{M,D}(\mathbf{R}_{M(D)})} \circ \mathit{inr} \circ M(h^e \circ \mathit{inl})$
$=$ ⟨ *Composition with identity* ⟩
$[\,\eta_D, \mu_D \circ M(\mathit{run}_D)\,] \circ \mathit{inr} \circ M(h^e \circ \mathit{inl})$
$=$ ⟨ *Theorem 2.3.5* ⟩
$\mu_D \circ M(\mathit{run}_D) \circ M(h^e \circ \mathit{inl})$
$=$ ⟨ M *is a functor* ⟩
$\mu_D \circ M(\mathit{run}_D \circ h^e \circ \mathit{inl})$
$=$ ⟨ *Unfolding* ***fix*** *in the definition of* run_D *(3.5.2)* ⟩
$\mu_D \circ M([\,\eta_D, \mu_D \circ M(\mathit{run}_D)\,] \circ h^p \circ h^e \circ \mathit{inl})$
$=$ ⟨ *Theorem 3.4.1* ⟩
$\mu_D \circ M([\,\eta_D, \mu_D \circ M(\mathit{run}_D)\,] \circ \mathit{id}_{\mathbf{F}_{M,D}(\mathbf{R}_{M(D)})} \circ \mathit{inl})$
$=$ ⟨ *Composition with identity* ⟩
$\mu_D \circ M([\,\eta_D, \mu_D \circ M(\mathit{run}_D)\,] \circ \mathit{inl})$
$=$ ⟨ *Theorem 2.3.5* ⟩
$\mu_D \circ M(\eta_D)$
$=$ ⟨ M *is a monad, 2nd Monad Law* ⟩
$\mathit{id}_{M(D)}$ □

Function ***prom***, which lifts a computation of type $\mathsf{R}(M)(D)$ to a computation of type $M(\mathsf{R}(M)(D))$, is useful in the rest of this section where we establish that $\mathsf{R}(M)(D)$ can be defined as a multi-monad and a strong monad. These two properties of $\mathsf{R}(M)(D)$ will also be used in Section 4.4.

Definition 3.5.3 $\mathit{prom}_D : \mathsf{R}(M)(D) \to M(\mathsf{R}(M)(D))$ *is the continuous function defined by:*

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

Let us now assume that M is a multi-monad and that $\|_{\mathsf{M}}$ is a *non-deterministic option* operator for computations represented by monad M. It is easy to extend this behavior to the monad $\mathsf{R}(M)$.

Definition 3.5.4 *Let M be a multi-monad. Let D be a domain. We define the binary operation $\|_{\mathsf{R(M)}} : \mathsf{R}(M)(D) \times \mathsf{R}(M)(D) \to \mathsf{R}(M)(D)$ by:*

$$x \parallel_{\mathsf{R(M)}} y \;=\; h^e\ (\textbf{\textit{inr}}\ (\textbf{\textit{prom}}\ x \parallel_{\mathsf{M}} \textbf{\textit{prom}}\ y))$$

Monad $\mathsf{R}(M)$ with $\|_{\mathsf{R(M)}}$ is a multi-monad.

Furthermore, we can introduce a way to create a new interleaved computation of type $\mathsf{R}(M)(A \times B)$ given two existing computations of types $\mathsf{R}(M)(A)$ and $\mathsf{R}(M)(B)$. Here we prefer to use monads M and $\mathsf{R}(M)$ in the functional way. If one of the two computations does not require the execution of any atomic step, i.e. if one of the two computations has already been completed, then the other computation is executed and the two results are combined. Otherwise, if both computations require at least one atomic step, we choose non-deterministically which computation will start executing.

Definition 3.5.5 *Let M be a multi-monad. Let A and B be domains. We define the binary operation $\bowtie_{\mathsf{R(M)}} : \mathsf{R}(M)(A) \times \mathsf{R}(M)(B) \to \mathsf{R}(M)(A \times B)$ by:*

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

Monad $\mathsf{R}(M)$ with $\bowtie_{\mathsf{R(M)}}$ is a strong monad.

Chapter 4

An implementation of RMT

In this section we present our implementation of the *Resumption Monad Transformer (RMT)* in *JavaScript*. Firstly, in Section 4.1 we will define the basic monads needed to later define our semantics of a simple imperative JavaScript-like language. In Section 4.2 we define the *State Monad* and the *State Monad Transformer* that we will need to introduce side-effects in our simple imperative language. Then, in Section 4.3 we define in *JavaScript* the *RMT* described in Chapter 3 alongside with its additional operations from Section 3.5, which will be necessary for the running programs in our semantics. Finally, we use our implementation of the *RMT* to define the semantics of our concurrent language.

4.1 Monads

4.1.1 Identity monad

A reasonable choice for stateless computations and left-to-right evaluation order in our semantics is the *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 List monad

To enable ambiguity in our expressions' evaluation we cannot use the Identity monad. A monad supporting multiple results must me used instead. The *power-domain* monad, often mentioned as the *List monad* in functional programming languages, is the obvious choice here. The *List* monad must be an instance of the *multi-monad* class and support a *"union"* of different computation results, which is achieved with the `multi` method defined below.

```
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 Set monad

As an extend to the List monad, we define the *Set monad*. The Set monad basically has the same interface with the List monad but, instead of concatenating the results, it keeps track of the same ones alongside with a counter of their appearances. The implementation depends heavily on the *State* we used, which we will define in Section 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 Monads and states

4.2.1 States

The notion of state is a very important one in the study of impure languages. A state is an element of a type which supports two main operations, `load` and `store`, for retrieving and updating the contents of a variable in memory. A distinguished element of this type is

the `initial` state, typically a state with all variables uninitialized. We first define a simple stack-based `State` and then we present two different versions of it that proved to have better performances in our tests.

```
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 State monad

The class of monads that are aware of the state is obviously necessary for our implementations. Class `StateM` supports two operations as an interface between computations and the state. Method `modify` updates the state by applying its argument function and returns a computation of the old state. Method `get` simply returns a computation of the current state.

```
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 State monad transformer

It is also useful to define a monad transformer implementing the direct semantics approach. For every state type `s` given as a parameter, the *State monad transformer* can be defined as follows. Parameter `M` specifies the monad representing the stateless computations.

```
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

Monads constructed using `StateT` are aware of the state and therefore monad `StateT(M)` is an instance of *State monad* for a state type s. We notice here that if in the `StateT(M)` we use the *Identity monad*, we get the simple `StateM` defined in Subsection 4.2.2.

4.3 RMT in JavaScript

In this section we define the *resumption monad transformer* which implements the tree-like branching semantics we defined in Chapter 3. Resumptions are constructs which split a computation in a single atomic step (to be executed first) and a resumed part, which corresponds to the rest of the computation. A resumption therefore can be either *Computed* or has to be *Resumed*.

Our original implementation is the following one. This implementation is an exact translation of the definitions we gave in Chapter 3 in JavaScript. But due to too many object instantiations and too much information kept in each object, we found that our optimized version, presented later in this section, works a lot better. What is more, the original version actually takes a monad `M` as an argument, but as this is always the *State Monad* in our tests, we incorporate this in our optimized version to exploit better method implementations.

```
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 Resumption monad transformer

We define here the superclass that implements the optimized version of our RMT. This class will be extended from the classes of the *Computed* RMTs and the *Resumed* parts of them. As we defined in subsection 3.5 RMTs have two additional operations in order to execute our semantics. The `stepR` method produces a computation of just one atomic step, therefore it is defined here as a static method. The `runR` and `bind` methods are different for each class, therefore they are later defined in each of them. We also implement the `get` and `modify` methods of the `StateM`, which here lift the RMT computations into state-aware ones, providing this way an interface between resumptions and the State monad. Finally, resumptions must be an instance of multi-monads, therefore we define the `multi` method which implements the necessary $\|_{R(M)}$ operation.

```
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 Case of *Computed*

The implementation of the *Computed* resumptions is presented here. A computed resumption stores its computed value and the `bind` method applies the argument function on that value. The `runR` method has no atomic steps to evaluate in a computed resumption therefore it just returns a new state monad with the computed value of the current resumption.

```
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 Case of *Resume*

The implementation of the *Resumed* resumptions is presented here. A resumed resumption stores a computation within its monad parameter and the `bind` method applies the argument function on that computation. The `runR` method fully evaluates the resumption by performing all the atomic steps of the given resumption.

```
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 Resumptions as strong monads

Since the aforementioned implementation of resumptions must also be an instance of strong-monads, we have to equip them with the $\bowtie_{R(M)}$ operation mentioned in subsection 3.5. The following implementation is again an optimized version used for our benchmarks that chooses randomly a path of all the possible 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

For the baseline executions of our benchmarks that use just the State Monad, we followed a simple left-to-right evaluation enforced by the simple following code.

```
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 A modular semantics of concurrency

4.4.1 The example language

Consider the simple sequential imperative language whose abstract syntax is given below.

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

It features an empty statement, assignment, sequential composition of statements, a structure for conditional and one more for *while* loops. The symbol $x \in \mathbf{Var}$ represents a variable.

The language of expressions e is the following.

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

The language of expressions e consists of all the numbers and variables, the common operations like addition, subtraction, etc., increment and decrement operations and all basic logical expressions like $<$, $<=$, $==$, etc.. Here, we omitted defining some of them since their semantics will be defined the same way in our implementation. The symbol $n \in \mathbf{Num}$ represents a number, which can either be an integer or a real number. The symbol $x \in \mathbf{Var}$ again represents a variable and is here used to access the value of a variable in our language.

4.4.2 The example language semantics

We define the denotational semantics of this language, assuming that the values of expressions are elements of the semantic domain $\mathbf{V}$. The *program state*, mapping variables to their current values, is an element of the domain $\mathbf{S} = \mathbf{Var} \to \mathbf{V}$.

As a provision for what will follow, we define a monad transformer D implementing the *direct semantics* approach.[1] If M is a monad, we define the monad $\mathsf{D}(M)$ as:

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

State computations created by the direct semantics monad transformer are functions (elements of $\mathsf{D}(M)(D)$) that take the initial program state (an element of $\mathbf{S}$) and return a stateless computation that yields the computed value (an element of D) and the final program state (an element of $\mathbf{S}$). The implementations of $\textit{unit}_{\mathsf{D}(\mathsf{M})}$ and $*_{\mathsf{D}(\mathsf{M})}$ carry out the propagation of the program state.

We also define an operation for the assignment of values to variables.[2]

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

By taking the *identity monad* Id as the argument of D, we obtain the monad M that models our simple notion of computation (ordinary direct semantics).

[1] This is the state monad transformer, as defined in [Lian95, Lian98].

[2] If A and B are domains, $f : A \to B$, $a \in A$ and $b \in B$, we use the notation $f\{a \mapsto b\}$ to denote a function $f' : A \to B$ such that $f'(a) = b$ and, for all $x \neq a$, $f'(x) = f(x)$.

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

The meaning of a statement s is a computation $[\![s]\!]$ of type $\mathsf{M}(\mathbf{U})$. Non-termination is represented by the bottom element. We also assume that the meaning of an expression e is a computation $[\![e]\!]$ of type $\mathsf{M}(\mathbf{V})$. The semantic function for the statements of our simple imperative language is completely straightforward.

$$
\begin{aligned}
&[\![\textbf{skip}]\!] \;=\; \textit{unit}\ \textit{u}\\
&[\![x := e]\!] \;=\; [\![e]\!] \ast (\textit{store}\ x)\\
&[\![s_1 \,;\, s_2]\!] \;=\; [\![s_1]\!] \ast (\lambda u.\, [\![s_2]\!])\\
&[\![\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2]\!] \;=\; [\![e]\!] \ast (\lambda b.\ \text{if } b \text{ then } [\![s_1]\!] \text{ else } [\![s_2]\!])\\
&[\![\textbf{while}\ e\ \textbf{do}\ s]\!] \;=\; \textit{fix}\ (\lambda g.\, [\![e]\!] \ast (\lambda b.\\
&\qquad\qquad\qquad\qquad \text{if } b \text{ then } [\![s]\!] \ast (\lambda u.\ g) \text{ else } \textit{unit}\ \textit{u}))
\end{aligned}
$$

The semantics function for the expressions is also pretty straightforward.

$$
\begin{aligned}
&[\![n]\!] \;=\; \textit{unit}\ n\\
&[\![x]\!] \;=\; \textit{load}\ x\\
&[\![e_1 + e_2]\!] \;=\; [\![e_1]\!] \ast (\lambda u_1.\, [\![e_2]\!] \ast (\lambda u_2.\, \textit{unit}\ u_1 + u_2))\\
&[\![e_1 \ast e_2]\!] \;=\; [\![e_1]\!] \ast (\lambda u_1.\, [\![e_2]\!] \ast (\lambda u_2.\, \textit{unit}\ u_1 \ast u_2))\\
&\ldots\\
&[\![x{+}{+}]\!] \;=\; [\![x]\!] \ast (\lambda u.\, \textit{store}\ x\ (u+1))\\
&\ldots\\
&[\![e_1 < e_2]\!] \;=\; [\![e_1]\!] \ast (\lambda u_1.\, [\![e_2]\!] \ast (\lambda u_2.\, \textit{unit}\ u_1 < u_2))\\
&[\![e_1 == e_2]\!] \;=\; [\![e_1]\!] \ast (\lambda u_1.\, [\![e_2]\!] \ast (\lambda u_2.\, \textit{unit}\ u_1 == u_2))\\
&\ldots
\end{aligned}
$$

In our implementation in JavaScript, we define a class for each token of our abstract syntaxes for statements s and expressions e. Those classes have a constructor method for keeping their token parameters (e.g. in the statement $x := e$ we keep the string of the variable x and the expression e) and the `sem` method which implements each token's semantics.

For example, the semantics of the *if* statement are implemented like this:

```
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));
    }
}
```

The implementation of the *while* statement is the following:

```
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()));
    }
}
```

The *while* statement uses the *fixed-point combinator*, which we define like the following code in our implementation.

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

4.4.3 Concurrency in the example language

Let us now introduce non-determinism and concurrency in our language, by extending it with three new constructs.

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

Operator $\oplus$ executes exactly one of the statements that are given as its operands. The selection is non-deterministic. On the other hand, operator $\parallel$ executes both statements that are given as its operands in an interleaved way. Finally, the construct $\langle s \rangle$ executes the statement s in a single atomic step, with no interleaving permitted during its execution.

Before we proceed with the semantics of our extended language, we have to modify the definition of M. By choosing the power-domain monad P as the argument of D, we obtain a multi-monad that can support non-determinism.

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

The option operator $\|_{\mathsf{M}}$ is defined as:

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

where $\uplus^{\natural}$ is the union operation on power-domains.

In the semantics of the extended language, we use the monad $\mathsf{R}(\mathsf{M})$ to model interleaved computations. According to Definition 3.5.4, $\mathsf{R}(\mathsf{M})$ is a multi-monad equipped with a non-deterministic option operator $\|_{\mathsf{R}(\mathsf{M})}$. Also, according to Definition 3.5.5, $\mathsf{R}(\mathsf{M})$ is a strong monad and operator $\bowtie_{\mathsf{R}(\mathsf{M})}$ can be used to model the interleaving of computations. Furthermore, the ***store*** operation can easily be lifted onto the new domain of computations.

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

The equations defining the meaning of existing language constructs do not require any changes, except for the implicit change that the meanings of statements and expressions are now elements of the semantic domains $\mathsf{R}(\mathsf{M})(\mathbf{U})$ and $\mathsf{R}(\mathsf{M})(\mathbf{V})$ respectively. On the other hand, the semantics of the additional constructs can be easily expressed in terms of $\mathsf{R}(\mathsf{M})$ operations.

$$\begin{array}{l} [\![s_1 \oplus s_2]\!] \;=\; [\![s_1]\!] \;\|_{\mathsf{R}(\mathsf{M})}\; [\![s_2]\!] \\ [\![s_1 \parallel s_2]\!] \;=\; ([\![s_1]\!] \;\bowtie_{\mathsf{R}(\mathsf{M})}\; [\![s_2]\!]) \;*\; (\lambda p.\; \textit{unit}\;\; \mathbf{u}) \\ [\![\langle s \rangle]\!] \;=\; \textit{step}\;(\textit{run}\;[\![s]\!]) \end{array}$$

The above operators $\oplus$, $\parallel$ and the construct $\langle s \rangle$ of our extended example language are simply implemented as follows:

```
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()); }
}
```

We also implemented an alternative of the $\|$ operator, which have the same semantics but take an array of statements instead of two. The use of this operator will be more clear in the benchmark of 5.1.6 were we create more than two threads.

```
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));
    }
}
```

This operator also uses a modified version of the `tensor` function of RMTs that executes the given array of threads in a cyclic manner. Its implementation is the following:

```
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}}))
    }
}
```

The aforementioned denotational semantics of our example language are used to execute our benchmarks presented in Section 5.1. The monad transformer D implementing the *direct semantics* approach, which is described at the start of this section, with the *identity monad* Id as its argument, is used to evaluate our baseline *sequential* programs. Then, by choosing the power-domain monad P as the argument of D, we obtain a multi-monad that can support non-determinism, and passing that as M to the monad R(M), we can model interleaved computations and use that for our *concurrent* benchmarks.

Chapter 5

Performance results

5.1 Benchmarks

In this section we describe all the algorithms that were used to measure the performance of our RMT implementation and the baseline implementations with Promises. All benchmarks were written in *JavaScript* and were executed in the *Node.js* run-time environment [Node18], using version *10.3.0*. The benchmarks are basically simple and common algorithms of various time complexities e.g. $O(n)$, $O(n\ logn)$, $O(n^2)$, $O(2^n)$. We provide here the pseudocode for both the sequential and the concurrent versions of our benchmark implementations.

In order to evaluate the run-time performance of our RMT implementation for the purpose of this thesis, the algorithms described below were written using our semantics. The *Sequential* ones were executed with our semantics, passing them as a parameter the State Monad Transformer lifting the Identity Monad, and for the *Concurrent* ones, passing them the RMT lifting the same transformer as their sequential counterparts.

For the benchmark baselines, we used the same algorithms implemented in plain JavaScript for the *Sequential* versions and transformed the same code using JavaScript's *Promises* for the *Concurrent* ones. The core implementation though remains the same in both RMT and Promises versions.

5.1.1 Sieve of Eratosthenes

Our first benchmark is the classic algorithm for the sieve of Eratosthenes. It has a time complexity of $O(n\ loglogn)$ and the basic loop can be broken into two (or more) separate threads for the concurrent version. The common sequential algorithm is presented in Algorithm 1. In the concurrent algorithm - Algorithm 2 - two threads are created that have the same body as the while loop in 1, but start on different numbers and have a step of 2 in each iteration. The problem here is reading or writing the global array *primes*, where the results of whether the relative index is prime or not is kept. A lock must be present in order to avoid race conditions and secure that the same operations as the sequential one will be executed. In our semantics, this can be simply done using our *unit* operator, while in the promises implementation we have to keep the whole operation of reading or writing in one atomic step inside a separate promise in order to achieve the same results.

Algorithm 1 Sequential sieve of Eratosthenes

```
 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
```

Algorithm 2 Concurrent sieve of Eratosthenes

```
 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 Pi approximation

Our second benchmark is a simple algorithm to approximate Pi. It has a time complexity of $O(n)$ and the basic loop can be broken into two separate threads for the concurrent version. The sequential algorithm is presented in Algorithm 3. In the concurrent algorithm - Algorithm 4 - two threads are created that have the same body as the while loop in 3, but start on different numbers (1 and 2) and have a step of 2 in each iteration. That way we can avoid in the concurrent version the check if i is even or odd. Each thread runs either for the odds or the evens separately. The problem here is the same as the sieve above 2; reading or writing the global variable pi, where the approximation of pi is kept. In our semantics, we simply use again our *unit* operator, while in the promises implementation we have to keep the whole operation of reading or writing in one atomic step inside a separate promise in order to achieve the same results.

Algorithm 3 Sequential approximation of 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
```

Algorithm 4 Concurrent approximation of 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 Primality test

Our third benchmark is a classic primality test for a given integer. It has a time complexity of $O(\sqrt{n})$ and again the basic loop can be broken into two separate threads for the concurrent version. This test was of course executed for very large primes. The sequential algorithm is presented in Algorithm 5. In the concurrent algorithm - Algorithm 6 - two threads are created that have the same body as the basic while loop in 5, but start on different numbers and each thread does half of the check avoiding some operations.

Algorithm 5 Sequential primality test

```
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
```

Algorithm 6 Concurrent primality test

```
 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 Insertion sort

Our fourth benchmark is insertion sort. It basically has a time complexity of $O(n^2)$, specifically in our case, where the tests consist of arrays reversely sorted. The sequential algorithm is presented in Algorithm 7. For the concurrent versions we run the same sequential algorithm, but without any threads actually interleaving. This way we can calculate the overhead that our implementation and promises have over their sequential counterparts on executing the same number of computations.

Algorithm 7 Insertion sort

```
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 Array reduction

Our fifth benchmark is a simple reduction of a given array, in order to get the sum of the array elements. It has, of course, a time complexity of $O(n)$ and again the basic loop can be broken into two separate threads for the concurrent version. This test was of course executed for large arrays. The sequential algorithm is presented in Algorithm 8. In the concurrent algorithm - Algorithm 9 - two threads are created that have the same while loop as in 8, but each thread tackles a different half of the array. The result is stored in a global variable x, so again we have to "lock" x to avoid data races.

Algorithm 8 Sequential reduce of array

```
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
```

Algorithm 9 Concurrent reduce of array

```
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 Matrix-vector multiplication

Our sixth benchmark is a matrix-vector multiplication of a given matrix and a given vector, which has a time complexity of $O(n^2)$. The sequential algorithm is presented in Algorithm 10. In the concurrent algorithm - Algorithm 11 - each thread performs a vector-vector multiplication. Therefore, in this benchmark we create *dim* threads, where *dim* is the number of rows in the input matrix, and store them in a vector *threads*. As for the final result, each thread keeps its result stored in a different position of the global vector y, so no "locks" are needed here. But we must define a way for that many threads in the vector *threads* to be executed. For the promises version we can use $Promise.all(threads)$ but in the RMT version we have to define a similar operator. **runMany** does exactly that. It executes the threads basic computations in a cyclical manner as $Promise.all()$ is expected to do, and has similar semantics as **run** but for an array of threads, as shown in the comment of line *15* in Algorithm 11.

Algorithm 10 Sequential matrix-vector multiplication

```
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
```

Algorithm 11 Concurrent matrix-vector multiplication

```
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 Combinations

Our last benchmark is finding all different combinations of size m out of n numbers (range 1 to n for our tests). It has a time complexity of $O(2^n)$ since our algorithm simply enumerates all possible combinations. The sequential algorithm is presented in Algorithm 12. As we did in Algorithm 7, for the concurrent versions we run the same sequential algorithm but without any threads actually interleaving, in order to just calculate the overhead of our implementations.

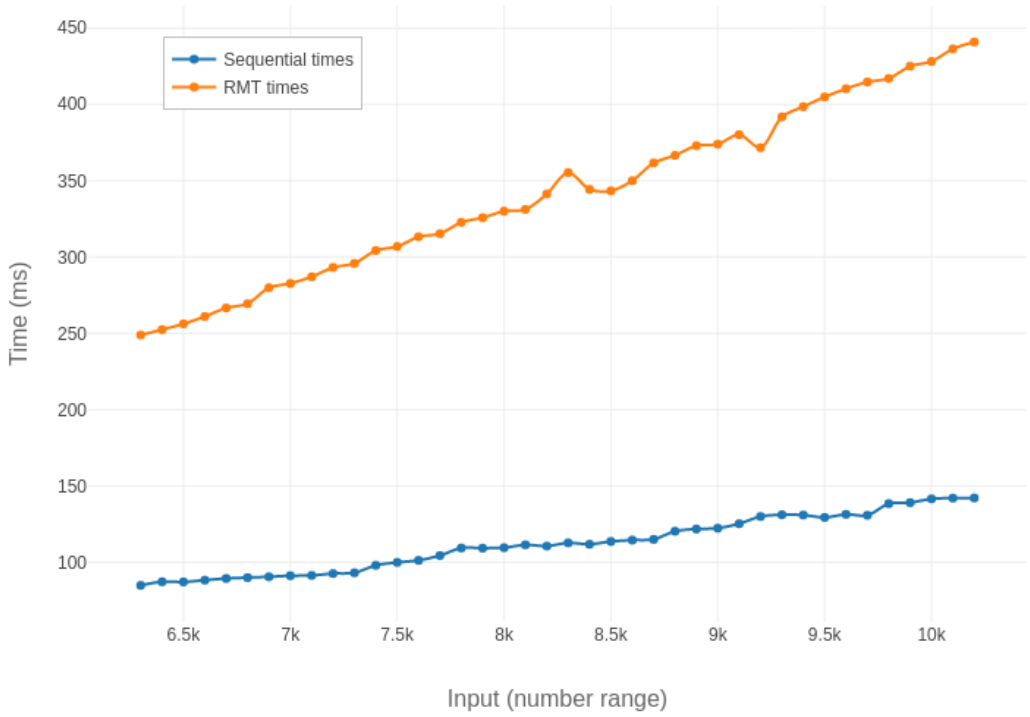
Algorithm 12 Combinations (m comb n)

```
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
```

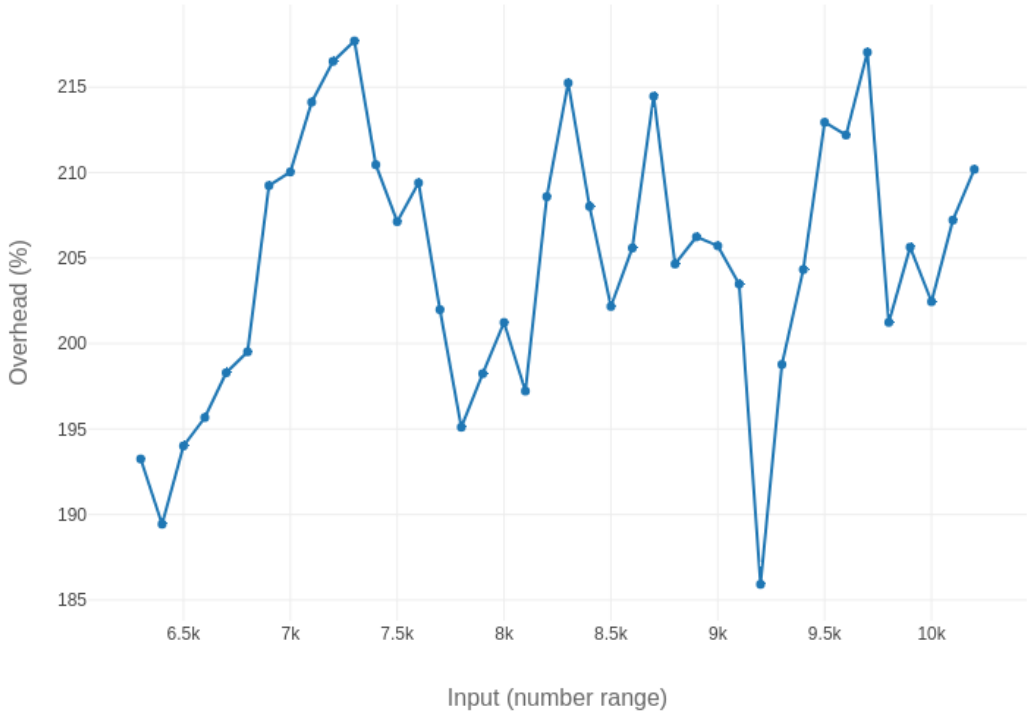
5.2 Results

5.2.1 Performance test results

Each benchmark described in Section 5.1 was executed 100 times and their average run-times are presented here. Additionally, we present the overhead that our RMT programs have over their simple sequential counterparts in separate figure for each benchmark.

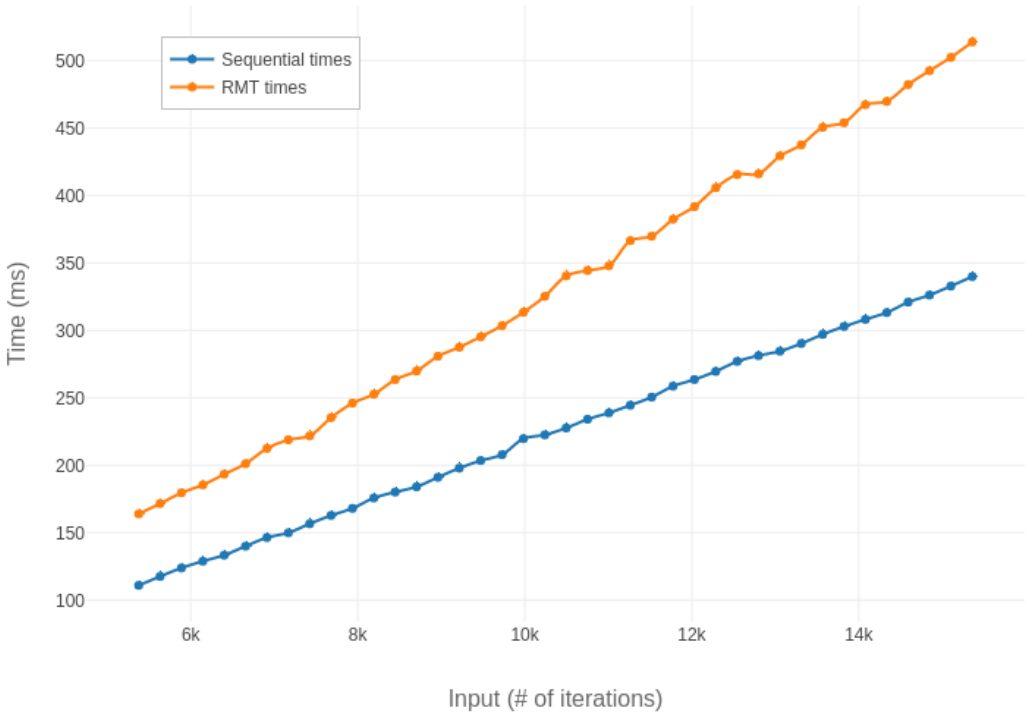


(a) Run-times

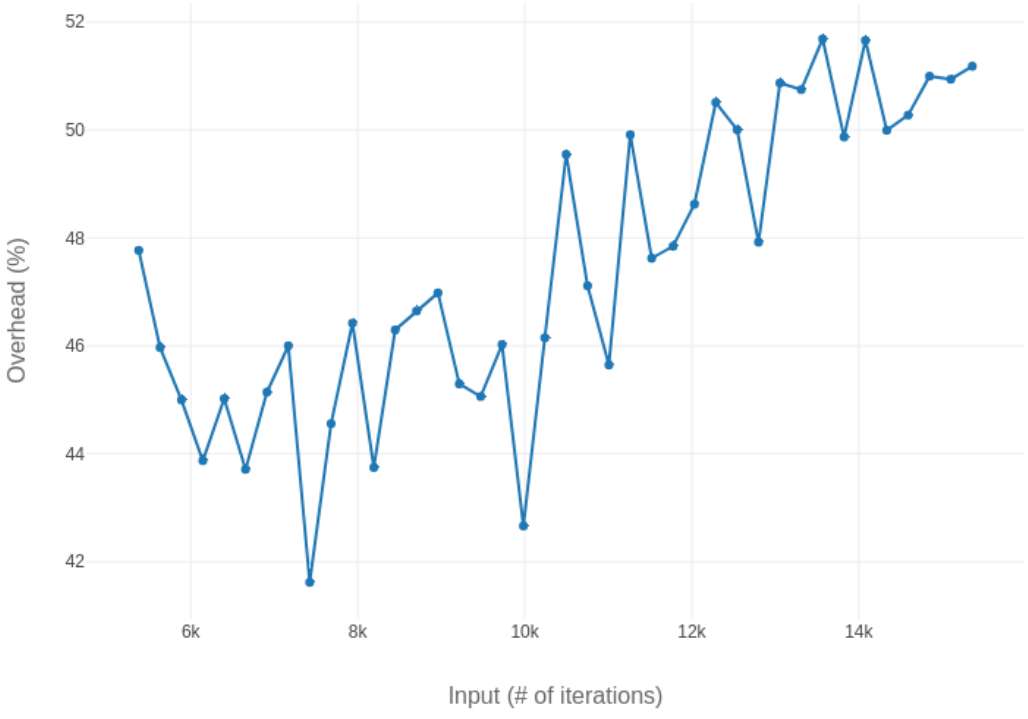


(b) Overhead

Figure 5.1: Sieve of Eratosthenes

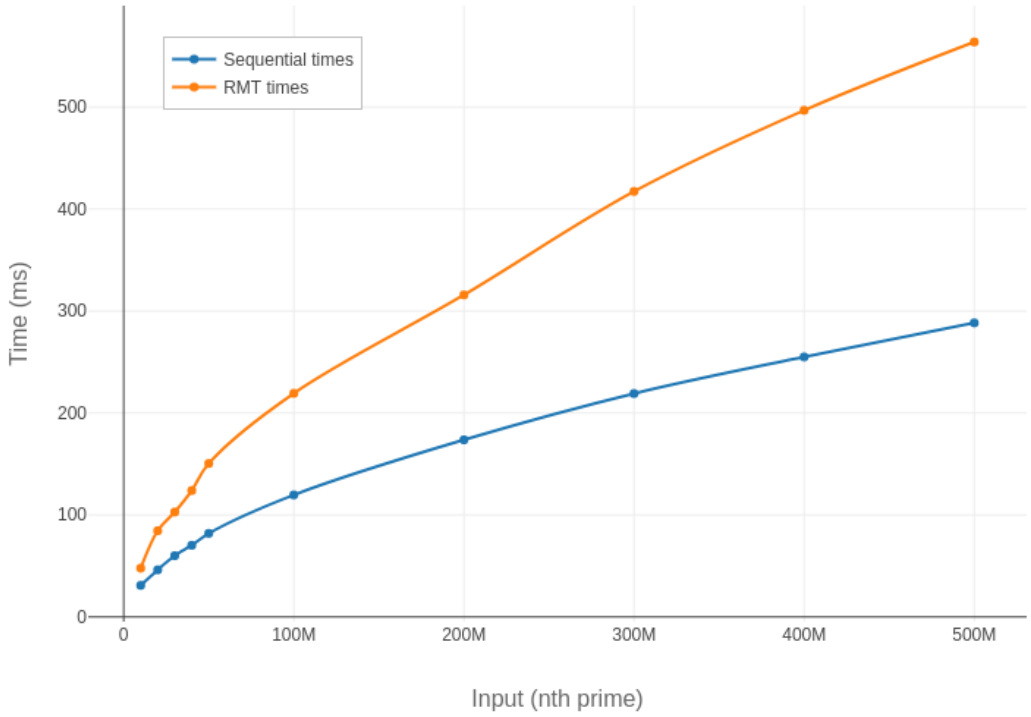


(a) Run-times

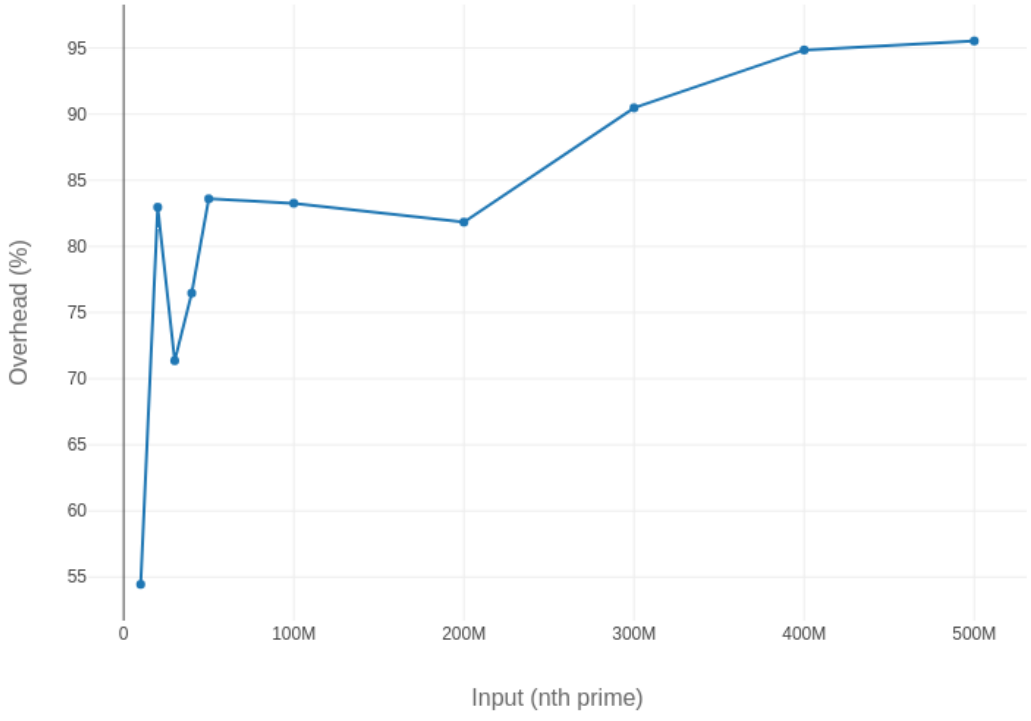


(b) Overhead

Figure 5.2: Approximation of Pi

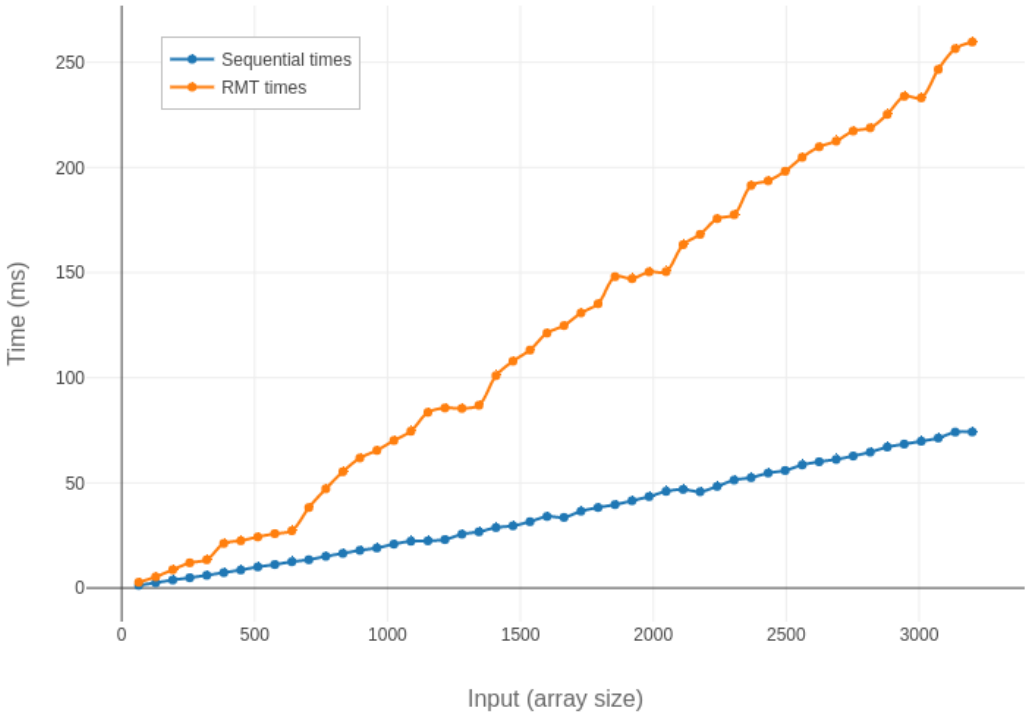


(a) Run-times

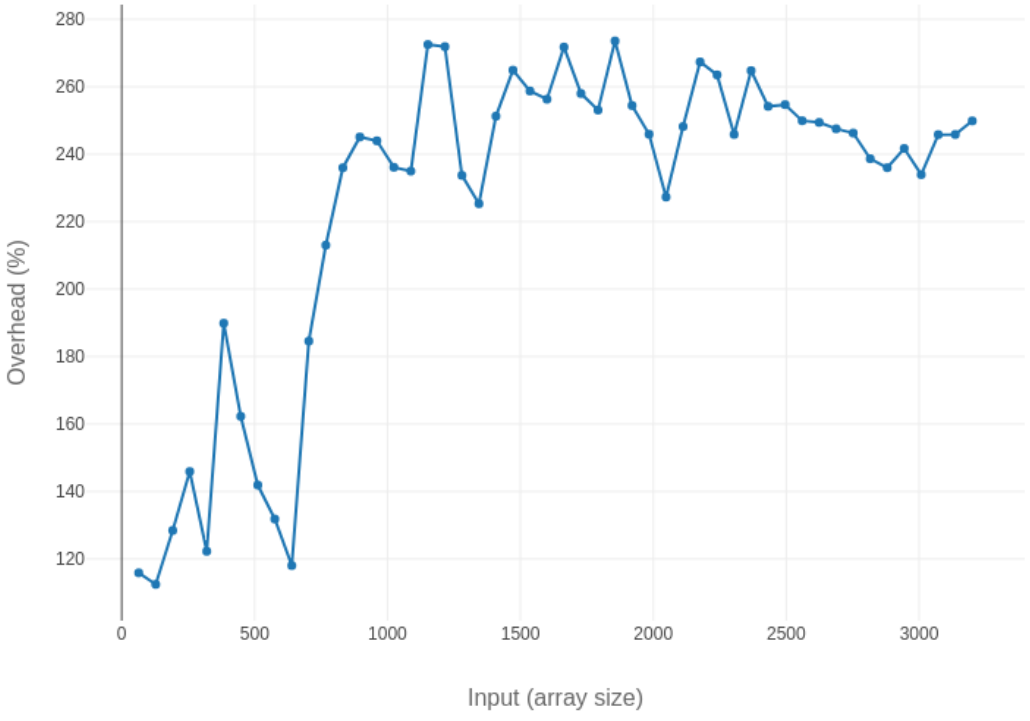


(b) Overhead

Figure 5.3: Primality test

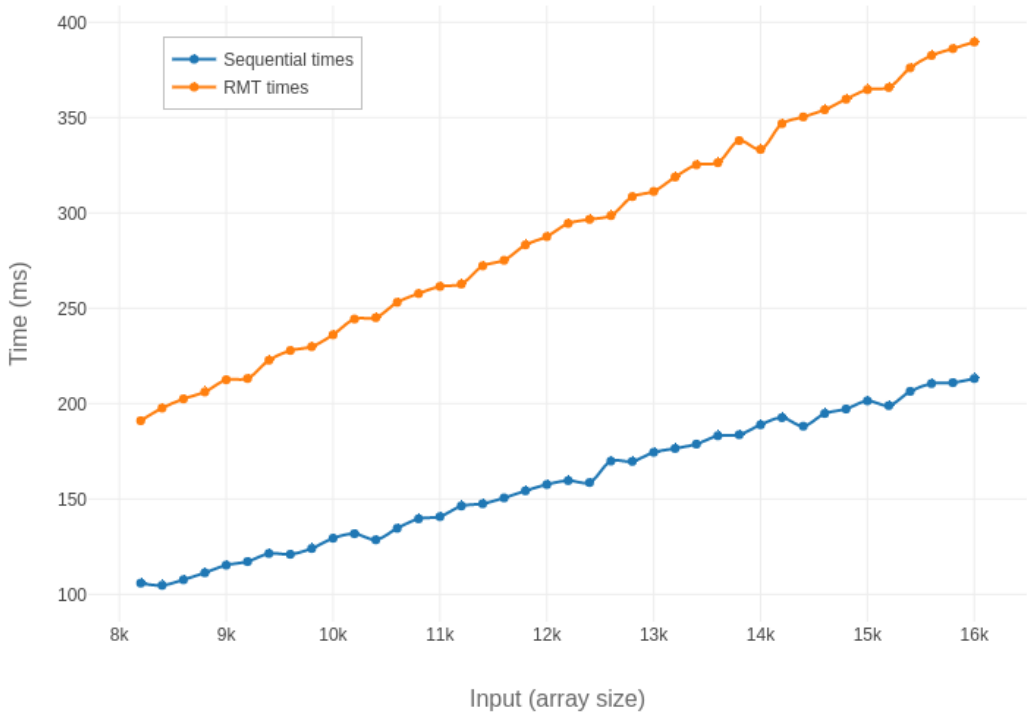


(a) Run-times

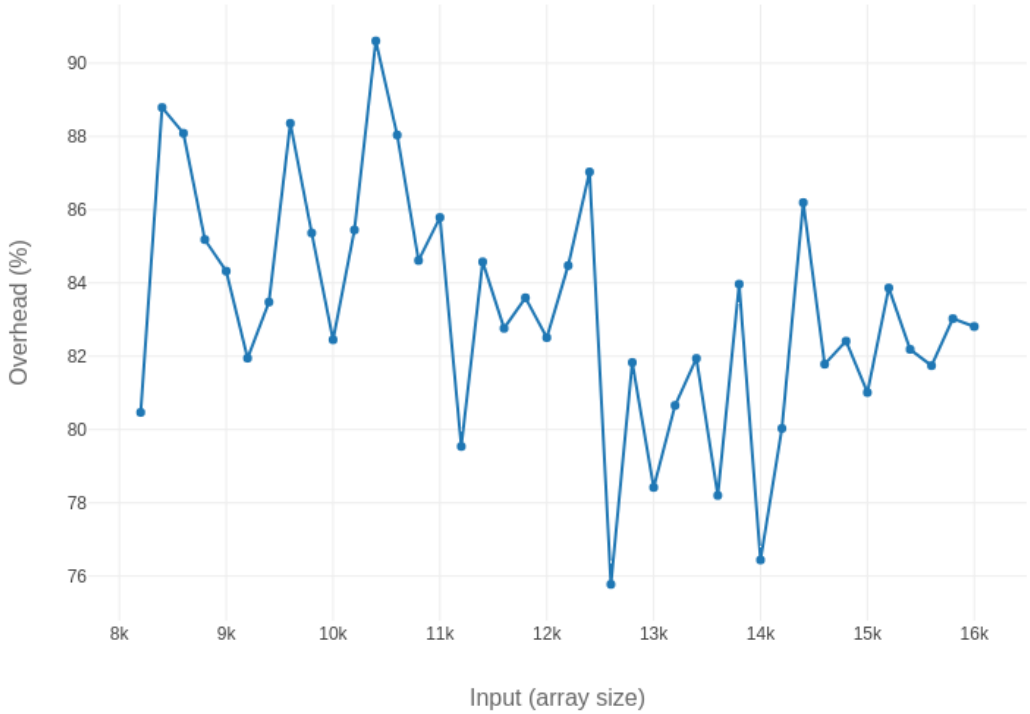


(b) Overhead

Figure 5.4: Insertion sort

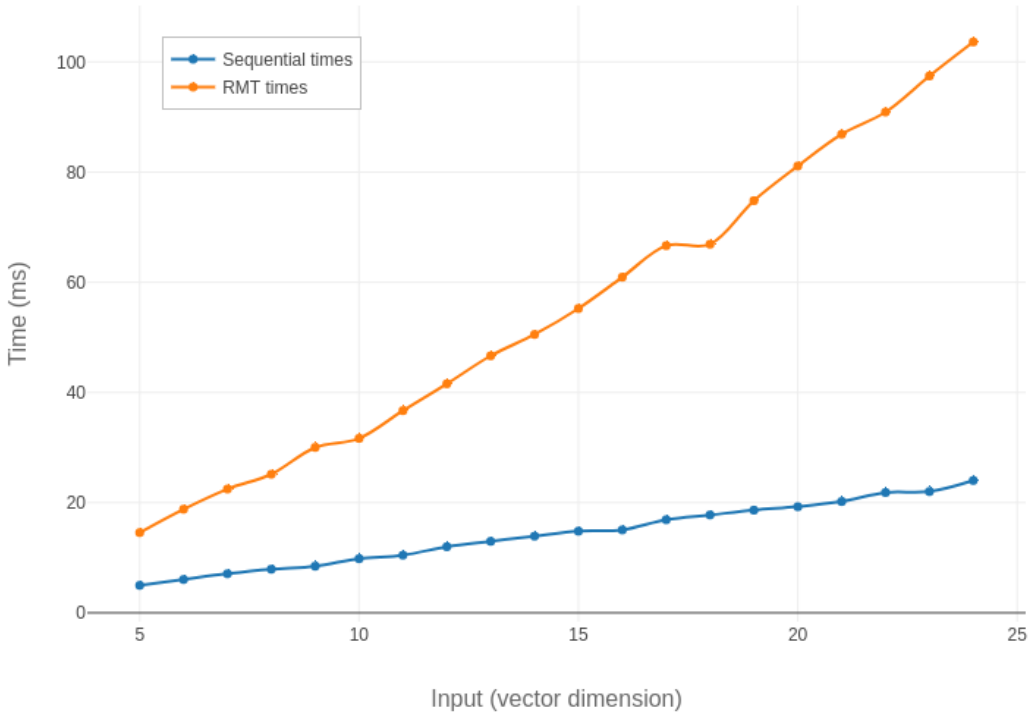


(a) Run-times

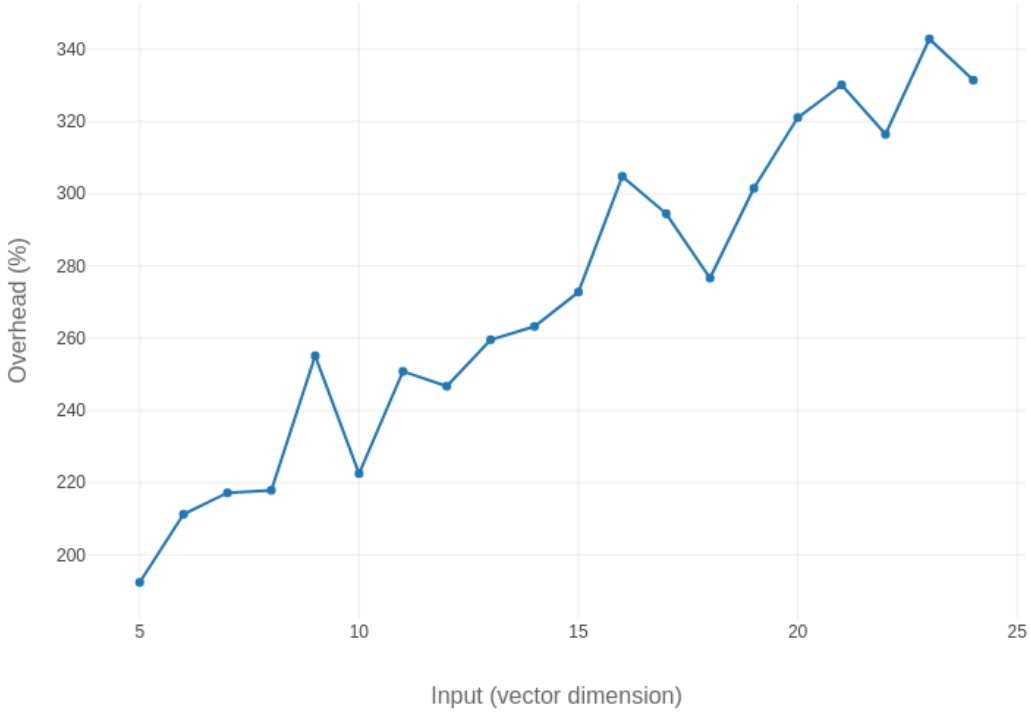


(b) Overhead

Figure 5.5: Reduce of array

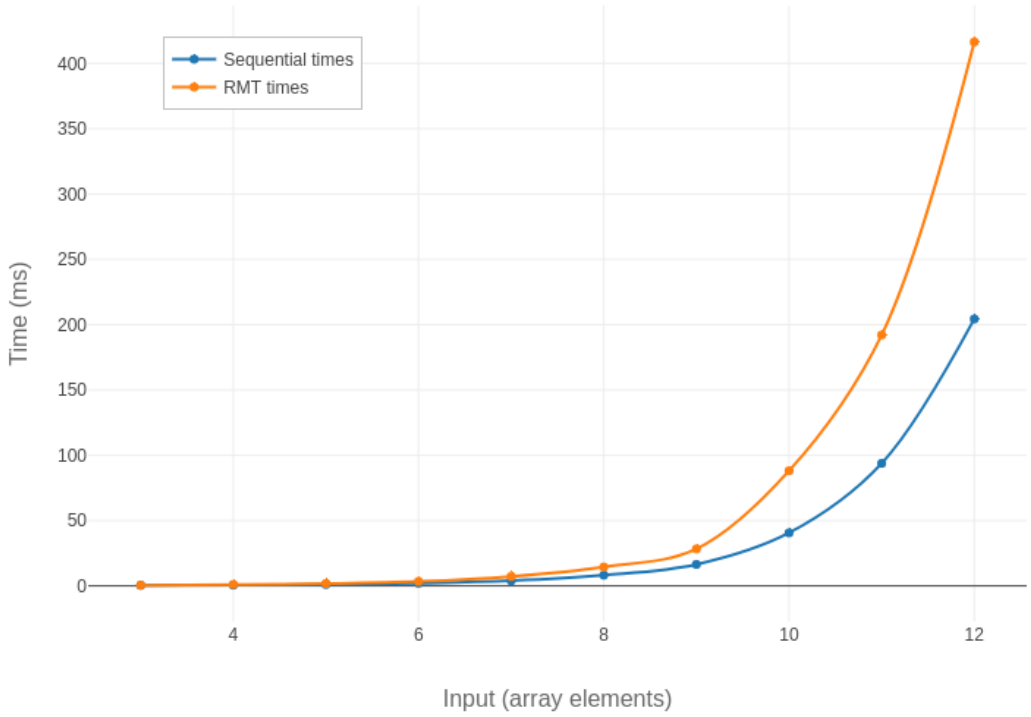


(a) Run-times

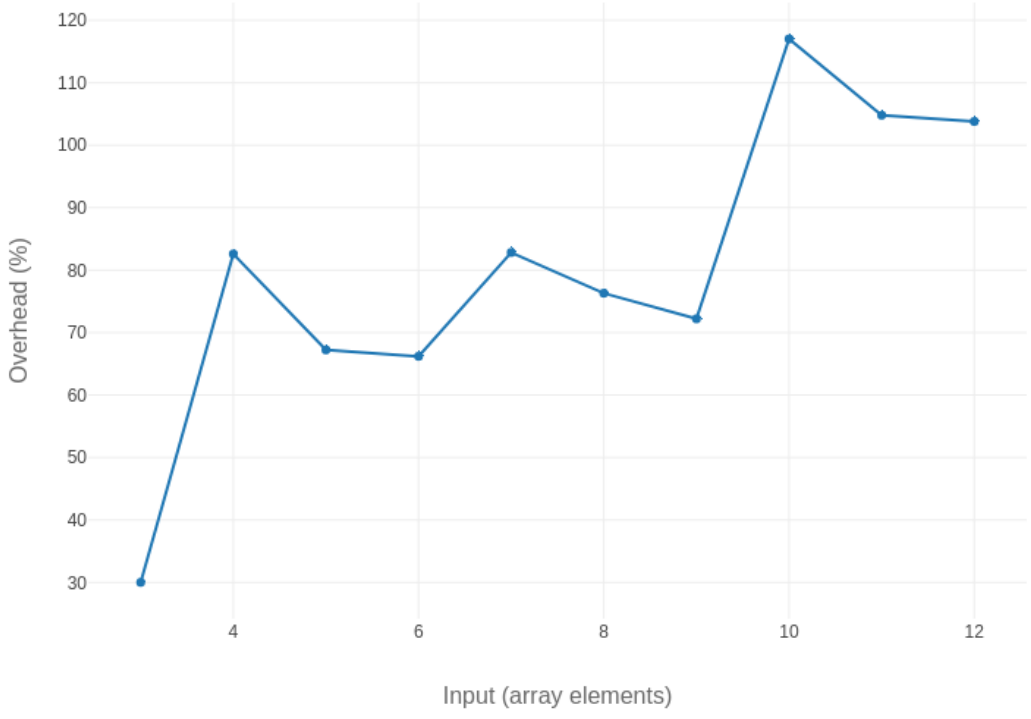


(b) Overhead

Figure 5.6: Matrix-vector multiplication



(a) Run-times



(b) Overhead

Figure 5.7: Combinations (m comb n)

5.2.2 Cumulative results

In this subsection we present the cumulative results of our implementations and compare them with the relative Promises benchmark baselines.

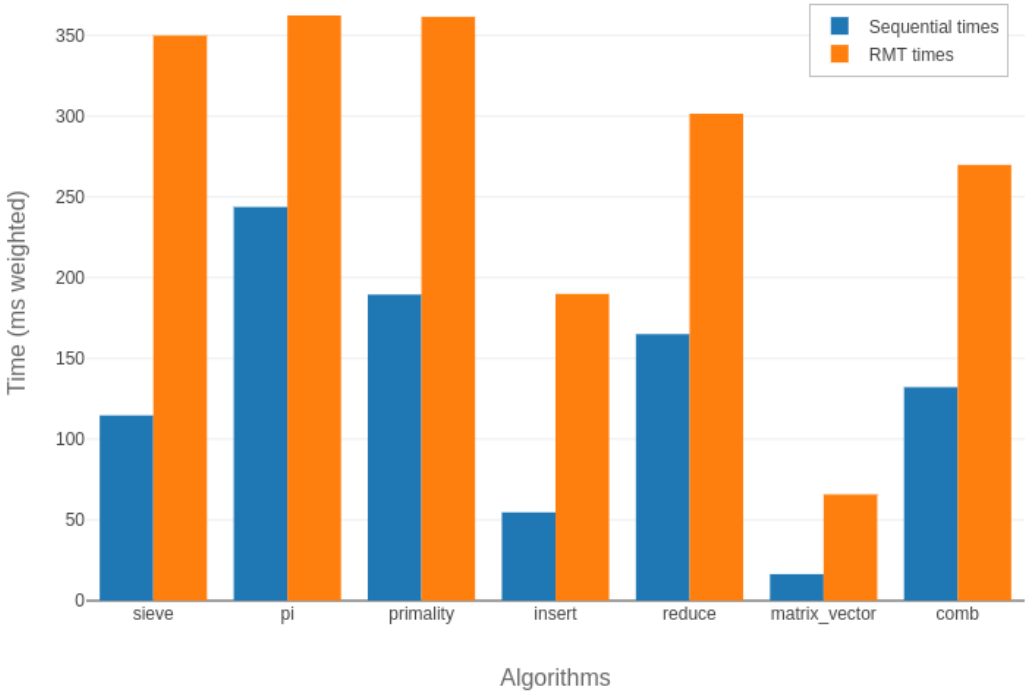


Figure 5.8: Weighted run-time averages by algorithm time complexity

In figure 5.8 we present a weighted run-time mean for each benchmark. The weights we used for each benchmark depend on their algorithm's time complexity. In Section 5.1, we mentioned the time complexity in big-O notation for each algorithm. For the weight of each input we substitute that time complexity with the relative input each time and then use that result as a weight. The same weights were also used in Figure 5.9, but there, we also present the overheads that the Promises implementation had.

In Figure 5.9, we can see that our RMT generally achieved smaller overheads than the relative Promises baselines. In some cases (pi, primality, insert), where a lot of computations are encapsulated in Promises or RMTs relatively, we observe that the promises implementation produces a lot more overhead concluding that creating a great number of Promises objects, does not scale as good as in the case of our RMT. In the case of matrix-vector multiplication, we get very similar results due to the number of threads created in this performance test. In that test we use *Promise.all*() to run all threads in the promises implementation and **runMany** in the RMT one. Although here, we also have a great number of Promises created, *Promise.all*() achieves a slightly more efficient scheduling than **runMany**, resulting in an overhead much closer to the RMT one than expected.

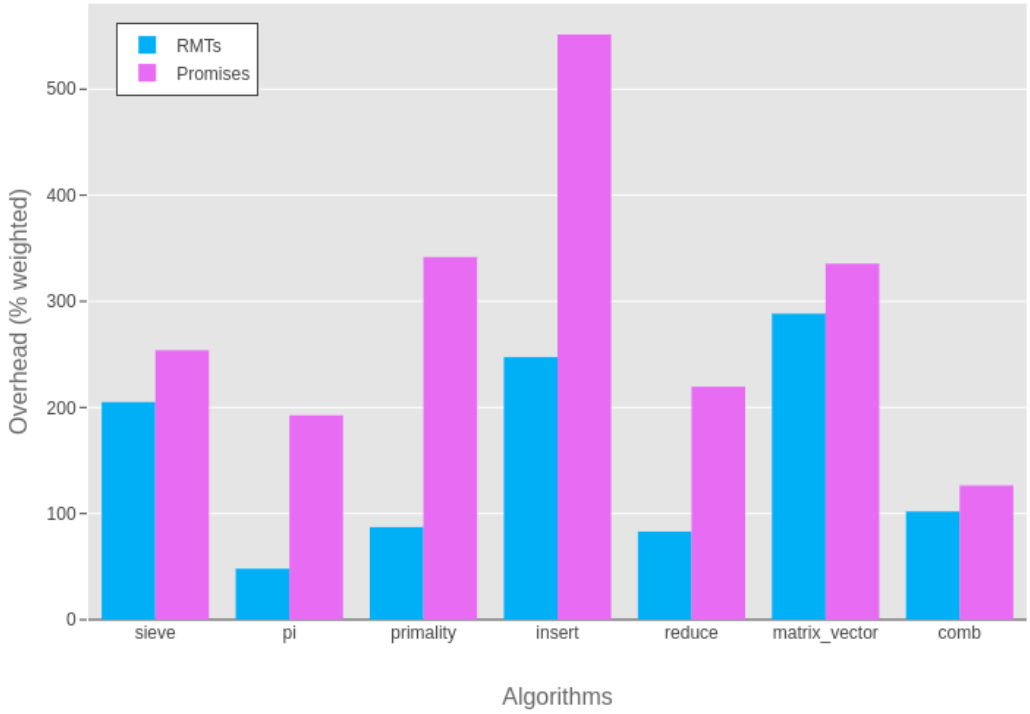


Figure 5.9: Weighted overheads averages by algorithm time complexity

Table 5.1 shows the exact results for the ten biggest inputs we could run during our tests. We can see that RMTs outperform Promises in every benchmark expect the matrix-vector multiplication. However, those overheads are close and the reason that Promises are that close may be that our **runMany** is not as optimizes as *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

Table 5.1: Overheads for the ten biggest inputs for every benchmark tested

In Figure 5.10, we present the average overheads of all the benchmarks for the ten biggest inputs that we used for each one, when we executed the performance tests. For this purpose, we first normalize each benchmark's overheads separately, which means that we transform the data to have a mean of 0 and a standard deviation of 1. This way, we ignore temporally the magnitude of the overheads and thus can find the mean of each input (i.e. the mean of all benchmarks' 1st input) caring only about the actual distances between the data. Then we do the reverse procedure of the normalization and transform the result's mean to be equal to the average of all benchmarks' means and the standard deviation to be equal to the average of of all benchmarks' standard deviations (which is also the error shown in Figure 5.10).

Using the aforementioned technique, we produce the above lines which are basically independent of the benchmarks and the benchmarks' inputs. Thus we compare their actual distances and their slopes to see how our RMT and the Promises implementations behave generally. We observe that the promises give bigger overheads across the whole line and not even the errors of the two lines intersect. We also notice that the RMT line has a smaller slope than the Promises line meaning that the RMTs actually scale better than the promises implementation as the input sizes increase.

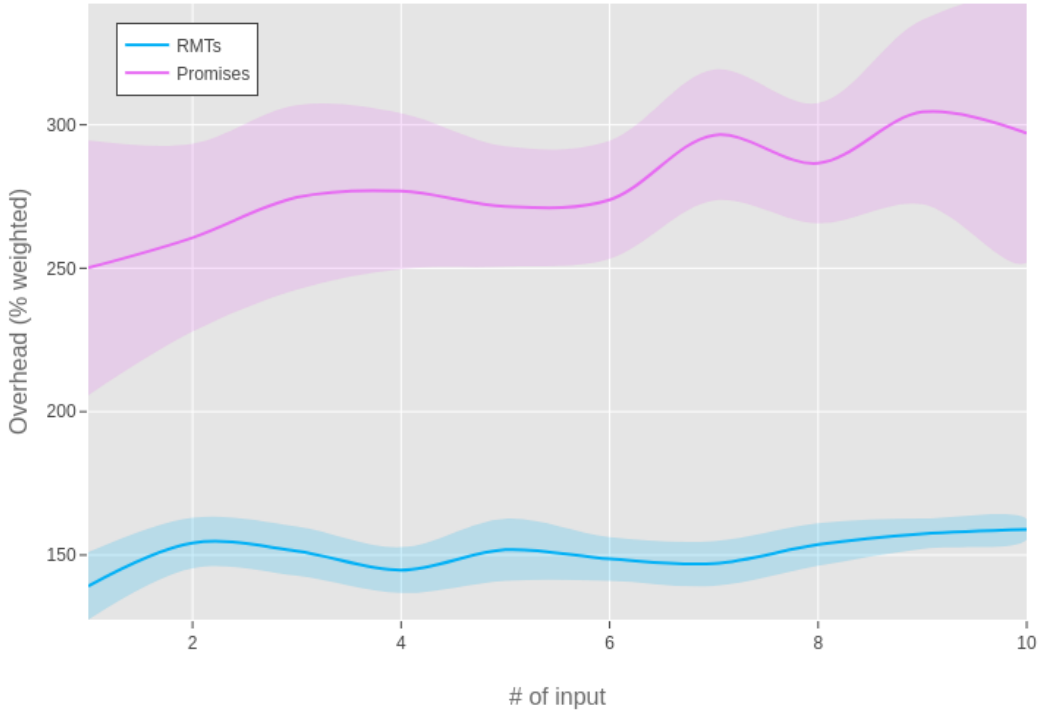


Figure 5.10: Average overheads for the ten biggest inputs of each benchmark

Chapter 6

Epilogue

6.1 Conclusions

In this thesis, we defined a general theoretical framework for formalizing the denotational semantics of interleaved computations in concurrent programming. We present here our conclusions for the *Resumption Monad Transformers*. We showed that resumptions can be a modular expression of the semantics of concurrency. The modularity of resumptions allowed us to easily define the semantics of our example language and extend it with new operations that introduce concurrency on such a language. But apart from its application in the semantics of concurrency, the resumption monad transformer proposed here can be used in the semantics of deterministic languages with unspecified evaluation order, such as C.

The main contribution of this thesis though, is the performance results of RMTs. We showed with our implementation of the RMTs that resumptions are low-overhead constructs that can be used to define the semantics of concurrency. Our comparison with JavaScript's Promises showed that we outperformed them by 100%, making RMTs a reasonable choice for defining the denotational semantics of concurrent programming languages and to be used in real-world applications.

6.2 Future work

Future research should investigate in detail the lifting properties of the resumption monad transformer, i.e. the exact way in which operations supported by the monad of atomic computations M can be passed on to the monad of resumptions $\mathsf{R}(M)$.

Additionally, a further expansion of this work should include an even better implementation of Resumption Monad Transformers. JavaScript nowadays has implemented in its core *generator* functions. *Generator* functions allow to lazily produce results with *yield* rather than *returning* them in a function. This could be much useful in our implementation of *bind* in the Resumption Monad Transformers.

Another direction that would be interesting, is implementing Resumption Monad Transformers in another language. Perhaps we could utilize better tools implementing them in a compiled language like C++. JavaScript also allows using libraries compiled from C++ as add-ons, which could give even better performance results.

Bibliography

[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.