YAŞAR UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

A PARALLELIZATION APPROACH TO HASKELL LANGUAGE THROUGH CATEGORY THEORETIC IMPLEMENTATIONS

Burak EKİCİ

Thesis Advisor: Dr. Ahmet Hasan KOLTUKSUZ

Department of Computer Engineering

Bornova-İZMİR
June 2012

ÖZET

KATEGORİ KURAMI UYGULAMALARI ALTINDA HASKELL DİLİNE BİR PARALELİZASYON YAKLAŞIMI

EKİCİ, Burak

Yüksek Lisans Tezi, Bilgisayar Mühendisliği Bölümü
Tez Danışmanı: Dr. Ahmet Hasan KOLTUKSUZ

Haziran 2012, 189 sayfa

Kategori teorisi, cebirsel yapıların evrensel bileşenlerini görselleştirmemizi ve bazı farklı yapıların aralarındaki ilişkileri kurmamızı sağlayan güçlü bir kuramsal çerçeve ve soyut cebirsel dildir. Teori son yıllarda, bilgisayar bilimlerinde alt uygulama alanları bulmuş, özellikle fonksiyonel programlama dilleri alanında birçok yeniliğin ortaya çıkmasına katkıda bulunmuştur. Bu bağlamda; çalışma, kategori teorisinin fonktörleri, doğal transformasyonları ve monadları ile birlikte gelen soyutlama yetisi ile; çözümlerine katkıda bulunduğu ya da alternatif bakış açıları getirdiği problemlerin ve bu problemlerin ait oldukları alt alanların, "fonksiyonel bir programlama dilinin saflığından, yarı-belirgin paralelizasyon uygulamalarına" kadar, incelenmesini hedeflemektedir.

Anahtar Kelimeler: Kategori Teori, Fonktörler, Doğal Dönüşümler, Monadlar, Fonksiyonel Programlama, Kategori Teorisinin Bilgisayar Bilimleri Uygulamaları: Haskell dilinin Maybe, List, IO, State ve Eval Monadları, Yarı-Belirgin Paralelizasyon, Paralel Programlama Stratejileri: Orjinal ve İkinci Nesil Paralel Haskell Stratejileri.

ABSTRACT

A PARALLELIZATION APPROACH TO HASKELL LANGUAGE THROUGH CATEGORY THEORETIC IMPLEMENTATIONS

EKİCİ, Burak

M.Sc. in Computer Engineering
Supervisor: Dr. Ahmet Hasan KOLTUKSUZ

June 2012, 189 pages

Category theory is a powerful abstract algebraic language and a conceptual framework that lets us visualize universal components of structures of given types and how those structures of different types are interrelated. In recent years, category theory has found new application areas in theoretical computer science and has contributed to developments of new logical systems, especially in the area of functional programming languages. In that sense, this study aims to indicate the areas to which category theory brings alternative solution methods by increasing the number of abstraction layers together with the usage of its functors, natural transformations and monads varying from "purity of a functional programming language" to "semi-explicit parallelization in functional programming".

Keywords: Category Theory, Functors, Natural Transformations, Monads, Functional Programming, Usage of Category Theory in Computer Science: Maybe, List, IO, State and Eval Monads of Haskell, Semi-explicit Parallelization, Parallel Programming Strategies: Original and Second Generation Strategies of parallel Haskell.

Acknowledgements

First of all; I am heartily thankful to my supervisor, Dr. Ahmet Hasan KOLTUKSUZ who has provided never ending support and a guidance at each level of the study from beginning to the end which enabled me to reach an understanding of the subject.

Special thanks and respects are also going to Dr. Mehmet TERZİLER who has provided theoretical basis to the project by letting me know *what is theory of categories* and *the reason why it stands for*. I have additional thanks to him also for permitting me to exploit his lecture notes on *category theory*.

Dr. Günter Egon SCHUMACHER deserves my kind regards and lots of thanks since under his supervision (when I was working as a trainee at JRC-Ispra, Italy), I have learned many things about the issues of “the scientific vision on problem solving” and “the way of behavior in a multi-cultural and scientific environment”.

Many thanks are also going to Dr. Simon Peyton JONES, Dr. Simon MARLOW and Dr. Andres LÖH for their kind attitudes in replying my question-full mails with pinpointing the actual answers regarding parallel Haskell issues.

Furthermore, I have great number of appreciations for Dr. Stephan KLINGER and Mr. Ertuğrul SÖYLEMEZ for letting me exploit their source codes in the declarations and illustrations of `IO` and Maybe monads.

I owe my kind regards and thanks to Dr. Serap ŞAHİN for her always purely optimistic guidance and having never been lost kind attitude even at worst of times.

I have also great respects and many thanks to Dr. Hüseyin HIŞIL for his intensively logical support in any phase of the whole project by sharing valuable experiences, shedding light on even the darkest issues and providing high level support for the usage of LaTeX libraries.

During that process, I also got great number of helps and suggestions from my classmates (more importantly friends): Çağatay YÜCEL, Görkem KILINÇ and Erdem SARILI in both theoretical and moral evaluation of the process without which I may fail. Therefore, they deserve billions of thanks, too.

My father, Mehmet EKİCİ, my mother Zeynep EKİCİ and my brother Çağın EKİCİ have provided me always the best conditions, in any case, for this project to be concluded. Many thanks are also going to them but with knowing that I never can thank them enough.

Lastly, I offer my best of regards and blessings to all of those who supported me in any case during whole process, especially to each member of HASKELL-CAFE.

TEXT OF OATH

I declare and honestly confirm that my study titled “A Parallelization Approach to Haskell Language Through Category Theoretic Implementations”, and presented as Master’s Thesis has been written without applying to any assistance inconsistent with scientific ethics and traditions and all sources I have benefited from are listed in bibliography and I have benefited from these sources by means of making references.

22 / 06 / 2012

Burak EKİCİ

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Category theory is a powerful abstract algebraic language and a conceptual framework that lets us to see;

1. the universal components of structures,
2. how those structures of different types are interrelated.

It is also has been given as an alternative to the set theory as a foundation in abstract mathematics eliminating paradoxes that are involved in the set theory. [1]

In recent years, category theory found new application areas such as theoretical computer science, and has contributed to developments of new logical systems and semantic programming, exemplary *Categorical Logic* by Pitts [2], *Algebra, Categories* and *Databases* by Plotkin [3] and *Some Aspects of Categories in Computer Science* by Scott [4] in 2000. Even in theoretic physics, higher dimensional category theory is exploited by [5] starting from 2001, in order to study "quantum groups" and "quantum field theory".[6] [1]

From the perspective of computational sciences, category theoretic objects such as *functors*, *natural transformations* and especially *monads* are being used in functional programming languages in order to provide abstraction layers that have already been used in object oriented programming paradigm for the encapsulation of the objects. In particular, these objects corresponds to lambda expressions in functional programming.

A purely functional programming language *Haskell* [7] [8], serves the opportunity to its users to be able to implement above mentioned category theoretical objects in its own environment by providing *functor* and *monad* classes. Therefore, special *type constructors* of the language could be defined as *functor* or *monad* class instances together with the help of some *natural transformations* in order to supply new solution strategies to the problems we are having in computing science, for instance:

- *The Maybe monad* [9] of Haskell is standing for the representations of the computations that are analog to the productions over an assembly line. If any worker on the line does not return a product, then the whole of the computation will not be able to return one.
- *The List monad* [9] provides the opportunity to model the non-deterministic programs that might end up with multiple results for any input query.
- *The State monad* [10] brings the feature called *referential transparency* to the language.
- *The IO monad* [9] provides *purity* that indicates that any observable interaction with calling functions or the outside world.
- *The Identity monad* is a base for the creations of monad transformers, *Eval* monad and also represents identity structure in the category that Haskell programming language involves.
- *The Eval monad*, by Marlow et al [11] [12], defines the evaluation order of the computations which could be in serial or in parallel via the encapsulation of some annotations that the language provides called *par*, *pseq* and *seq*, in order to create second generation strategies by [11] which solves some memory management problems arise in original strategies by [13] and to separate the parallel pragmas from the source code. By this way, some useful discipline and structure is brought to the language which simplifies writing parallel programs from coders point of view.

Therefore, monads of category theory implemented in Haskell effectively and brought new line of vision to the current functional programming manner. Especially, providing purity in order to obey the rule of function based referential transparency and controlling the evaluation order of the computations.

After the representations of all above mentioned monads implemented by some specific type constructors and also illustrations of each, in the last chapter of the project, parallelism provided by *eval* monad [11] (Marlow et al) is investigated and exemplified via parallelization of some recursive algorithms such as quicksort, calculating Fibonacci numbers, Karatsuba multiplication and n-queens problem. As an example to the data parallelization, encryption and decryption schemes of RSA crypto-system are given.

Performance evaluations of the mentioned algorithms are also sketched and compared with the ones that are not parallelized in the sense of monadic manner. (Jr. et al) [14]

1.1 Road Map

- *Chapter 1* gives mathematical background on the notions: *graphs*, *categories*, *functors*, *natural transformations* and *monads*.
- *Chapter 2* defines *programming languages* associated with the definitions of *algorithm* stated by Alan Turing no his well known machine and by Alonzo Church Lambda Calculus. By this way, the theoretical differences between functional programming and the other programming paradigms are shown. In the last part of the chapter, Haskell programming language with its type system is explained.
- *Chapter 3* relates Haskell programming language with category theory:
  - *Hask* structure of Haskell is proven to be a category.
  - Functor and Monad type classes of Haskell represent category theoretic functor and monads.
  - Some type constructors of Haskell are proven to be functors and/or monads with some natural transformations.
  - Relevant Examples are provided.
- *Chapter 4* completely focuses on the parallelization issue in Haskell programming environment via both *original* and *second generation strategies* which is based on Haskell's *Eval monad*.
- *Chapter 5* figures out the differences between original and second generation strategies by the realization of *Karatsuba multiplication* and *encryption, decryption* schemes of *RSA cryptosystem*.

- *Chapter 6* indicates the performance evaluations of the mentioned algorithms, conclusion and future work.

In Figure 1.1 the flowchart of the thesis is given.

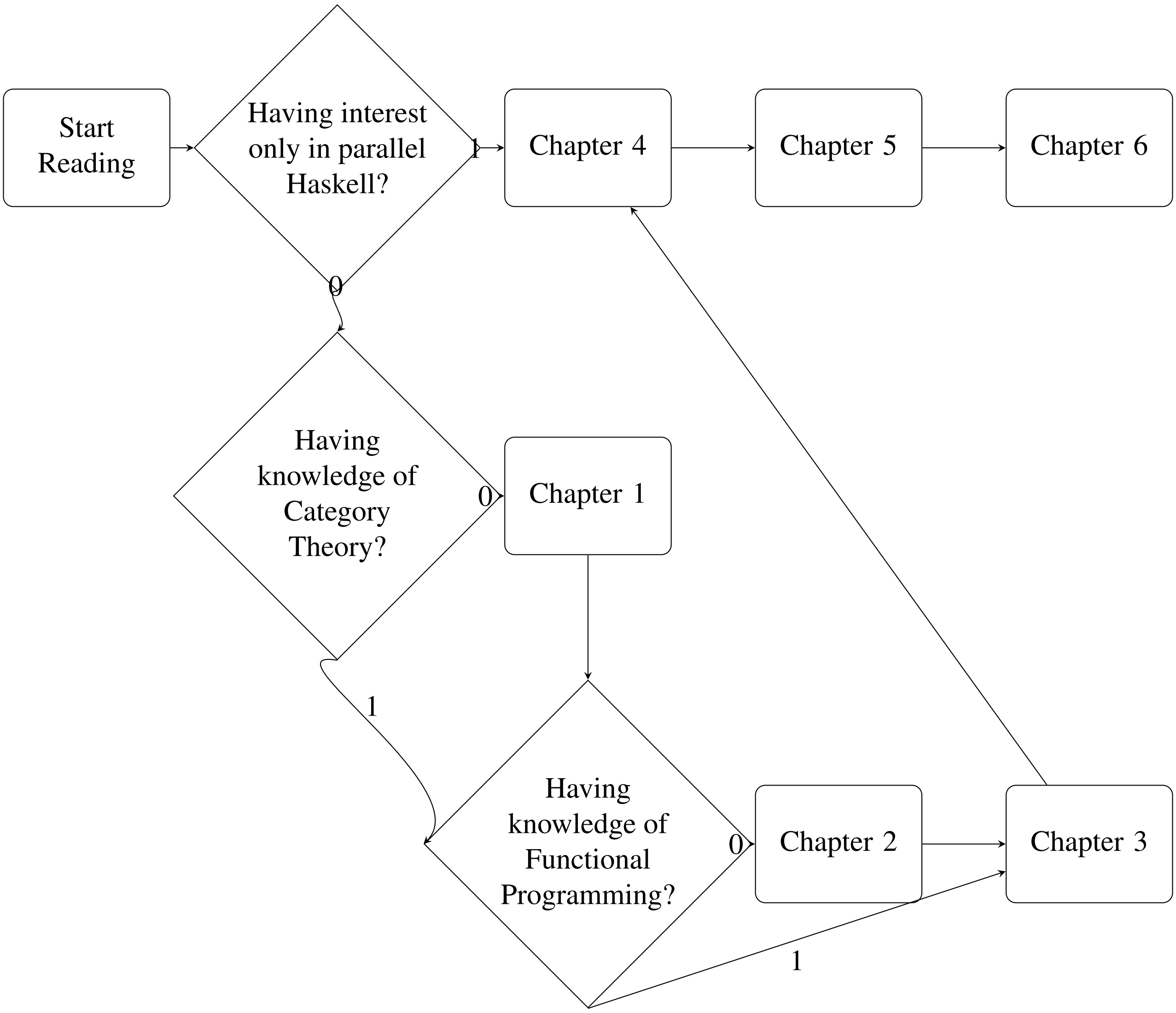
Start Reading
Having interest only in parallel Haskell?
1
Chapter 4
Chapter 5
Chapter 6
0
Having knowledge of Category Theory?
0
Chapter 1
1
Having knowledge of Functional Programming?
0
Chapter 2
Chapter 3
1

FIGURE 1.1: Road Map for the reader

Chapter 2

Mathematical Background

2.1 Preliminaries: Graphs

A graph is the abstract representation of a set of objects, some of which are connected by links called arrows. The specific type of graphs, we discuss in this section is directed multi-graphs with loops, the concept of which is very important in the commutative diagram definitions to express equations from the point of view of categorists. Above all, it can be roughly said that a category is a graph whose paths can be composed. Formally speaking; [15]

Definition 2.1.1. A graph is an ordered pair $G = (V, E)$ forming a set V of vertices or nodes together with a set E of edges or lines, which are 2-element subsets of V.

Definition 2.1.2. An undirected graph is a graph $G = (V, E)$ for which the relations between pairs of vertices are symmetric with the following properties:

1. The first component, V, is a finite, nonempty set.

2. The second component, E, is a finite set of sets. Each element of E is a set that is comprised of exactly two (distinct) vertices.

Example 2.1.3. *Consider the undirected graph $G = (V, E)$ comprised of four vertices and four edges: (see Figure 2.1)*

$V = \{a, b, c, d\}$

$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$

Definition 2.1.4. A directed graph or digraph is an ordered pair $D = (V, A)$ with the following properties:

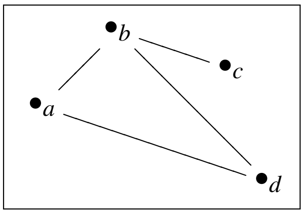

FIGURE 2.1: Undirected Graph

- The first component, V, is a finite, nonempty set.
- The second component, A, is a finite set of ordered pair of vertices called directed edges. Each element of A is a set that is comprised of exactly two (distinct) vertices

The notation $f : a \to b$ means that f is a vertex, a and b are the domain and codomain nodes, respectively.

In the above notation, if $b = a$, that means that a loop is created on the node a by vertex f.

A directed graph having at least one loop on it, is called directed graphs with loops.

Example 2.1.5. *Consider the directed graph $G = (V, E)$ comprised of four vertices and four edges:*

$V = \{a, b, c, d\}, \;\; E = \{\{a, b\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}\}.$

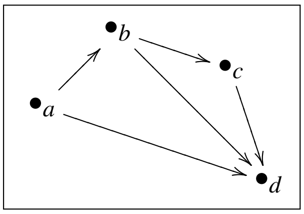

FIGURE 2.2: Directed Graph

Example 2.1.6. *Consider the directed graph with loops $G = (V, E)$ comprised of four vertices and four edges:*

$V = \{a, b, c, d\}, \;\; E = \{\{a, a\}, \{a, b\}, \{a, d\}, \{b, b\}, \{b, c\}, \{b, d\}, \{c, c\}, \{c, d\}\}.$

Definition 2.1.7. Multi-graphs with loops are directed graphs on which parallel edges and loops are allowed.

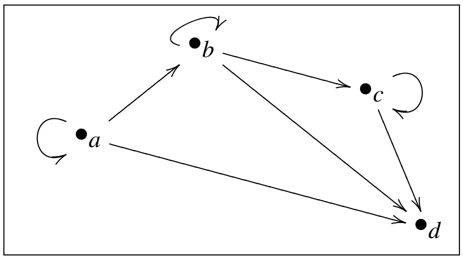

FIGURE 2.3: Directed Graph with Loops

Example 2.1.8. *Consider the directed multi-graph with loops graph $G = (V,E)$ composed of four vertices and four edges:*

$V = \{a,b,c,d\}, \;\; E = \{\{a,a\},\{a,b\},\{a,d\},\{b,b\},\{b,c\},\{b,d\},\{c,c\},\{c,d\}\}.$

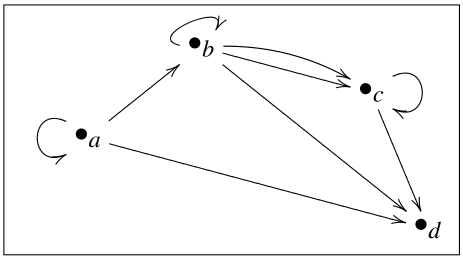

FIGURE 2.4: Directed Multi-Graph with Loops

Categories are specific types of graphs, for that reason, basic definitions and examples of graph theory are given in this section as an introduction. From next section on, notions of Category Theory starts from categories and ends with monads are defined in details.

2.2 Category Theory

Category theory is a general mathematical theory of structures and systems which is still evolving in the sense that its functions are correspondingly developing and expanding. From another point of view, it is a powerful language and a conceptual framework that lets us to study the universal components of structures of given types and the structures of different types that are interrelated.

The Category Theory is an alternative theory to the Set Theory in abstract mathematics. This feature brings much more consistency to category theory in comparison to Set Theory. For instance, Russell's paradox; which is the most famous of the logical or set-theoretical paradoxes. The paradox arises within naive set theory by considering the set of all sets that are not members of themselves. Such a set appears to be a member of itself if and only if it is not a member of itself, hence the paradox.

Note 2.2.1. *Paradoxes come from the "primitive" or "undefined" terms of the Set Theory; especially,* $\in$ *(membership relation) causes troubles.*

In order to avoid this paradox; Category Theory has been proposed as a theory of concrete universals by both satisfying the laws of being a universal, doubtlessly, and involving itself as a member with the help of unique identity morphisms. Therefore, the membership paradox indicated by Bertrand Russell is eliminated.

The evolution of category theory was started in 1945 by Samuel Eilenberg and Saunders Mac Lane who published an article that introduces the basic concepts of what later became the mathematical theory of categories and functors, so called Category Theory. [6] In fact, the French mathematician Dieudonné is the first to introduce these notions.

After their paper in 1945, the clarity of newly developed concepts as a convenient mathematical language was questionable. That condition lasted for about next fifteen years. In 1952, by Eilenberg & Steenrod, category theory was implemented on the foundations of algebraic topology and in 1956, by Cartan & Eilenberg, homological algebra was considered in the manner of category theory. These two approaches gave birth to the opportunity for the new generation mathematicians learn algebraic topology and homological algebra directly in a categorical sense. Indeed, without the method of diagram chasing, many results in these two books seem inconceivable, or at the very least would have required a considerably more intricate presentation.

In 1957, the situation was radically changed by Grothendieck's landmark paper entitled "Sur quelques points d'algébre homologique", in which the categories were used to define more general theories such as algebraic geometry. In 1958, Kan showed that some crucial concepts of limits and co-limits could be demonstrated via the usages of adjoint functors.

Herein after, Category Theory became a convenient language together with the help of two crucial developments:

1. Axiomatic method and language of categories are defined in an abstract fashion types of categories which showed how to perform various constructions in these categories, and proved various results about them. In the core, Grothendieck showed how to develop part of homological algebra in an abstract setting of this sort.
2. Category theorists gradually came to see the pervasiveness of the concept of adjoint functors by published works of Freyd and Lawvere. By the early 1970's, the concept of adjoint functors was seen as central to category theory.

Starting from 1980s, category theory found new application areas such as theoretical computer science, and has contributed to developments of new logical systems and semantic programming (by Pitts, Plotkin and Scott in 2000). Monads of Category Theory are implemented in Haskell functional programming environment providing some important features given in Chapter 4 and 5 that are bringing newline of vision to the functional programming paradigm. Especially, providing purity in order to obey the rule of function based referential transparency. Even in theoretic physics, higher dimensional category theory is exploited by Baez & Dolan starting from 2001, in order to study "quantum groups" and "quantum field theory".[6] [1]

2.2.1 Categories

A category is a graph with a rule to compose arrows from head to tail in order to give another arrow. This rule is subject to certain conditions which will be precisely given in this session.

Definition 2.2.2. A category $\mathbb{C}$ consists of following data:

- Objects: $A, B, C, \ldots$
- Arrows (Morphisms): $f, g, h, \ldots$
- For each $f{:}A \to B$, there are given two objects :
  - dom (f) = A and
  - cod (f) = B

- Given arrows $f{:}A \to B$ and $g{:}B \to C$ with dom (f) = cod (g) there must be an arrow h called the composite of f and g.
  - $h = g \circ f$
- For each object A, there exist a morphism $1_A{:}A \to A$, called the identity or unit on A.

These data are required to satisfy the following laws:

- $\forall f,g,h \in \mathbb{C}{:} f \circ (g \circ h) = (f \circ g) \circ h$ (Associativity of Composition)
- $\forall f \in \mathbb{C}{:} f \circ 1_A = 1_B \circ f = f$ which means $A \xrightarrow{1_A} A \xrightarrow{f} B \xrightarrow{1_B} B$. (Identity)

Note 2.2.3. *In category theory, unlike in set theory; objects do not have to be sets and morphisms functions. In this sense, category theory is a general abstraction of mathematical concepts.*

Definition 2.2.4. A **small** category is a category whose objects and morphisms constitute sets, otherwise it is **large** .

Definition 2.2.5. If A and B are the objects of category C, then set of arrows from A to B is denoted by $Hom_C(A,B)$ or by only $Hom(A,B)$ and is called **homset**.

Hence, for each triple $Hom(B,C) \times Hom(A,B) \to Hom(A,C)$

Proposition 2.2.6. *For any path $(f_1, f_2, ..., f_n)$ in a category C and any integer k with $1 < k < n$:*

$$(f_1 \circ ... \circ f_k) \circ (f_{k+1} \circ ... \circ f_n) = (f_1 \circ ... \circ f_n) \tag{2.1}$$

Fact 2.2.7. *When a binary operation is associative, it turns out that parentheses can be removed.*

Proof 2.2.8. *As mentioned in the definition of categories, composition operation of morphisms in a category is associative.*

Thus;

$$(f_1 \circ ... \circ f_k) \circ (f_{k+1} \circ ... \circ f_n) = f_1 \circ ... \circ f_k \circ f_{k+1} \circ ... \circ f_n$$

for any k with $1 < k < n$

$$f_1 \circ ... \circ f_k \circ f_{k+1} \circ ... \circ f_n = f_1 \circ ... \circ f_n$$

□

Examples of Categories

1. **Category of Sets**

   Definition 2.2.9. The category of sets is the category whose objects are sets and morphisms are functions with the following properties:

   Let **Set** denote the category of sets, then for each object (set) A in category **Set**, there exist an identity function:

   - $1_A{:}A \to A$

   For every pair of morphisms $f{:}A \to B$ and $g{:}B \to C$ in category **Set**, there exist a composite function h, where:

   - $h = g \circ f{:}A \to C.$

2. **Category of Finite Sets**

   Definition 2.2.10. The category of finite sets, denoted by **Fin**, is the category whose objects are finite sets and morphisms are functions between finite sets.

   Function composition and identity function are identical to the ones in category of sets.

3. **Category of Partial Ordered Sets**

   Definition 2.2.11. A Poset (partially ordered set) is a set $(A, \leq_A)$ with a binary relation $a \leq_A b$, which is reflexive, antisymmetric and transitive.

   Definition 2.2.12. Let $(A, \leq_A)$ and $(B, \leq_B)$ be two given posets. Then a function
   $m{:}(A, \leq_A) \to (B, \leq_B)$ is monotone (isotone, increasing, etc...) if $a \leq_A b$ implies $f(a) \leq_B f(b)$.

   Definition 2.2.13. The category of partially ordered sets is the category whose objects are partially ordered sets and morphisms are monotone functions with the following properties:

   (a) Let **PoSet** denote the category of posets. Then for each object (poset) $(A, \leq_A)$ in category **PoSet**, there exist an identity function:

   - $1_{(A,\leq_A)}{:}\ (A, \leq_A) \to (A, \leq_A)$
     $a \leq_A b \mapsto a \leq_A b$

(b) For every pair of monotone function $f\colon (A, \leq_A) \to (B, \leq_B)$ and $g\colon (B, \leq_B) \to (C, \leq_C)$ in category **PoSet**, there exist a composite monotone function h, where:

- $h = g \circ f\colon (A, \leq_A) \to (C, \leq_C)$.
- $h = g \circ f\colon a \leq_A b \mapsto g(f(a)) \leq_C g(f(b))$.

4. **Monoid as a Category**

Definition 2.2.14. A monoid (or semi-group with an identity element) is a set M with a binary operation $\bullet\colon M \times M \to M$ that satisfies the following axioms:

(a) $\forall a, b \in M, (a \bullet b) \in M$ (Closure)

(b) $\forall a, b, c \in M, (a \bullet b) \bullet c = a \bullet (b \bullet c)$ (Associativity)

(c) there exists an element $e \in M$ such that $\forall a \in M, e \bullet a = a \bullet e = a$

Definition 2.2.15. Equivalently, a monoid is a category with one object. The morphisms of the category are the elements of some monoid, the identity morphism is e (identity element of monoid) and composition operation is $\bullet$.

Also given any set X, the set of functions from X to X, $\mathrm{Hom}(X, X)$, is a monoid under the operation of composition.

More generally, for any object A in a category $\mathcal{C}$, the set of arrows from A to A forms a monoid, written $Hom_C(A, A)$.

5. **Category of Groups**

Definition 2.2.16. A group $< G, + >$ is a set G closed under the binary operation "$+$" such that

(a) $+$ is associative, which means that $(a + b) + c = a + (b + c)$

(b) $\exists e$ such that $e + x = x = x + e$ for all $x \in G$. (Identity Element)

(c) for each $a \in G$, $\exists a' \in G$ with $a + a' = e$. (Inverse Element)

Definition 2.2.17. A map ϕ of a group $< G, + >$ into $< G, +' >$ is a homomorphism if $\phi(a + b) = \phi(a) +' \phi(b)$ for all $a, b \in G$.

Definition 2.2.18. The category of groups is the category whose objects are groups and morphisms are group homomorphisms with the following properties:

Let **Grp** denote the category of groups. Then for each object (group) $< G, + >$ in category **Grp**, there exist an identity homomorphism:

- $1_G\colon G \to G$
  $a \mapsto a$

For every pair of group homomorphisms $f\colon G_1 \to G_2$ and $g\colon G_2 \to G_3$ in category **Grp**, there exist a composite group homomorphism h, where:

- $h = g \circ f\colon G_1 \to G_3$.

6. **Category of Relations**

   Definition 2.2.19. The category of relations is the category whose objects are sets and morphisms are relations with the following properties:

   Let **Rel** denote the category of relations. Then for each object (sets) A in category **Rel**, there exist an identity morphism:

   - $1_A = \{(a,a) \mid a \in A\} \subseteq A \times B$.
   - $A \xrightarrow{1_A} A \xrightarrow{R} B, A \xrightarrow{R} B \xrightarrow{1_B} B$.

   For every pair of relations $R \subseteq A \times B$ and $S \subseteq B \times C$ in category **Rel**, a composite morphism $T = S \circ R$ could be defined by:

   - $(a,c) \in (S \circ R) \iff (\exists b \in B)[(a,b) \in R \wedge (b,c) \in S]$

7. **Category of Matrices**

   Definition 2.2.20. The category of matrices is the category whose objects are finite sets and morphisms are rectangular matrices of natural numbers.

   Identity morphism is identity matrix and the composition operation is well-known matrix multiplication.

Constructions on Categories

This section requires some level of abstract algebra. Additionally, the reader should be aware that a structure may have "substructures" which are subsets closed under predefined operations and "free" structures of given types.

Above mentioned structures could be performed in categories, as well. These structures are all outlined in this section.

Definition 2.2.21. A **subcategory** $\mathcal{D}$ of category $\mathcal{C}$ whose objects $\text{Obj}(\mathcal{D}) \in \text{Obj}(\mathcal{C})$ and morphisms $\text{Morph}(\mathcal{D}) \in \text{Morph}(\mathcal{C})$ such that

- for each $A \in \text{Obj}(\mathcal{D})$, the identity morphism of $A, \text{id}_{\mathcal{A}} \in \text{Morph}(\mathcal{D})$.
- for each $f \in \text{Morph}(\mathcal{D})$, $dom(f)$ and $cod(f) \in \text{Obj}(\mathcal{D})$.
- for every pair $f, g \in \text{Morph}(\mathcal{D})$ such that $f \circ g$ exists, then $f \circ g \in \text{Morph}(\mathcal{D})$.

Example 2.2.22. *The category of finite sets,* **Fin***, is a* **subcategory** *of category* **Set***.*

Let A and B be finite sets, then $Hom_{\textbf{Fin}}(A,B) = Hom_{\textbf{Set}}(A,B)$, which means every arrow of **Set** *between objects of* **Fin** *is also an arrow of* **Fin***.*

Example 2.2.23. **Set** *is a subcategory of a category whose objects are sets and morphisms are partial functions.*

Definition 2.2.24. If category $\mathcal{D}$ is subcategory of category $\mathcal{C}$ and for every pair of objects A,B of $\mathcal{D}$, $Hom_{\mathcal{D}}(A,B) = Hom_{\mathcal{C}}(A,B)$, then $\mathcal{D}$ is called **full subcategory** of $\mathcal{C}$.

From this definition, it can be inferred that **Fin** is full subcategory of **Set**.

$$Hom_{\textbf{Fin}}(A,B) = Hom_{\textbf{Set}}(A,B) \tag{2.2}$$

Example 2.2.25. *A category ζ, whose objects are sets and morphisms are surjective (onto) mappings is a subcategory of* **Set***.*

Proof 2.2.26. *Any identity $1_A{:}A \to A$ is surjective, where $1_A{:}A \in Morp(\zeta)$.*

Given two surjective mappings $f{:}A \to B$ and $g{:}B \to C$, compositions of them $h = g \circ f{:}A \to C$ is surjective, as well.

$\forall b \in B, \exists a \in A{:} f(a) = b$; because f is surjective.

$\forall c \in C, \exists b \in B{:} g(b) = c$; because g is surjective.

Therefore, $g \circ f \in Morph(\zeta)$. □

Definition 2.2.27. Let $\mathcal{C}$ and $\mathcal{D}$ are two categories, the product $\mathcal{C} \times \mathcal{D}$ is also the category whose objects are all ordered pairs (C,D) where $C \in \mathcal{C}$ and $D \in \mathcal{D}$ and in which an arrow $(f,g){:}(C,D) \to (C',D')$ is a pair of arrows $f{:}C \to C' \in \mathcal{C}$ and $g{:}D \to D' \in \mathcal{D}$. The identity morphism of (C,D) is (id_C, id_D), where:

- $(id_C, id_D){:}(C,D) \to (C,D)$.

The composition of pair of morphisms $(f,g)\colon(C,D)\to(C',D')$ and $(f',g')\colon(C',D')\to(C'',D'')$ is another pair of morphism $(h,j)\in\mathcal{C}\times\mathcal{D}$, where:

- $(h,j)=(f,g)\circ(f',g')$.
- $(f,g)\circ(f',g')=(f\circ f',g\circ g')\colon(C,D)\to(C'',D'')$.

Definition 2.2.28. Let $\mathcal{C}$ be a category. The dual or opposite of $\mathcal{C}$ is the category $\mathcal{C}^d$ or $\mathcal{C}^{op}$ whose objects are the objects of $\mathcal{C}$ and whose arrows are the morphisms of $\mathcal{C}$ in the reverse order.

- $f{:}A\to B$ in $\mathcal{C}$ yields $f{:}B\to A$ in $\mathcal{C}^{op}$.

Any property satisfied by $\mathcal{C}$ is also satisfied by $\mathcal{C}^{op}$.

Definition 2.2.29. For any given graph $\mathcal{G}$, there exists a category $\mathcal{C}(\mathcal{G})$ whose objects are nodes of $\mathcal{G}$ and morphisms are paths in $\mathcal{G}$.

Associative composition is defined like:

- $(f_1\circ\ldots\circ f_k)\circ(f_{k+1}\circ\ldots\circ f_n)=(f_1\circ\ldots\circ f_n)$

For each object $A\in\mathcal{C}(\mathcal{G})$, there exists an identity morphism 1_A, where:

- $1_A{:}A\to A$.

Properties of Objects and Morphisms in a Category

The specific properties, in the sense of the roles that objects and morphisms have in a category, are called categorical properties. In this section; the definitions and examples of some categorical properties such as isomorphisms, initial-terminal objects, mono and epimorphisms are given, in detail.

Isomorphism

Definition 2.2.30. Let $\mathcal{C}$ be a category. Let A,B be objects $A,B\in\mathcal{C}$. A morphism $f{:}A\to B$ is an **isomorphism** if and only if there exists a **unique** morphism $g{:}B\to A$ such that $g\circ f=1_A$ and $f\circ g=1_B$. In this case, A is said to be **isomorphic to** B and written $A\cong B$.

Proof 2.2.31. *Assume that there exists $h_1{:}B\to A$, $h_2{:}B\to A$ such that $f\circ h_1=1_B$ and $h_2\circ f=1_B$. It is enough to show that h_1 and h_2 are equal.*

$$h_1=1_A\circ h_1=(h_2\circ f)\circ h_1=h_2\circ(f\circ h_1)=h_2\circ 1_B=h_2 \tag{2.3}$$

□

Initial, Terminal and Zero Objects

Definition 2.2.32. An object U of $\mathcal{C}$ is said to be an **initial object** if for every object X in $\mathcal{C}$, $f{:}U \to X \in Morph_{\mathcal{C}}$ is a singleton.

Definition 2.2.33. An object U of $\mathcal{C}$ is said to be a **terminal object** if for every object X in $\mathcal{C}$, $f{:}X \to U \in Morph_{\mathcal{C}}$ is a singleton.

Example 2.2.34. *In* **Set** *the initial object is empty set, because for any $X \in Obj(\mathbf{Set})$, there exists a unique mapping, f, from empty set to X.*

- $f{:}\varnothing \to X$

Every set with a single element in category **Set** *are terminal objects. Because, for any $X \in Obj(\mathbf{Set})$ there exist a unique mapping, f, from X to one element sets.*

- $f{:}X \to \{a\}$

Definition 2.2.35. If an object Z is both initial and terminal, it is called the **zero object**.

Note 2.2.36. *A category which has initial and terminal objects do not need to have a zero object.*

Example 2.2.37. *In* **Set**, *there is no zero objects, because initial and terminal objects are not intersecting.*

Mono and Epimorphisms

Definition 2.2.38. A mapping $f{:}Y \to Z$ is left-cancellable if

- $\forall X, \forall g_1{:}X \to Y \wedge g_2{:}X \to Y$
- $f \circ g_1 = f \circ g_2 \implies g_1 = g_2$

$$\begin{tikzcd} X \arrow[r, shift left, "g_1"] \arrow[r, shift right, "g_2"'] & Y \arrow[r, "f"] & Z \end{tikzcd}$$

Definition 2.2.39. Let $\mathcal{C}$ be a category, a morphism $f{:}A \to B$ in $\mathcal{C}$ is said to be **monic** or **monomoprhism** if it is left-cancellable.

Note 2.2.40. *In concrete categories, every injective morphism is monic, however the converse, in general, is not true.*

Definition 2.2.41. A mapping $f{:}X \to Y$ is right-cancellable if

- $\forall X, \forall g_1, g_2 : Y \to Z, g_1 \circ f = g_2 \circ f \Longrightarrow g_1 = g_2$

$$\begin{tikzcd} X \arrow[r, "f"] & Y \arrow[r, shift left, "g_1"] \arrow[r, shift right, "g_2"'] & Z \end{tikzcd}$$

Definition 2.2.42. Let $\mathcal{C}$ be a category, a morphism $f : A \to B$ in $\mathcal{C}$ is said to be **epic** or **epimorphism** if it is right-cancellable.

Note 2.2.43. *In concrete categories, every surjective morphism is epic, however the converse, in general, is not true.*

Assume that $f \circ g$ is defined;

1. if f, g are monic, then $f \circ g$ is monic.
   - $(f \circ g) \circ h = (f \circ g) \circ k \Rightarrow g \circ h = g \circ k \Rightarrow h = k$
2. if f, g are epic, then $f \circ g$ is epic.
   - $h \circ (f \circ g) = k \circ (f \circ g) \Rightarrow h \circ f = k \circ f \Rightarrow h = k$
3. if $f \circ g$ is monic, then g is monic.
   - $g \circ h = g \circ k \Rightarrow f \circ g \circ h = f \circ g \circ k \Rightarrow h = k$
4. if $f \circ g$ is epic, then f is epic.
   - $h \circ f = k \circ f \Rightarrow h \circ f \circ g = k \circ f \circ g \Rightarrow h = k$

2.2.2 Functors

In category theory, morphisms that are relation between objects are much more important than objects, themselves. In this section, a wider notion of morphisms called functor, a prescription between categories, is defined in detail.

Definition 2.2.44. Let $\mathcal{C}$ and $\mathcal{D}$ be two categories, a **covariant functor** F from $\mathcal{C}$ to $\mathcal{D}$ is a prescription that assigns every object A of $\mathcal{C}$ to objects $F(A)$ of $\mathcal{D}$ and every morphism $\alpha : A \to B$ of $\mathcal{C}$ to morphisms $F(\alpha) : F(A) \to F(B)$ of $\mathcal{D}$ such that;

1. $F_{1_A} = 1_{F(A)}$
2. if $\beta \circ \alpha$ is defined in $\mathcal{C}$, then there exist $F(\beta) \circ F(\alpha)$ in $\mathcal{D}$ where:
   - $F(\beta) \circ F(\alpha) = F(\beta \circ \alpha)$

Definition 2.2.45. A functor F is called **endofunctor** if it is defined from any category $\mathbf{C}$ to itself.

- $F{:}\mathbf{C} \to \mathbf{C}$.

Examples of Functors

1. Let $\mathcal{C}$ be a category, consider the assignment $1_{\mathcal{C}}{:}\mathcal{C} \to \mathcal{C}$ described by setting $1_{\mathcal{C}}(X) = X$ for every object of $\mathcal{C}$ and $1_{\mathcal{C}}(f) = f$ for every morphism of $\mathcal{C}$. $1_{\mathcal{C}}$ is the functor called **identity functor** on $\mathcal{C}$.

2. Let $\mathcal{C}$ be a subcategory of category $\mathcal{D}$, consider the assignment $I{:}\mathcal{C} \to \mathcal{D}$ described by $I(X) = X$ for every object X of $\mathcal{C}$ and $I(f)$ for every morphism in $\mathcal{C}$ then I is a functor from $\mathcal{C}$ to $\mathcal{D}$, called **inclusion functor**.

3. Let $(P,\leq)$ and $(Q,\leq)$ be preorders seen as categories, then

   - $F{:}P \to Q$

     – $\forall a,b \in P, a \leq b \Rightarrow F(a) \leq F(b)$

4. Let $(M,\bullet,1)$ and $(N,\diamond,1)$ are two monoids (a category with single object). A morphism $F{:}(M,\bullet,1) \to (N,\diamond,1)$ is a functor.

   - $F(1) = 1$.
   - $\forall m,n \in M$, there exist $F(m),F(n) \in N$ wnhere $F(m \bullet n) = F(m) \diamond F(n)$.

5. A **bifunctor** F is a functor of the form: $F{:}\mathcal{A} \times \mathcal{B} \to \mathcal{C}$.

   Note 2.2.46. *The notion of* **bifunctor** *could be generalized under the notion of* **multifunctors**.

   Given mappings $f{:}A \to C$ and $g{:}B \to D$, we know from the universal property of products that there exists a unique mapping $f \times g{:}A \times B \to C \times D$ such that the diagram

   is commutative, naturally that is given by $(f \times g)(a,b) = (f(a),g(b))$. We define a functor $F{:}\mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$ by assigning $F(A,B) = A \times B$ and $F(f,g) = f \times g$. This functor is called **cartesian product bifunctor**.

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_A"'] \arrow[r, "\pi_B"] \arrow[d, "f \times g"] & B \arrow[d, "g"] \\
C & C \times D \arrow[l, "\pi_C"] \arrow[r, "\pi_D"] & D
\arrow[from=1-1, to=2-1, "f"]
\end{tikzcd}$$

FIGURE 2.5: Cartesian Bifunctor - Commutative Diagram

2.2.3 Natural Transformations

Natural transformations are one of the most important issues in category theory. They map a functor into another with preserving the internal structure and also could be counted as "morphisms between functors". From another point of view, natural transformations are used to formalize the categories of functors.

Definition 2.2.47. Define $F:\mathcal{C} \to \mathcal{D}, G:\mathcal{C} \to \mathcal{D}$ are functors, with categories $\mathcal{C}$ and $\mathcal{D}$. A **transformation** from F to G is a rule that assigns each object $A \in \mathcal{C}$ to a morphism $\eta_A : F(A) \to G(A) \in \mathcal{D}$.

Definition 2.2.48. If $F:\mathcal{C} \to \mathcal{D}, G:\mathcal{C} \to \mathcal{D}$ are functors, from category $\mathcal{C}$ to $\mathcal{D}$, then a **natural transformation** $\eta : F \to G$ is a rule that assigns each object $A \in \mathcal{C}$ to a morphism $\eta_A : F(A) \to G(A) \in \mathcal{D}$ in such a way that associated with every morphism $f:A \to B \in \mathcal{C}$, there exists a commutative diagram:

$$\begin{tikzcd}
F(A) \arrow[r, "\eta A"] \arrow[d, "F(f)"] & G(A) \arrow[d, "G(f)"'] \\
F(B) \arrow[r, "\eta B"] & G(B)
\end{tikzcd}$$

FIGURE 2.6: Natural Transformation - Commutative Diagram

Example 2.2.49. *Let* $U:\textbf{Mon} \to \textbf{Set}$ *be a functor between monoid and set categories. Then we can define* $U \times U:\textbf{Mon} \to \textbf{Set}$ *where:*

- $(U \times U)(M) = U(M) \times U(M)$ *for a monoid* M.
- $h:M \to N$, $(U \times U)(h)(m,n) = (h(m),h(n))$ *for a monoid homomorphism*

Let $\mu : U \times U \to U$ *and* $\mu M : U(M) \times U(M) \to U(M)$ *defined by* $\mu M(m,m') = mm' \in M$, *then* μ, *monoid multiplication, is a natural transformation if following diagram commutes.*

$$\begin{tikzcd}
(U \times U)(M) \arrow[r, "\mu M"] \arrow[d, "(U\times U)(h)"] & (U)(M) \arrow[d, "h"'] \\
(U \times U)(N) \arrow[r, "\mu N"] & U(N)
\end{tikzcd}$$

FIGURE 2.7: Natural Transformation - Commutative Diagram 2

Let $(m,n) \in (U \times U)(M)$, then $\mu M(m,n) = (m \cdot n)$ (the product of m and n in M) $h(m \cdot n) = h(m) \cdot h(n) = h \circ \mu M(m,n)$.

$(U \times U)(h)(m,n) = (h(m),h(n))$ $\mu N(h(m),h(n)) = h(m).h(n) = \mu N \circ (U \times U)(h)$.

The above diagram is commutative, from the fact that h is a homomorphism.

$$\mu N \circ (U \times U)(h) = h \circ \mu M(m,n) = h(m) \cdot h(n) \tag{2.4}$$

2.2.4 Monads

Monads are a nice abstract way of talking about various algebras together with the idea of adjunction whose part of structure is an endofunctor. On the other hand, they play very critical and important role as a wrapping tool in the manner functional programming that are defined in detail in the upcoming chapters, however in this session, monads in the sense of category theory is defined and illustrated particularly.

Definition 2.2.50. Let $\mathbf{C}$ be a category, $T{:}\mathbf{C} \to \mathbf{C}$ an endofunctor, $\eta{:}Id_{\mathbf{C}} \to T$ and $\mu{:}T^2 \to T$ are two natural transformations, then the triple (T,μ,η) is called a **monad** if and only if both of the below diagrams are commutative which means the equalities:

$$\mu \circ T\mu = \mu \circ \mu T \text{ and } \mu \circ T\eta = id_T = \mu \circ \eta T$$

are being satisfied. As shown in the Figure 2.8.

Example 2.2.51. *Let $F{:}\mathbf{Set} \to \mathbf{Set}$ be an endofunctor, for any set $A \in \mathbf{Set}$, if $F(A) = \{a_1,a_2,a_3,...\}$ represents set of all possible lists may be constructed by elements in A including empty one, then together with two natural transformations $\eta_A = A \to F(A)$ and $\mu_A{:}F(F(A)) \to F(A)$; triple (F,η_A,μ_A) constructs a monad.*

Example 2.2.52. *Let $T{:}\mathbf{Set} \to \mathbf{Set}$ be an endofunctor, for any set $A \in \mathbf{Set}$, and let T maps them into their power sets, represented by $T(A)$ and any function $f{:}A \to B$ into*

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"] & T^2 \arrow[d, "\mu"'] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, "T\eta"] \arrow[d, "\eta T"'] \arrow[dr, equal, "id"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

FIGURE 2.8: Commutative Diagrams - Monad

$T(f){:}T(A) \to T(B)$, then together with two natural transformations $\eta_A = A \to T(A)$ and $\mu_A{:}T(T(A)) \to T(A)$; triple (T,η_A,μ_A) constructs a monad.

Note 2.2.53. *Some definitions, theorems and proofs given in this chapter are taken from* Prof. Dr. Mehmet Terziler*'s 2010-2011 fall term semester* Category Theory *lecture notes in Yaşar University, Izmir, Turkey.*

Chapter 3

Haskell Programming Language

3.1 Programming Languages

In plain terms, an algorithm can be defined to be a collection of simple instructions for carrying out a task. Formal definitions are made by Alan Turing via his machine so called "Turing Machine", by Gödel–Herbrand–Kleene in the definition recursive functions, by Alonzo Church in his Lambda Calculus and by Emile Post in "Formulation 1".

In Alan Turing's model, there are three important notions to be defined, just in order to be able to comprehend what algorithm means.

1. **Turing Machine** :

   A Turing Machine is a hypothetical device that was described by Alan Turing in 1936. It mainly consists of three parts:

   (a) an **infinite tape**, which is divided into cells, on which symbols are being manipulated,

   (b) a **head** that is able to read and write symbols on the tape and also move the tape left and right one cell at a time and

   (c) a **control unit** which defines state transitions like either erasing or writing a symbol on the tape and moving the tape right or left direction.

   Formally; a Turing Machine M is defined by the following multi tuple.

In formal terms;

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

where

Q is the set of states,
Σ is the input alphabet,
Γ is the finite set of symbols. (called tape alphabet),
$\delta = Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the state transition function,
$q_0 \in Q$ is the initial state,
$\square \in \Gamma$ is the blank symbol,
$F \subseteq Q$ is the set of final states.

In any time slot, we know two things about a Turing Machine:

- the content of each cell of the tape, especially the one under the tape.
- the current state of the machine.

2. **Turing Transducers**:

   A Turing Machine which transduces an input sequence into an output sequence.

   $$\omega = f(\mathrm{w}) \Rightarrow q_0 \ \mathrm{w} \vdash^* q_f f(\mathrm{w}) \text{ with } q_f \in F$$

   where w represents input, ω represents output sequences and $\vdash^*$ is used to define all of the transformations in between input and output sequences.

3. **Turing Computable Functions**:

   A function f with domain D is said to be a Turing computable if there exist some Turing Machine M such that

   $$q_0 \ \mathrm{w} \vdash^* q_f f(\mathrm{w}) \text{ with } q_f \in F \text{ and } \forall \mathrm{w} \in D$$

Definition 3.1.1. Let a function $f : \Sigma^* \rightarrow \Sigma^*$ is a computable function. If there exist a Turing Machine transducer T such that on every input w. Then T halts with $f(w)$ on its tape. Such a Turing Machine is itself called an **algorithm**.

In 1936, Alonzo Church also came up with a formalism to define the notion of algorithm relying on his Lambda Calculus. However, the formalism done by Alan Turing and Alonzo Church were shown to be equivalent, as indicated in the below definition. That means that programming languages based on Turing's computational model and Lambda Calculus computational model are intuitively equivalent.

In Lambda Calculus part of the thesis, the simulation of Lambda Calculus with using Turing Machines is also given.

Intuitive notion of **algorithms** $\equiv$ Turing Machine **algorithms**.

Definition 3.1.2. Over an alphabet Σ, Σ^* is defined as the set of all possible combinations of words that could be generated by the elements of the given alphabet Σ.

Definition 3.1.3. A **formal language** over an alphabet Σ is some fixed subset, L, of Σ^* whose members are called words.

Definition 3.1.4. A **programming language** is a **formal language** which expresses **algorithms** in order to control the flow of instructions.

Programming languages include some elementary building blocks that are described by syntactic and semantics rules, in order to define the process of the data or the data, itself.

Syntax

Syntax of any programming language refers to the ways that symbols gather to create well-formed sentences in the language. The syntax of textual programming languages are totally the combinations of **regular expressions**, from lexical sense, and **Backus – Naur Form**, which is one of the two main notation techniques for context-free grammars, from the grammatical point of view.

Definition 3.1.5. A **grammar** G is defined by

$$G = (\Sigma, N, P, S)$$

where

Σ is the set of terminal symbols which cannot be broken down into smaller units.
N is the set of non-terminal symbols that can be broken down into other symbols.
P is the productions or rules to define non-terminal symbols.
S is the start symbol which is a distinguished non-terminal symbol.

Example 3.1.6. *Here is an example to the traditional grammar notations in programming languages, known as* **Bakus – Naur Form***:*

<declaration> ::= var <variable list> : <type>;

where

"**var**", ":" *and* ";" *are terminal symbols.*
"::=" *is the description part for grammars which means "is defined to be".*

Definition 3.1.7. The part of the grammar which composed of its terminal and non-terminal symbols is called **vocabulary**.

Within the field of formal computer science; **Chomsky Hierarchy** categorizes the formal grammars into four types, from type 0 to type 3.

1. **Type 0:**

   Type 0 grammars are the most general ones, so called **unrestricted grammars** in which there is no requirements rather than at least one non-terminal symbol should occur on the left side of a rule like "α ::= β".

   For example, the grammar G with non-terminals $N = \{S, A, B, C\}$, terminals $\Sigma = \{a, b, c\}$, start $S = \{S\}$and rules $P =$

   $$\begin{aligned}
   &\mathbf{S} \Rightarrow \mathbf{SS} \mid \mathbf{ABC} \mid \varepsilon\\
   &\mathbf{AB} \Rightarrow \mathbf{BA}\\
   &\mathbf{BA} \Rightarrow \mathbf{AB}\\
   &\mathbf{AC} \Rightarrow \mathbf{CA}\\
   &\mathbf{CA} \Rightarrow \mathbf{AC}\\
   &\mathbf{BC} \Rightarrow \mathbf{CB}\\
   &\mathbf{CB} \Rightarrow \mathbf{BC}\\
   &\mathbf{A} \Rightarrow \mathbf{a}\\
   &\mathbf{B} \Rightarrow \mathbf{b}\\
   &\mathbf{C} \Rightarrow \mathbf{c}
   \end{aligned}$$

   is a type 0 grammar.

   Strings could be generated: $\varepsilon, \mathbf{abc}, \mathbf{aabbcc}, \mathbf{cabcab}, \ldots$

2. **Type 1:**

   In type 1 grammar, the only restriction is that right side should not contain fewer symbols than left one. They are also known as **context sensitive grammars**. The form of the rule in type 1 grammars is "$\alpha < B > \gamma ::= \alpha\beta\gamma$" where $B \in N$ and α, β, γ are the vocabularies together with the rule that β is not an empty string.

   For example, the grammar G of the language $\{a^n b^n c^n\}$is context sensitive.

   Non-terminals $N = \{S, A, B\}$, terminals $\Sigma = \{a, b, c\}$, start $S = \{S\}$and rules $P =$

   $$\begin{aligned} &\mathbf{S} \Rightarrow \mathbf{abc} \mid \mathbf{aAbc} \mid \varepsilon \\ &\mathbf{Ab} \Rightarrow \mathbf{bA} \\ &\mathbf{Ac} \Rightarrow \mathbf{Bbcc} \\ &\mathbf{bB} \Rightarrow \mathbf{Bb} \\ &\mathbf{aB} \Rightarrow \mathbf{aa} \mid \mathbf{aaAc} \end{aligned}$$

   Strings could be generated: $\varepsilon, \mathbf{abc}, \mathbf{aabbcc}, \mathbf{aaabbbccc}, \ldots$

3. **Type 2:**

   In another saying **context free grammars** require that left side be a single nonterminal producing rules of the form "$< A >::= \alpha$" where $A \in N$ and $\alpha \in \{N \cup \Sigma\}$.

   For example, the grammar G of the language $\{a^n b^n\}$is context free.

   Non-terminals $N = \{S, A\}$, terminals $\Sigma = \{a, b\}$,start $S = \{S\}$and rules $P =$

   $$\begin{aligned} &\mathbf{S} \Rightarrow \varepsilon \mid \mathbf{A} \\ &\mathbf{A} \Rightarrow \mathbf{aAb} \mid \mathbf{ab} \end{aligned}$$

   Strings could be generated: $\varepsilon, \mathbf{abc}, \mathbf{aabb}, \mathbf{aaabbb}, \ldots$

4. **Type 3:**

   Type 3 grammars are the most restricted ones, also known as **regular grammars** permitting only a terminal or a terminal followed by one nonterminal on the right side. The form of the rule is "$< A >::= a$" or "$< A >::= a < A >$" where $A \in N$ and $a \in \Sigma$.

   For example, the grammar G with non-terminals $N = \{S, A\}$, terminals $\Sigma = \{a, b, c\}$, start $S = \{S\}$and rules $P =$

$$\begin{aligned} \mathbf{S} &\Rightarrow \mathbf{aS} \mid \mathbf{bA} \\ \mathbf{A} &\Rightarrow \varepsilon \mid \mathbf{cA} \end{aligned}$$

is regular.

Strings could be generated: $\mathbf{b}, \mathbf{abc}, \mathbf{abcc}, \mathbf{aabc}, \mathbf{aaabcc}$...

Definition 3.1.8. Let Σ be an alphabet. R is a **regular expression**, if it is in the from of:

- a, for any $a \in \Sigma$, denoting any literal character
- ε, denoting empty string
- $\varnothing$, denoting empty set

additionally; if R_1 and R_2 are regular expressions, then

- $(R_1 \cup R_2)$ (union)
- $(R_1 \circ R_2)$ (concatenation)
- R_1^* (Kleene Star)

are also regular expressions with the prior condition that number of application of operations should be finite.

For example, regular expression $a^*b^* \mid c$ corresponds to the set $\{\varepsilon, c, ab, abab, abababbbb, ...\}$

Semantics

The concept semantics was firstly used by M. Breal in his “Studies in the science of meaning”, published in 1900. At that time, the concept was used to mean “the study of the way that word changes its meaning”. However, nowadays in linguistics or in computer science, the notion points out the meaning more generally that “the study of the meaning“.

The notion of semantics of programming languages deals with the meaning of instructions in a language unlike the one called syntax which only checks if the expression is composed of valid or invalid symbols. The question that semantics look for solutions is that what, actually, the programs do when they are executed. On the other hand, it is hard to formalize the semantics of programming languages unlike grammatical issues of it. [16]

Formalization of semantics are used to increase comprehensibleness of general behaviors of programs and they also build a mathematical model which is useful for program analysis and verification. [17]

There are three kinds of semantics: Operational, Denotational and Axiomatic Semantics.

1. **Operational Semantics**:

   Operational semantics describe the meaning of the programming language by pointing out how it executes on an abstract machine.

   The Structured Operational Semantics notation:

   $$\sigma(e) \Rightarrow v$$

   is a statement about the computation of value v from expression e in state σ.

   - if e is a constant, then v is corresponding value in semantic domain.
   - If e is a variable, then v is the value of variable in state σ.
   - v can also be a new state if e has side-effects.

   An execution rule

   $$\frac{\textbf{premise}}{\textbf{conclusion}}$$

   means "if premise is true, then conclusion is true". [18]

   Example 3.1.9. *An execution rule for addition:*

   $$\frac{(\sigma(e_1) \Rightarrow v_1) \,\wedge\, (\sigma(e_2) \Rightarrow v_2)}{\sigma(e_1 + e_2 \Rightarrow v_1 + v_2}$$

2. **Denotational Semantics**:

   Denotational semantics define the meaning of programming languages by mathematical concepts. They also provide deep and widely applicable techniques for various languages.

   Example 3.1.10. *Composing denotations of a programming language. Consider the expression "3 + 2". In this case, compositionality means "3 + 2" with respect to separate meanings of "3", "2" and "+".*

- *meaning for types of the language, $[\![T]\!]$ is standing for the representation of type T, i.e $[\![nat]\!] = \mathbb{N}_\perp$ for natural numbers.*
- *meaning for typing contexts, $[\![x_1 : T_1, ..., x_n : T_n]\!] = [\![T_1]\!] \times ... \times [\![T_n]\!]$, i.e $[\![x : nat, y : nat]\!] = \mathbb{N}_\perp \times \mathbb{N}_\perp$.*
- *meanings of each program-fragment-in-typing-context, Let P be a program fragment of type σ, in typing context $\ulcorner$, then the meaning of this program-in-typing-context must be a continuous function $[\![\ulcorner \vdash P : \sigma]\!] : [\![\ulcorner]\!] \to [\![\sigma]\!]$, i.e $[\![\vdash 15 : \sigma]\!]$ represents the constant function "15".*
- *the meaning of "3 + 2" is represented by compositions of three functions: $[\![\vdash 3 : \sigma]\!] : 1 \to \mathbb{N}_\perp$, $[\![\vdash 2 : \sigma]\!] : 1 \to \mathbb{N}_\perp$ and $[\![x : nat, y : nat \vdash 1 : x + y : nat]\!] : \mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$.*

3. **Axiomatic Semantics** :

   Axiomatic semantics give the meaning of a programming construct by axioms or proof rules in a program logic. They are more generally used in developing and verifying programs.

There are also different styles of semantics that are dependent on each other.

- In order to check the correctness of proof rules of axiomatic semantics, underlying denotational or operational semantics might be used.
- With respect to denotational semantics, to show the correctness of the implementation, it is needed to show the agreements of operational and denotational semantics.
- In order to verify an operational semantics, use a denotational semantics to get rid of implementation details which are not that important so that high-level computational behavior becomes easier to comprehend.

Classifications of Programming Languages

There are many programming languages standing for different programming manners available, nowadays. For that reason, the classifications of them could be done via many ways. However, the most fundamental way is to do the categorization by using programming paradigms of each.

All of the programming paradigms provide different code execution strategies to the programmer. In this session, some of the most important ones such as imperative or the procedural, the declarative and the object oriented programming paradigms are explained and illustrated via some number of comparisons among them. In the mean time, it should be kept in mind that some of the programming languages embody features of different paradigms, at the same time.

1. **Imperative Programming Paradigm** :

   Imperative programming paradigm is defined by programming together with a mutable state and some number of ordered commands. Each program has a start state, list of commands to complete and a set of final states which should not be an empty one. In that sense, this manner of programming could be characterized by finite state machine computation model.

   Imperative programming approach is called procedural if it is equipped with procedures that might be functions, subroutines and methods. The idea is to divide the whole program into relatively smaller pieces, so called procedures, in order to make it easier for programmers to understand and maintain program structure.

   Programming languages like C, Pascal, Algol, Cobol are well-known examples of imperative programming paradigm.

2. **Declarative Programming Paradigm** :

   In contrast to imperative manner, in declarative programming paradigm, programmers do not have to prescribe *"how to do"* in terms of sequences of actions to be taken. On the other side, *"what to do"* always have to be described in both cases.

   Declarative manner is divided into two sub-manners namely functional and logical programming paradigms. In functional programming, which is expressed in much more detail in the next section, the computations are expressed as the evaluations of mathematical functions. Unlike in imperative programming, values are never modified in functional manner. Instead of this, values are transformed into another values and computations are performed via applying functions to these values. For example; $(*\mathbf{2}\ \mathbf{5}) = \mathbf{10}$.

   In logical programming paradigm, computations are expressed solely in terms of mathematical logic. In comparison to the functional paradigm which emphasizes the idea of function applications, the logical manner focuses on the predicate logic for which the relations are the basis. They are generally used to solve problems when the problem can not obviously be solved by functional manner.

Prolog is the very well-known example to the logical programming languages. Haskell and Scheme could be given as examples to functional programming languages. [19]

3. **Object Oriented Programming Paradigm** :

   The newest paradigm compared to the others is object oriented one. In this programming paradigm, the designer specifies both the data structures and the types of operations that can be applied to those data structures. These data together with the applicable operations construct the notion of objects.

   Fundamental characterization of object oriented paradigm was done by Alan Kay, as follows:

   - Everything in the language is modeled as objects.
   - Communications of objects in the model is done by message passing method.
   - Similar types of objects are instances of the contexts called classes.
   - The relationship between classes are giving birth to the notion called inheritance.

   Unlike imperative programming in which data are passive instead procedures are active, object oriented manner combines data with procedures, by this way, data take the active role.

   Programming languages such as C#, C++ and Java are well-known examples of object oriented programming paradigm.

Comparisons of Programming Languages

1. **Imperative Programming vs Declarative Programming**

   In declarative programming, the programmer only specifies *"what to do"* but the data organization and sequencing are done by the interpreter, in other words, *"how to do"* part is not the job of programmer. On the other hand, in imperative programming, both *"what and how to do"* should be specified by the programmer.

   Imperative programming languages were constructed on the basis of Turing Machine computational model while functional ones are relying on Lambda Calculus computational model. In imperative programming, the fundamental concepts are *variables* that are possible to modify numerously, representing cells on the tape (memory), together with the operations of *assigning* values to that variables and *storing* them into a specific hard drive, if exists. Besides, the approach to accomplish a desired goal is *iteration* and that process is called *execution*. While in functional programming, the concept *variable* could be *assigned* only once that is also called binding or initializing. Accomplishing the goal is done via *recursive* model and the process is called *evaluation* instead of execution.

2. **Imperative Programming vs Object Oriented Programming**

   In imperative programming, the emphasis is on procedural abstraction while in object oriented manner, data is the main focus of abstraction. Imperativeness brings the obligatory of top-down design manner due to the stepwise refinement, however in object oriented one, both aspects of top-down and bottom-up could be utilized.

   Object oriented design manner provides data re-usability for that reason, it is suitable to use this manner in large programming aspects, on the other side of the comparison, imperative programming do not show the property of re-usability so that they are suitable for short coding aspects.

3. **Declarative Programming vs Object Oriented Programming**

   Rather than the differences that are already explained in the above sessions the one might be given in this part is that especially in pure functional programming languages; functions are easier to execute and parallelize due to having lack of side effects, compared to object oriented design methodology.

3.2 Functional Programming

Functional programming is the style of programming in which neither procedures nor objects are the fundamental building blocks. Instead, everything is done via mathematical function applications.

In the above session, functional programming has been mentioned without examining thoroughly. However, in this section, functional manner of coding is evaluated from the point of view of Lambda Calculus and then functional programming languages are being counted as categories that are defined and illustrated in detail in the mathematical background 1.1 part of the thesis.

Informally, the term **computation** refers to transfer information from an implicit form to an explicit form by doing any kind of calculation. However, the notion computation has solely no formal meaning, unless a formal computational model is defined.

The concept of computing could be formalized in several different ways, that is by means of several different computational models such as Turing Machine and Lambda Calculus. Now then, it could be specified that the term **programming** carries the meaning that expressing a specific computation on a computational model by the help of programming languages. For example, as described in comparisons of programming languages session, programming in any imperative language means a sequence of instructions for a Turing Machine while in functional programming languages, it specifies a collection of functions to be evaluated by the rules defined in Lambda Calculus.

3.2.1 Definition, Syntax and Semantics of λ - Calculus

Lambda Calculus was invented by Alonzo Church in 1936 in order to formalize computable mathematical functions. In other words, it is a functional language based on mathematical function theory. As also mentioned in the previous sessions, today's implementation area of lambda calculus is to develop functional programming languages. It is also equipped with the rules to make functional programming languages pure.

The λ - Calculus Syntax

In λ calculus, we have only anonymous functions given by:

$$e \ ::= \ x \,|\, e_1 \ e_2 \,|\, \lambda x.e$$

which are called $\lambda -$ **expressions** where

x	represents variables.
$e_1 \ e_2$	standing for function applications.
$\lambda x.e$	for function abstractions.

The abstraction $\lambda x.e$ is an anonymous function with variable x and function body e.

Example 3.2.1. *In anonymous function $\lambda x.x^3$, the variable is x and return value is the cube of x.*

In application $e_1\ e_2$; e_1 represents the function that is applied to expression e_2.

Example 3.2.2. *In this anonymous function application* $(\lambda x.x^3)$ 5*, the variable x takes the value 5 and then anonymous function* $\lambda x.x^3$ *calculates cube of 5 and returns 125 as the application result.*

Note 3.2.3. *Lambda expressions are extensible as far to the right as possible. However, consider that function application is left associative.*

Example 3.2.4. $(\lambda x.x)\ y\ z$ *is the same as* $((\lambda x.x)\ y)\ z$.

Note 3.2.5. *In lambda calculus, a variable is* **bound** *if it can be linked to a lambda abstraction. Other variables that are not bound are called* **free***.*

Example 3.2.6. *In lambda expression* $\lambda x.xy$*;variable x is bound while y is free.*

Definition 3.2.7. $\alpha -$**conversion** renames bound variables. For instance, $\lambda x.x$ can be lambda converted into $\lambda y.y$.

Definition 3.2.8. $\lambda x.x$ and $\lambda y.y$ are called $\alpha -$**equivalent**.

The λ - Calculus Semantics

$\beta -$Reduction

As defined earlier, function application $(\lambda x.e_1)\ e_2$, applies the function $(\lambda x.e_1)$ to the expression e_2. Consider that application operation could be also done by replacing the free occurrences of variable x in expression e_1 by expression e_2 which is called **substitution** and represented as $e_1\{e_2/x\}$.

Example 3.2.9. $(\lambda x.x + 10)5$ *could be written as* 15.

Therefore, it is obvious now to remark that the expressions $(\lambda x.e_1)\ e_2$ and $e_1\{e_2/x\}$ are equivalent, which is known as $\beta -$**equivalence**. The operation of rewriting $(\lambda x.e_1)\ e_2$ as $e_1\{e_2/x\}$ is called $\beta -$**reduction**.

So far, the only evaluation methodology known is β-reduction, hence the crucial decision is ordering the evaluations of function applications. With regard to this, we have different variations of semantics such as *call by value* and *call by name* semantics.

Call by Value Semantics

In call by value semantics, the most important concern is:

- Arguments of the functions are always evaluated to their **values** form before the application of it.

  Note 3.2.10. *In applied lambda calculus,* **value** *is an expression that can not be reduced any more. [20]*

Here are the operational semantics for call-by-value execution of the lambda calculus: [21]

$$x \xrightarrow{cbv} x$$

$$(\lambda x.e) \xrightarrow{cbv} (\lambda x.e)$$

$$\frac{e_1 \xrightarrow{cbv} (\lambda x.e) \qquad e_2 \xrightarrow{cbv} e_2^{'} \qquad e[e_2^{'}/x] \xrightarrow{cbv} e^{'}}{(e_1\ e_2) \xrightarrow{cbv} (e^{'})}$$

For instance; here is the call by value reduction of lambda expression $(e_1\ e_2)$ to $(e^{'})$:

$$e_1\ e_2 \xrightarrow{\beta\ \mathbf{reduction\ on\ e_2}} e_1\ e_2^{'}$$

$$e_1 \equiv (\lambda x.e)$$

$$e_1\ e_2^{'} \rightarrow (\lambda x.e)\ e_2^{'}$$

$$(\lambda x.e)\ e_2^{'} \xrightarrow{\beta\ \mathbf{equivalence}} e[e_2^{'}/x]$$

$$e[e_2^{'}/x] \xrightarrow{\beta\ \mathbf{reduction\ on\ e_2^{'}}} e^{'}$$

In the sense that; before the application of function e_1 which is $(\lambda x.e)$, the argument, e_2 is evaluated into its value form (impossible to reduce any more) by using β-reduction which is represented with $e_2^{'}$ and then substitution is done as the last step of the evaluation to get the result of function application, $e^{'}$.

Example 3.2.11. *Here is a basic example to call by value semantics. Notice that function is applied after the argument is reduced. [22]*

$$(\lambda f.\ f\ 10)\ ((\lambda x.\ x\ x)\ \lambda y.y+2) \longrightarrow (\lambda f.\ f\ 10)\ ((\lambda y.y+2)\ (\lambda y.y+2))$$

$$(\lambda f.\ f\ 10)\ ((\lambda y.y+2)\ (\lambda y.y+2)) \longrightarrow (\lambda f.\ f\ 10)\ (\lambda y.y+2+2)$$

$$(\lambda f.\ f\ 10)\ (\lambda y.y+2+2) \longrightarrow (\lambda f.\ f\ 10)\ (\lambda y.y+4)$$

$$(\lambda y.y+4)(10) \longrightarrow 4+10 \longrightarrow 14$$

Call by Name Semantics

In call by name semantics, the concern to be noticed is:

- The function is applied to its argument before the argument's evaluation. In other words, the argument is not reduced to its value form before the application of the function.

Below is the operational semantics for call-by-name execution of the lambda calculus: [21]

$$x \xrightarrow{cbn} x$$

$$(\lambda x.e) \xrightarrow{cbn} (\lambda x.e)$$

$$\frac{e_1 \xrightarrow{cbn} (\lambda x.e) \qquad e[e_2/x] \xrightarrow{cbn} e'}{(e_1\ e_2) \xrightarrow{cbn} (e')}$$

For instance; here is the call by name reduction of lambda expression $(e_1\ e_2)$ to (e'):

$$e_1 \equiv (\lambda x.e)$$

$$e_1\ e_2 \rightarrow (\lambda x.e)\ e_2$$

$$(\lambda x.e)\ e_2 \xrightarrow{\beta\ \textbf{equivalence}} e[e_2/x]$$

$$e[e_2/x] \xrightarrow{\beta\ \textbf{reduction on}\ \mathbf{e_2}} e'$$

Example 3.2.12. *Here is a basic example to call by name semantics. Notice that function is applied before the argument is reduced. [22]*

$$(\lambda f.\ f\ 10)\ ((\lambda x.\ x\ x)\ \lambda y.y+2) \longrightarrow ((\lambda x.\ x\ x)\ \lambda y.y+2)(10)$$

$$(\lambda x.\ x\ x)\ \lambda y.y+2)(10) \longrightarrow ((\lambda y.y+2)\ (\lambda y.y+2))(10)$$

$$((\lambda y.y+2)\ (\lambda y.y+2))(10) \longrightarrow (\lambda y.y+2+2)(10)$$

$$(\lambda y.y+2+2)(10) \longrightarrow (\lambda y.y+4)(10)$$

$$(\lambda y.y+4)(10) \longrightarrow 4+10 \longrightarrow 14$$

3.2.2 Identity Function, Function Application, Function Composition and Recursive Functions

In this session of the thesis, possible operations that could be implemented by using lambda calculus semantic rules are given and illustrated via lambda calculus syntax.

Identity Function

Identity function is the one that returns the given argument as also its result.

$$\mathbf{id} = (\lambda\mathbf{x}.\ \mathbf{x})$$

For example;

$$(\lambda x.\ x)\ M \equiv M \text{ for every term } M.$$

Selection Function

Selection functions take two input as arguments and returns first or second argument, depending on the user’s choice, as its result.

$$\mathbf{fst} = (\lambda\mathbf{x}.\lambda\mathbf{y}.\ \mathbf{x})$$

For example;

$$(\lambda x.\lambda y.\ x)\ M\ N \equiv M \text{ for every term } M \text{ and } N.$$

$$\mathbf{snd} = (\lambda \mathbf{x}.\lambda \mathbf{y}.\ \mathbf{y})$$

For example;

$$(\lambda x.\lambda y.\ y)\ M\ N \equiv N \text{ for every term } M \text{ and } N.$$

Function Application

The lambda term application takes a function and an argument in order to apply that function on that lambda term.

$$\mathbf{apply} = (\lambda \mathbf{f}.\lambda \mathbf{x}.\ \mathbf{f}\ \mathbf{x})$$

For example;

$$(\lambda f.\lambda x.\ f\ x)\ M\ N \equiv M(N) \text{ for every function } M \text{ and argument } N.$$

Note 3.2.13. *The number of function application could be increased, arbitrarily.*

Example 3.2.14. *Here are the lambda expressions for the application of the same function twice and three times.*

$$\mathbf{twice} = (\lambda \mathbf{f}.\lambda \mathbf{x}.\ \mathbf{f}\ (\mathbf{f}\ \mathbf{x}))$$

$$\mathbf{three\_times} = (\lambda \mathbf{f}.\lambda \mathbf{x}.\ \mathbf{f}\ (\mathbf{f}\ (\mathbf{f}\ \mathbf{x})))$$

Function Composition

If there are functions $f : A \rightarrow B$ and $g : B \rightarrow C$, then there exists another function $h = g \circ f : A \rightarrow C$ called compositions of f and g.

In lambda calculus, the term composition takes two functions and an argument as inputs in order to first take the composition of the functions and then apply the composition to the argument. These operations are represented in lambda calculus syntax as:

$$\textbf{composition} = (\lambda\mathbf{g}.\lambda\mathbf{f}.\lambda\mathbf{x}.\ \mathbf{g}\ (\mathbf{f}\ \mathbf{x}))$$

For example;

$$(\lambda g.\lambda f.\lambda x.\ g\ (f\ x))\ M\ N\ a \equiv M(N(a))\ \text{for functions}\ M,N\ \text{and argument}\ a.$$

Self Application

Self application is a lambda calculus term that is not logical from the point of view of mathematical function theory. It, roughly, takes a function and applies the input function to itself.

$$\textbf{self\_application} = (\lambda\mathbf{x}.\ \mathbf{x}\ \mathbf{x})$$

For example;

$$\text{id id} = (\lambda x.\ x)\ \text{id} = \text{id}$$

$$\text{snd snd} = (\lambda x.\lambda y.\ y)\ \text{snd} = (\lambda y.\ y) = \text{id}$$

The Y Combinator

Y combinator is a kind of fixed point combinator in lambda calculus that was discovered by Haskell B. Curry. It, more or less, looks like the self application term apart from the involvement of an additional function represented by **t**. Here is the lambda calculus expression of Y- combinator:

$$\mathbf{Y}\ t = (\lambda x.\ t\ (x\ x))(\lambda x.\ t\ (x\ x))$$

$$(\lambda x.\ t\ (x\ x))(\lambda x.\ t\ (x\ x)) \xrightarrow{\beta\ \textbf{reduction}} t\ ((\lambda x.\ t\ (x\ x))(\lambda x.\ t\ (x\ x)))$$

$$t\ ((\lambda x.\ t\ (x\ x))(\lambda x.\ t\ (x\ x))) = t\ (\mathbf{Y}\ t)$$

Therefore, **Y** *t* equals to the function *t* applied to itself. Besides, it could be repeatedly unfold.

$$(\mathbf{Y}\ t) = t\ (\mathbf{Y}\ t) = t\ (t\ (\mathbf{Y}\ t)) = t\ (t\ (t\ \mathbf{Y}\ t)) = \ldots$$

This equality provides an crucial feature in order to define **recursive functions** in lambda calculus.

For instance, lets have look at the lambda calculus definition and evaluation of the recursive factorial function.

1. **Definition** :

$$t = \lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$$

2. **Evaluation of Factorial 3** :

   We already know that $(\mathbf{Y}\ t) = t\ (\mathbf{Y}\ t)$, therefore: [23]

$$(\mathbf{Y}\ (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)))\ 3 = (t\ (\mathbf{Y}\ t))3$$

$$(t\ (\mathbf{Y}\ t))3 = ((\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1))\ (\mathbf{Y}\ t))3$$

   $((\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1))\ (\mathbf{Y}\ t))3 = ((\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * (\mathbf{Y}\ t)(n-1)))3$

   $((\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * (\mathbf{Y}\ t)(n-1)))3 = ((\ \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * (\mathbf{Y}\ t)(3-1)))$

$$t\ (\mathbf{Y}\ t)(3) = 3 * (\mathbf{Y}\ t)(2)$$

$$3 * (\mathbf{Y}\ t)(2) = 3 * t\ (\mathbf{Y}\ t)(2)$$

$$t\ (\mathbf{Y}\ t)(2) = 2 * (\mathbf{Y}\ t)(1)$$

$$2 * (\mathbf{Y}\ t)(1) = 2 * t\ (\mathbf{Y}\ t)(1)$$

$$t\ (\mathbf{Y}\ t)(1) = 1 * (\mathbf{Y}\ t)(0)$$

   consequently;

   $(\mathbf{Y}\ t)(0) = 1 \longrightarrow t\ (\mathbf{Y}\ t)(1) = 1 \longrightarrow t\ (\mathbf{Y}\ t)(2) = 2 \longrightarrow t\ (\mathbf{Y}\ t)(3) = 6.$

3.2.3 Simulation of Turing Machines using Lambda Calculus and Vice Versa

As declared by Church in his thesis or conjuncture; the effectively Turing computable functions are also definable in the pure lambda calculus. That thesis still cannot be proved due to the lack of informal definition of the notion “effectively computable function”. However, since all methods developed to be computed by Turing Machines have been proved to be no more powerful than the lambda calculus. On the other side, undecidability aspect of Turing Machines, so called Halting Problem that states that there is no algorithmic way to decide whether or not a Turing Machine will stop running on a given input, also exists in Lambda Calculus. There exist some lambda expressions in Lambda Calculus for which it is not possible to find a reduction combination to end the evaluation such as:

$$\lambda f.\ (\lambda x.\ (f\ (x\,x))\ \lambda x.\ (f\ (x\,x))).$$

Here are the simulations to demonstrate equivalence of both models in computability:

1. Everything computable by λ Calculus can be computed using the Turing Machine: In order to simulate λ calculus with Turing Machine:

    - The initial tape is filled up with initial lambda expressions.
    - Reduction rules in finite number could be represented by finite state automaton involved in Turing Machine.

      Note 3.2.15. *Informally, a* Turing Machine *consists of a finite state automaton as control unit that includes states, alphabets and state transition functions; a tape on which operations are happening and a head that moves over the tape to read or write symbols on it.*

    - In order to stop the execution, Turing Machine has to encounter with the final lambda expression on the tape; otherwise, it continues forever that means that β reductions never end.

2. Everything computable by Turing Machine can be computed with λ Calculus. In order to simulate Turing Machine with λ calculus: [21]

    - Processing in Turing Machines is done via ways to make decisions that are if then else condition sentences.

$$\textbf{if\_then\_else} = \lambda x.\lambda y.\lambda z.\ ((x)\ y)\ z$$

$$\textbf{true} = \lambda x.\lambda y.\ x$$

$$\textbf{false} = \lambda x.\lambda y.\ y$$

- The arithmetical and control unit operations that could be implemented over a Universal Turing Machine, are also possible to represent in Lambda Calculus.

  Representations of numbers and arithmetical operations:

  $$\mathbf{0} \equiv \lambda f\ x.\ x, \mathbf{1} \equiv \lambda f\ x.\ f(x), \mathbf{2} \equiv \lambda f\ x.\ f\ (f(x))\ ...$$

  $$\textbf{successor} \equiv \lambda n\ f\ x.\ f(n\ f\ x)$$

  $$\textbf{add} \equiv \lambda xy.\ x\ \textbf{successor}\ (y)$$

  $$\textbf{multiply} \equiv \lambda xy.\ x\ \textbf{add}\ (y)\ 0$$

  $$\textbf{power} \equiv \lambda xy.\ y\ x$$

  $$\textbf{predecessor} \equiv \lambda nfx.\ n\ (\lambda gh.\ h\ (g\ f))\ (\lambda u.\ x)\ (\lambda u.\ u)$$

  $$\textbf{subtraction} \equiv \lambda xy.\ y\ \textbf{predecessor}\ (x)$$

  Here are the representations of logical expressions:

  $$\textbf{logical\_and} = \lambda x.\lambda y.\ x\ y\ x$$

  $$\textbf{logical\_or} = \lambda x.\lambda y.\ x\ x\ y$$

  Logical and: ('true' 'and' 'false')

  - $(\lambda x.\lambda y.\ x\ y\ x)$ true false $\xrightarrow{\beta\ \textbf{reduction}}$ true false true
  - true false true = $(\lambda x.\lambda y.\ x)$ false true $\xrightarrow{\beta\ \textbf{reduction}}$ false

  Logical or: ('true' 'or' 'false')

  - $(\lambda x.\lambda y.\ x\ x\ y)$ true false $\xrightarrow{\beta\ \textbf{reduction}}$ true true false
  - true true false = $(\lambda x.\lambda y.\ x)$ true false $\xrightarrow{\beta\ \textbf{reduction}}$ true

- Instead of Turing Machine tapes, mutable lists are used in Lambda Calculus.

  A list could be either a pair whose second element is a list or a null.

$$\mathbf{pair} \equiv \lambda xyf.\ f\ x\ y$$

$$\mathbf{first} \equiv \lambda x.\ x\ \mathbf{true}$$

$$\mathbf{second} \equiv \lambda x.\ x\ \mathbf{false}$$

$$\mathbf{null} \equiv \lambda z.\ z\ (\lambda xy.\ \mathbf{false})$$

As shown above, Turing Machines and Lambda Calculus are equivalent in terms of computability in automation theory. That also proves that programming languages based on Turing's computational model and Lambda Calculus computational model are intuitively equivalent.

3.2.4 Functional Programming Languages as Categories

Functional programming languages involve:

- Primitive data types that are already involved in programming language, not user created ones.
- Constants for each type.
- Operations that are functions between data types.
- Data type and operation constructors in order to allow user to create new data types and operations.

Together with the above mentioned fundamental features, if two assumptions and an "innocent" feature is added on, then any functional programming language L corresponds in a canonical way to a category $C(L)$.

Assumptions:

1. There should be a do-nothing operation id_A for each type A in order to satisfy the identity morphism rule in mathematical category definition.
2. The language should have a composition constructor that takes two operations as inputs. One of which, namely f, takes something in type A as input and returns something in type B as output while the other one, namely g, takes something in type B as input and returns something in type C, since composition constructor is expected to return another operation, namely $h = g \circ f$, takes

something in type A and returns something in type C. Besides, composition should be associative. By this way, the composition of morphisms feature of category theory is also satisfied.

The Innocent addition:

1. The type called 1 is the additional type which has the property that from every type A there exists a unique operation to 1. That means that for each constant c of type A, there is an arrow $c : 1 \to A$. By this way, the constants are involved in the set of operations, in other words, they will no longer appear as separate data.

If the above mentioned features are added, then a functional programming language L satisfies the rules to have the category structure $C(L)$ for which:

- The objects of $C(L)$ are the types of L.
- The arrows of $C(L)$ are the operations of L.
- Input and output types of any operation in L are domains and co-domains of the morphisms in $C(L)$.
- Composition of the morphisms in $C(L)$ are the composition operation in L in the reverse order.
- Identity morphisms in $C(L)$ are the do-nothing operations in L.

Example 3.2.16. *A language L with three primitive data types:* BOOL *(booleans),* CHAR *(characters) and* NAT *(natural numbers) together with following properties constructs a category $C(L)$:*

1. NAT *has a constant;* $0 : 1 \to \mathrm{NAT}$ *and an operation; successor* $: \mathrm{NAT} \to \mathrm{NAT}$.
2. BOOL *has two constants;* $\mathrm{true},\ \mathrm{false} : 1 \to \mathrm{BOOL}$ *and a 'negation' operator;* $\neg : \mathrm{BOOL} \to \mathrm{BOOL}$.
3. CHAR *has one constant;* $c : 1 \to \mathrm{CHAR}$ *for each character.*
4. *L also has two type conversion operations;* $\mathrm{ord} : \mathrm{CHAR} \to \mathrm{NAT}$ *and* $\mathrm{chr} : \mathrm{NAT} \to \mathrm{CHAR}$ *where* $\mathrm{ord} \circ \mathrm{chr} = id_{\mathrm{CHAR}}$. *For sure, composition operation and identity morphisms for each data type are defined.*

The objects of category $C(L)$ are CHAR*,* BOOL*,* NAT *and 1.*

The morphisms are combinations of all operations. For instance:

$$\mathrm{chr} \circ \mathrm{successor} \circ \mathrm{ord} \circ \mathrm{chr} \circ \mathrm{ord} : \mathrm{CHAR} \rightarrow \mathrm{CHAR}$$

3.3 Haskell Programming Language and its Type System

Haskell is a *purely functional*, *lazy* and *polymorphically, statically and strongly typed* programming language. The name Haskell is coming from the name of Haskell Brooks Curry who played very crucial role in the foundation of functional programming languages. Besides, it relies on Lambda Calculus like the other functional programming languages, actually for that reason, it uses the symbol λ as its logo.

Haskell offers some number of advantages to its programmers such as:

- Easily maintainable and shorter codes.
- Reliability.
- Nearly, no semantic gap between programmer and the language.
- A wide-range of programmer community.

The notion *pure* in functional programming requires following two features to be satisfied:

1. The rule *referential transparency* should hold. That means that any function in the language has to return the same result for the same input query, independent of current state.
2. No *side effects* should be involved in the language which indicates the notion that state modification and observable interaction with outside world are restricted.

In addition, *laziness* carries the meaning that Haskell never evaluates functions unless they are forced to return a result.

It is also given in the definition that Haskell is a *polymorphically statically typed* language. In order to be able to explain the issue, type systems of programming languages should be pointed out.

Type Systems

A *data type* is an agenda of the data which fundamentally consists of up to below given three features:

1. The physical representation of the data which involves how the data is stored.
2. The operations that can or can not be implemented over the data.
3. Some outside program controls that are able to or surrounded with the right to make changes on it.

For instance, some basic data types that are used nearly all of the programming languages are *integers*, *characters*, *booleans* and etc...

In addition to the above definition, we can say a language *L* is *strongly typed*, if it is not allowing automatic type conversions such as converting a float type to integer type. Otherwise, it is called *weakly typed* which means type conversions are allowed to be done, although they cause information losses.

Now, we can speak of the term *type systems* whose fundamental aim is to prevent any kind of errors may happen in the running or compiling phases of the program. They are generally composed of some rules in order to check the consistency of programs. This process of verifying the consistency or checking if the constraints of *data types* are obeyed or not is called *type checking*.

There are two kinds of type checking:

1. *Static Type Checking*: A programming language performs *static typing* if type checking is done in compilation phase. Error detection before run time phase could be spoken as an advantage of this kind.
2. *Dynamic Type Checking*: A programming language performs *dynamic typing* if type checking is done in execution phase. The advantage is that utilizations of the functions which execute on arbitrary data are permitted in this kind. This option is also allowed in static type checking but together with the usage of some algebraic data type implementation which brings additional effort in the construction stage.

Furthermore, Haskell's type system also permits the usages of *polymorphic types*. The term *polymorphism* refers to the types that are universally quantified in some way, over all types.

In the light of such information given above, lets also illustrate Haskell's type system in order to convert the abstract definitions into concrete working examples.

Basic Haskell Types

Haskell has a number of basic types of values, including:

- Int ⟶ for 32-bit integers
- Integer ⟶ for arbitrarily long integers
- Char ⟶ for single characters
- Bool ⟶ for logical values
- String ⟶ for list of characters
- Float ⟶ for floating point numbers

Not only values but also functions have types in Haskell. For instance, *lowercase* and *uppercase* functions take *Char* value type as input and returns the same value type as output. Here are the *type signatures* of *lowercase* and *uppercase* functions:

$$lowercase, uppercase :: Char \rightarrow Char$$

In Haskell, for any functions with multiple arguments, anyone can distinguish between the input and output types by using the idea that in type signatures of these functions, the last type represents output value while others are standing for input values.

For example:

$$anonymous :: Char \rightarrow Int \rightarrow Int$$

In the signature of the above anonymous function; there are two value types for inputs: *Char* and *Int* while the returned value type is *Int*.

The other important notion in type theory is *type variables*. They range over all types defined in the programming language.

For instance, lets consider the *fst* function in Haskell whose type signature is:

$$fst :: (a, b) \to a$$

Function takes two types which are not obliged to be different (a and b might have the same types) and returns the value of the first type. However, the crucial thing here is that a and b could be any of the types involved in the programming language. It could be, obviously, inferred that $fst\ (1, \text{“}abba\text{”})$ will return 1 as its result.

Furthermore, some types gather under the same root in order to express a specific behavior. These new constructions are called *type classes* and they first appeared in Haskell Programming Language.

Here is an example to type classes in Haskell syntax:

$$\begin{array}{l}\textit{class}\ \ \textbf{Equality}\ \ a\ \ \textit{where}\\ \qquad \begin{array}{lll} eq & :: & a \to a \to Bool \qquad \text{— function returns ’true’ if inputs are the same}\\ neq & :: & a \to a \to Bool \qquad \text{— function returns ’false’ if inputs are the same}\end{array}\end{array}$$

A class Equality contains two function types each of which admits equality. The instance declaration could be done for all a types involved in the language. Let’s see, what does it look like if a is selected as *Integer*.

$$\begin{array}{l}\textit{instance}\ \ \textbf{Equality Integer}\ \ \textit{where}\\ \qquad \begin{array}{lll} eq & = & (x == y)\\ neq & = & not\ (x \mathbin{/}= y)\end{array}\end{array}$$

Apart from the primitive types that have been already stored in Haskell, any user is also able to create specific types for specific issues by using the keyword called **data**. There are two types of constructors.

1. Data Constructors:

   As the name suggests; data constructors are gathering in order to produce new data types.

   $$\textbf{data}\ \ Bool = True \mid False$$

The left part of the above assignment represents the new data type which is *Bool*. On the right hand side, we have two *value or data constructors* that are *True* and *False*, specifying different values might be involved in this new data type. Rather, the symbol — exemplifies logical *or* operator. Therefore, the above assignment could be read as: New data type *Bool* can have a value of *True* or *False*.

Here, the most important inference is that types are composed of data constructors.

For instance:

data *Int* = −2147483647 | ... | 0 | 1 | ... | 2147483647

values ranging from - 2147483647 to 2147483647 are the data constructors of the type *Int* which consists of 32 bits: 31 bits for data and 1 bit for the sign.

2. Type Constructors:

   A type constructor is a parameterized type definition used with polymorphic types.

   data *Maybe a* = *Nothing* | *Just a*

   In the above assignment, *a* is the type parameter, *Maybe* is the type constructor, on the other side; *Nothing* and *Just* are the data constructors of Maybe. In a sense, the Maybe type constructor can produce types as *Maybe Integer*, *Maybe Char*, *Maybe String*, etc.

   For instance; *Just 10* has the type of *Maybe Integer* while *Just 'x'* is having the type of *Maybe Char*.

In parallel with these information given so far, in the next chapter of the thesis, some special type constructors in Haskell are proved to satisfy the algebraic rules of functors and monads.

Chapter 4

Functors & Monads in Haskell

Category theoretic objects such as functors, natural transformations and monads can be represented in Haskell programming language. Especially, usages of monads bring a great number of features such as *purity* to Haskell.

In this chapter, the representations of all Category Theoretical notions in the language and also proofs that are demonstrating some type constructors of the Haskell language satisfy the algebraic rules of functors, natural transformations and monads are given and illustrated by Haskell code examples.

4.1 HASK Category of Haskell

In order to be able to define functors, natural transformations and monads, first of all, the category description should be done. The category of Haskell programming language is called **HASK** which involves *Haskell types* as objects and *Haskell functions* as morphisms.

As mentioned in the mathematical background part of the thesis, in order to construct a category, the identity morphism for each object and the associative composition operation must be included.

Fact 4.1.1. **HASK** *is a category.*

Proof 4.1.2. *In* **HASK** *, the identity morphism is defined and named as* id. *Furthermore, the composition function is also defined and represented with '.' symbol.*

1. $\forall A,B,C \in \text{Obj(HASK)}$ *and* $\forall f : A \rightarrow B, g : B \rightarrow C \in \text{Morph(HASK)}$*, there also exists*
   $g.f = h : A \rightarrow C \in \text{Morph(HASK)}$

2. $\forall A,B \in \text{Obj(HASK)}$ *and* $\forall f : A \rightarrow B \in \text{Morph(HASK)}$

   $$id_B \ (B \rightarrow B) \,.f \ = f \,.\, id_A \ (A \rightarrow A) \ = f$$

3. $\forall f,g,h \in \text{Morph(HASK)}$*:*

   $$(f.g).h \ = f.(g.h)$$

HASK *is now said to be a category; since all algebraic properties, for constructing a category, are satisfied.*

4.2 Functor and Monad Type Classes of Haskell

Haskell functional programming language has some number of type classes like Equality class that is described in the last part of the previous section. In order to represent category theoretic concepts, it also involves specific type classes such as Functor and Monad.

Here are the mentioned type classes with their category theoretic connections.

4.2.1 Functor Type Class in Haskell

```
class Functor F where
    fmap :: (a → b) → F a → F b
```

As visible in the above definition suggests, functor type class has a *fmap* method which takes a function whose domain is a type *a* and co-domain is another type *b* and a type constructor *F* with its argument type *a* as inputs and returns the same type constructor *F*, this time with its argument type *b*.

Informally, it can be thought like: The method *fmap* takes a function $(a \rightarrow b)$ and a container *F* in which an arbitrary value of type *a* is stored. Firstly, it opens the container, then applies the input function over the inside value which means inside type value of a is converted into another type b value and lastly type b value is returned inside the same container.

Formally, in order to be able to speak of that any instances f and g of type class **Functor** is representing the category theoretic functors, it should satisfy the rules that are originating from the underlying theory such as:

$$\begin{array}{ll} \mathit{fmap}\ \mathit{id} = \mathit{id} & \text{—}\, F_{1_A} = 1_{F(A)} \\ \mathit{fmap}\ (f.g) = \mathit{fmap}\ f\ .\ \mathit{fmap}\ g & \text{—}\, F(\beta) \circ F(\alpha) = F(\beta \circ \alpha) \end{array}$$

Note 4.2.1. *It should be also noted that a functor defined in Haskell programming language is an* **endo − functor** *whose domain is category* **HASK** *and co-domain category is* **func** *which is a subcategory of HASK.*

4.2.2 Monad Type Class in Haskell

There are two different ways to represents monads in Haskell language. That means that two different type class definitions could be done.

The first one more explicitly demonstrates category theoretic rules to be satisfied. It involves a functor class which has the *fmap* method representing the endo-functor and a monad class in which *natural transformation* methods are included.

```
class Functor m where
    fmap   :: (a → b) → m a → m b


class Monad m where
    join   :: m (m a) → m a    — corresponding to μ :: T^2 → T
    return :: a → m a          — corresponding to η :: 1_ℂ → T
```

In the above monad class definition, it is obvious that m is a functor class type constructor, we can suppose now that m is an endo-functor which is able to map any type a (could be a function type) from category *HASK* to *func* with the *fmap* method. We know that, $m \circ m$ is another functor in *HASK*; therefore, the mapping, namely *join* should be proven as a natural transformation via satisfying the rule that below diagram commutes for all $f : a \to b \in HASK$:

$$\begin{tikzcd}
m\ (m\ a) \arrow[r, "join_a"] \arrow[d, "fmap\ fmap\ f"'] & m\ a \arrow[d, "fmap\ f"'] \\
m\ (m\ b) \arrow[r, "join_b"] & m\ b
\end{tikzcd}$$

FIGURE 4.1: Natural Transformation Diagram for Haskell Types to be satisfied

which means:

$$(join_b \circ fmap\ fmap\ f)\ [m\ (m\ a)] = (fmap\ f \circ join_a)\ [m\ (m\ a)] = m\ b$$

Furthermore, the mapping called *return* should also be proven as a natural transformation by showing the commutativity given in Figure 4.2.

$$\begin{tikzcd}
\mathrm{id}_{HASK}(a) \arrow[r, "return_a"] \arrow[d, "f"'] & m\ a \arrow[d, "fmap\ f"'] \\
\mathrm{id}_{HASK}(b) \arrow[r, "return_b"] & m\ b
\end{tikzcd}$$

FIGURE 4.2: Natural Transformation Diagram for Haskell Types to be satisfied

In particular, this means:

$$(return_b \circ f)\ [a] = (fmap\ f \circ return_a)\ [a] = m\ b$$

After proving these two methods are representations of natural transformations, then monadic laws also should be demonstrated satisfying which exactly means proving commutativity of below diagrams in order to be able to speak of that any instance *m* of type class **Monad** is representing the category theoretic monads:

$$\begin{tikzcd}
m\ (m\ (m\ a)) \arrow[r, "join\ (fmap\ id)"'] \arrow[d, "fmap\ join"'] & m\ (m\ a) \arrow[d, "join"'] & m\ a \arrow[r, "return\ (fmap\ id)"'] \arrow[d, "fmap\ return"'] & m\ (m\ a) \arrow[d, "join"'] \\
m\ (m\ a) \arrow[r, "join"] & m\ a & m\ (m\ a) \arrow[r, "join"] & m\ a
\end{tikzcd}$$

FIGURE 4.3: Commutative Diagrams - Monad

which means:

1. $join \circ join\ (fmap\ id) = join \circ (fmap\ join)$
2. $join \circ return\ (fmap\ id) = join \circ (fmap\ return) = id$

   Actually, these two features are sufficient to prove Haskell Monad class instances are the representations of category theoretic monads, however, in some literatures, two additional equalities are shown to be needed. Indeed, additional laws are not requirements but they are the expected things about how monads behave.
3. $return \circ f = (fmap\ f) \circ return$
4. $join \circ (fmap\ (fmap\ f)) = (fmap\ f) \circ join$

Here is the second way to implement monads in Haskell via using only *return* and *bind* methods:

```
class Monad m where
    bind     :: m a → (a → m b) → m b
    return   :: a → m a
```

Implementation could be done in the way shown above, because of the reason that both *fmap* and *join* methods are involved in *bind* method.

Informally, *bind* takes a container ($m\ a$) and a morphism ($a \rightarrow m\ b$) as inputs; then it opens the container, applies the function over the type constructor a, the result of which is ended in another container, namely $m\ b$. Therefore, at this level, we have a new type constructor b, saved in two containers one within the other, $m\ (m\ b)$. Finally, it joins these two containers together and returns the result type b inside only one container, $m\ b$.

Formally;

$$fmap\ [(m\ a), (a \rightarrow m\ b)] = [m\ (m\ b)]$$
$$join\ [m\ (m\ b)] = [m\ b]$$

Therefore;

$$bind\ [X,\ f] = join \circ fmap\,[X,\ f\,]$$

For instance; the Haskell demonstration of above equality via *list* type constructor is given below:

- For *Integer* type:

$f\,x\ =\ return\ \ (x\ +\ 20)$ — function deceleration

$join\ \ (fmap\ f)\ [20]\ =\ [40]\ =\ bind\ \ [20]\ f$ — function calls

$$join.fmap = bind$$

4.3 Proofs: Particular Type Constructors of Haskell Behave as Functors and Monads

In this section of the thesis, some number of Haskell type constructors are proven to satisfy the algebraic rules of being a functor and a monad together with the help of Haskell programming language. At the same time, it could also be explained as these type constructors are the instances of the functor and monad type classes in Haskell. Here, the list of mentioned type constructors, each of which were designed to develop solutions for specific issues, is given below:

- Maybe type constructor
- List type constructor
- State type constructor
- IO type constructor
- Identity type constructor
- Eval type constructor

In addition to this, do consider that while proving any above type constructor as a monad, the methods of the monad type classes such as *return* and *join* are shown to obey the algebraic rules of being a natural transformation.

4.3.1 Maybe Type Constructor as Functor and Monad Class Instances

Maybe is a type constructor which enables the opportunity of creating new concrete types in Haskell programming language. Besides, the data constructors of *Maybe* are *Just a* and *Nothing* that means that a value of a *Maybe* type could either be just a data value such as *Just 10* or nothing which represents no value is contained in the type, at that time.

The deceleration of *Maybe* in the prelude class of Haskell was made as:

$$\mathit{data}\ \ \mathit{Maybe}\ \ a\ =\ \mathit{Just}\ a \mid \mathit{Nothing}$$

Maybe as a Functor Class Instance

Haskell code that defines *Maybe* as an *instance of functor class*:

$$\begin{aligned}
&\mathit{class}\ \ \textbf{Functor}\ \ F\ \ \mathit{where}\\
&\qquad \mathit{fmap}\ \ ::\ \ (a \to b) \to F\ \ a \to F\ \ b\\
&\mathit{instance}\ \ \textbf{Functor}\ \ \textbf{Maybe}\ \ \mathit{where}\\
&\qquad \mathit{fmap}\ f\ \ \mathit{Nothing}\ =\ \mathit{Nothing}\\
&\qquad \mathit{fmap}\ f\ \ (\mathit{Just}\, x)\ =\ \mathit{Just}\ \ (f\ \ x)
\end{aligned}$$

Lemma 4.3.1. *Type constructor* Maybe : HASK $\longrightarrow$ func *is a functor.*

Proof 4.3.2. *Category theoretic rules, to represent a functor, that are given in subsection 4.2.1 must be shown to satisfy for all of the data constructors of given type which are* Nothing *and* Just, *here.*

```
fmap id Nothing = id Nothing = Nothing
fmap id (Just a) = id (Just a) = (Just a)
(fmap g . fmap h) Nothing = (fmap g.h) Nothing = Nothing
(fmap g . fmap h) (Just a) = (fmap g.h) (Just a) = (Just (g.h(a)))
```

Let's assign Integer *and* Char *types to* type variable a, *respectively and then check all of the above equalities in Haskell environment.*

1. *proof via* Maybe Integer *type:*

```
-- FIdA = IdFA;  fmap id = id
1. Main.fmap Main.id Nothing = Main.id Nothing = Nothing
2. Main.fmap Main.id (Just 10)  = Main.id (Just 10)  = Just 10
-- F(g . h)  = F g . F h; fmap (f . g)  = (fmap f)  . (fmap g)
3. Main.fmap h(Main.fmap f Nothing)
   = (Main.fmap (h.f)) Nothing = Nothing
4. Main.fmap h(Main.fmap f (Just 10))
   = (Main.fmap (h.f)) (Just 10) = Just 25
where g, h :: Integer -> Integer; g x = x + 5,  h x = x + 10 and
id x = x
```

2. *proof via* Maybe Char *type:*

```
-- FIdA = IdFA; fmap id = id
1. Main.fmap Main.id Nothing = Main.id Nothing = Nothing
2.  (Main.fmap Main.id)  (Main.Just 'a')  = Main.id (Just 'a')
   = Just 'a'
-- F(g . h)  = F g . F h; fmap (f . g)  = (fmap f)  . (fmap g)
3.  (Main.fmap Char.toLower(Main.fmap Char.toUpper Nothing))
   = (Main.fmap (h.f)) Nothing = Nothing
4.  (Main.fmap Char.toLower)  (Main.fmap Char.toUpper(Main.Just 'a'))
   = (Main.fmap (Char.toLower. Char.toUpper))  (Main.Just 'a')
   = Just 'a'
where g, h :: Char -> Char; g = Char.toLower ,  h = Char.toUpper
and id x = x
```

As seen above, the equalities coming from the functor algebra are also satisfied and illustrated by codes written in Haskell via *Maybe Integer* and *Maybe Char* types.

Here, the crucial idea should be inferred is that Maybe maps objects and morphisms of a category namely *HASK* into objects and morphisms of another category called *func* whose objects are instances of functor class (Maybe a, ...) and morphisms are the functions between these objects (Maybe (a → a), ...). As mentioned before, *func* is the subcategory of *HASK*.

Example 4.3.3. *Type of* 'a' $\in Obj\ (HASK)$ *is Char;* Char.toUpper *function is* $(Char \rightarrow Char) \in Morph\ (HASK)$ *while type of* Just 'a' *is* Maybe Char $\in Obj\ (func)$ *and* Just (Char.toUpper) *is Maybe* $(Char \rightarrow Char) \in\ Morph\ (func)$.

The test in Haskell environment:

```
:t 'a' = Char
:t Main.Just ('a') =  Main.Maybe Char
:t Char.toUpper = (Char -> Char)
:t Main.Just (Char.toUpper) = Main.Maybe (Char -> Char)
```

Note that "*:t*" function returns the data types of its arguments. Eventually;

- $Maybe : Obj\,(HASK) \longrightarrow Obj\,(func)$
  $$Integer \longmapsto Maybe\;Integer$$
  $$Char \longmapsto Maybe\;Char$$
  $$Float \longmapsto Maybe\;Float$$
- $Maybe : Morph\,(HASK) \longrightarrow Morph\,(func)$
  $$(Integer \to Integer \to \ldots) \longmapsto Maybe\;(Integer \to Integer \to \ldots)$$
  $$(Char \to Char \to \ldots) \longmapsto Maybe\;(Char \to Char \to \ldots)$$
  $$(Float \to Float \to \ldots) \longmapsto Maybe\;(Float \to Float \to \ldots)$$

Maybe is said to be a functor defined between category *HASK* and its subcategory *func*.

Maybe as a Monad Class Instance

```
class Functor F where
    fmap :: (a → b) → F a → F b
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
class Monad m where
    join   :: m (m a) → m a
    return :: (Show (m a)) => a → m a
    bind   :: m a → ( a → m b) → m b
instance Monad Maybe where
    return x = Just x
    join (Just (Nothing)) = Nothing
    join (Just (Just x)) = Just x
    bind Nothing g = Nothing
    bind (Just x) g = g x
```

Lemma 4.3.4. join *and* return *are representations of natural transformations in Haskell environment.*

Proof 4.3.5. *Rules that are given in section 4.2.2 should be satisfied.*

1. proof via Maybe Integer *type:*

```
-- (join . fmap fmap f) [m(m a)] = (fmap f . join) [m (m a) ]= m  b
1. join((fmap (fmap f)) (Just(Just 10)))
   = fmap f (join (Just(Just 10))) = Just 15
-- (return . f) [ a ] = (fmap f . return) [ a ]= m b
2. ((return (f 10)) :: Maybe Integer)
   = fmap f ((return 10) :: Maybe Integer)  = Just 15
where f, h :: Integer -> Integer; f x = x + 5,  h x = x + 10 and
id x = x
```

Lemma 4.3.6. *The triple* (Maybe + fmap, join, return) *is a monad in Haskell environment.*

Proof 4.3.7. *The naturality of transformations* join, return *and functorial behavior of* Maybe + fmap *are already proven. Therefore, category theoretic rules, to represent a monad, that are given in subsection 4.2.2 must be shown to satisfy for all of the data constructors of given type which are* Nothing *and* Just, *here.*

```
1. join . join (fmap id) (Just(Just(Just a)))
   = join . join (Just(Just(Just a))) = Just a
   =  join . (fmap join)  (Just(Just(Just a)))
2. join . return (fmap  id)  (Just a) = Just a
   = join . (fmap  return)(Just a)
3. return . f (a) = Just (f a) = (fmap f) . return (a)
4. join . (fmap (fmap f))(Just(Just a)) = join . (Just(Just(f a)))
   = Just(f a) = (fmap f) . join (Just(Just a)) =  join (Just(Just (f a))
```

Note 4.3.8. *Note that proofs could be illustrated for all primitive and user created data types in Haskell programming environment. Specific types* $(Here:$ Integer *and* Char$)$ *are used in order just to convert the abstract and logical idea into concrete working examples. Since, as shown above, the feature of being a functor or a monad class instance is originating from the reason that data constructors of given algebraic data types are satisfying the category theoretic rules of building a functor or a monad. In the next sections of the thesis, proofs are done via limited number of data types* $(1$ *or* $2)$ *without emphasizing all types might be used to represent the category theoretic features of concerned type constructor.*

Let’s assign Integer *and* Char *types to* type variable ’a’, *respectively and then check all of the above equalities in Haskell environment.*

1. *proof via* Maybe Integer *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. join(join((fmap id)(Just(Just(Just 15)))))
   = join(fmap join(Just(Just(Just 15))))
 -- join . return (fmap  id)  = join . (fmap  return)
2. join(return(fmap id (Just 15))) =  join(fmap return(Just 15))
-- return . f  = (fmap f) . return
3. (return(f 10):: Maybe Integer)
   = fmap f ((return 10) :: Maybe Integer)
-- join . (fmap (fmap f)) = (fmap f) . join
4. join(fmap(fmap f)(Just(Just 10))) = fmap f(join(Just(Just 10)))
where f :: Integer -> Integer; f x = x + 5 and id x = x
```

2. *proof via* Maybe Char *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. join(join (.fmap id(Just(Just(Just ’a’)))))
   = join(fmap join(Just(Just(Just ’a’))))
-- join . return (fmap  id)  = join . (fmap  return)
2. join(return (fmap id (Just ’a’))) = join(fmap return(Just ’a’))
-- return . f  = (fmap f) . return
3. return(Char.toUpper ’s’) :: Maybe Char
   = ((fmap (Char.toUpper))(return ’s’) :: Maybe Char)
-- join . (fmap (fmap f)) = (fmap f) . join
4. join((fmap(fmap (Char.toUpper)))(Just(Just ’a’)))
   = (fmap (Char.toUpper)) (join(Just(Just ’a’)))
```

As the above equalities hold, the triple $(Maybe + fmap, return, join)$ or equally $(Maybe, return, bind)$ is proven as a monad which combines a chain of computations each of which could return *Nothing* or any other value within the container of *Just*.

The crucial idea here is that if any computation fails to return a *Nothing* value, then the result of the whole chain will also be *Nothing*.

For example, think a chain of computations that calculates if any given integer has n^{th} roots (suppose n is even) or not by first checking whether it has a square root;

if so, then checking whether it has $n/2^{th}$ root. In the case that a given value has no square roots (this part of the chain returns *Nothing*) that obviously means that it does not have n^{th} root, either (the whole calculation ends up with *Nothing*, either). Since, the value of *Nothing* is binded to the second part of the chain in which exactly no calculations done.

```
sqrt      :: Integer → Maybe Integer
sqrt x    =  sqrt' x (0, 0)
        where
                sqrt' x (s, r)
                    | s > x       = Nothing
                    | s == x      = Just r
                    | otherwise   = sqrt' x (s + 2 * r + 1, r + 1)
4th_root      :: Integer → Maybe Integer
4th_root x    =  sqrt x >>= sqrt
8th_root      :: Integer → Maybe Integer
8th_root x    =  sqrt x >>= sqrt >>= sqrt
```

In example, calculation of 4^{th} integer root of any given integer:

- First, calculate whether it has a square root, if so then bind the result of the square root computation to the square root function one more in order to detect whether it has a 4^{th} root or not. Otherwise, return *Nothing* value as the result of the whole computation (source code of *sqrt* function belongs to Mr. E. Söylemez).

Additionally, *Maybe* monad plays an important role in the error detection area in Haskell programming environment. For instance, a very well known *divide-by-zero error* could be caught easily.

```
divby0          :: Float → Float → Maybe Float
divby0   x 0    =  Nothing
divby0   x y    =  Just  (x/y)
```

Above code, detects and returns *Nothing* if divider is inputted as zero, otherwise, the result of the division operation is returned within the *Just* container.

Besides, in database management systems, the operation of looking up a record that might be missing is able to be managed smoothly by these *Maybe* monadic coding manner. Here, a basic Haskell code is given in order to illustrate the issue.

```
stdCrs =fromList([(“Cagatay”, “Thesis”), (“Erdem”, “ParallelProg”),
(“Burak”, “Network”)])
crsInst =fromList([(“Thesis”, “DrKoltuksuz”), (“ParallelProg”, “DrSahin”),
(“Network”, “DrHisil”)])
InstUni =fromList([(“DrKoltuksuz”, “YasarUni”), (“DrSahin”, “Iztech”),
(“DrHisil”, “YasarUni”)])
stdUni :: String → Maybe String
stdUni stdName = do
    course < − Data.Map.lookup stdName stdCrs
    instructor < − Data.Map.lookup course crsInst
    Data.Map.lookup instructor InstUni
```

Basically, in the above code:

1. Lookup lists, namely *stdCrs, crsInst* and *InstUni* that are analogies to database tables, are created and for each table, some tuples are inserted.
2. Then, a method, *stdUni* which takes a sting query as an input and returns a Maybe Sting type object as an output is given.
    - This method checks if a given student is registered in which university in *Just* container. If there is no student having the name inputted, then *Nothing* value is returned to the user.
    - For example; if you give the input query *stdUni “Burak”*, then you will get *Just “YasarUni”*. For any name that is not stored in the first table, *Nothing* will be returned as an output.

4.3.2 List Type Constructor as Functor and Monad Class Instances

List is another type constructor like Maybe in Haskell language whose data constructors are empty list, denoted by [ ], and an operation that prepends an element to the front of the list that is represented as ':'. Below, you see the definition of [] type in Haskell:

$$\mathit{data}\ [\,]\ a = [\,]\ |\ a : [a]$$

However, please note that no-one is allowed to use the above definition in any Haskell code, since this definition is a special one that is already defined in language specifics.

List as a Functor Class Instance

Haskell code that defines [ ] as an *instance of functor class*:

$$\begin{aligned}
&\mathit{class}\ \mathbf{Functor}\ F\ \ \mathit{where}\\
&\qquad \mathit{fmap}\ :: \ (a \to b) \to F\ a \to F\ b\\
&\mathit{instance}\ \mathbf{Functor}\ [\,]\ \mathit{where}\\
&\qquad \mathit{fmap}\ f\ \ [\,] = [\,]\\
&\qquad \mathit{fmap}\ f\ \ (x : xs) = (f\ \ x)\ :\ (\mathit{fmap}\, f\ xs)
\end{aligned}$$

Lemma 4.3.9. *Type constructor* [ ] : HASK ⟶ func *is a functor.*

Proof 4.3.10. *Category theoretic rules, to represent a functor, that are given in subsection 4.2.1 must be shown to satisfy for all of the data constructors of given type which are* empty list *and* prepending operation, *here.*

1. proof via [*Integer*] type:

```
-- FIdA = IdFA; fmap id = id
1. fmap id [ ] = id [ ]
2. fmap id ([1,2,3,4,5]) = id ([1,2,3,4,5])
--  F(g . h) = F g . F h; fmap (f . g) = (fmap f) . (fmap g)
3. fmap h(fmap f [ ]) = (fmap (h.f)) [ ]
4. fmap h(fmap f ([1,2,3,4,5])) = (fmap (h.f)) ([1,2,3,4,5])
where g, h :: Integer -> Integer; g x = x + 5,  h x = x + 10 and
id x = x
```

2. proof via [*Char*] type:

```
-- FIdA = IdFA; fmap id = id
1. fmap id [] = id []
2. fmap id ([``ax``, ``bx``]) = id ([``ax``, ``bx``])
-- F(g . h) = F g . F h; fmap (f . g) = (fmap f) . (fmap g)
3. fmap (map Char.toLower) ((fmap (map Char.toUpper))[])
   = (fmap ((map Char.toLower).(map Char.toUpper)) [])
4. fmap (map Char.toLower) ((fmap (map Char.toUpper))
   [``ax``, ``bx``])
   = fmap ((map Char.toLower).(map Char.toUpper))[``ax``, ``bx``])
where g, h :: Char -> Char; g = Char.toLower ,  h = Char.toUpper
and id x = x
```

Eventually;

- $[\,] \,:\, Obj\,(HASK) \longrightarrow Obj\,(func)$

  $$Integer \longmapsto [\,Integer\,]$$
  $$Char \longmapsto [\,Char\,]$$
  $$Float \longmapsto [\,Float\,]$$

- $[\,] \,:\, Morph\,(HASK) \longrightarrow Morph\,(func)$

  $$(Integer \to Integer \to \ldots) \longmapsto [\,Integer \to Integer \to \ldots\,]$$
  $$(Char \to Char \to \ldots) \longmapsto [\,Char \to Char \to \ldots\,]$$
  $$(Float \to Float \to \ldots) \longmapsto [\,Float \to Float \to \ldots\,]$$

Such as *Maybe* type constructor; [ ] is also said to be a functor defined between category *HASK* and its subcategory *func*.

List as a Monad Class Instance

class **Functor** *F* *where*

$\quad fmap \;::\; (a \to b) \to F\;a \to F\;b$

instance **Functor** [ ] *where*

$\quad fmap\;f\;[\,] = [\,]$

$\quad fmap\;f\;(x : xs) = (f\;\;x) : (fmap\,f\,xs)$

class **Monad** *m* *where*

$\quad join \;::\; m\;(m\;a) \to m\;a$

$\quad return \;::\; (Show\;(m\;a)) \Rightarrow a \to m\;a$

$\quad bind \;::\; m\;a \to (a \to m\;b) \to m\;b$

```
instance Monad [] where
    return x = [x]
    join [[x]] = [x]
    join [[ ]] = [ ]
    join (x:xs) = x ++ join (xs)
    bind (x:xs) g = (fmap g (x:xs))
    bind [] g = []
```

Lemma 4.3.11. *The triple* ([] + fmap, join, return)*, equivalently* ([], bind, return) *is a monad in Haskell environment.*

Note 4.3.12. *Naturalities of join and return transformations are proven in previous session, 4.3.1, for* Maybe type constructor. *The same proof also for* [] *type or any other type constructors in the next sections will not be given. Since, by the application of the same methodology performed for* Maybe*, it is very elementary to prove it also for any other type constructors. Additionally, the proof of functorial behavior of* [] + *fmap is done in this section.*

Proof 4.3.13. *Therefore, category theoretic rules, to represent a monad, that are given in subsection 4.2.2 must be shown to satisfy for all of the data constructors of given type which are empty list and ':', here.*

1. *proof via* [Integer] *type:*

```
-- join . join (fmap id)   = join . (fmap join)
1. join(join((fmap id)[[[1,2],[3,4]],[[5,6],[7,8]]]))
   =  join(fmap join([[[1,2],[3,4]],[[5,6],[7,8]]]))
-- join . return (fmap  id)  = join . (fmap  return)
2. join(return((fmap id)[1,2,3])) = join(fmap return[1,2,3])
-- return . f  = (fmap f) . return
3. return(f 10) :: [Integer] = fmap f(return 10 :: [Integer])
-- join . (fmap (fmap f)) = (fmap f) . join
4. join(fmap(fmap f)([[1,2],[3,4],[5,6]]))
   = fmap f(join([[1,2],[3,4],[5,6]]))
where f :: Integer -> Integer; f x = x + 5 and id x = x
```

2. *proof via* [Char] *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. join(join((fmap id)[[[``ax``,``bx``],[``cx``,``dx``]],
   [[``ex``,``fx``],[``gx``,``hx``]]]))
   = join(fmap join([[[``ax``,``bx``],[``cx``,``dx``]],
   [[``ex``,``fx``],[``gx``,``hx``]]]))
-- join . return (fmap  id)  = join . (fmap  return)
2. join(return ((fmap id)[``ax``,``bx``,``cx``]))
   = join(fmap return[``ax``,``bx``,``cx``])
-- return . f  = (fmap f) . return
3.  ((return (map Char.toUpper ``ax``)) :: [[Char]])
   = (fmap (map Char.toUpper))(return ``ax`` :: [[Char]])
-- join . (fmap (fmap f)) = (fmap f) . join
4. join((fmap(fmap (map Char.toUpper))) ([[``ax``,``bx``],
   [``cx``,``dx``]]))
   = (fmap (map Char.toUpper))
   (join [[``ax``,``bx``],[``cx``,``dx``]])
```

As the above equalities hold, the triple $([] + fmap, return, join)$ or equally $([], return, bind)$ is proven as a monad. The main idea behind the implementation of list monad is to model the non-deterministic programs that might end up with multiple results for any input query. The first basic example could be *calculating power set of a given set* by using Haskell [] monad.

$$\begin{aligned} &powerset\,[\,] \;=\; [[\,]] \\ &powerset\,(x:xq) \;=\; l\;{+}{+}\;map\;(x:)\;l \\ &\qquad \textit{where}\; l \;=\; powerset\,xq \end{aligned}$$

Lets have a look at how code generates the power set of $[0,1,2]$:

Recursion points are: $0:[1,2], 1:[2]$ and $2:[]$ which is also the base case. Then, lets point out the calculation steps:

1. $[[]] + +map(2:)[[]] = [[]] + +[[2]] = [[],[2]]$
2. $[[],[2]] + +map(1:)[[],[2]] = [[],[2]] + +[[1],[1,2]] = [[],[2],[1],[1,2]]$
3. $[[],[2],[1],[1,2]] + +map(0:)[[],[2],[1],[1,2]] = [[],[2],[1],[1,2]] + +$ $[[0],[0,2],[0,1],[0,1,2]] = [[],[2],[1],[1,2],[0],[0,2],[0,1],[0,1,2]]$

Compared to the previous one, more meaningful example will be the implementation of a non-deterministic finite state automaton in Haskell language, using the abstraction layer that [] monad brings.

```
data NFAState id = NFAState {StateId :: id,
                             isFinal :: Bool,
                             transitionF0 :: [NFAState id],
                             transitionF1 :: [NFAState id]}
expressiontoAutomata = [state1]
      where
          state1 = NFAState 1 False [state1,state2,state3] [state1]
          state2 = NFAState 2 False [state4][state3]
          state3 = NFAState 3 True  [state4][state4]
          state4 = NFAState 4 False [state4][state4]
strAccept start_states str = any (\id → acceptP id str) start_states
      where
          acceptP (NFAState _ isFinal _ _) [] = isFinal
          acceptP (NFAState _ _ t0 t1) (x : xs) = strAccept (if x then t1 else t0) xs
```

In the above Haskell coded non-deterministic finite state automaton (acceptor):

1. First part defines a new data type so-called *NFAState*, which represents the states in the automation involving each ones id, a boolean value indicates whether it is the final state or not and transition functions for the inputs both 1 and 0.

2. Second part defines the start state as state1 and also transition functions for each state. For example; from state1 with input 0, possible transformations are: state might not be changed, (staying in state1), changed into state2 or state3 which is the only part yields the non-determinism.

3. Third part of the code, checks all possible state transformations might be ended up with, then returns the Boolean value which demonstrates if the input string is accepted or not.

   In example, the input string, $strAccept\ expressiontoAutomata(map\ (>0)\ [0,0,0,1,0,1,1])$, is not accepted by the automation while the string $strAccept\ expressiontoAutomata\ (map\ (>0)\ [0,0,0,1,0])$ is accepted.

4.3.3 State Type Constructor as Functor and Monad Class Instances

Similar to [] and *Maybe*; *State s* is also a type constructor in Haskell environment, but, from another point of view, unlike *Maybe* and []; *State s* has a function which takes a state value as an input and returns an intermediate value together with some new state value, as of the data constructor. Below, you see the definition of *State s* type in Haskell:

$$\mathit{data}\ (\mathit{State}\ s)\ a\ =\ \mathit{State}\ (s\ \rightarrow\ (a,s))$$

In order to eliminate the confusions might arise, it should be noted that the term *State s* standing on the left hand side of the deceleration represents the type constructor where the one on the right hand side is standing for the indication of data constructors of any type created by *State s* .

Indeed, *State s* type constructor is nothing more than an encapsulation to the function that takes a state value, s, and returns an intermediate value and some new state value, (a,s). Moreover, in order to be able to unwrap the inside function, another function called *runState* is used. [10]

$$\begin{aligned}&\mathit{runState}\ ::\ \mathit{State}\ s\ \ a\ \rightarrow\ s\ \rightarrow\ (a,\ s)\\&\mathit{runState}\ (\mathit{State}\ f)\ \ s\ =\ f\ \ s\end{aligned}$$

Note 4.3.14. *Note that the data constructor here is* "State s" *not only* "State".

State as a Functor Class Instance

Haskell code that defines *State s* as an *instance of functor class*:

$$\begin{aligned}&\mathit{class}\ \ \mathbf{Functor}\ \ F\ \ \mathit{where}\\&\qquad \mathit{fmap}\ \ ::\ (a\ \rightarrow\ b)\ \rightarrow\ F\ \ a\ \rightarrow\ F\ \ b\\&\mathit{instance}\ \ \mathbf{Functor}\ \ (\mathbf{State\,s})\ \ \mathit{where}\\&\qquad \mathit{fmap}\,f\ m\ =\ \mathit{State}\ (\backslash\ k\ \rightarrow\ \ \mathit{let}\ (a,\ s)\ =\ \mathit{runState}\ m\ k\ \ \mathit{in}\ (f\ a,\ s))\end{aligned}$$

Lemma 4.3.15. *Type constructor* State s : HASK ⟶ func *is a functor.*

Proof 4.3.16. *Category theoretic rules, to represent a functor, that are given in subsection 4.2.1 must be shown to satisfy.*

1. *proof via* State Integer Integer *type:*

```
-- FIdA = IdFA; fmap id = id
1. runState(fmap id (encap 10))'a'
   = runState((fmap (h.f)) (encap 10))'a'
-- F(g . h) = F g . F h; fmap (f . g) = (fmap f) . (fmap g)
2. runState(fmap h(fmap f (encap 10)))'a'
   = runState((fmap (h.f)) (encap 10))'a'
where f, h :: Integer -> Integer; f x = x + 5,  h x = x + 10 and
encap :: Integer -> State s Integer; encap a  = State (\s -> (a,s))
```

2. *proof via* State Integer Char *type:*

```
-- FIdA = IdFA; fmap id = id
1. runState(fmap id (encap 10))'a'
   = runState((fmap (h.f)) (encap 10))'a'
--  F(g . h) = F g . F h;  fmap (f . g) = (fmap f) . (fmap g)
2. runState(fmap h(fmap f (encap 10)))'a'
   = runState((fmap (h.f)) (encap 10))'a'
where f, h :: Integer -> Integer; f x = x + 5,  h x = x + 10 and
encap :: Integer -> State s Integer; encap a  = State (\s -> (a,s))
```

Eventually;

- $State\ s\ :\ Obj\ (HASK) \longrightarrow Obj\ (func)$

$$\begin{aligned} Integer &\longmapsto State\ s\ Integer \\ Char &\longmapsto State\ s\ Char \\ Float &\longmapsto State\ s\ Float \end{aligned}$$

- $State\ s\ :\ Morph\ (HASK) \longrightarrow Morph\ (func)$

$$\begin{aligned} (Integer \rightarrow Integer \rightarrow \ldots) &\longmapsto State\ s\ (\ Integer \rightarrow Integer \rightarrow \ldots\ ) \\ (Char \rightarrow Char \rightarrow \ldots) &\longmapsto State\ s\ (\ Char \rightarrow Char \rightarrow \ldots\ ) \\ (Float \rightarrow Float \rightarrow \ldots) &\longmapsto State\ s\ (\ Float \rightarrow Float \rightarrow \ldots\ ) \end{aligned}$$

Such as *Maybe* and [] type constructors; *State s* is also said to be a functor defined between category *HASK* and its subcategory *func*.

State as a Monad Class Instance

```
class Functor F where
    fmap   :: (a → b) → F a → F b
instance Functor (State s) where
    fmap f m = State (\ k → let (a, s) = runState m k in (f a, s))
class Monad m where
    join    :: m (m a) → m a
    return  :: a → m a
    bind    :: m a → ( a → m b) → m b
instance Monad (State s) where
    return a = State (\ s - > (a,s))
    bind (State x) f = State (\ s - > let (a,s′) = x s in runState (f a) s′)
    join xss = State (\ s - > uncurry runState (runState(xss) s))
```

Lemma 4.3.17. *The triple* (State s + fmap, join, return), *equivalently* (State s, bind, return) *is a monad in Haskell environment.*

Proof 4.3.18. *Therefore, category theoretic rules, to represent a monad, that are given in subsection 4.2.2 must be shown to satisfy for all of the data constructors of given type that is a basic encapsulated function, here.*

1. *proof via* State Integer Integer *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. runState(join( join  (fmap id(encap  (encap(encap 4))))))5
   = runState(join(fmap join  (encap(encap(encap 4)))))5
-- join . return  (fmap  id)  = join .  (fmap  return)
2. runState(join(return(fmap id(encap 4))))5
   = runState(join(fmap return(encap 4)))5
-- return . f  = (fmap f) . return
3. runState(return(f 10))30 = runState(fmap f(return 10))30
-- join . (fmap (fmap f)) = (fmap f) . join
4. runState(join(fmap(fmap f)(encap(encap 10))))30
   = runState(fmap f(join(encap(encap 10))))30
where f :: Integer -> Integer; f x = x + 5 and id x = x
```

2. *proof via* State String Integer *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. runState(join( join (fmap id(encap (encap(encap ``ax''))))))5
   = runState(join(fmap join (encap(encap(encap ``ax'')))))5
-- join . return (fmap  id)  = join . (fmap  return)
2. runState(join(return(fmap id(encap ``ax''))))5
   = runState(join(fmap return(encap ``ax'')))5
-- return . f  = (fmap f) . return
3. runState(return(map Char.toUpper ``ax''))5
   = runState((fmap (map Char.toUpper))(return ``ax''))5
-- join . (fmap (fmap f)) = (fmap f) . join
4. runState(join((fmap(fmap(map Char.toUpper)))
    (encap(encap ``ax''))))5
   = runState((fmap (map Char.toUpper))
    (join(encap(encap ``ax''))))5
where f :: Integer -> Integer; f x = x + 5 and id x = x
```

As the above equalities hold, the triple $((State\ s) + fmap, return, join)$ or equally $((State\ s),\ return, bind)$ is proven as a monad. The main idea behind the implementation of state monad is to bring the feature called "referential transparency", that certifies that a function returns the same result for the same input regardless of its current state, to the language and also model the input output system of it with disabling side effects which is explained detailedly in the next section.

A random number generator function in Haskell could be given as an example to the implementation of the state monad. Since, consider that random number generation is not even a function in large number of programming languages. For instance, in C programming language, random number generation is such a method whose signature is *int rand* $(void)$. That means that *rand* returns different values even without taking a parameter, therefore this situation yields in violation of referential transparency rule. In order to avoid this violation, random number generation work, in Haskell, is done with great amount of help of the state monad.

```
genRand :: State StdGen Integer
genRand = do generator <- get
             let ( value, newGenerator ) = randomR (1,566564653213) generator
             put newGenerator
             return value
```

Primarily, *get* function takes the current state of the monad and assigns it to a variable called *generator*. Secondly, *randomR* function is called with two integer parameters and a generator that was derived from the state monad itself. Then, results returned from *randomR* function are assigned to *value* and *newGenerator* variables. Lastly, after the assignment of *newGenerator* to the state monad as a new state, the random number, inside the *value* parameter, is returned to the user.

4.3.4 IO Type Constructor as Impure Monad Class Instances

IO type constructor of Haskell is a specific type of the state constructor. In this section of the thesis, input-output operations of Haskell programming language are explained and illustrated together with disabling side effects that indicates that any observable interaction with calling functions or the outside world, by performing all I/O operations inside the IO monad.

Here is the definition of IO type constructor:

$$\mathit{data}\ \mathit{IO}\ a = \mathit{IO}\ ((\mathit{String},\mathit{String}) \rightarrow (a,(\mathit{String},\mathit{String})))$$

Informally, *IO* type constructor, correspondingly *State s*, involves a function inside the *IO* wrapper which takes a pair of strings, matches up to *s* (state value in the State monad), as an input and returns an intermediate value, *a* (stays unchanged compared to State monad), together with some new pair of strings, as of the data constructor. Furthermore, in order to be able to unwrap the inside function, *runIO* function should be used:

$$\begin{aligned}&\mathit{runIO} :: \mathit{IO}\ a \rightarrow (a,(\mathit{String},\mathit{String}))\\&\mathit{runIO}\ (\mathit{IO}\ p) = (p\ \mathit{undefined})\end{aligned}$$

runIO function takes another function inside the *IO* wrapper, first unwraps it, then returns the intermediate value and the pair of strings as output regardless of intermediate value's type which is reported by the usage of *undefined* pragma to the compiler.

After taking above things into consideration, it could be said that *IO* type constructor is the specific case of *State s* type constructor and undoubtedly a monad which generates the below instance:

```
instance Monad (IO) where
    return a = IO(\_ → (a,(“ ”, “ ”)))
    (IO t0) >>= f = IO(\ s0 → let (a,s1) = t0 s0
                                  (IO t1) = f a
                              in t1 s1 )
```

Note 4.3.19. IO *type constructor together with* (bind + return) *or* (fmap, join+ return) *natural transformations could be proven as a monad, like the ones done for before type constructors. However, due to being a specific type of* State s *monad, this proof is not given. Since, the crucial inference is that if* State s *with mentioned transformations is a monad, then* IO *also constructs a monad with the same transformations.*

Let’s basically illustrate the I/O functions such as *getChar*, *putChar*, *getLine* and *putStr*, in Haskell environment by the usage of *IO* monad.

1. Initialization of the environment in which I/O operations are done.

    - The void type environment. (State is represented by pair of strings, intermediate value is a void type one)

      ```
      init :: IO ()
      init = IO(\_ → ((),(“burak”,“”)))
      ```

    - The Char type environment. (State is represented by pair of strings, intermediate value is a Char type one)

      ```
      initChar :: IO Char
      initChar = IO(\_ → (' ',(“burak”,“”)))
      ```

2. Get character and put character functions.

    ```
    getc :: IO Char
    getc = IO(\((i:is),os) → (i,(is,os)))
    ```

    where *is* and *os* represent input and output streams, respectively.

    Function *getc* operates inside the environment which is already initialized. It takes the first element of input stream and puts that character into the intermediate variable, does not append it into output stream, and returns the pair of strings with its new configuration.

For instance:

$$\begin{aligned} t0 \;=\; & runIO\;(\;\textit{do}\;initChar \\ & \qquad\qquad getc) \end{aligned}$$

returns $('b',(\text{“}urak\text{”},\text{“”}))$ as output. Meanwhile, *do* is called *syntactic sugar* in Haskell environment which is used instead of binding inside the *IO* monad in order to ease the operations.

$$\begin{aligned} &getc2 \;::\; Char \rightarrow IO\;Char \\ &getc2\,i \;=\; IO(\backslash((i:is),os) \rightarrow (i,(is,os))) \end{aligned}$$

If *getc* is defined in the above form, then $runIO(Main.initChar >>= getc2)$ returns the same result, $('b',(\text{“}urak\text{”},\text{“”}))$, with $t0$. That means that the *do* block provides the opportunity to bind values to functions in a much more easier way inside the monad.

$$\begin{aligned} &putc \;::\; Char \rightarrow IO\;() \\ &putc\,c \;=\; IO(\backslash(is,os) \rightarrow ((),(is,os++[c]))) \end{aligned}$$

Function *putc* takes the intermediate value inside the monad and then assigns it to the output stream. For instance:

$$\begin{aligned} t1 \;=\; & runIO\;(\;\textit{do}\;initChar \\ & \qquad\qquad x <- getc \\ & \qquad\qquad putc\;\;x\;) \end{aligned}$$

returns $((),(\text{“}urak\text{”},\text{“}b\text{”}))$ as output where *getc* takes the char *b* from the intermediate variable *a*, then *putc* takes it as input and appends it to the output stream.

For sure, binding could be used instead of syntactic sugar *do*:

$$\begin{aligned} &putc2 \;::\; Char \rightarrow IO\;() \\ &putc2\,c \;=\; IO(\backslash(is,os) \rightarrow ((),(is,os++[c]))) \end{aligned}$$

Query: $runIO(initChar >>= getc2 >>= putc2)$ returns the same result, $((),$ $(\text{“}urak\text{”},\text{“}b\text{”}))$, with $t1$.

```
putstr :: String → IO ()
putstr “” = putc ' '
putstr (x : xs) = ( do putc x
                       putstr xs )
```

Above *putstr* function which corresponds to *putStr* in Haskell, is nothing more than recursively called *putc* function till a base case (empty string) is detected.

For instance, the below function returns the first element of the given input string with the explanation sentence: “*1st char is*”:

```
t2 = runIO ( do initChar
                x < − getc
                putstr “1st char is”
                putc  x )
initString :: IO String
initString = IO(\_ → (“ ”, (“burak”, “”)))
getline :: IO String
getline = IO(\(is, os) → (is, (“ ”, os)))
```

Function *getline* transfers the input string to the intermediate value which is also initialized as string in *IO String* monad (the originals of example codes for *IO* monad belong to S. Klinger [9]). After the illustrations of all basic I/O operations operate in Haskell *IO* monad, let's also give a basic working example:

```
main :: IO()
main =  do putStrLn “What is your name?”
           x < − getLine
           if x == “burak”
               then putStrLn “You are right!”
               else putStrLn “You are wrong!”
```

As seen in the above code, *main* function is defined as an *IO* monad type and all of the I/O operations are done inside the monad in order to avoid the side effects that were going to occur due to the interaction with the real world. [9]

4.3.5 Identity Type Constructor as Functor and Monad Class Instances

Similar to [], *Maybe* and *State s*; *Identity* is also a type constructor in Haskell environment, but, from another point of view, unlike *Maybe*, [] and *State s*, there is no computational reason to use *Identity* monad, since it only applies the function was bound inside the monad to its arguments without making any changes (only a simple wrapper). However, it plays very critical role in the creation of *Eval* monad which provides the *evaluation order* to the computation via parallel and sequential strategies and also in the area of *monad transformers*, that are beyond the scope of this project. Below, you see the definition of *Identity* type in Haskell:

```
data Identity a = Identity a
```

In order to unwrap the inside type, *runIdentity* function, whose signature given below, is used:

```
runIdentity :: Identity a → a
runIdentity (Identity a) = a
```

Identity as a Functor Class Instance

Haskell code that defines *Identity* as an *instance of functor class*:

```
class Functor F where
    fmap :: (a → b) → F a → F b
instance Functor Identity where
    fmap f m = Identity (f (runIdentity m))
```

Lemma 4.3.20. *Type constructor* Identity : HASK ⟶ func *is a functor.*

Proof 4.3.21. *Category theoretic rules, to represent a functor, that are given in subsection 4.2.1 must be shown to satisfy.*

1. *proof via* Identity Integer *type:*

```
-- FIdA = IdFA; fmap id = id
1. runIdentity(fmap id (return 10)) = runIdentity(id (return 10))
-- F(g . h) = F g . F h; fmap (f . g) = (fmap f) . (fmap g)
2. runIdentity(fmap h(fmap f (return 10)))
   = runIdentity((fmap (h.f)) (return 10))
where f, h :: Integer -> Integer; f x = x + 5,  h x = x + 10
```

2. *proof via* Identity Char *type:*

```
-- FIdA = IdFA; fmap id = id
1. runIdentity(fmap id (return 'a')  = runIdentity(id (return 'a'))
--  F(g . h) = F g . F h; fmap (f . g) = (fmap f) . (fmap g)
2. runIdentity(fmap (Char.toLower)(fmap(Char.toUpper)(return 'a')))
   = runIdentity((fmap ((Char.toLower).(Char.toUpper)))
   (return 'a'))
```

Eventually;

- $Identity\ :\ Obj\ (HASK) \longrightarrow Obj\ (func)$

  $Integer \longmapsto Identity\ Integer$

  $Char \longmapsto Identity\ Char$

  $Float \longmapsto Identity\ Float$

- $Identity\ :\ Morph\ (HASK) \longrightarrow Morph\ (func)$

  $(Integer \to Integer \to ...) \longmapsto Identity\ (\ Integer \to Integer \to ...\ )$

  $(Char \to Char \to ...) \longmapsto Identity\ (\ Char \to Char \to ...\ )$

  $(Float \to Float \to ...) \longmapsto Identity\ (\ Float \to Float \to ...\ )$

Such as *Maybe*, [] and *State s* type constructors; *Identity* is also said to be a functor defined between category *HASK* and its subcategory *func*.

Identity as a Monad Class Instance

class **Functor** *F* *where*

  fmap :: $(a \to b) \to F\ a \to F\ b$

instance **Functor Identity** *where*

  fmap f m = *Identity* (*f* (*runIdentity m*))

```
class Monad m where
    join   :: m (m a) → m a
    return :: a → m a
    bind   :: m a → (a → m b) → m b
instance Monad Identity where
    return a = Identity a
    m >>= k = k (runIdentity m)
    join m = runIdentity m
```

Lemma 4.3.22. *The triple* (Identity + fmap, join, return)*, equivalently* (Identity, bind, return) *is a monad in Haskell environment.*

Proof 4.3.23. *Therefore, category theoretic rules, to represent a monad, that are given in subsection 4.2.2 must be shown to satisfy for all of the data constructors of given type that is a basic encapsulated function, here.*

1. *proof via* Identity Integer *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. runIdentity(join(fmap join (encap(encap(encap 4)))))
   = runIdentity(join( join (fmap id(encap (encap(encap 4))))))
-- join . return (fmap  id)  = join . (fmap  return)
2. runIdentity(join(fmap return(encap 4)))
   = runIdentity(join(return(fmap id(encap 4))))
-- return . f  = (fmap f) . return
3. runIdentity(return(f 10)) = runIdentity(fmap f(return 10))
-- join . (fmap (fmap f)) = (fmap f) . join
4. runIdentity(join(fmap(fmap f)(encap(encap 10))))
   = runIdentity(fmap f(join(encap(encap 10))))
where f :: Integer -> Integer; f x = x + 5 and id x = x
```

2. *proof via* Identity Char *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. runIdentity(join(fmap join (return(return(return 'a')))))
   = runIdentity(join( join (return(return(return 'a')))))
-- join . return (fmap  id)  = join . (fmap  return)
2. runIdentity(join(fmap return(return 'a')))
   = runIdentity(join(return(return 'a')))
-- return . f  = (fmap f) . return
3. runIdentity(return(Char.toUpper 'a'))
   = runIdentity(fmap Char.toUpper(return 'a'))
-- join . (fmap (fmap f)) = (fmap f) . join
4. runIdentity(join(fmap(fmap Char.toUpper)(return(return 'a'))))
   = runIdentity(fmap Char.toUpper(join(return(return 'a'))))
where id x = x
```

As the above equalities hold, the triple $(Identity + fmap, return, join)$ or equally $(Identity, return, bind)$ is proven as a monad. As mentioned, the main idea behind the implementation of *Identity* monad is to create the *Eval* monad in order to define the evaluation strategies to parallelize the computations, if possible.

Next chapter of the project is completely relying on the parallelization issue via original strategies and the second generation ones that are encapsulated by *Eval* type constructor with the proof demonstrates that it satisfies the rules of constructing a monad by the help of natural transformations: $bind + return$ or $(join, fmap) + return$.

Chapter 5

Semi Explicit Parallelism in Haskell

5.1 Terminology: Parallel Computing

Parallel computing is splitting up the solution of a large computational problem into smaller tasks and performing them simultaneously over multiple processors to speed up the computations by exploiting the underlying hardware. In order to implement parallel computation in the solution of any problem; techniques and strategies to be applied and the subtasks where the potential parallelism might occur, should already been defined.

Parallel computing has been used for many years, mostly in high-performance computing, however, interest in it has grown in recent years due to the development and improvement of multi core chips. After the improvements happened in hardware, parallel computation has become a dominant paradigm in software design, especially creating parallel algorithmic solutions to the problems which seems to become a hot topic in computing science.

In accordance with the above mentioned innovations, programming languages have also began to provide parallel programming support. For instance, Intel has come up with a library, called OpenMp, which involves some number of external pragmas to bring the parallel programming opportunity for C++ users. As a functional programming language, Haskell also provides the opportunity to create parallel programs to its users via specific methodologies called *strategies* that are defined detailedly in this section of the project.

Haskell seems very convenient language for parallel computations due to the reasons:

- Being *purely* functional, means there is no side effects.
- Having a *strong type system* (type castings are not allowed).
- Including a runtime system (GHC) that supports light-weight threading called *sparks*.
- Involving second generation strategies that are *non-strict*. Assume function evaluation $f\,x$:
  - f might never need x, then no problem occurs.
  - f might need x immediately, then no problem occurs but parallelization is interrupted.
  - f might not need x immediately which means parallelization is provided.

On the other hand, parallelization internally has some number of disadvantages should be envisaged before the implementation of the process:

- Creation of threads or sparks brings an overhead to the system.
- Garbage collection should be evaluated in the sense of parallelization.
- Non-strictness in second generation strategies might be problematic in some cases such as:
  - How far to be evaluated seems unclear.
  - The interaction of it with garbage collection should be made clear.

Note 5.1.1. *Due to the above mentioned problematic cases, implicit or fully automatic parallelization still seems a future goal. For the time being, the only thing can be done is to help the compiler by using basic annotations that are providing parallelization. [12]*

However, Haskell eliminates such number of traditional error-prone issues of parallel programming due to being a safe (in terms of type casting and consistency of semantics) programming language:

1. Parallel programming in Haskell is *deterministic*: The algorithm is called *deterministic* parallel algorithm, if it returns the same result for the same input query regardless of the number of processors it has been run over. Therefore, anyone could debug the parallel Haskell programs without running them in parallel.

2. Parallel programs in Haskell do not explicitly deal with the *synchronization* and *communication* issues.

    - *Synchronization* is the term used to indicate the action of waiting for the other task to complete by an arbitrary task, most probably, because of the dependencies between them.
    - *Communication* defines the interactions done in between the tasks, running over different cores in parallel.

    Synchronization is done by the GHC runtime system and/or the parallelism libraries. From the point of *communication* view, it could be noted that communication is also done internally by the runtime system GHC, since all the tasks share the same heap and share data without any constraints which means that although there is no communications in the program level, it is done in hardware level as the transmission of data between the caches of the different cores.

Deterministic computation, *Synchronization* and *Communication* terms are guaranteed to be done by the runtime system of Haskell. On the other hand, Haskell programmer has to deal with some issues given below to get the tasks working in parallel:

1. *Partitioning* refers to the meaning that a whole work is divided into subtasks that could execute in parallel.

    - *Granularity* indicates the sizes of subtasks. If tasks are divided into relatively small subtasks, then the thread or spark overhead that removes the aim of parallelism, occurs. If tasks are divided into relatively large subtasks, this time the parallelization potential decreases. Ideally, granularity of the subtasks should be large enough the gain the parallelization potential together with minimum overhead.

2. *Data dependencies* are the mathematical dependency in between any subtasks of an arbitrarily given task which forces the serialization of these dependent tasks. [12]

As it is inferred from the above work-sharing, the notion *Semi-explicit parallelization* originates from the feature that both compiler and programmer have, nearly, the same load of responsibilities in creating the parallelization. The notion, *Explicit* *parallelization*, is spoken of when all of the responsibility is put on the programmer, while everything to be done, in terms of parallelization, is expected from compiler (still a future goal), to define the computation the term *Implicit parallelization* is used.

Furthermore, parallelization process involves so many approaches depending on its own demands such as:

1. *Data parallelism*, in which data is divided into many pieces and the general operation is implemented over each subpart (piece) in parallel.
2. *Task parallelism*, in which the whole work is divided into its subtasks and these tasks are evaluated (executed) in parallel.

In this part of the thesis project, we are completely focusing on the special *strategies* that are defining the evaluation of a structure with components in sequence or in parallel without generating interesting results.

Haskell programming language involves two versions of *Second Generation strategies*:

1. *Original Strategies*.
2. *Second Generation Strategies* which are the evolved versions of the original strategies encapsulated by *Eval monad*.

The parallel programming model in Haskell is based on two annotations:

- $par \ \ :: a \rightarrow b \rightarrow b$
- $pseq :: a \rightarrow b \rightarrow b$

As inferred from their type signatures, both *par* and *pseq* annotations take two arguments, *a, b*, as inputs and return a value in the form of second argument, *b*. Therefore, computationally, no difference exists between these two functions. However, the key point is hidden in the sense of evaluation order that is under their guidance which means that *par* annotation hints to the Haskell implementation that it might be beneficial to evaluate the first argument in parallel while *pseq* guarantees the second argument is evaluated after the evaluation of the first one (in sequence). [14][24]

- *par* annotation stores its first argument, *a*, as a *spark* which is a light-weighted thread, into the *spark pool*, then resumes the evaluation of its second argument, *b*. The idea is that an idle processor might find that spark in the pool and evaluates it. Therefore, an uncertified parallelization is supplied by the GHC run time system in between the evaluations of the arguments *a* and *b*.

- *pseq* is used to sequentialize the computation. Since, in *lazy* programming languages such as Haskell, the sequence of evaluation is undefined. By the usage of the *pseq* annotation sequence is converted clear by evaluating first argument before than second one. [12]

Note 5.1.2. *Both* par *and* pseq *annotations evaluate their first arguments to* weak head normal form (WHNF) *and second arguments to* normal form (NF). *[25]*

Note 5.1.3. Normal Form (NF) *is the notion indicating the situation that if there is no more evaluation could be performed over an arbitrary expression, which means that the expression, under discussion, is already reduced to its value form and values are always in the normal form. On the other hand, if there is at least one evaluation to be done then, that expression is still not evaluated until its value form, therefore it is called in* Weak Head Normal Form (WHNF).

An expression is called in WHNF, *if and only if it is either:*

- *a function applied to too few arguments such as* $(*)8$, fib
- *a constructor that is already applied to arguments such as* Just 100, $[100]$
- *or a lambda expression in the form of* $(\backslash x \rightarrow expression)$.

5.2 Original Strategies

Original evaluation strategies, generated by [26] Trinder et al, are actually higher order functions which indicate the program's *dynamic evaluation behavior* without making any change on the algorithmic construction. Therefore, the program, now, could be said to be composed of two parts: *Algorithm* and *Strategies*.

Here is the definition of original strategies:

```
type Strategy a = a → ()
```

which is a function, takes an argument *a* as input and returns $()$, unit type, as output, since it only defines nothing more than the dynamic behavior of the program.

5.2.1 Strategies for the Control of Evaluation Degree: r0, rwhnf, rnf.

In this session, original strategies that are dealing with evaluation degree not with order are defined and demonstrated via their signatures.

The first one, *r*0, is the most basic one:

```
rnf r0  :: Strategy a
r0 _ = ()
```

deals with, actually, nothing. Although it seems to make effort only on the evaluation degree of the computation, it does not cause any reduction over the given expression.

In Haskell, an arbitrary expression could be either in *WHNF* or in *NF*. Reducing an expression into *WHNF* is done by a function whose signature is given below:

```
rwhnf   :: Strategy a
rwhnf  x = x `pseq` ()
```

Therefore, *rwhnf* function takes an expression *x*; passes it into *pseq* annotation as its first input and () as the second one. By this way, expression *x* is evaluated into *WHNF* and then () is returned.

In order to be able to reduce an arbitrary expression into its *NF* or value form, *rwhnf* function is called recursively till the evaluation ends up with a value.

```
class NFData a  where
    rnf    :: Strategy a
    rnf    =  rwhnf
```

Since, different datatypes require different ways to be reduced into their *NFs*, above type class is defined to encapsulate each datatype. Consider the list datatype is demanded to be reduced into *NF*:

```
instance NFData [a]  where
    rnf [] =  ()
    rnf (x : xs)  = rnf x `pseq` rnf xs
```

In order to be able to apply strategies, *using* combinator is used:

```
using :: a → Strategy a → a
x ‘using‘ s = s x ‘pseq‘ x
```

It takes an argument, *a*, and a strategy, *Strategy a*. Evaluation of the argument depending on the inputted strategy is returned as the result.

5.2.2 Combined Original Strategies for Sequentialization and Parallelization: seqList, parList, seqMap, parMap.

Due to being just higher order function, second generation strategies could be combined to be passed like parameters or to be composed by the function composition operation. This combination is done either in sequence or in parallel.

For instance, strategy *seqList* applies the given strategy to every element of a given list in sequence:

```
seqList :: Strategy a → Strategy [a]
seqList strat [] = ()
seqList strat (x : xs) = strat x ‘pseq‘ (seqList strat xs)
```

There is no doubt that the same strategy could be changed as *parList* which applies the input strategy to each element of the given list in parallel:

```
parList :: Strategy a → Strategy [a]
parList strat [] = ()
parList strat (x : xs) = strat x ‘par‘ (parList strat xs)
```

Besides, a function could be applied to elements of a given list either in parallel or in sequence:

```
seqMap :: Strategy b → (a → b) → [a] → [b]
seqMap strat f xs = map f xs ‘using‘ seqList strat
```

seqMap takes a strategy, *Strategy b*, a function, $(a \rightarrow b)$, and a list, $[a]$, then applies the function together with the light of the inputted strategy to every element of the given list in sequence and returns the new generated list as its output.

$$
\begin{aligned}
&\mathit{parMap} :: \mathit{Strategy}\ b \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
&\mathit{parMap}\ \mathit{strat}\ f\ \mathit{xs} = \mathit{map}\ f\ \mathit{xs}\ \text{`}\mathit{using}\text{`}\ \mathit{parList}\ \mathit{strat}
\end{aligned}
$$

In contrast to *seqMap*; *parMap* applies the function to every element of the list with the input strategy in parallel and returns the new list as output.

5.3 Second Generation Strategies via Eval Monad

Second generation strategies by [11] do not change some number of features that are also provided by original ones. Here are the preserved features:

- The feature that introduces evaluation order which is left unspecified in Haskell environment stays unchanged.
- The abstraction layer which distinguishes between the parallel pragmas and algorithm is also left unchanged.

Second generation strategies bring some additional benefits onto the already existing ones, due to returning the arguments of computations inside a new data type constructor, *Eval*, (instead of returning unit types) which is proven as a monad in the next session, such as:

- Unlike original compositional strategies; the ones that are defined by the help of *Eval monad* allows garbage collector to define already evaluated (fizzled) sparks and discard them from the spark pool.
- Speculative parallelism is supported in second generation strategies which is impossible to implement in original ones due to only relying on the *root* garbage collection policy.

5.3.1 Eval Type Constructor as a Functor and Monad Instance

Eval monad, like Identity; does not make any change in the computation but it effects the evaluation order of it. For instance, some parts of the computation might be evaluated in parallel while others are done in a serial manner, with encapsulating the basic annotations of par and pseq. Below, you see the definition of *Eval* type in Haskell:

$$\mathit{data}\ \ \mathit{Eval}\,a\ =\ \mathit{Done}\,a$$

In order to unwrap the inside type, *runEval* function, whose signature given below, is used:

$$\begin{aligned}&\mathit{runEval}\ ::\ \mathit{Eval}\,a\ \rightarrow\ a\\&\mathit{runEval}\,(\mathit{Done}\,a)\ =\ a\end{aligned}$$

Eval as a Functor Class Instance

Haskell code that defines *Eval* as an *instance of functor class*:

$$\begin{aligned}&\mathit{class}\ \ \textbf{Functor}\ f\ \ \mathit{where}\\&\qquad \mathit{fmap}\ \ ::\ \ (a\rightarrow b)\rightarrow f\ \ a\rightarrow f\ \ b\\&\mathit{instance}\ \ \textbf{Functor}\ \ \textbf{Eval}\ \mathit{where}\\&\qquad \mathit{fmap}\,f\,x\ =\ (\mathit{return}\,.\,f)(\mathit{runEval}\,x)\end{aligned}$$

Lemma 5.3.1. *Type constructor* Eval : HASK ⟶ func *is a functor.*

Proof 5.3.2. *Category theoretic rules, to represent a functor, that are given in subsection 4.2.1 must be shown to satisfy.*

1. *proof via* Eval Integer *type:*

```
-- FIdA = IdFA; fmap id = id
1. runEval(Main. fmap Main. id (encap 10))
   = runEval(Main. id (encap 10))
--  F(g . h) = F g . F h; fmap (f . g) = (fmap f) . (fmap g)
2. runEval(Main. fmap h(Main. fmap f (encap 10)))
   = runEval((Main. fmap (h. f))
where f, h :: Integer -> Integer; f x = x + 5,  h x = x + 10
```

2. *proof via* Eval Char *type:*

```
-- FIdA = IdFA; fmap id = id
1. runEval(Main. fmap Main. id (Main. return 'a'))
   = runEval(Main. id (Main. return 'a'))
-- F(g . h) = F g . F h; fmap (f . g) = (fmap f) . (fmap g)
2. runEval(Main. fmap (Char. toLower)
   (Main. fmap(Char. toUpper)(Main. return 'a')))
   = runEval((Main. fmap ((Char. toLower). (Char. toUpper)))
   (Main. return 'a'))
```

Eventually;

- $Eval\ :\ Obj\ (HASK) \longrightarrow Obj\ (func)$

  $Integer \longmapsto Eval\ Integer$

  $Char \longmapsto Eval\ Char$

  $Float \longmapsto Eval\ Float$

- $Eval\ :\ Morph\ (HASK) \longrightarrow Morph\ (func)$

  $(Integer \rightarrow Integer \rightarrow ...) \longmapsto Eval\ (\ Integer \rightarrow Integer \rightarrow ...\ )$

  $(Char \rightarrow Char \rightarrow ...) \longmapsto Eval\ (\ Char \rightarrow Char \rightarrow ...\ )$

  $(Float \rightarrow Float \rightarrow ...) \longmapsto Eval\ (\ Float \rightarrow Float \rightarrow ...\ )$

Such as *Maybe*, [], *State s* and *Identity* type constructors; *Eval* is also said to be a functor defined between category *HASK* and its subcategory *func*. **Eval as a Monad Class Instance**

$$\begin{aligned}
&\mathit{class}\ \textbf{Functor}\ f\ \ \mathit{where}\\
&\quad fmap\ \ ::\ (a \rightarrow b) \rightarrow f\ \ a \rightarrow f\ \ b\\
&\mathit{instance}\ \textbf{Functor}\ \ \textbf{Eval}\ \mathit{where}\\
&\quad fmap f\ x\ =\ (return\ .f)(runEval\ x)\\
&\mathit{class}\ \textbf{Monad}\ \ m\ \ \mathit{where}\\
&\quad join\quad\ \ ::\ m\ \ (m\ \ a) \rightarrow m\ \ a\\
&\quad return\ \ ::\ a \rightarrow m\ \ a\\
&\quad bind\quad\ ::\ m\ \ a \rightarrow (\ a \rightarrow m\ \ b) \rightarrow m\ \ b\\
&\mathit{instance}\ \textbf{Monad}\ \ \textbf{Eval}\ \ \mathit{where}\\
&\quad return\ x\ =\ Done\ x\\
&\quad join\ x\ =\ runEval\ x\\
&\quad x >>= k\ =\ k\ x
\end{aligned}$$

Lemma 5.3.3. *The triple* (Eval + fmap, join, return)*, equivalently* (Eval, bind, return) *is a monad in Haskell environment.*

Proof 5.3.4. *Therefore, category theoretic rules, to represent a monad, that are given in subsection 4.2.2 must be shown to satisfy for all of the data constructors of given type that is a basic encapsulated function, here.*

1. *proof via* Eval Integer *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. runEval(Main. join(Main. fmap
   Main. join (encap(encap(encap 4)))))
   = runEval(Main. join( Main. join (Main. fmap
   Main. id(encap (encap(encap 4))))))
-- join . return (fmap  id)  = join . (fmap  return)
2. runEval(Main. join(Main. fmap Main. return(encap 4)))
   = runEval(Main. join(Main. return(Main. fmap
   Main. id(encap 4))))
-- return . f  = (fmap f) . return
3. runEval(Main. return(f 10))
   = runEval(Main. fmap f(Main. return 10))
-- join . (fmap (fmap f)) = (fmap f) . join
4. runEval(Main. join(Main. fmap(Main. fmap f)(encap(encap 10))))
   = runEval(Main. fmap f(Main. join(encap(encap 10))))
where f :: Integer -> Integer; f x = x + 5 and id x = x
```

2. *proof via* Eval Char *type:*

```
-- join . join (fmap id)  = join . (fmap join)
1. runEval(Main. join(Main. fmap Main. join (Main. return
   (Main. return(Main. return 'a')))))
   = runEval(Main. join( Main. join (Main. return(Main. return
   (Main. return 'a')))))
-- join . return (fmap  id)  = join . (fmap  return)
2. runEval(Main. join(Main. fmap Main. return(Main. return 'a')))
   = runEval(Main. join(Main. return(Main. return 'a')))
-- return . f  = (fmap f) . return
3. runEval(Main. return(Char. toUpper 'a'))
   = runEval(Main. fmap Char. toUpper(Main. return 'a'))
-- join . (fmap (fmap f)) = (fmap f) . join
4. runEval(Main. join(Main. fmap(Main. fmap Char. toUpper)
   (Main. return(Main. return 'a'))))
   = runEval(Main. fmapChar. toUpper(Main. join
   (Main. return (Main. return 'a'))))
 where id x = x
```

As the above equalities hold, the triple $(Eval + fmap, return, join)$ or equally $(Eval, return, bind)$ is proven as a monad. As mentioned, the main idea behind the implementation of *Eval* monad is to encapsulate the *par* and *pseq* annotations of Haskell language to affect the evaluation order and solve some memory management problems arise in original strategies, by this way parallelization is getting closer to implicit one since the amount of help done for compiler is increasing.

Together with the help of *Eval* monad, below you see the new definition of strategies:

```
type Strategy a = a → Eval a
```

which is again a function, takes an argument *a* as input, lifts or puts that input into the *Eval* monad (container) and returns that container.

5.3.2 Original Strategies Revisited (Second Generation Strategies): rpar, rseq, rdeepseq, parList

In this session, second generation strategies are defined and demonstrated via their type signatures.

The first one is correspondence of *r*0 in original strategies. Again, it does not have any effects in evaluation order or reduction, it only lifts or puts the input argument into *Eval* monad. Its name also remains unchanged:

```
r0 :: Strategy a
r0 x = return x
```

The second one corresponds to *rwhnf* in original strategies. Similarly, it takes an argument, *x*, evaluates it into its *WHNF* and then returns its lifted version which is in the *Eval* monad, in serial.

```
rseq :: Strategy a
rseq x = x ‘pseq‘ return x
```

The third one does not exist in original strategies. Similar to *rseq*, it takes an argument, *x*, evaluates it into its *WHNF* and returns its lifted version in the *Eval* monad, but in parallel.

```
rpar :: Strategy a
rpar x = x ‘par‘ return x
```

The fourth strategy, namely *rdeepseq* and correspondingly *rnf* in original strategies, takes an argument as an input and evaluates it into its *NF* and lifts the evaluated version into *Eval* monad and returns it as output.

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x ‘pseq‘ return x
```

Analogous to original strategies, in second generation strategies, an input strategy could be applied to each element of a given list in parallel.

```
parList :: Strategy a → Strategy [a]
parList s = evalList (rpar ‘dot‘ s)
```

where

```
evalList :: Strategy a → Strategy [a]
evalList s [] = ()
evalList s (x : xs) = do x′ < − s x
                         xs′ < − evalList s xs
                         return (x′ : xs′)
dot :: Strategy a → Strategy a → Strategy a
s2 ‘dot‘ s1 = s2 . runEval . s1
```

evalList traverses the list and forces each element for a strategy to be applied while *dot* combines given two strategies and returns the combined strategy.

In the sense of *evalList* strategy in second generation ones, input strategy is applied to each element of the list in parallel.

5.3.3 New Memory Management Methodology

New generation of strategies, that are defined in the previous session, supply the additional feature of returning arguments of an arbitrary strategy inside the *Eval* monad, instead of returning a void pointer. This innovation lets compiler do memory management much more feasibly by using a smart garbage collector.

In oder to be able to make benefits of second generation strategies visible, first of all, the problem arising from the usages of original strategies should be explained. For that reason, let's consider the *par* annotation which saves a pointer to the heap node, namely *spark pool*, to represents its first argument inside. At exactly this point, the management of that *spark pool* comes to the surface which involves the problematic case of *when to remove the spark from the pool.*

Let's go a little deeper in spark pool management mechanisms. Here, two main policies are given by using the terminology of [14]:

1. Weak Policy: In this policy weak references are used. Weak references permit garbage collector to collect any object although it could be accessed by main program. They are generally used in the programs demand too much memory space to run. It is tempting to remember that collecting them is as easy as recreating them. (does not keep the object it refers to alive.)

2. Root Policy: Unlike weak policy, in the root policy, strong references are used which means while main program has an access to the referenced object in a way, garbage collection cannot collect it, only when main program loses the access or reference, then this object could be collected. (keeps the object it refers to alive.)

Both policies end up with problematic issues, if they are used together with the original strategies:

1. Weak policy causes all potential parallelism to be lost, since in the case of garbage collector collects all of the created sparks, it is not possible to recreate the references again. Because, original strategies do not return any information to its caller.

2. On the other side, Root policy supports potential parallelism to be implemented. However, if there are no enough processors available, then *most of the sparks* will never be fizzled and will be referenced as space leaks, unless their reference is canceled by the main program (if they are not needed any more). [25]

5.3.3.1 Fizzled Sparks

A spark fizzles when the expression that it points is evaluated to its normal form (not possible to reduce it any more) by the main thread, not by the one it is assigned to be evaluated in parallel. Therefore, if a spark is fizzled, then it should be dismissed from the pool. However, composed original strategies do not provide this kind of a feature, since they do not return information about the expression's evaluation degree to its caller (program). In other words, no mater if it is fizzled or not, most of the sparks are converted into real operating system threads causing space leaks, although they are potentially fizzled.

In second generation strategies, fizzled sparks are not allowed to be converted into real operating system threads; instead they are pruned. Since together with the usage of *rpar* annotation, the evaluation degree information of every expression has been put in spark pool is returned to the caller into a container that is *Eval* monad. By this way, garbage collection could be done in a better way in order to prevent *space leaks* in the memory which means that while an expression has already been evaluated to its *NF* is discarded, another one that is still under evaluation could be reached depending on the demand of the main program (caller).

5.3.3.2 Speculative Parallelism

Speculative parallelism could be used when there is an inner dependency among the threads expected to be run in parallel without waiting affiliated thread to halt, dependent thread could start its evaluation by predicting the demanded value, soon after, affiliated thread returns a result, predicted value is compared with it, if any difference arises, then result of the dependent thread is scaled in association with the result of the affiliated thread.

In *root garbage collection* policy, *speculative parallelism* is not supported, since any speculative spark would become a space leak. On the other hand, *weak policy prunes* speculative sparks from the pool, if they are not reachable any more.

5.3.3.3 Life Cycle of a Spark

In advance of mentioning about the performance analyses, most important notice should be the life cycle of a GHC spark. In this sense; a spark is called:

- *overflowed*, if the spark pool is already full.
- *dud*, if it is already evaluated before the creation.
- *created*, if it is put in the spark pool.

Soon after a sparks is created, then it could be threated in three ways. It is said; a spark could be:

- *converted*, if it runs in parallel with the others by the Haskell Execution Context it was assigned before.
- *fizzled*, if it is evaluated into its Normal Form by the main thread and called *pruned* when it is dismissed out of the pool by the garbage collector during the evaluation.
- *garbage collected*, if it is not needed during the evaluation. The idea is discarding these sparks right after the evaluation (for sure) with the help of garbage collector. This option is only valid for second generation strategies due to the usage of *weak references* as the garbage collection policy which enables not to evaluate unnecessary sparks for the computation.

Note 5.3.5. *The lifecycle of a spark, installation and usage details of program called Threadscope could be found on this website:* `http://www.haskell.org/haskellwiki/ThreadScope_Tour`.

Note 5.3.6. *Different number of threads could be created when the same parallelization effort spent for the same algorithm by second generation and original strategies. The reason for that originates from GHC that has a non-deterministic execution model in which a particular expression may be evaluated multiple times at runtime.*

Basically; implementation allows multiple processors to evaluate any thunk at the same time that is not tried to be prevented. The actual attempt of second generation strategies focuses on detecting large amount of duplicate evaluations. That means that number of sparks created in the evaluation of an arbitrary thunk completely depends on the number of processors performed the evaluation in question at the same time. (from the explanation of Dr. Simon Marlow.)

Note 5.3.7. *In both second generation and original strategies* total number of created sparks *might not be equal to* created sparks + fizzled sparks *that means that there might be some number of sparks created but neither converted nor fizzled. Since, suppose that an arbitrary thunk, involving* par *annotations, is started to be evaluated by more than one core which yields in the creation of multiple number of sparks for the same job (spark pools are actually circular heaps, existing one for each core, that are managed by* lock-free *and* work stealing *manner for load balancing issues).*

From that point on, some of these sparks will be fizzled, some others will be converted into OS threads to provide parallelism and the rest will not be needed during the evaluation, since sub-tasks assigned to them are already evaluated by other spark(s).

Note 5.3.8. *The number of created sparks for each parallelization attempt performed by using original and second generation strategies are more or less the same that demonstrates that both of the two formulations are expressing the same parallelism.*

Following chapter of the project demonstrates the implementations of some algorithms that could be parallelized by using both original and second generation strategies, comparatively in the senses of memory management and elapsed time performance issues.

Chapter 6

Comparisons of Original and Second Generation Strategies

In order to be able to illustrate and compare original and second generation strategies, here, in this chapter of the project, parallelized versions of calculating demanded Fibonacci number within the series, sorting a given list with quick-sort algorithm, RSA encryption, decryption of given texts and Karatsuba multiplication of large numbers are given as examples both with original and second generation strategies with their performance analyses, comparatively.

6.1 Parallel Fibonacci

The well-known function which calculates the intended element of the Fibonacci series is written in both of the parallelization manners and performance evaluations are given in this part. The recursive function also checks the depth of the parallelization to be able to finalize it at a demanded level.

First of all, lets evaluate the parallel code without using strategies, only by annotations provided by Haskell runtime system, that takes the current parallelization depth, limit of it with the sequence of intended Fibonacci number in the series and returns the corresponding value.

Function first checks whether current parallelization depth is equal to or greater than the limit, if yes, then it no more creates sparks to be evaluated in parallel, instead it calls the serial version of the function which is also recursive. Otherwise, function creates a spark for its each recursive call to calculate the Fibonacci series values of two numbers, one of which is one and the other is two less than the input value, in parallel. These recursive calls proceed until demanded parallelization depth or the base case $(n <= 1)$ is reached.

```
fibParWs :: Integer → Integer → Integer → Integer
fibParWs currentDepth limit n
          | n <= 1 = 1
          | currentDepth >= limit = fibSer n
          | otherwise = x ‘par‘ (y ‘pseq‘ x + y)
             where
              x = fibParWs (currentDepth + 1) limit (n − 1)
              y = fibParWs (currentDepth + 1) limit (n − 2)
```

In the above code, algorithm and the annotations providing parallelism, (*par and pseq*), are not split off each other. As emphasized in the section which explains the reason why strategies are standing for, it is better to distinguish between the algorithm and the parallel annotations in parallel with the motto *Algorithm* + *Strategies* = *Parallelism*.

```
fibParOr :: Integer → Integer → Integer → Integer
fibParOr currentDepth limit n
          | n <= 1 = 1
          | currentDepth >= limit = fibSer n
          | otherwise = x + y ‘using‘ strategy
             where
              x = fibParOr (currentDepth + 1) limit (n − 1)
              y = fibParOr (currentDepth + 1) limit (n − 2)
              strategy res = (rnf x) ‘par‘ (rnf y)
```

In both of the codes above and below, the crucial idea is to divide the problem that indicates the calculation of the correspondence of the input values in the Fibonacci series, into two sub problems (calculation of the values fib (n-1) and (n-2)) after than assigning them to the GHC sparks and calling the function recursively until the base case or enough depth is reached. In other words, a kind of tree structure, demonstrated below, is constructed. At each level of the tree, one GHC spark is created and assigned to the relevant calculation (evaluation of 'x') which is expected to be evaluated to their *normal forms* by the usages of *rnf* and *rpar* functions. Evaluation of 'y' is already done by main thread, so there is no need to create one spark also for it.

```
fibParEval :: Integer → Integer → Integer → Integer
fibParEval currentDepth limit n
          | n <= 1 = 1
          | currentDepth >= limit = fibSer n
          | otherwise = x + y ‘using‘ strategy
             where
              x = fibParEval (currentDepth + 1) limit (n − 1)
              y = fibParEval (currentDepth + 1) limit (n − 2)
              strategy s = do {rpar x; rseq y}
```

Furthermore, the main difference between the semantics of the functions *fibParOr* and *fibParEval* is the memory management mechanism of second generation strategies which allows fizzled sparks to be discarded out of the pool without evaluated it once again (due to usage of Eval monad) which supplies prevention of possible space leaks in the main memory.

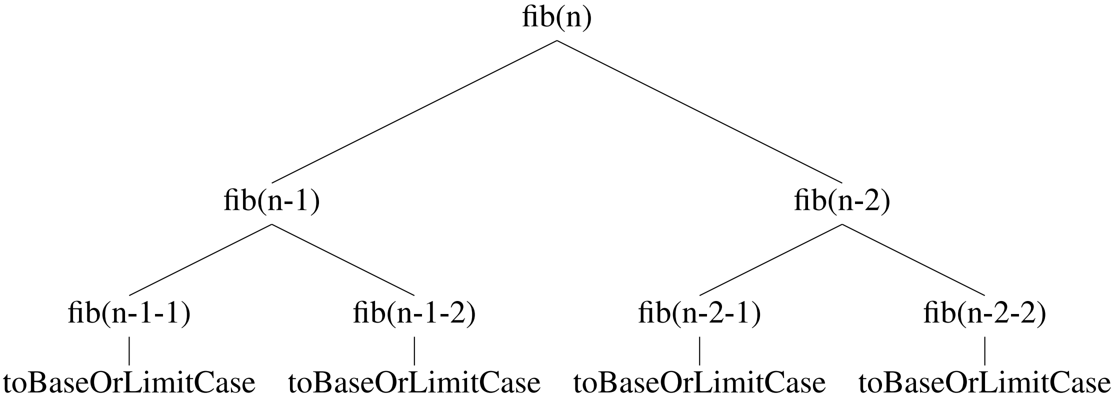

FIGURE 6.1: Fibonacci n Calculation Tree

Performance analyses of the calculations of Fibonacci 35, 37 and 40, in both original and second generation strategies, are comparatively plotted on graphs and the program called *threadscope* is used to evaluate *eventlog* extended files of the calculations visually to be able to see the occurrence of the parallelization.

Note 6.1.1. *The performance analyses that are given in this part is carried out on Intel Core™ 2 Quad Processor Q6600 (8M Cache, 2.40 GHz, 1066 MHz FSB) CPU and 64-bit Ubuntu 11.10 OS running, environment.*

In all of the performance analysis tables, for each parallelization depth from 1 to 32, handled elapsed times of each evaluation, total number of sparks with created and pruned ones and maximum amount of heap space used (maximum heap residency) together with total memory usages are given, respectively.

As seen in the Fibonacci 35, 37 and 40's performance tables; total number of created sparks both for original and second generation strategies at each level are more or less the same; which demonstrates that the same conditions are satisfied for the comparison issue, therefore comparison could now be focused on the parts showing the handled elapsed times, number of created sparks and maximum amount of heap spaces occupied during the parallel evaluation.

For the evaluation of Fibonacci 35 over 2 cores; *Second Generation Strategies* give the best improvement in the elapsed time of 0.83 seconds at 16^{th} level with maximum heap residency of 15224 bytes by using only 28 sparks converted. On the other side of the comparison, *Original Strategies* serve the best elapsed time as 0.82 seconds again at parallelization depth 16 with 22384 bytes of maximum heap residency by converting 29741 sparks. Since, original strategies evaluate all of the sparks although some of them are not actually needed for the whole computation (sparks to which the same sub-calculation is assigned), due to root garbage collecting policy.

The same evaluation over 3 and 4 cores end up with more or less same situations. Over 3 cores, the best time handled by *Original Strategies* is 0.59 seconds at depth 4 with 15 sparks converted and 38752 bytes of heap residency. *Second Generation Strategies* could reach its best performance at parallelization depth number 8 with 0.60 seconds, only 9 of 255 sparks is converted with the same heap residency of 38752 bytes.

Over 4 cores, *Original Strategies* provide its best performance in elapsed times at both 8^{th} and 12^{th} levels with 0.51 seconds. At level 8; all of the created 274 sparks are converted into real OS threads resulting in 39776 bytes of maximum heap residency. *Second Generation Strategies* returns the result of the evaluation in the fastest way at depth 16 in 0.53 seconds by only converting 50 sparks into real OS threads with the 38752 bytes of maximum heap residency.

In the above table, performances of calculating Fibonacci 35 is examined detailedly. Even though, original strategies seem a little ahead of second generation strategies in terms of elapsed time performances, heap residencies demonstrate that the same job is done by using less space allocation in the heap by second generation strategies.

TABLE 6.1: Fibonacci 35

Number of Cores	*Strategy Type*	*Best Time (sec.)*	*Average Time (sec.)*	*Max. Heap Residency of Best (bytes)*	*Avg. Heap Residency (bytes)*
2	Original	0.82	0.88	22384	14285
	Sec.	0.83	0.85	15224	15598
3	Original	0.59	0.66	38752	45560
	Sec.	0.60	0.70	38752	34532
4	Original	0.51	0.60	39776	41423
	Sec.	0.53	0.61	38752	38030

TABLE 6.2: Fibonacci 37

Number of Cores	*Strategy Type*	*Best Time (sec.)*	*Average Time (sec.)*	*Max. Heap Residency of Best (bytes)*	*Avg. Heap Residency (bytes)*
2	Original	1.57	1.66	13640	13116
	Sec.	1.63	1.69	13664	11131
3	Original	1.28	1.39	27584	18364
	Sec.	1.25	1.37	11864	13691
4	Original	1.14	1.30	24312	21896
	Sec.	1.16	1.31	24200	18734

TABLE 6.3: Fibonacci 40

Number of Cores	*Strategy Type*	*Best Time (sec.)*	*Average Time (sec.)*	*Max. Heap Residency of Best (bytes)*	*Avg. Heap Residency (bytes)*
2	Original	5.47	5.88	13952	12049
	Sec.	5.50	5.86	14072	11662
3	Original	3.98	4.69	19488	15519
	Sec.	3.96	4.42	18632	14666
4	Original	3.62	4.38	24608	18933
	Sec.	3.44	4.01	23584	17582

The same examination is also done for evaluations of Fibonacci 37 and 40. In these cases, more or less, same conditions arise. Some number of created sparks are fizzled and discarded from the spark pool preventing possible space leaks on the memory.

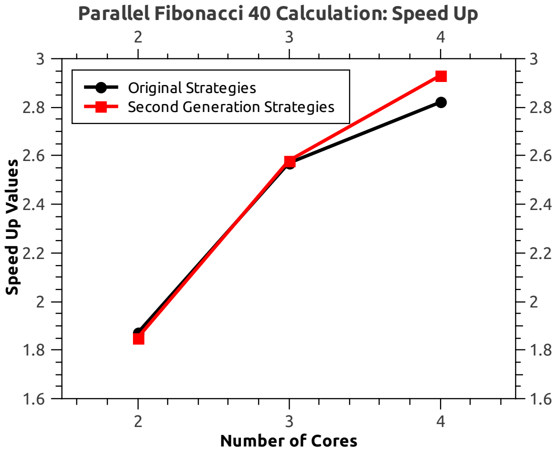

FIGURE 6.2: Parallelization of Fibonacci 40: Speed Up Values

For more details, related tables in the appendix part could be examined, in the version which parallelization implemented by original strategies, maximum heap residencies are behind of second generation ones but they perform better timings than second generation ones. Since, in second generation strategies, fizzled sparks are discarded as soon as being noticed, but in original ones, to be able to understand whether a spark is fizzled or not it should be evaluated once again creating extra workload for CPU resulting in waste of time that indicates that extra evaluation in original strategies took less time than big amount of garbage collection performed in second generation ones.

6.2 Parallel QuickSort

Quicksort is a very well-know sorting algorithm in computer science which also could be implemented by recursive coding style as defining a number in Fibonacci series. Algorithm divides the input list into its sublists recursively until a base point is reached and sorts the handled sublists depending on the selected pivot elements: lower elements are put left part of the pivot while higher ones are placing on the right part. If pivot element is selected randomly, then algorithm is called randomized quicksort.

Parallelization is provided by assigning a spark to sort each sublist. Parallelized quicksort algorithm which are coded both by original and second generation strategies are given below.

```
orQuickSortPar :: Integer → Integer → [Integer] → [Integer]
orQuickSortPar _ _ [] = []
orQuickSortPar currentDepth limit (x : xs)
                 | currentDepth >= limit = sort (x : xs)
                 | otherwise = (k ++ (x : l)) `using` strategy
                    where
                     k = orQuickSortPar (currentDepth + 1) limit [y | y <- xs, y <= x]
                     l = orQuickSortPar (currentDepth + 1) limit [y | y <- xs, y > x]
                     strategy res = (rnf k) `par` (rnf l) `pseq` (rnf res)
```

In the above code, a compositional strategy *res* is used to reduce the recursive jobs *k* and *l* to their normal forms in parallel which means for every recursive call of the main function, one spark is created and assigned to evaluate sorting operation, *k*, completely in parallel with main threads job, *l*. Evaluation of strategy itself to normal form, *rnf res*, provides the guarantee to the fully evaluations of the jobs *k* and *l*. It is tempting to note that if *rnf res* is removed from the code, result will not change and the difference between the elapsed times, with and without it, will be negligible.

```
EvalQuickSortPar :: Integer → Integer → [Integer] → [Integer]
EvalQuickSortPar _ _ [] = []
EvalQuickSortPar currentDepth limit (x : xs)
                 | currentDepth >= limit = sort (x : xs)
                 | otherwise = (k ++ (x : l)) `using` strategy
                    where
                     k = EvalQuickSortPar (currentDepth + 1) limit [y | y <- xs, y <= x]
                     l = EvalQuickSortPar (currentDepth + 1) limit [y | y <- xs, y > x]
                     strategy res = do (rpar k)
                                       (rseq l)
                                       (rdeepseq res)
```

The same sparking manner is also used for second generation strategies by sparking job *k* in parallel with *l*. However, this time *rdeepseq res* had to be used in order to evaluate jobs *k* and *l* to their normal forms. Since, *rpar* and *rseq* are not the strategies to evaluate their parameters completely.

Performance analyses of sorting 1 and 2 millions of randomly generated numbers, are summarized into below tables such as the ones standing to demonstrate performances of parallel Fibonacci algorithm. Over 2, 3 and 4 cores, best elapsed times, their corresponding heap residencies; average elapsed times measured with average heap residencies according to parallelization depths are depicted in the tables.

TABLE 6.4: Quicksort 1.000.000 Random Numbers

Number of Cores	*Strategy Type*	*Best Time (sec.)*	*Average Time (sec.)*	*Max. Heap Residency of Best (bytes)*	*Avg. Heap Residency (bytes)*
2	Original	3.83	3.99	55265480	55747120
	Sec.	3.55	3.74	58246468	57351696
3	Original	3.79	3.90	50415960	51295302
	Sec.	3.54	3.71	50045216	51767145
4	Original	3.69	3.78	49815776	63333515
	Sec.	3.47	3.66	57118432	53385031

In the above case, second generation strategies seem ahead of original ones in the sense of elapsed time performances of the evaluation whose serial measured time is 7.08 seconds. Additionally, averages of maximum heap residencies during the second generation are also by second generation strategies' side. Since, maximum heap residencies of best measured time could convince the reader due to the reason why they could be measured at different parallelization depths which does not let comparison to be done. In order to avoid this condition, it is much more meaningful to take average values into account in both cases.

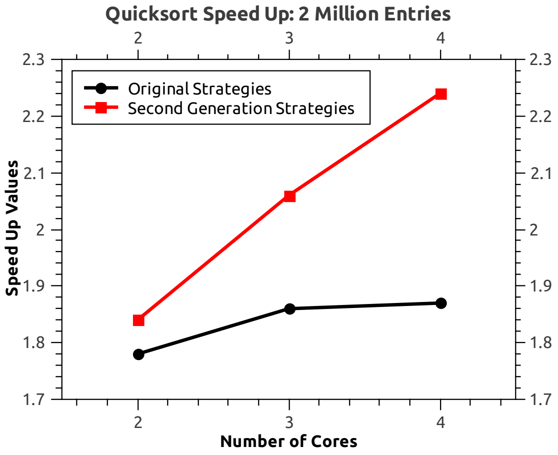

FIGURE 6.3: Quicksort Speed Up: 2 Million Entries

TABLE 6.5: Quicksort 2.000.000 Random Numbers

Number of Cores	*Strategy Type*	*Best Time (sec.)*	*Average Time (sec.)*	*Max. Heap Residency of Best (bytes)*	*Avg. Heap Residency (bytes)*
2	Original	9.53	11.11	84299840	128311643
	Sec.	8.87	9.40	128483984	127573634
3	Original	8.78	10.44	84786832	115139868
	Sec.	7.91	8.64	96010296	107357999
4	Original	8.73	9.26	102512456	105088678
	Sec.	7.29	8.14	82052784	91115522

Parallelization with second generation strategies are again ahead of original strategies, in both senses, for sorting 2 million randomly generated numbers which serially takes 16.33 seconds to succeed. For more detailed information tables in the appendix part could be checked. These tables involve measurements for both manner of strategies such as measured times, maximum heap residencies, total number of sparks, converted sparks, pruned sparks achieved at each parallelization depth (1, 2, 4, 8, 12, 13, 14 and 16).

6.3 Parallel RSA Cryptosystem

RSA is a well-known public key (asymmetric) cryptosystem developed by Ron Rivest, Adi Shamir and Leonard Adleman in 1978 whose presumed difficulty is based on prime factorization of a given integer. In this part of the project; encryption and decryption schemes of cryptosystem are parallelized and compared with regards to both original and second generation strategies.

Parallelization is not thought in mathematical calculation parts in recursive manner, instead it is implemented as dividing the given plain or cipher texts into sub texts, applying the same function over them and lasts with the combination of the texts which were encrypted or decrypted in parallel.

6.3.1 Parallel RSA Encryption Scheme

In this section encryption scheme of RSA cryptosystem is parallelized; codes and measured results with their log files that were processed by threadscope are given.

```
split4ToEncOr :: [Integer] → [Integer]
split4ToEncOr (PUB n e) [] = []
split4ToEncOr (PUB n e) (x : xs) = (d ++ c ++ b ++ a) `using` strategy
          where
           len = length (x : xs)
           (firstHalf, secondHalf) = Main.splitAt (len `quot` 2) (x : xs)
           (secondPart1, secondPart2) = Main.splitAt (len `quot` 4) secondHalf
           a = (ersa (PUB n e) firstPart1)
           b = (ersa (PUB n e) firstPart2)
           c = (ersa (PUB n e) secondPart1)
           d = (ersa (PUB n e) secondPart2)
           strategy res = (rnf a) `par` (rnf b) `par` (rnf c) `par` (rnf d)
```

Above and below codes divide the given integer list (after text to integer conversion) into 4 sublists and create a spark to apply encryption function namely, *ersa*, to each sublist in parallel over 4 cores. Totally; three sparks are created to evaluate the jobs *a*, *b*, *c* and main thread does the evaluation of job *d* into their normal forms by using both original and second generation strategies. Other versions of the function also exist such as dividing the given list into 2 and 3 sub-lists to make it executed over 2 and 3 cores.

```
split4ToEncEval :: [Integer] → [Integer]
split4ToEncEval (PUB n e) [] = []
split4ToEncEval (PUB n e) (x : xs) = (d ++ c ++ b ++ a) `Main.using` strategy
          where
           len = length (x : xs)
           (firstHalf, secondHalf) = Main.splitAt (len `quot` 2) (x : xs)
           (secondPart1, secondPart2) = Main.splitAt (len `quot` 4) secondHalf
           a = (ersa (PUB n e) firstPart1)
           b = (ersa (PUB n e) firstPart2)
           c = (ersa (PUB n e) secondPart1)
           d = (ersa (PUB n e) secondPart2)
           strategy s = do  (rpar `dot` rdeepseq a)
                            (rpar `dot` rdeepseq b)
                            (rpar `dot` rdeepseq c)
                            (rseq `dot` rdeepseq d)
```

Performance analyses of parallel encryption scheme for 50, 100 and 125K[1]of plain texts with 186 bits of encryption key are given below whose serial elapsed times are 7.36, 7.73, 8.09 seconds that are measured during the division process of input plain text into 4, 3 and 2 sub-lists + their encryption processes implemented over a single core for 50K of input text, respectively. Corresponding results for 100K are 23.79, 24.94 and 26.89 seconds and 34.86, 36.61, 40.05 seconds for 125K of input plain text.

TABLE 6.6: RSA Encryption of 50K plain text

Number of Cores	*Strategy Type*	*Parallel Time (sec.)*	*Max. Heap Residency of Best (bytes)*
2	Original Sec.	6.18 6.13	2444576 2335496
3	Original Sec.	5.50 5.27	2449304 2474656
4	Original Sec.	5.09 4.92	1722824 1719392

In other words; serial encryption of 50K list (after text to integer conversion) on a single core after dividing it into 2, 3 and 4 sub-lists takes 8.09, 7.73 and 7.36 seconds to complete. Therefore, parallelization in this case causes to speed the evaluation up 1.32, 1.47 and 1.50 times for second generation; 1.31, 1.41 and 1.45 times for original strategies over 2, 3 and 4 cores.

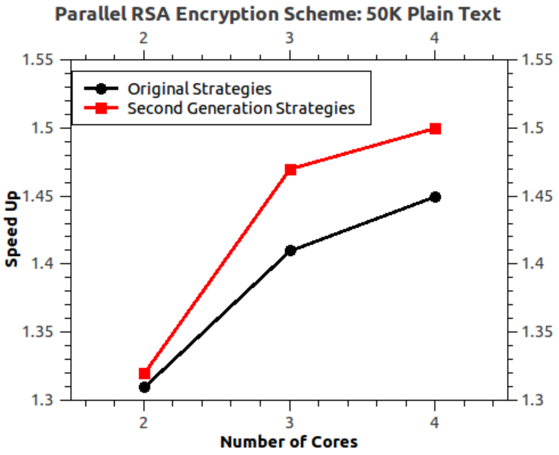

FIGURE 6.4: Parallel RSA Encryption Scheme: 50K Plain Text

For serial encryption of 100K list (after text to integer conversion)on a single core after dividing it into 2, 3 and 4 sub-lists with takes 26.89, 24.94 and 23.79 seconds

[1] 1K = 1024 bytes

TABLE 6.7: RSA Encryption of 100K plain text

Number of Cores	*Strategy Type*	*Parallel Time (sec.)*	*Max. Heap Residency of Best (bytes)*
2	Original	22.26	4892176
	Sec.	22.16	4992200
3	Original	20.31	5433968
	Sec.	20.28	5296960
4	Original	19.46	5104312
	Sec.	18.94	5032104

to complete. Therefore, parallelization speeds the evaluation up 1.21, 1.23 and 1.26 times for second generation; 1.21, 1.22 and 1.22 times for original strategies over 2, 3 and 4 cores.

For 125K list, more or less, the same performances are handled with speed up values of 1.10, 1.15 and 1.18 for second generation; 1.06, 1.15 and 1.17 for original strategies. For the details; below table could be examined.

TABLE 6.8: RSA Encryption of 125K plain text

Number of Cores	*Strategy Type*	*Parallel Time (sec.)*	*Max. Heap Residency of Best (bytes)*
2	Original	37.87	5656744
	Sec.	36.99	5575016
3	Original	31.83	6168648
	Sec.	31.73	6128688
4	Original	29.85	5949632
	Sec.	29.55	5816496

Above three tables demonstrate that second generation and original strategies have nearly the same performance results both in the elapsed time and maximum heap space allocated during the evaluation. The differences are all negligible, since they originate from the benchmarking process performed one after another. Since, the number of created, converted, fizzled and unneeded sparks are completely same. The main difference in the design criteria of second generation and original strategies is that in second generation ones, the whole effort is spent to catch large amounts of duplicate spark evaluations and to discard them. In this case, all created sparks are converted to encrypt different sub-lists therefore, there was no duplication to be caught causing performance differences.

6.3.2 Parallel RSA Decryption Scheme

The same division and sparking manner in parallel encryption scheme is also used to parallelize decryption scheme of RSA into 2, 3 and 4 sub-lists to be able to run them over 2,3 and 4 cores in parallel. Full working codes could be found in the appendix part of the thesis. Measured results are given in the same manner with the above one. Below code parallelizes RSA decryption scheme by original strategies:

```
split4ToDecOr :: [Integer] → [Integer]
split4ToDecOr (PRIV n d) [] = []
split4ToDecOr (PRIV n d) (x:xs) = (e ++ c ++ b ++ a) ‘Main.using‘ strategy
                where
                 len = length (x:xs)
                 (firstHalf, secondHalf) = Main.splitAt (len ‘quot‘ 2) (x:xs)
                 (secondPart1, secondPart2) = Main.splitAt (len ‘quot‘ 4) secondHalf
                 a = (drsa (PRIV d e) firstPart1)
                 b = (drsa (PRIV d e) firstPart2)
                 c = (drsa (PRIV d e) secondPart1)
                 d = (drsa (PRIV n e) secondPart2)
                 strategy res = (rnf a) ‘par‘ (rnf b) ‘par‘ (rnf c) ‘par‘ (rnf e)
```

An the code parallelizing RSA decryption scheme by second generation strategies:

```
split4ToDecEval :: [Integer] → [Integer]
split4ToDecEval. (PRIV n d) [] = []
split4ToDecEval. (PRIV n d) (x:xs) = (e ++ c ++ b ++ a) ‘Main.using‘ strategy
                where
                 len = length (x:xs)
                 (firstHalf, secondHalf) = Main.splitAt (len ‘quot‘ 2) (x:xs)
                 (secondPart1, secondPart2) = Main.splitAt (len ‘quot‘ 4) secondHalf
                 a = (drsa (PRIV n d) firstPart1)
                 b = (drsa (PRIV n d) firstPart2)
                 c = (drsa (PRIV n d secondPart1)
                 d = (drsa (PRIV n d) secondPart2)
                 strategy s = do  (rpar ‘dot‘ rdeepseq a)
                                  (rpar ‘dot‘ rdeepseq b)
                                  (rpar ‘dot‘ rdeepseq c)
                                  (rseq ‘dot‘ rdeepseq e)
```

Performance analyses of parallel decryption scheme for 50, 100 and 125K of cipher texts with 188 bits of decryption key are given below whose serial elapsed times are 7.14, 7.35, 7.94 seconds that are measured during the division process of input cipher text into 4, 3 and 2 sub-lists + their decryption processes implemented over a single core for 50K of input text, respectively. Corresponding results for 100K are 23.86, 25.17 and 26.84 seconds and 37.80, 38.61, 40.02 seconds for 125K of input cipher text.

TABLE 6.9: RSA Decryption of 50K cipher text

Number of Cores	*Strategy Type*	*Parallel Time (sec.)*	*Max. Heap Residency of Best (bytes)*
2	Original	6.07	2401464
	Sec.	6.05	2317680
3	Original	5.52	2477016
	Sec.	5.26	2499048
4	Original	5.12	1772792
	Sec.	4.93	1719392

For decryption process of 50K cipher text, speed up values are 1.31, 1.40 and 1.45 for second generation; 1.31, 1.33 and 1.39 for original strategies over 2,3 and 4 cores.

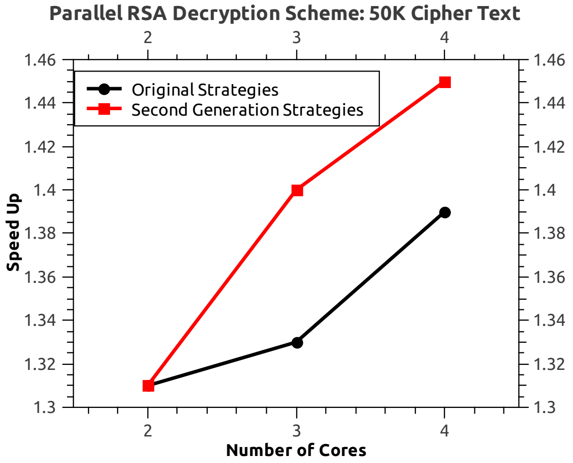

FIGURE 6.5: Parallel RSA Decryption Scheme: 50K Cipher Text

Speed up values for the decryptions 100K and 125K of cipher texts are 1.21, 1.22, 1.24; 1.08, 1.13, 1.17 for second generation; 1.21, 1.22, 1.22; 1.06, 1.15 and 1.17 for original strategies, respectively.

Similar to encryption scheme, also in decryption one; performance results are more or less the same. The reason for small differences is again benchmarking effort

TABLE 6.10: RSA Decryption of 100K cipher text

Number of Cores	*Strategy Type*	*Parallel Time (sec.)*	*Max. Heap Residency of Best (bytes)*
2	Original	22.39	5113736
	Sec.	22.17	4871304
3	Original	20.59	5432688
	Sec.	20.55	5325880
4	Original	19.55	5044568
	Sec.	19.21	5068864

which follows one another exploiting the better caching effect, since number of created, converted, fizzled and unneeded sparks are completely same that hides the advantages of second generation strategies against original ones.

TABLE 6.11: RSA Decryption of 125K cipher text

Number of Cores	*Strategy Type*	*Parallel Time (sec.)*	*Max. Heap Residency of Best (bytes)*
2	Original	38.20	5960984
	Sec.	37.00	5703536
3	Original	34.38	6227008
	Sec.	34.28	6181032
4	Original	32.75	5979888
	Sec.	32.23	5922952

6.4 Parallel Karatsuba Multiplication

Karatsuba algorithm is mainly used to multiply large scale of numbers within the range of $1K$ and $4K$ with the computational time complexity of $O(n^{log_2 3})$ for cryptographic implementation reasons. As the other algorithms given above; parallelization of Karatsuba multiplication is also done within the same scope via both original and second generation strategies.

Here is the main function which supports parallelism by the manner of original strategies:

```
OrKaratsuba :: Int → [Bool] → [Bool] → [Bool]
OrKaratsuba _ [] _ = []
OrKaratsuba _ _ [] = []
          | (l < 32 || currentDepth >= limit) = mul xs ys
          | otherwise = (x ‘add‘ (replicate l False ++ (z ‘add‘ (replicate l False ++ y)))) ‘using‘
             where
             l = (min (length xs) (length ys)) ‘div‘ 2
             (xs0, xs1) = splitAt l xs
             (ys0, ys1) = splitAt l ys
             x = (normalize (OrKaratsuba (currentDepth + 1) xs0 ys0))
             y = (normalize (OrKaratsuba (currentDepth + 1) xs1 ys1))
             z = (normalize (OrKaratsuba (currentDepth + 1) (add xs0 xs1) (add ys0 ys1))
             v = z ‘sub‘ x ‘sub‘ y
             strategy res = (rnf x) ‘par‘
                            (rnf y) ‘par‘
                            (rnf z) ‘par‘
                            (rnf v) ‘pseq‘
                            (rnf res)
```

Here is the main function which supports parallelism by the manner of second generation strategies:

```
EvalKaratsuba :: Int → [Bool] → [Bool] → [Bool]
EvalKaratsuba _ [] _ = []
EvalKaratsuba _ _ [] = []
          | (l < 32 || currentDepth >= limit) = mul xs ys
          | otherwise = (x ‘add‘ (replicate l False ++ (z ‘add‘ (replicate l False ++ y)))) ‘using‘
             where
             l = (min (length xs) (length ys)) ‘div‘ 2
             (xs0, xs1) = splitAt l xs
             (ys0, ys1) = splitAt l ys
             x = (normalize (EvalKaratsuba (currentDepth + 1) xs0 ys0))
             y = (normalize (EvalKaratsuba (currentDepth + 1) xs1 ys1))
             z = (normalize (EvalKaratsuba (currentDepth + 1) (add xs0 xs1) (add ys0 ys1))
             v = z ‘sub‘ x ‘sub‘ y
             strategy res = do
             (Main.rpar ‘Main.dot‘ Main.rdeepseq) (x)
```

$(Main.rpar\ `Main.dot`\ Main.rdeepseq)\ (y)$
$(Main.rpar\ `Main.dot`\ Main.rdeepseq)\ (z)$
$(Main.rpar\ `Main.dot`\ Main.rdeepseq)\ (v)$
$(Main.rdeepseq)\ (res)$

Both in second generation and original strategies; parallelization idea was just diving the given inputs to be multiplied recursively, that are actually in base 2, into sub numbers until the length of any input number is decreased to 32 bits or below, and then multiplications of each sub-number pairs are performed in parallel.

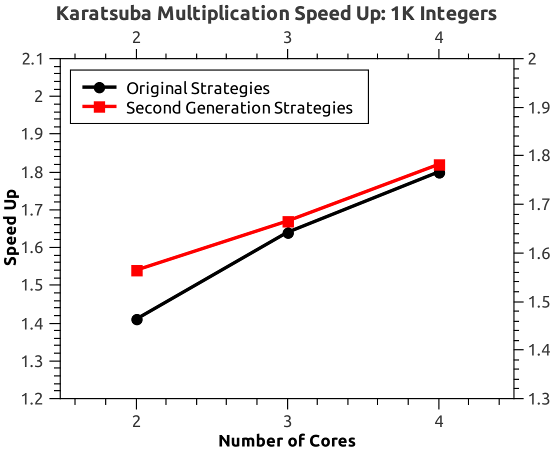

FIGURE 6.6: Parallel Karatsuba Multiplication: 1K Integers

TABLE 6.12: Karatsuba Multiplication of 1K integers

Number of Cores	*Strategy Type*	*Best Time (sec.)*	*Average Time (sec.)*	*Max. Heap Residency of Best (bytes)*	*Avg. Heap Residency (bytes)*
2	Original	0.85	1.22	1159200	1527382
	Sec.	0.78	1.16	1130272	1300553
3	Original	0.73	1.06	38752	851410
	Sec.	0.72	1.07	38752	699193
4	Original	0.67	1.06	38752	910059
	Sec.	0.66	1.04	38640	658821

Here, in th below table; the measured results from the multiplication of 1Kb long integers are demonstrated. As could be inferred; from both speed up and memory allocation amount points of view, second generation strategies are ahead of the original ones due to missing CPU cycles in order to evaluate already fizzled (evaluated) sparks. For more and detailed information, corresponding table involved in Appendix A could be checked.

Note 6.4.1. *Serial (Original) Versions of codes:*

1. *RSA Scheme is taken from the paper published by David Gray named as* Implementing Public-Key Cryptography in Haskell *[27]*

2. *Karatsuba Multiplication: from Goerch's corner.* `http://goerchs-corner.blogspot.com/2007/03/karatsuba-in-haksell.html`

3. *Quicksort Scheme: Haskell-Wiki page.* `http://www.haskell.org/haskellwiki/Introduction#Quicksort_in_Haskell`

Chapter 7

Conclusion & Future Work

Functional programming paradigm is reclaimed by category theory which alternates to set theory without any paradoxes. In this project, the general aim was to examine how objects of category theory are represented in a functional programming language called Haskell, in which cases they offer better solution methodologies via functors, natural transformations and monads. In that sense; some of the polymorphic data types of Haskell programming language such as Maybe, List, State, IO and Eval are proven as functors and monads together with some natural transformations like fmap, join and return.

For each mentioned polymorphic data type; the specific areas of computing science are explained and abstract solution manners are illustrated via some number of codes. Especially, the new memory management methodology that Eval monad provides to the original strategies or data structures in parallelization area is demonstrated, in detail. Calculating the correspondence of any input number within the Fibonacci series, Quicksort, Karatsuba multiplication algorithms are given as examples to parallelization of recursive functions. On the other side, RSA Encryption and Decryption schemes are the functions that parallelized to illustrate non-recursive approach. At the end of each parallelization trial; measured elapsed time and heap residencies together with handled speed up values for both original and second generation strategies, in which Eval monad is used, are given, comparatively.

Below table includes the demonstrates the reason of all possible parallelization outcomes in terms of elapsed time performances and allocated heap residency amounts from the point of second generation strategies' view, for sure under the same circumstances like identically applied parallelization manners and more or less the same total number of created sparks.

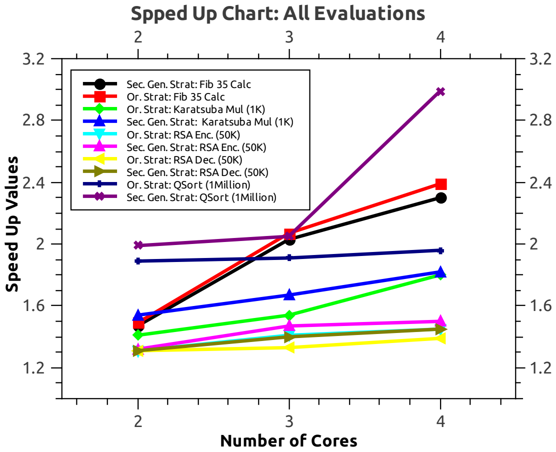

FIGURE 7.1: Speed Up Values for Each Parallel Evaluation

	Time Elapsed (seconds)	**Heap Residency (bytes)**	**Reason**
	–	–	less amount GC > CPU cycles
	–	✔	big amount GC > CPU cycles
	✔	–	less amount GC < CPU cycles
	✔	✔	big amount GC < CPU cycles

The first case, actually, explains the reason why second generation strategies stand behind of original ones in both performance aspects as less amount of garbage collection for second generation strategies takes more time than the CPU cycles lost during the evaluations of already fizzled sparks in original strategies.

In the second case; big amount of garbage collection for second generation strategies takes more time than lost CPU cycles of original ones, so that second generation strategies have better (less) amount of heap residency during the calculation but much amount of time required for that calculation to be done.

Third case is the one completely opposite of the second which explains better elapsed time performance and worse heap residency for second generation strategies with the indication that less amounts of garbage collection for second generation strategies takes less time than lost CPU cycles of original strategies.

The last case is actually the best one in the second generation strategies' sense. Being ahead of original strategies in accordance both with elapsed time performance and amount of allocated heap space.

According to our results; parallelization of Karatsuba Multiplication algorithm and some parts of Quicksort and Fibonacci Series calculation are included in the last case which declares that the aim of the project seem to be achieved.

Nearly all cases of RSA Encryption, Decryption schemes are seemingly participated in the best case, but as explained, the reason for that is not the advantages provided by second generation ones. The actual reason is the benchmarking effort done one after another causing better cache exploitations, since the number of created, converted and fizzled sparks are completely same.

For further information; detailed tables of each parallelization trial could be checked in Appendix A part.

In brief; second generation strategies provide ability to discard fizzled sparks by the help of category theoretic monad usage. For that reason, compared to original ones, they do not waste time (CPU cycles) to decide whether a spark is fizzled or not. For the other case which is not an improvement supplied by monads but the garbage collection policy (so that this could be valid for both second generation and original strategies), in which there exists some unnecessary ones created by at least two cores to perform the same job, these unnecessary sparks are referenced by *weak pointers* in the *memo tables* that are standing for the registration of the results of last evaluated functions provided by *tracing garbage collection policy* where referenced sparks do not have to be evaluated by the program if they are not needed (the same subtask was already assigned to another spark which finalized its evaluation and registered the result into the memo table, since for the same evaluation, memo table is checked and result found! No need to evaluate another spark for the same task). By that way, no CPU cycles are wasted for the evaluations of some number of sparks never become necessary during the evaluation. Additionally; *speculative parallelism* is also supported in second generation strategies which prunes any spark when it is proven to be never needed again due to the usages of weak references for the management of the spark pool.

Appendix A

Performance Analysis Tables

Serially calculated Fibonacci 35: 1.22 seconds,

Serially calculated Fibonacci 37: 2.86 seconds,

Serially calculated Fibonacci 40: 10.21 seconds,

Serial Quicksort of 1 million integers: 7.16 seconds,

Serial Quicksort of 2 million integers:16.33 seconds,

Serial RSA Encryption of 125Kb plain text via 2 sub-texts: 40.05 seconds,

Serial RSA Encryption of 125Kb plain text via 3 sub-texts: 36.31 seconds,

Serial RSA Encryption of 125Kb plain text via 4 sub-texts: 34.86 seconds,

Serial RSA Encryption of 100Kb plain text via 2 sub-texts: 26.89 seconds,

Serial RSA Encryption of 100Kb plain text via 3 sub-texts: 24.94 seconds,

Serial RSA Encryption of 100Kb plain text via 4 sub-texts: 23.79 seconds,

Serial RSA Encryption of 50Kb plain text via 2 sub-texts: 8.09 seconds,

Serial RSA Encryption of 50Kb plain text via 3 sub-texts: 7.73 seconds,

Serial RSA Encryption of 50Kb plain text via 4 sub-texts: 7.36 seconds,

Serial RSA Decryption of 125Kb cipher text via 2 sub-texts: 40.02 seconds,

Serial RSA Decryption of 125Kb cipher text via 3 sub-texts: 38.61 seconds,

Serial RSA Decryption of 125Kb cipher text via 4 sub-texts: 37.80 seconds,

Serial RSA Decryption of 100Kb cipher text via 2 sub-texts: 26.84 seconds,

Serial RSA Decryption of 100Kb cipher text via 3 sub-texts: 25.17 seconds,

Serial RSA Decryption of 100Kb cipher text via 4 sub-texts: 23.86 seconds,

Serial RSA Decryption of 50Kb cipher text via 2 sub-texts: 7.94 seconds,

Serial RSA Decryption of 50Kb cipher text via 3 sub-texts: 7.35 seconds,

Serial RSA Decryption of 50Kb cipher text via 4 sub-texts: 7.14 seconds,

Serial Karatsuba Multiplication of 1K integers, takes 1.20 seconds to conclude.

TABLE A.1: Performance Analysis of Parallel Fibonacci 35 over 2 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	0.92	1	1	0	5432	713 - 1
	Sec.	0.90	1	1	0	6592	713 - 1
2	Original	0.90	3	3	0	6592	713 - 1
	Sec.	0.90	3	1	0	6576	713 - 1
4	Original	0.90	15	12	3	11096	713 - 1
	Sec.	0.85	15	2	0	7704	713 - 1
8	Original	0.88	311	240	71	14504	713 - 1
	Sec.	0.84	255	3	0	38752	713 - 1
12	Original	0.89	4965	3814	1150	17976	713 - 1
	Sec.	0.83	4144	7	92	16584	713 -1
13	Original	0.89	10072	8011	2061	17936	713 - 1
	Sec.	0.85	8324	9	317	16672	713 - 1
14	Original	0.89	19848	15829	4019	18360	713 - 1
	Sec.	0.85	16673	12	631	16680	713 - 1
16	Original	0.82	65661	29741	3147	22384	713 - 1
	Sec.	0.83	66375	28	670	15224	713 - 1
32	Original	1.83	15199333	557031	49345	14880	713 - 1
	Sec.	1.50	15332997	57	49144	14344	713 - 1

TABLE A.2: Performance Analysis of Parallel Fibonacci 35 over 3 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	0.93	1	1	0	5432	1069 - 2
	Sec.	0.93	1	1	0	6560	1069 - 2
2	Original	0.66	3	3	0	38752	1069 - 2
	Sec.	0.86	3	3	0	7672	1069 - 2
4	Original	0.59	15	15	0	38752	1069 - 2
	Sec.	0.65	15	5	0	38752	1069 - 2
8	Original	0.63	282	282	0	38752	1069 - 2
	Sec.	0.60	255	9	0	38752	1069 - 2
12	Original	0.63	4523	4523	0	43920	1069 - 2
	Sec.	0.67	4404	31	0	38752	1069 - 2
13	Original	0.60	8596	8596	0	72400	1069 - 2
	Sec.	0.68	8721	14	0	55024	1069 - 2
14	Original	0.67	19808	19808	0	38752	1069 - 2
	Sec.	0.63	16757	36	0	42848	1069 - 2
16	Original	0.62	69765	56608	0	87760	1069 - 2
	Sec.	0.64	67456	36	0	47896	1069 - 2
32	Original	1.73	15570906	880437	49146	20992	1069 - 2
	Sec.	1.42	15420117	100	49139	19552	1069 - 2

TABLE A.3: Performance Analysis of Parallel Fibonacci 35 over 4 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	0.95	1	1	0	5432	1425 - 2
	Sec.	0.94	1	1	0	6560	1425 - 2
2	Original	0.66	3	3	0	38752	1425 - 2
	Sec.	0.68	3	3	0	38752	1425 - 2
4	Original	0.56	15	15	0	38752	1425 - 2
	Sec.	0.55	15	7	0	38752	1425 - 2
8	Original	0.51	274	274	0	39776	1425 - 2
	Sec.	0.55	262	21	0	38752	1425 - 2
12	Original	0.51	4251	4251	0	43920	1425 - 2
	Sec.	0.54	4275	42	0	45824	1425 - 2
13	Original	0.53	9209	9209	0	39776	1425 - 2
	Sec.	0.55	8770	32	0	39664	1425 - 2
14	Original	0.57	19114	18262	852	61136	1425 - 2
	Sec.	0.59	17786	116	62	57184	1425 - 2
16	Original	0.58	78481	70289	0	38752	1425 - 2
	Sec.	0.53	66580	50	0	38752	1425 - 2
32	Original	1.57	15195235	683734	34097	30256	1425 - 2
	Sec.	1.31	16305901	110	32759	28536	1425 - 2

TABLE A.4: Performance Analysis of Parallel Fibonacci 37 over 2 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
	Sec.	1.83	1	1	0	6840	713 - 1
2	Original	1.85	3	1	2	7016	713 - 1
	Sec.	1.85	3	1	1	6912	713 - 1
4	Original	1.64	15	6	9	10264	713 - 1
	Sec.	1.63	15	2	8	7016	713 - 1
8	Original	1.57	258	175	83	13640	713 - 1
	Sec.	1.64	258	5	149	10384	713 - 1
12	Original	1.59	4180	3067	1113	14048	713 - 1
	Sec.	1.63	4158	8	2419	13664	713 -1
13	Original	1.59	8415	6236	2179	14368	713 - 1
	Sec.	1.64	8319	8	4803	13704	713 - 1
14	Original	1.58	16756	12456	4300	14224	713 - 1
	Sec.	1.64	16641	11	9660	13816	713 - 1
16	Original	1.59	66621	34100	13912	14528	713 - 1
	Sec.	1.67	66383	45	16710	13944	713 - 1
32	Original	3.87	39325069	436748	204775	16056	713 - 1
	Sec.	3.31	39402722	71	134005	14504	713 - 1

TABLE A.5: Performance Analysis of Parallel Fibonacci 37 over 3 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	1.86	1	1	0	6885	1069 - 2
	Sec.	1.86	1	1	0	6904	1069 - 2
2	Original	1.38	3	3	0	8328	1069 - 2
	Sec.	1.35	3	2	1	8248	1069 - 2
4	Original	1.34	15	11	4	12728	1069 - 2
	Sec.	1.32	15	4	10	8392	1069 - 2
8	Original	1.29	278	252	26	21976	1069 - 2
	Sec.	1.25	257	10	185	11864	1069 - 2
12	Original	1.30	4774	3429	1345	22944	1069 - 2
	Sec.	1.25	4128	21	2960	18360	1069 - 2
13	Original	1.32	9871	7211	2660	23120	1069 - 2
	Sec.	1.37	8721	15	6015	18432	1069 - 2
14	Original	1.34	19550	14282	5268	23344	1069 - 2
	Sec.	1.29	16809	41	11994	18584	1069 - 2
16	Original	1.28	71184	55914	6916	27584	1069 - 2
	Sec.	1.27	66157	36	24573	18744	1069 - 2
32	Original	3.85	39600827	1403334	229348	21336	1069 - 2
	Sec.	3.15	39634470	103	196573	19696	1069 - 2

TABLE A.6: Performance Analysis of Parallel Fibonacci 37 over 4 Cores

Depth	*Strategy Type*	*Time* (*sec.*)	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage* (*MB*)
1	Original	1.89	1	1	0	6776	1425 - 2
	Sec.	1.89	1	1	0	6776	1425 - 2
2	Original	1.38	3	3	0	8344	1425 - 2
	Sec.	1.37	3	3	0	9408	1425 - 2
4	Original	1.26	15	13	2	12912	1425 - 2
	Sec.	1.19	15	7	1	12928	1425 - 2
8	Original	1.16	276	234	42	33176	1425 - 2
	Sec.	1.22	268	30	94	18512	1425 - 2
12	Original	1.20	4697	3960	737	24352	1425 - 2
	Sec.	1.21	4254	37	1562	20904	1425 - 2
13	Original	1.17	9245	6178	3067	36800	1425 - 2
	Sec.	1.16	8397	35	2808	24200	1425 - 2
14	Original	1.17	19139	13124	6015	28496	1425 - 2
	Sec.	1.21	16868	41	5963	24408	1425 - 2
16	Original	1.14	69527	45198	16146	24312	1425 - 2
	Sec.	1.23	69203	127	11674	32736	1425 - 2
32	Original	3.89	39930049	2426378	172142	26776	1425 - 2
	Sec.	3.03	39561515	77	163807	24792	1425 - 2

TABLE A.7: Performance Analysis of Parallel Fibonacci 40 over 2 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	6.72	1	1	0	6936	713 - 1
	Sec.	6.69	1	1	0	6888	713 - 1
2	Original	6.72	3	1	2	7144	713 - 1
	Sec.	6.61	3	1	2	7000	713 - 1
4	Original	5.79	15	2	13	10472	713 - 1
	Sec.	5.78	15	2	12	10200	713 - 1
8	Original	5.47	255	38	217	13952	713 - 1
	Sec.	5.56	255	4	246	13568	713 - 1
12	Original	5.58	4177	679	3498	14272	713 - 1
	Sec.	5.52	4096	8	3973	13784	713 - 1
13	Original	5.58	8371	1396	6974	14360	713 - 1
	Sec.	5.51	8194	10	7942	13840	713 - 1
14	Original	5.59	16786	3088	13698	14576	713 - 1
	Sec.	5.70	16525	9	16217	13928	713 - 1
16	Original	5.55	67039	12625	54414	14680	713 - 1
	Sec.	5.50	65566	9	63488	14072	713 - 1
32	Original	14.76	164535147	676337	802718	14848	713 - 1
	Sec.	12.79	165776604	238	666609	14848	713 - 1

TABLE A.8: Performance Analysis of Parallel Fibonacci 40 over 3 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	6.77	1	1	0	6888	1069 - 2
	Sec.	6.67	1	1	0	5952	1069 - 2
2	Original	6.78	3	2	1	8464	1069 - 2
	Sec.	4.44	3	2	1	8384	1069 - 2
4	Original	4.69	15	6	9	12016	1069 - 2
	Sec.	4.37	15	4	10	11680	1069 - 2
8	Original	4.04	261	114	147	19696	1069 - 2
	Sec.	3.97	255	9	236	15208	1069 - 2
12	Original	4.03	4234	1756	2478	19112	1069 - 2
	Sec.	3.96	4099	16	3851	18632	1069 - 2
13	Original	4.10	8707	4767	3940	23120	1069 - 2
	Sec.	3.98	8205	25	7662	18752	1069 - 2
14	Original	3.98	16963	5663	11296	19488	1069 - 2
	Sec.	3.93	16391	21	15412	18784	1069 - 2
16	Original	3.99	69065	25296	43679	20000	1069 - 2
	Sec.	4.03	66007	42	59385	18936	1069 - 2
32	Original	15.36	166166425	5832108	1045253	22008	1069 - 2
	Sec.	12.08	164402198	59	909123	20160	1069 - 2

TABLE A.9: Performance Analysis of Parallel Fibonacci 40 over 4 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	6.85	1	1	0	6952	1425 - 2
	Sec.	6.80	1	1	0	6888	1425 - 2
2	Original	4.63	3	3	0	9752	1425 - 2
	Sec.	4.64	3	3	0	9688	1425 - 2
4	Original	4.96	15	7	8	13456	1425 - 2
	Sec.	3.75	15	7	7	13120	1425 - 2
8	Original	3.69	256	120	136	23312	1425 - 2
	Sec.	3.53	258	20	214	16648	1425 - 2
12	Original	3.73	4497	3166	1331	23912	1425 - 2
	Sec.	3.49	4106	51	3172	23352	1425 - 2
13	Original	3.68	9060	7021	2039	24384	1425 - 2
	Sec.	3.53	8240	52	7079	23464	1425 - 2
14	Original	3.62	18263	13190	5073	24608	1425 - 2
	Sec.	3.44	16506	44	13956	23584	1425 - 2
16	Original	3.88	77096	57214	19882	25088	1425 - 2
	Sec.	3.53	65794	58	63128	23912	1425 - 2
32	Original	15.05	166303861	7292869	755777	27632	1425 - 2
	Sec.	12.03	165993284	285	687980	25272	1425 - 2

TABLE A.10: Performance Analysis of Quicksort of 1 Million Random Integers over 2 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	4.04	1	1	0	51165008	836 - 0
	Sec.	4.06	1	1	0	49918056	809 - 0
2	Original	4.00	3	1	2	55475600	811 - 0
	Sec.	3.98	3	1	2	49863632	806 - 0
4	Original	3.99	15	1	14	57220040	818 - 0
	Sec.	3.58	15	1	12	58246768	811 - 0
8	Original	3.83	251	31	210	55265480	858 - 0
	Sec.	3.58	256	1	231	59433568	824 - 0
12	Original	3.90	3547	1768	1779	47918984	827 - 0
	Sec.	3.71	3305	7	2041	61705090	831 - 0
13	Original	3.91	5773	2332	3541	62005832	849 - 0
	Sec.	3.70	5797	7	3514	58205792	830 - 0
14	Original	4.08	9922	2635	7287	57874096	840 - 0
	Sec.	3.71	9855	13	5778	61195584	825 - 0
16	Original	4.15	25006	10911	14095	59051920	837 - 0
	Sec.	3.64	24863	7	14701	60245080	826 - 0

TABLE A.11: Performance Analysis of Quicksort of 1 Million Random Integers over 3 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	3.93	1	1	0	49898894	1155 - 0
	Sec.	4.04	1	1	0	49918176	1165 - 0
2	Original	3.79	3	2	1	47269000	1162 - 0
	Sec.	3.73	3	2	1	50011920	1218 - 0
4	Original	3.97	15	4	11	47688984	1173 - 0
	Sec.	3.69	15	3	11	58842912	1183 - 0
8	Original	3.88	262	114	148	49086736	1193 - 0
	Sec.	3.54	255	11	33	50045216	1213 - 0
12	Original	3.79	3330	1318	2012	50415960	1213 - 0
	Sec.	3.78	3423	28	2174	52032600	1232 - 0
13	Original	3.87	5836	5265	571	54434312	1199 - 0
	Sec.	3.64	5790	25	3366	52446096	1220 - 0
14	Original	3.95	10607	9961	646	53068688	1197 - 0
	Sec.	3.67	9917	22	4775	51273584	1254 - 0
16	Original	4.03	25214	17184	8030	58499840	1209 - 0
	Sec.	3.56	25352	48	3206	49566664	826 - 0

TABLE A.12: Performance Analysis of Quicksort of 1 Million Random Integers over 4 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	3.70	1	1	0	49910592	1510 - 0
	Sec.	4.09	1	1	0	49917788	1521 - 0
2	Original	3.92	3	3	0	49455480	1511 - 0
	Sec.	3.78	3	3	0	49723576	1517 - 0
4	Original	3.71	15	6	9	49365984	1520 - 0
	Sec.	3.66	15	7	1	51164264	1511 - 0
8	Original	3.69	255	213	42	49814776	1526 - 0
	Sec.	3.56	255	18	81	52650336	1516 - 0
12	Original	3.75	3333	1884	1449	58627712	1533 - 0
	Sec.	3.47	3360	44	992	57118432	1519 - 0
13	Original	3.83	5771	4052	1819	55847544	1558 - 0
	Sec.	3.63	5994	69	1709	60207304	1524 - 0
14	Original	3.93	9894	7897	1997	146303360	1516 - 0
	Sec.	3.62	10282	56	4506	52614544	1526 - 0
16	Original	3.75	26163	19813	6350	47341672	1209 - 0
	Sec.	3.49	25200	44	7124	53684008	1517 - 0

TABLE A.13: Performance Analysis of Quicksort of 2 Million Random Integers over 2 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	9.53	1	1	0	84299840	955 - 0
	Sec.	9.25	1	1	0	99822088	912 - 0
2	Original	9.70	3	1	2	104963616	968 - 0
	Sec.	9.35	3	1	2	108163992	1011 - 0
4	Original	9.58	15	4	11	133324392	1057 - 0
	Sec.	9.65	15	2	9	94917200	1001 - 0
8	Original	9.93	255	31	210	55265480	858 - 0
	Sec.	8.87	255	4	193	128483984	973 - 0
12	Original	9.93	3313	1018	2295	1245588880	1048 - 0
	Sec.	9.11	3338	12	2516	162482400	1063 - 0
13	Original	12.25	5782	741	5041	128608936	1065 - 0
	Sec.	9.70	5770	8	4141	158590040	1036 - 0
14	Original	12.96	9769	2512	7257	129772672	1044 - 0
	Sec.	9.51	9721	7	7375	156675280	993 - 0
16	Original	13.97	24946	5238	19708	171955208	1098 - 0
	Sec.	10.08	24958	13	18363	111454088	1033 - 0

TABLE A.14: Performance Analysis of Quicksort of 2 Million Random Integers over 3 Cores

Depth	*Strategy*	*Time*	*Total*	*Converted*	*Pruned*	*Max.*	*Memory*
	Type	*(sec.)*	*Number*	*Sparks*	*Sparks*	*Heap*	*Usage*
			of Sparks			*Residency*	*(MB)*
1	Original	9.12	1	1	0	97491656	1330 - 0
	Sec.	9.22	1	1	0	99822072	1267 - 0
2	Original	9.65	3	2	1	87132568	1333 - 0
	Sec.	8.40	3	2	1	102132504	1409 - 0
4	Original	8.78	15	7	8	84786832	1313 - 0
	Sec.	8.78	15	3	8	93623968	1348 - 0
8	Original	8.95	258	118	140	121246328	1363 - 0
	Sec.	7.91	255	10	150	96010296	1354 - 0
12	Original	10.49	3369	2750	619	136572712	1338 - 0
	Sec.	8.88	3308	15	2039	100983592	1400 - 0
13	Original	11.09	5877	2373	3504	166950512	1390 - 0
	Sec.	8.58	5782	19	3564	108678712	1399 - 0
14	Original	12.41	9980	4993	4887	124999792	1420 - 0
	Sec.	8.99	9714	34	4320	109211856	1314 - 0
16	Original	13.07	25316	9946	15370	101938544	1404 - 0
	Sec.	8.35	25241	46	12645	148400992	1380 - 0

TABLE A.15: Performance Analysis of Quicksort of 2 Million Random Integers over 4 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	8.73	1	1	0	99823736	1623 - 0
	Sec.	9.19	1	1	0	49917788	1521 - 0
2	Original	10.93	3	3	0	98078200	1681 - 0
	Sec.	8.25	3	3	0	99503128	1634 - 0
4	Original	10.29	15	8	7	87821440	1718 - 0
	Sec.	8.27	15	7	7	91752496	1701 - 0
8	Original	10.10	255	122	133	121707384	1694 - 0
	Sec.	7.72	255	14	62	83049904	1738 - 0
12	Original	9.49	3361	2324	1037	97230248	1656 - 0
	Sec.	8.91	3339	99	1542	93719248	1689 - 0
13	Original	9.64	6116	3874	2242	105653904	1699 - 0
	Sec.	7.29	5821	41	2430	82052784	1705 - 0
14	Original	12.03	9961	5961	4000	110005304	1708 - 0
	Sec.	7.77	10127	127	5012	86798080	1679 - 0
16	Original	11.97	27824	20899	6925	117700488	1745 - 0
	Sec.	7.75	25070	47	12912	92224800	1321 - 0

TABLE A.16: RSA Encryption of 100Kb plain text

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
2	Original	22.16	1	1	0	4892176	1079 - 10
	Sec.	22.16	1	1	0	4992200	1079 - 10
3	Original	20.31	2	2	0	5433968	1435 - 10
	Sec.	20.28	2	2	0	5296960	1435 - 10
4	Original	19.46	3	3	0	5104312	1790 - 10
	Sec.	18.94	3	3	0	5032104	1793 - 12

TABLE A.17: Parallel RSA Encryption of 50Kb plain text

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
2	Original	6.18	1	1	0	24445976	932 - 0
	Sec.	6.13	1	1	0	2335496	932 - 0
3	Original	5.50	2	2	0	2449304	1287 - 0
	Sec.	5.27	2	2	0	2474656	1288 - 0
4	Original	5.09	3	3	0	1722824	1643 - 0
	Sec.	4.92	3	3	0	1719392	1643 - 0

TABLE A.18: Parallel RSA Decryption of 125Kb cipher text

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
2	Original	38.20	1	1	0	5960984	1082 - 12
	Sec.	37.00	1	1	0	5700536	1082 - 12
3	Original	34.38	2	2	0	6227008	1438 - 13
	Sec.	34.28	2	2	0	6181032	1438 - 13
4	Original	32.75	3	3	0	5979888	1793 - 12
	Sec.	32.23	3	3	0	5922952	1793 - 12

TABLE A.19: RSA Decryption of 100Kb cipher text

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
2	Original	22.39	1	1	0	5113736	1079 - 10
	Sec.	22.17	1	1	0	4871304	1079 - 10
3	Original	20.59	2	2	0	5432688	1435 - 10
	Sec.	20.55	2	2	0	5325880	1435 - 10
4	Original	19.55	3	3	0	5044568	1790 - 10
	Sec.	19.21	3	3	0	5068864	1793 - 12

TABLE A.20: RSA Decryption of 50Kb cipher text

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
2	Original	6.07	1	1	0	2401464	932 - 0
	Sec.	6.05	1	1	0	2317680	932 - 0
3	Original	5.52	2	2	0	2477016	1287 - 0
	Sec.	5.26	2	2	0	2499048	1288 - 0
4	Original	5.12	3	3	0	1772792	1643 - 0
	Sec.	4.93	3	3	0	1719392	1643 - 0

TABLE A.21: Performance Analysis of Karatsuba Multiplication of 1K Integers over 2 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	2.86	3	3	0	3058728	716 - 0
	Sec.	2.84	3	2	1	2474416	754 - 0
2	Original	1.84	12	3	9	2274696	749 - 0
	Sec.	1.88	12	5	7	1294400	714 - 0
4	Original	1.10	120	36	84	1485104	715 - 0
	Sec.	1.05	120	42	78	1104344	714 - 0
8	Original	0.86	9849	5929	3920	1141104	713 - 0
	Sec.	0.78	9810	4114	5277	1130272	713 - 0
12	Original	0.86	9969	6046	3923	1143744	713 - 0
	Sec.	0.78	9810	3939	5346	1138736	713 - 0
13	Original	0.86	9849	5911	3938	1247768	713 - 0
	Sec.	0.78	9810	3801	5319	1146416	713 - 0
14	Original	0.86	9888	5974	3914	1089184	713 - 0
	Sec.	0.78	9810	4500	5310	1143400	713 - 0
16	Original	0.85	9849	5932	3917	1159200	713 - 0
	Sec.	0.78	9810	4063	5323	1145528	713 - 0
32	Original	0.86	9969	6046	3923	1146912	713 - 0
	Sec.	0.80	9810	3866	5338	1127464	713 - 0

TABLE A.22: Performance Analysis of Karatsuba Multiplication of 1K Integers over 3 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	2.38	3	3	0	3296360	1072 - 0
	Sec.	2.35	3	3	0	3059672	1073 - 1
2	Original	1.78	12	5	7	2654120	1071 - 0
	Sec.	1.86	12	8	4	1628952	1070 - 0
4	Original	1.00	120	79	41	1479696	1070 - 0
	Sec.	1.04	123	102	21	1371968	1071 - 0
8	Original	0.73	9981	9981	0	38752	1069 - 1
	Sec.	0.72	10863	10863	0	38752	1069 - 1
12	Original	0.73	10050	10050	0	38752	1069 - 1
	Sec.	0.72	10878	10878	0	38640	1069 - 1
13	Original	0.73	9939	9939	0	38752	1069 - 1
	Sec.	0.73	10869	10869	0	38752	1069 - 1
14	Original	0.73	9810	9810	0	38752	1069 - 1
	Sec.	0.73	10965	10965	0	38608	1069 - 1
16	Original	0.74	10212	10212	0	38752	1069 -1
	Sec.	0.72	10956	10956	0	38752	1069 - 1
32	Original	0.75	10173	10173	0	38752	1069 - 1
	Sec.	0.72	10809	10809	0	38640	1069 - 1

TABLE A.23: Performance Analysis of Karatsuba Multiplication of 1K Integers over 4 Cores

Depth	*Strategy Type*	*Time (sec.)*	*Total Number of Sparks*	*Converted Sparks*	*Pruned Sparks*	*Max. Heap Residency*	*Memory Usage (MB)*
1	Original	2.73	3	3	0	3094272	1428 - 0
	Sec.	2.55	3	3	0	3072392	1428 - 0
2	Original	1.67	12	7	5	2966816	1439 - 1
	Sec.	1.71	12	11	1	1064072	1428 - 0
4	Original	0.98	123	92	31	1896928	1426 - 0
	Sec.	1.01	123	111	12	1560752	1427 - 0
8	Original	0.67	10950	10950	0	38752	1425 - 1
	Sec.	0.66	11307	11307	0	38640	1425 - 1
12	Original	0.69	10134	10134	0	38752	1425 - 1
	Sec.	0.68	12123	12123	0	38752	1425 - 0
13	Original	0.69	11229	11229	0	38752	1425 - 1
	Sec.	0.69	12084	12084	0	38752	1425 - 0
14	Original	0.70	10257	10257	0	38752	1425 - 1
	Sec.	0.69	11550	11550	0	38640	1425 - 0
16	Original	0.69	11121	11121	0	38752	1425 -1
	Sec.	0.67	11514	11514	0	38752	1425 - 0
32	Original	0.68	10791	10791	0	38752	1425 - 1
	Sec.	0.67	11604	11604	0	38752	1425 - 0

Appendix B

Threadscope Results

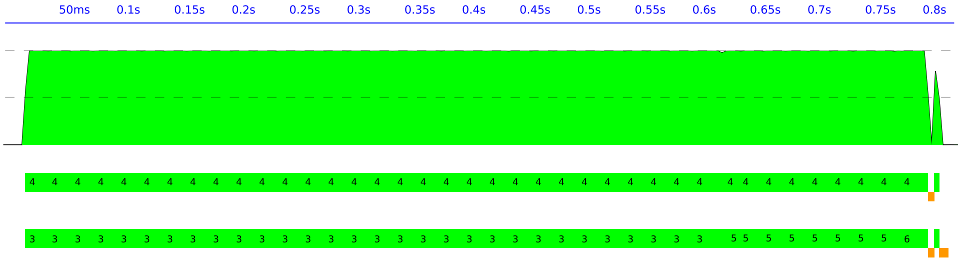

FIGURE B.1: Parallelization of Fibonacci 35 over 2 Cores - Original Strategies - Depth: 16 view

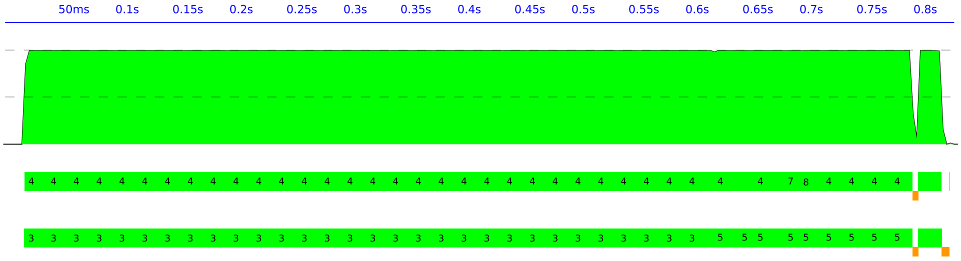

FIGURE B.2: Parallelization of Fibonacci 35 over 2 Cores - Evaluation Strategies - Depth: 16 view

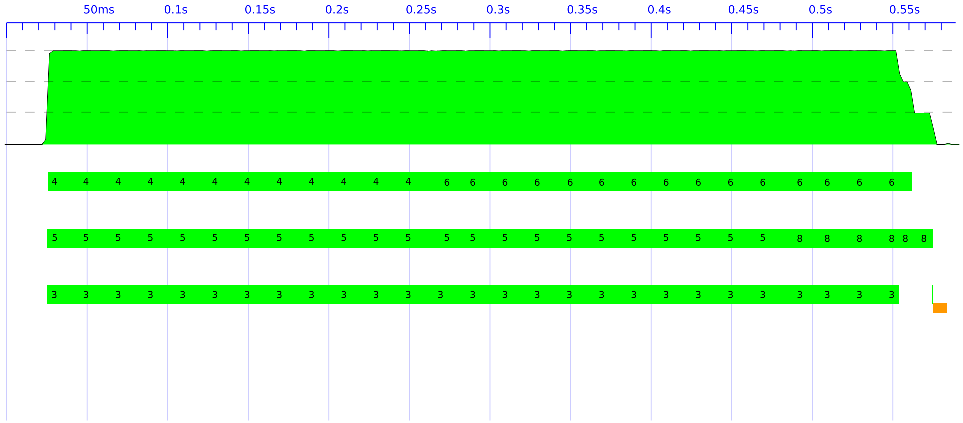

FIGURE B.3: Parallelization of Fibonacci 35 over 3 Cores - Original Strategies - Depth: 4 view

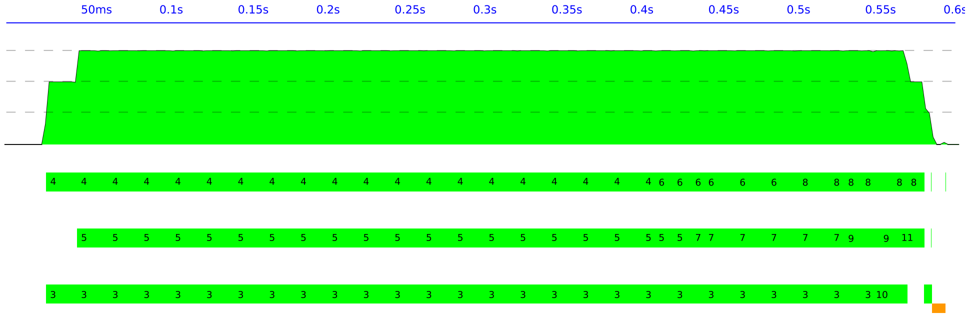

FIGURE B.4: Parallelization of Fibonacci 35 over 3 Cores - Evaluation Strategies - Depth: 8 view

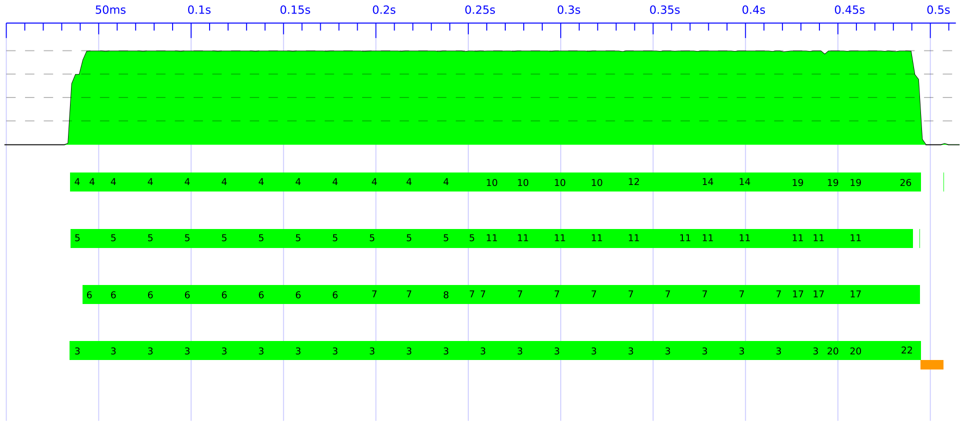

FIGURE B.5: Parallelization of Fibonacci 35 over 4 Cores - Original Strategies - Depth: 8 view

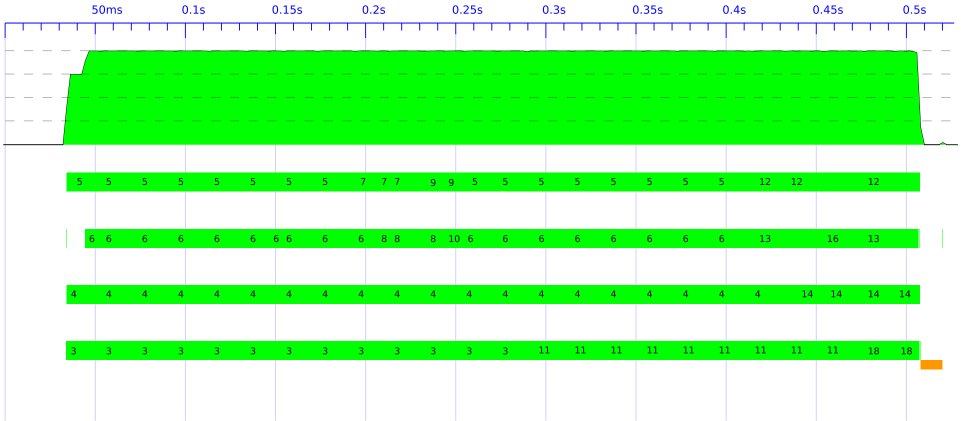

FIGURE B.6: Parallelization of Fibonacci 35 over 4 Cores - Evaluation Strategies - Depth: 16 view

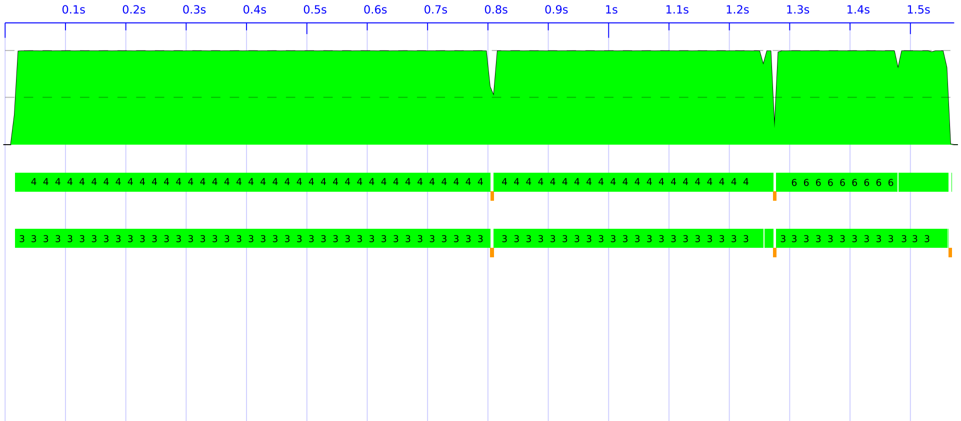

FIGURE B.7: Parallelization of Fibonacci 37 over 2 Cores - Original Strategies - Depth: 8 view

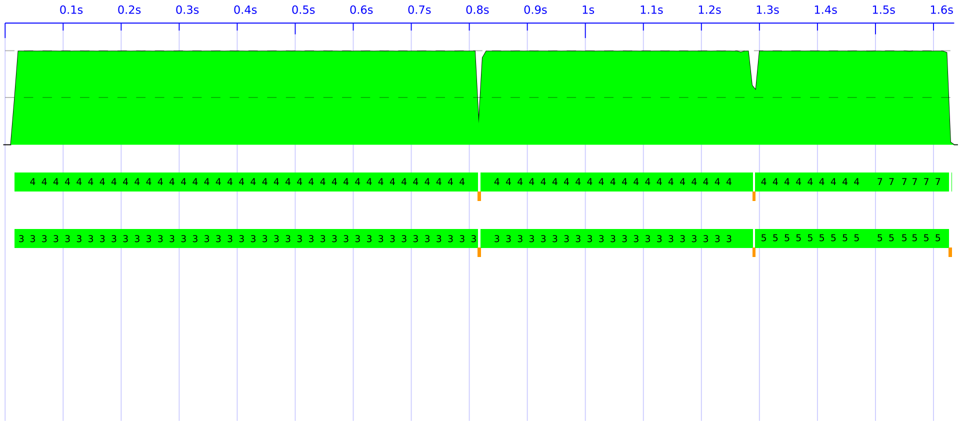

FIGURE B.8: Parallelization of Fibonacci 37 over 2 Cores - Evaluation Strategies - Depth: 12 view

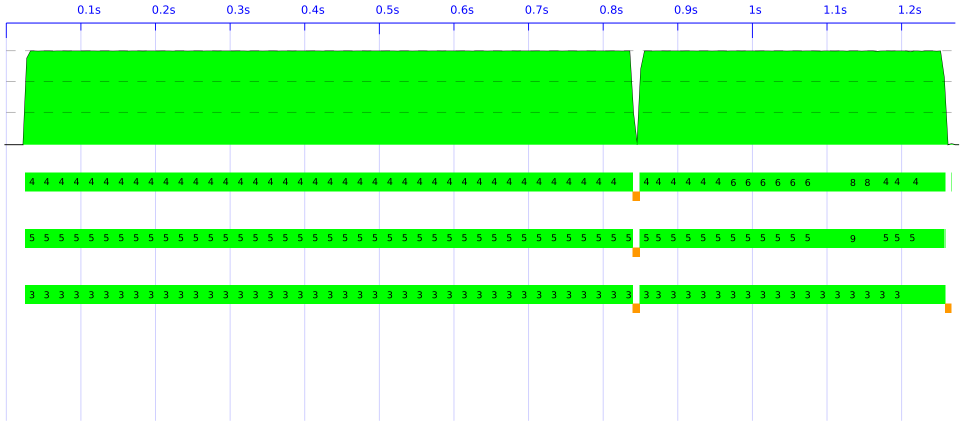

FIGURE B.9: Parallelization of Fibonacci 37 over 3 Cores - Original Strategies - Depth: 16 view

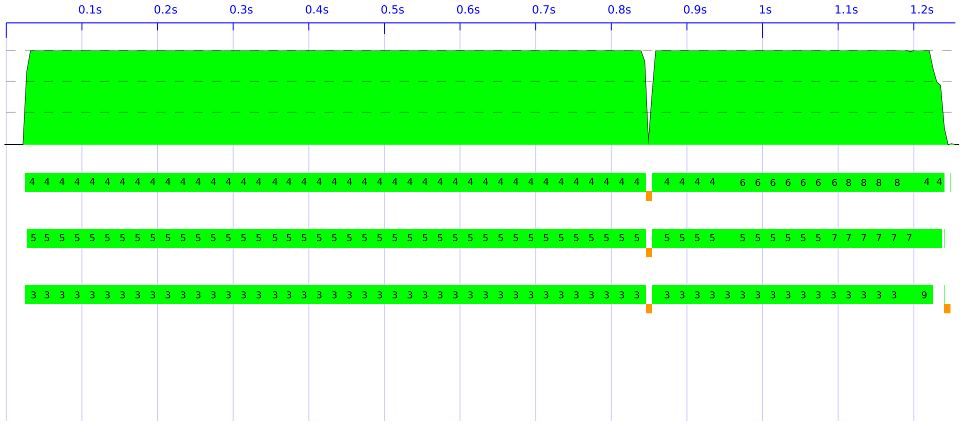

FIGURE B.10: Parallelization of Fibonacci 37 over 3 Cores - Evaluation Strategies - Depth: 8 view

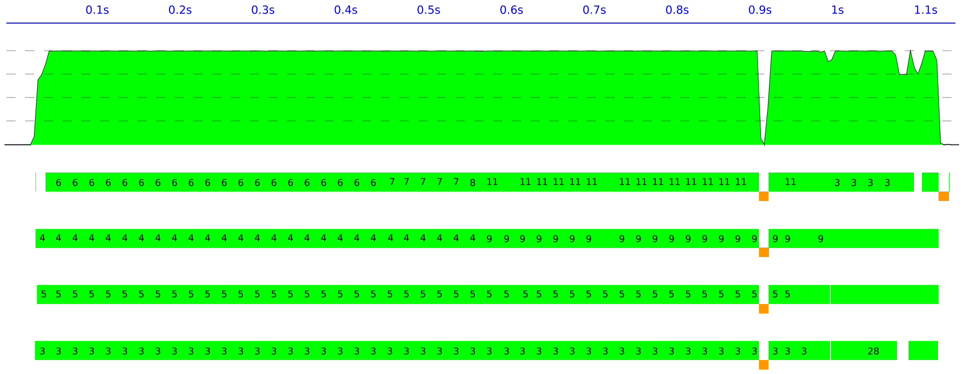

FIGURE B.11: Parallelization of Fibonacci 37 over 4 Cores - Original Strategies - Depth: 16 view

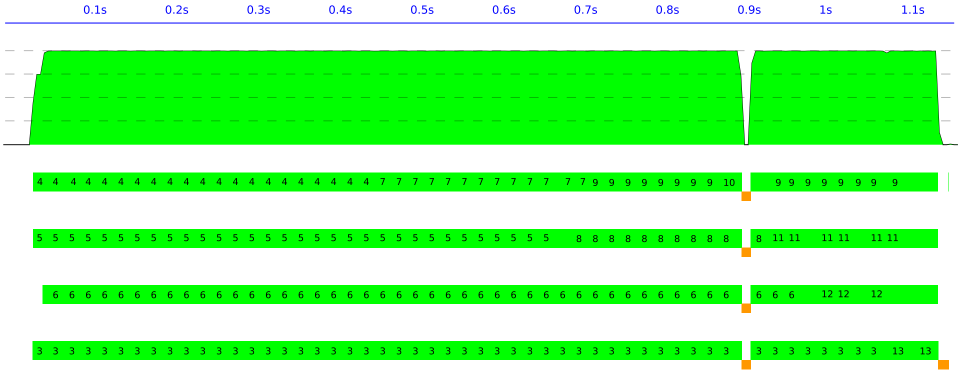

FIGURE B.12: Parallelization of Fibonacci 37 over 4 Cores - Evaluation Strategies - Depth: 13 view

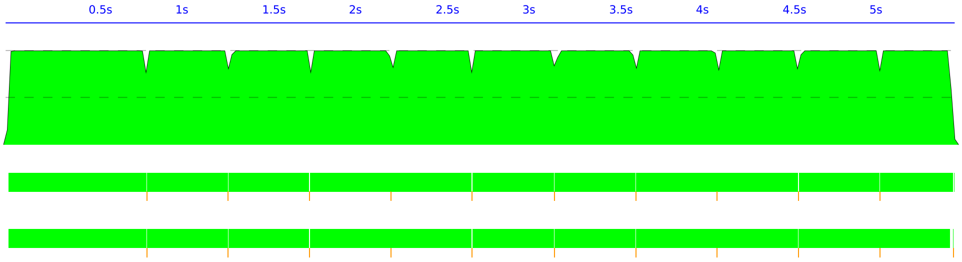

FIGURE B.13: Parallelization of Fibonacci 40 over 2 Cores - Original Strategies - Depth: 8 view

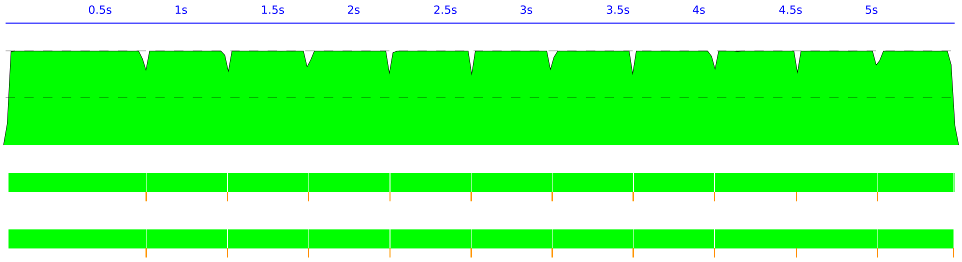

FIGURE B.14: Parallelizattheoryion of Fibonacci 40 over 2 Cores - Evaluation Strategies - Depth: 16 view

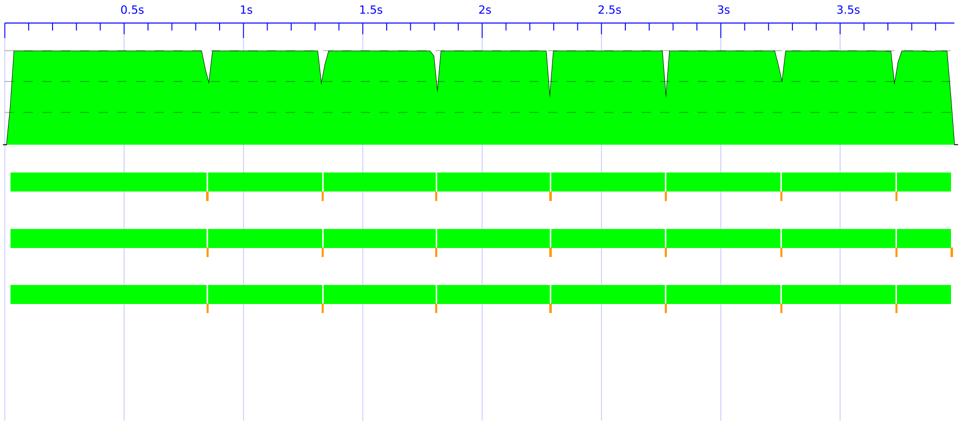

FIGURE B.15: Parallelization of Fibonacci 40 over 3 Cores - Original Strategies - Depth: 14 view

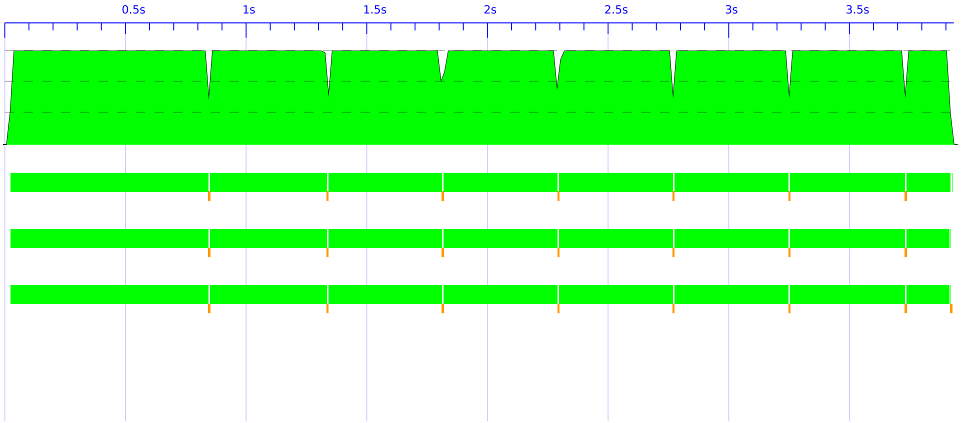

FIGURE B.16: Parallelization of Fibonacci 40 over 3 Cores - Evaluation Strategies - Depth: 14 view

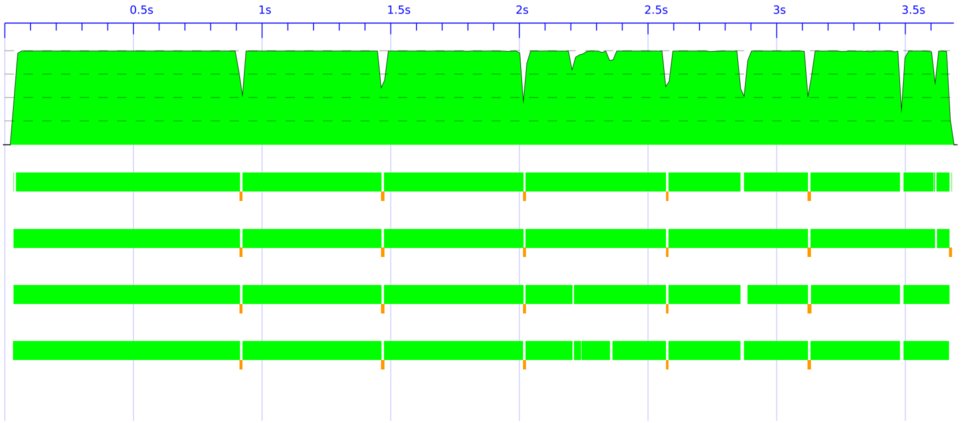

FIGURE B.17: Parallelization of Fibonacci 40 over 4 Cores - Original Strategies - Depth: 14 view

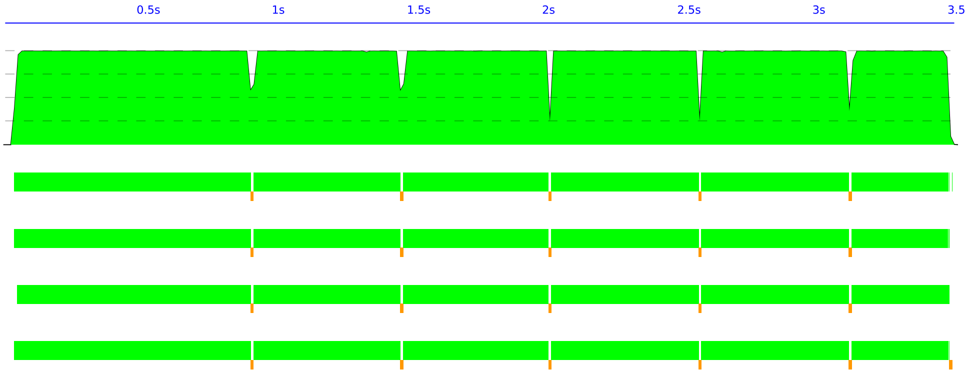

FIGURE B.18: Parallelization of Fibonacci 40 over 4 Cores - Evaluation Strategies - Depth: 14 view

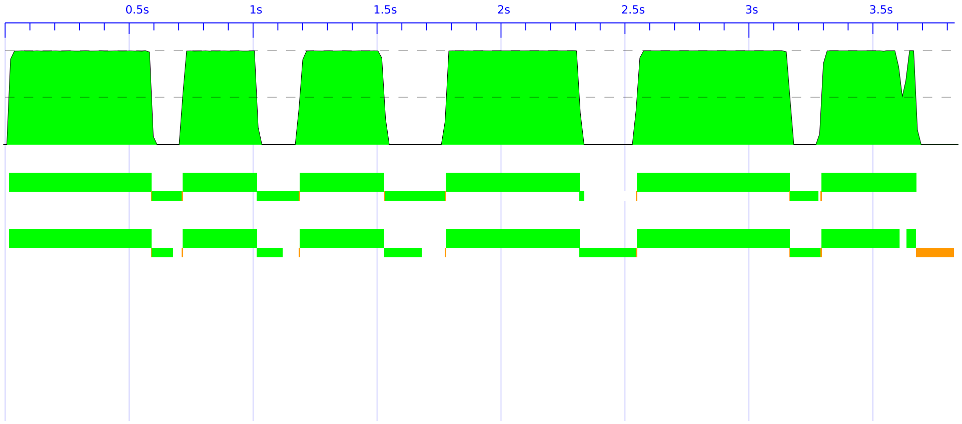

FIGURE B.19: Parallelization of QuickSort for 1 Million Entries over 2 Cores - Original Strategies -Depth: 8 view

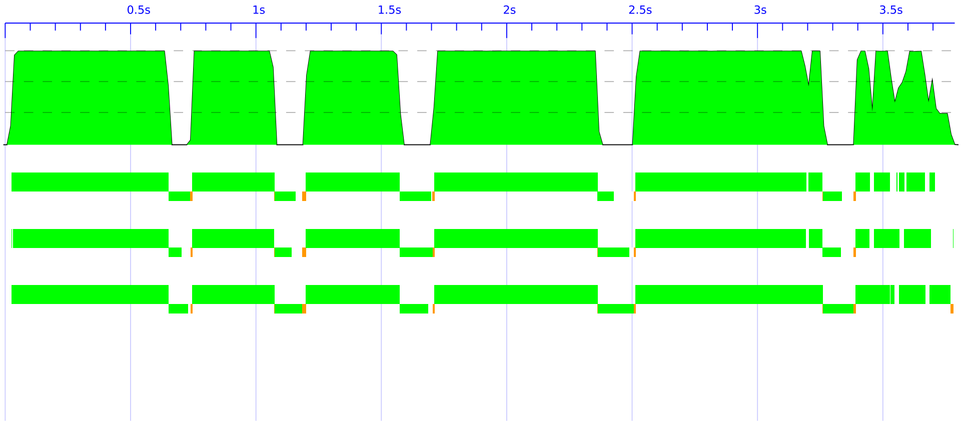

FIGURE B.20: Parallelization of QuickSort for 1 Million Entries over 3 Cores - Original Strategies -Depth: 12 view

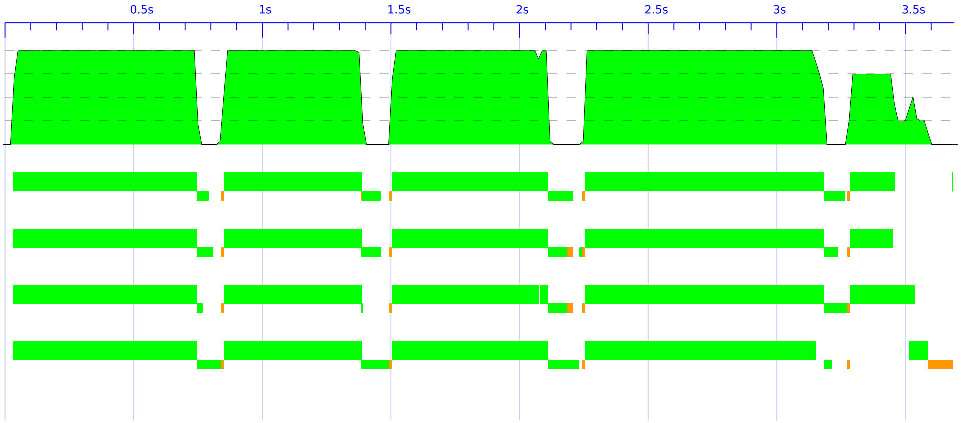

FIGURE B.21: Parallelization of QuickSort for 1 Million Entries over 4 Cores - Original Strategies -Depth: 8 view

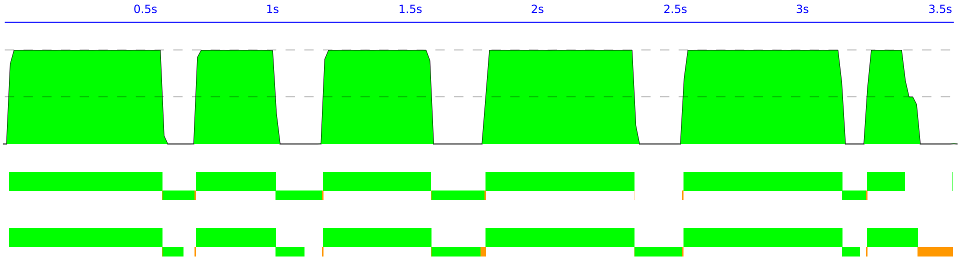

FIGURE B.22: Parallelization of QuickSort for 1 Million Entries over 2 Cores - Evaluation Strategies -Depth: 8 view

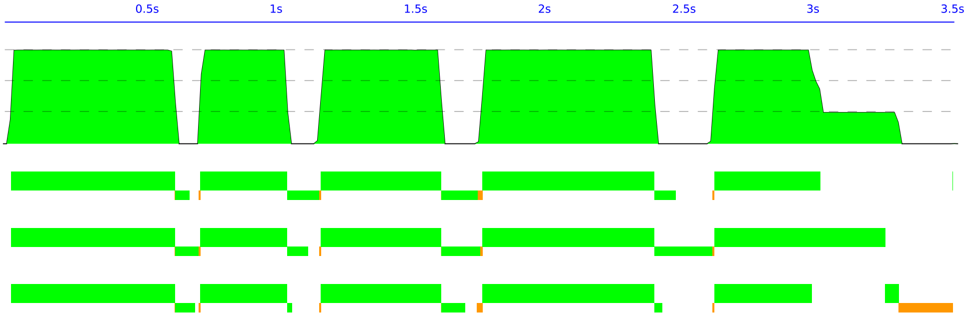

FIGURE B.23: Parallelization of QuickSort for 1 Million Entries over 3 Cores - Evaluation Strategies -Depth: 8 view

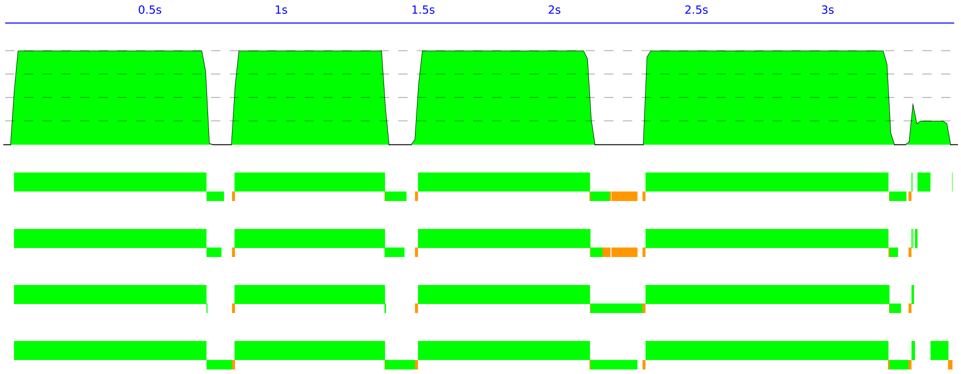

FIGURE B.24: Parallelization of QuickSort for 1 Million Entries over 4 Cores - Evaluation Strategies -Depth: 12 view

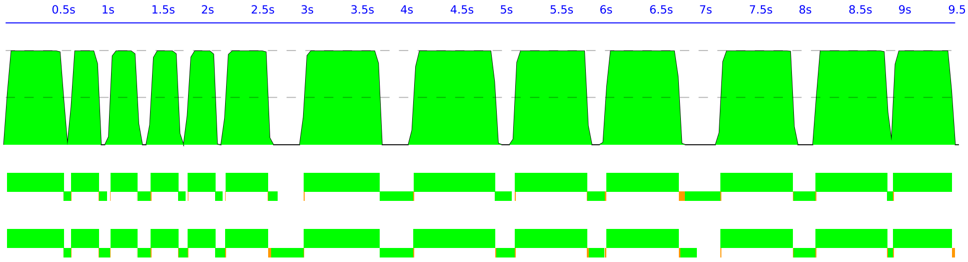

FIGURE B.25: Parallelization of QuickSort for 2 Million Entries over 2 Cores - Original Strategies -Depth: 1 view

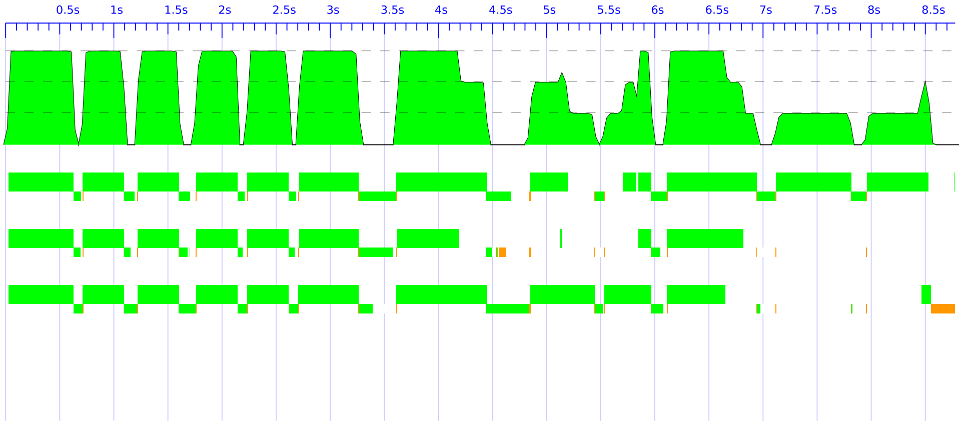

FIGURE B.26: Parallelization of QuickSort for 2 Million Entries over 3 Cores - Original Strategies -Depth: 4 view

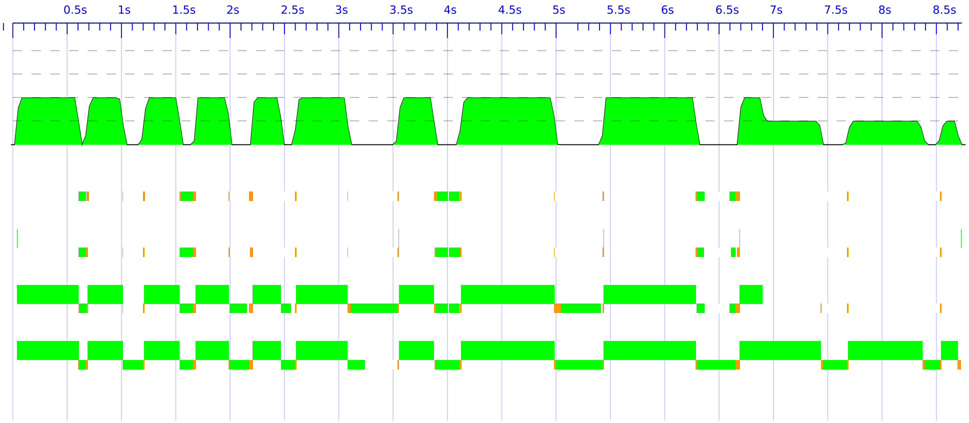

FIGURE B.27: Parallelization of QuickSort for 1 Million Entries over 4 Cores - Original Strategies -Depth: 1 view

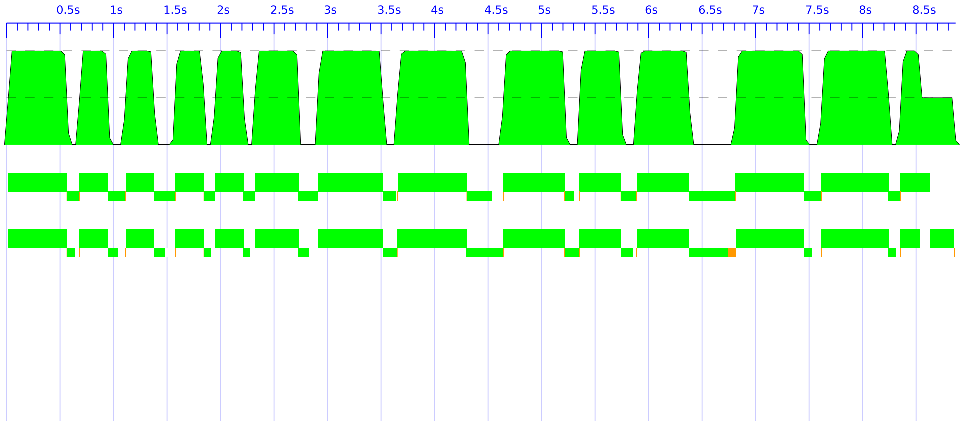

FIGURE B.28: Parallelization of QuickSort for 2 Million Entries over 2 Cores - Evaluation Strategies -Depth: 8 view

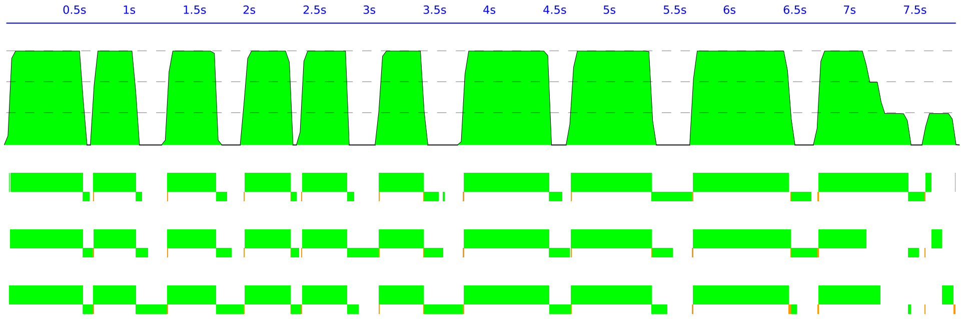

FIGURE B.29: Parallelization of QuickSort for 2 Million Entries over 3 Cores - Evaluation Strategies -Depth: 8 view

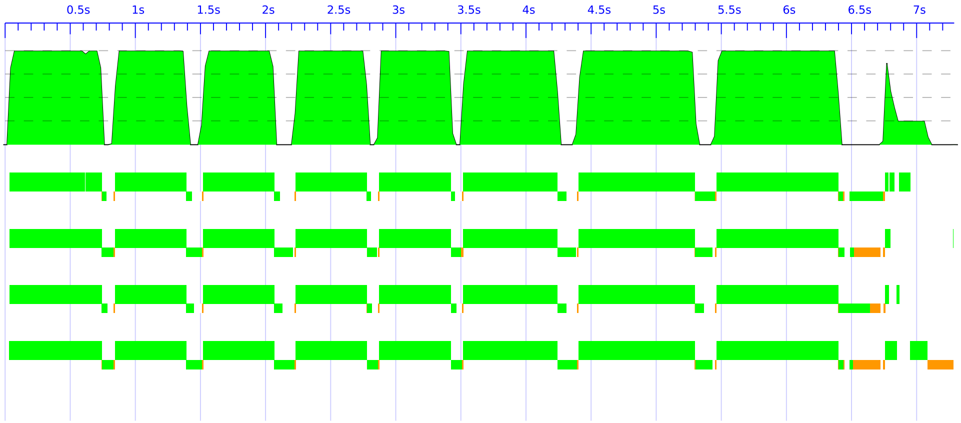

FIGURE B.30: Parallelization of QuickSort for 2 Million Entries over 4 Cores - Evaluation Strategies -Depth: 13 view

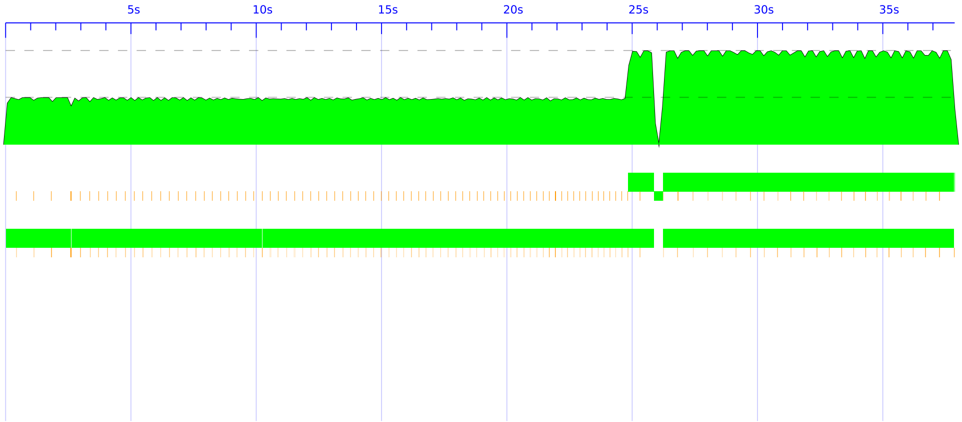

FIGURE B.31: RSA Encryption of 125K Plain-text over 2 Cores - Original Strategies

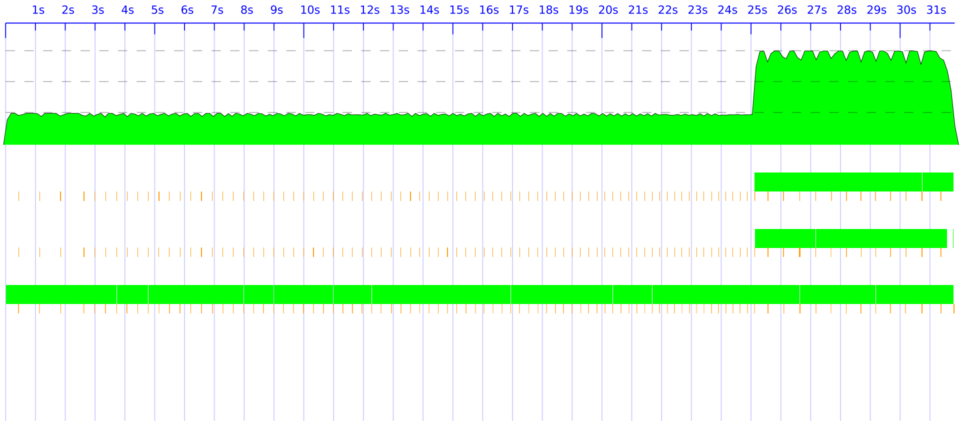

FIGURE B.32: RSA Encryption of 125K Plain-text over 3 Cores - Original Strategies

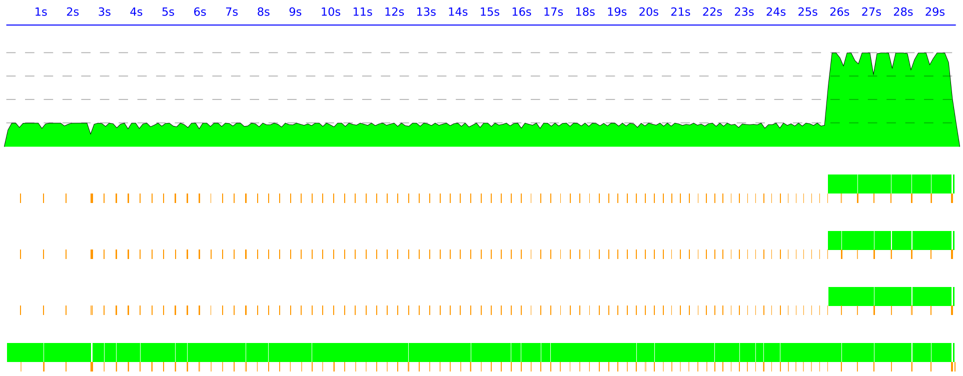

FIGURE B.33: RSA Encryption of 125K Plain-text over 4 Cores - Original Strategies

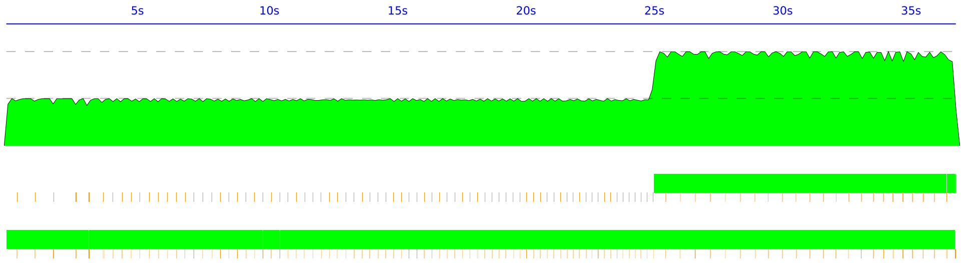

FIGURE B.34: RSA Encryption of 125K Plain-text over 2 Cores - Evaluation Strategies

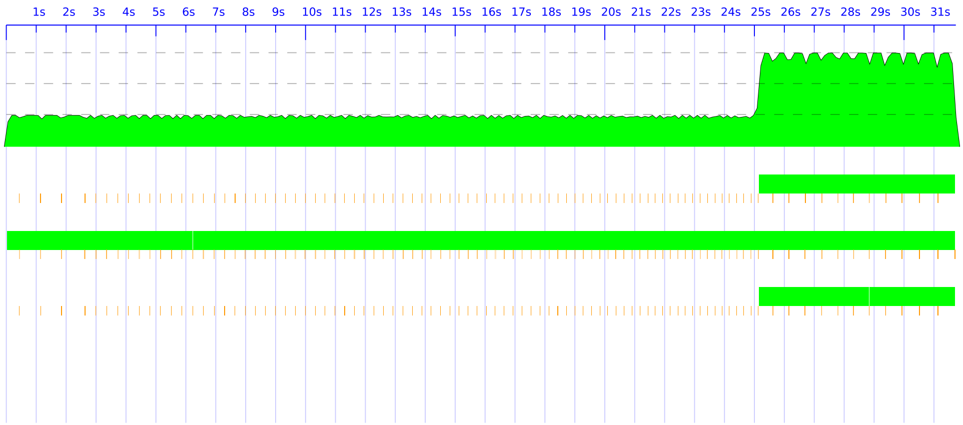

FIGURE B.35: RSA Encryption of 125K Plain-text over 3 Cores - Evaluation Strategies

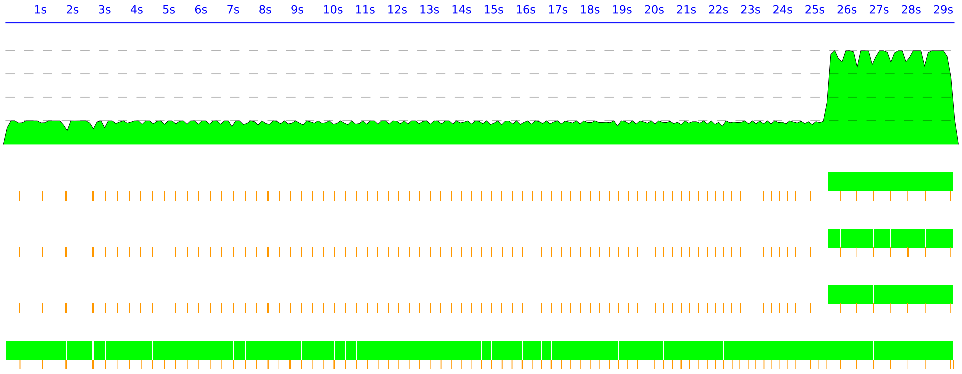

FIGURE B.36: RSA Encryption of 125K Plain-text over 4 Cores - Evaluation Strategies

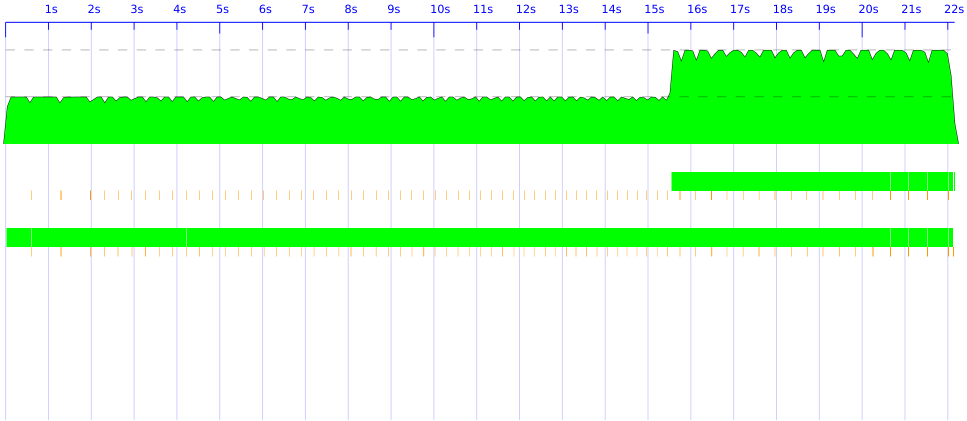

FIGURE B.37: RSA Encryption of 100K Plain-text over 2 Cores - Original Strategies

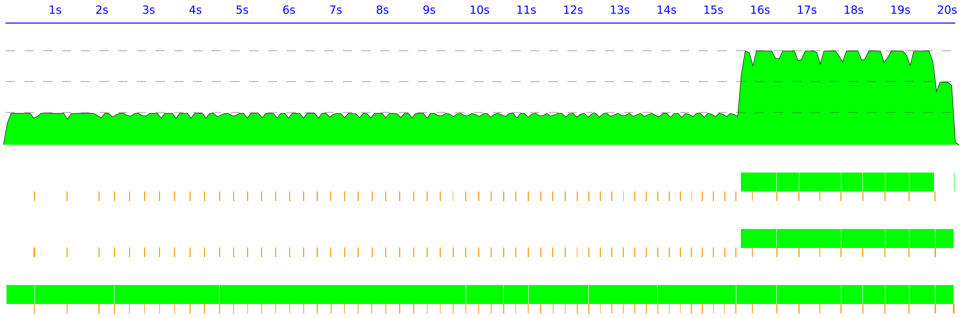

FIGURE B.38: RSA Encryption of 100K Plain-text over 3 Cores - Original Strategies

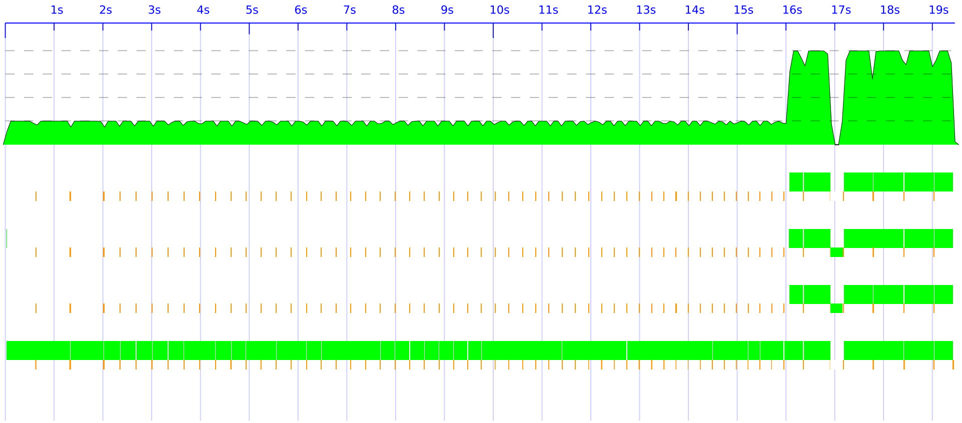

FIGURE B.39: RSA Encryption of 100K Plain-text over 4 Cores - Original Strategies

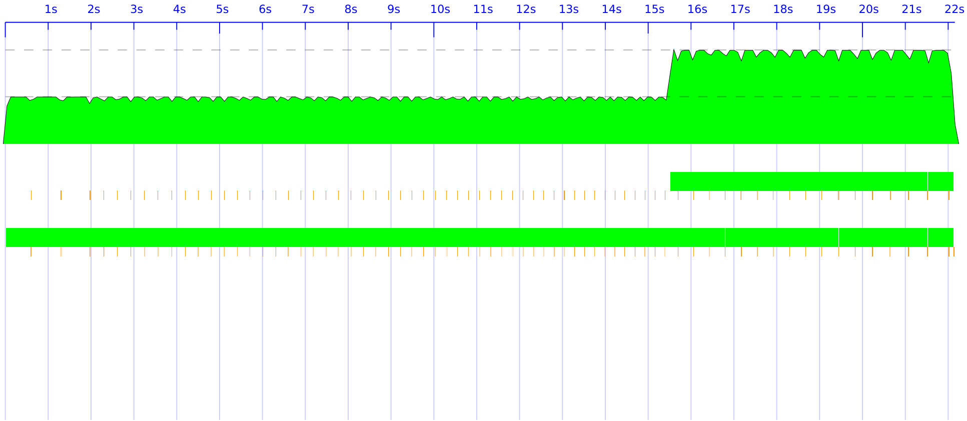

FIGURE B.40: RSA Encryption of 100K Plain-text over 2 Cores - Evaluation Strategies

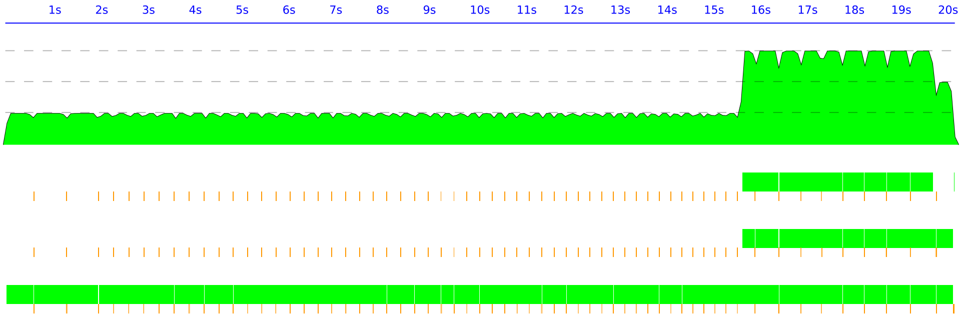

FIGURE B.41: RSA Encryption of 100K Plain-text over 3 Cores - Evaluation Strategies

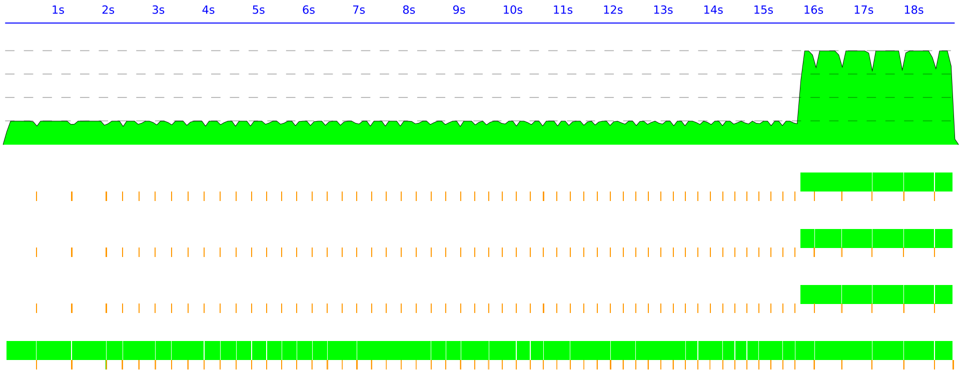

FIGURE B.42: RSA Encryption of 100K Plain-text over 4 Cores - Evaluation Strategies

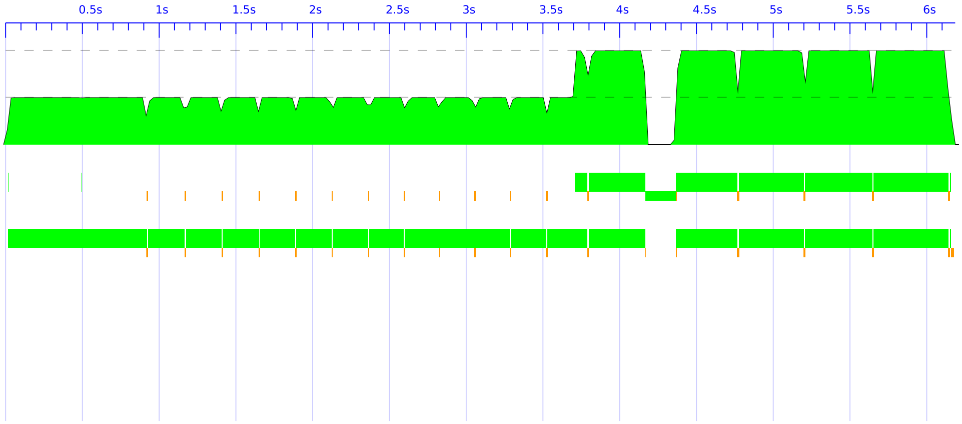

FIGURE B.43: RSA Encryption of 50K Plain-text over 2 Cores - Original Strategies

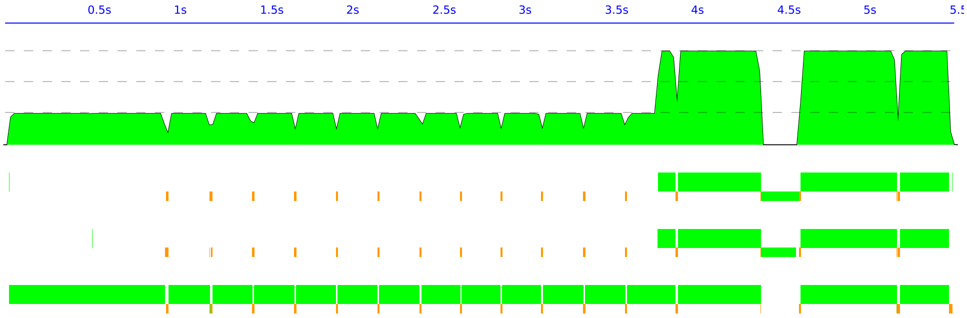

FIGURE B.44: RSA Encryption of 50K Plain-text over 3 Cores - Original Strategies

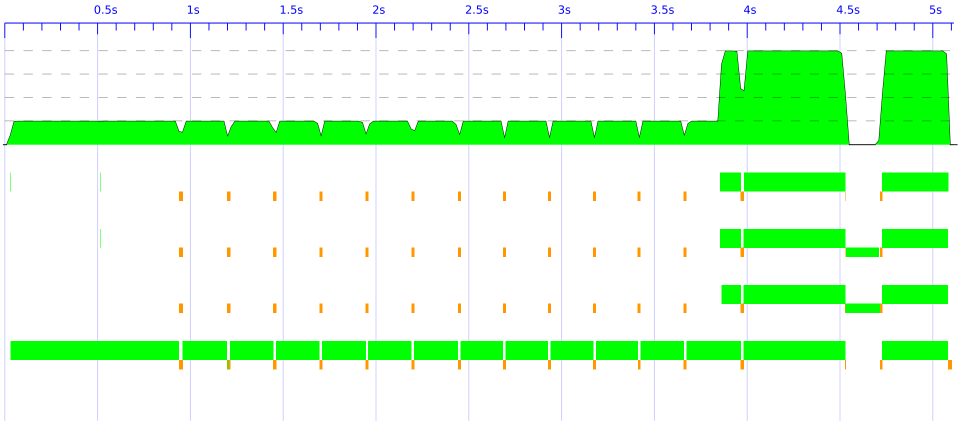

FIGURE B.45: RSA Encryption of 50K Plain-text over 4 Cores - Original Strategies

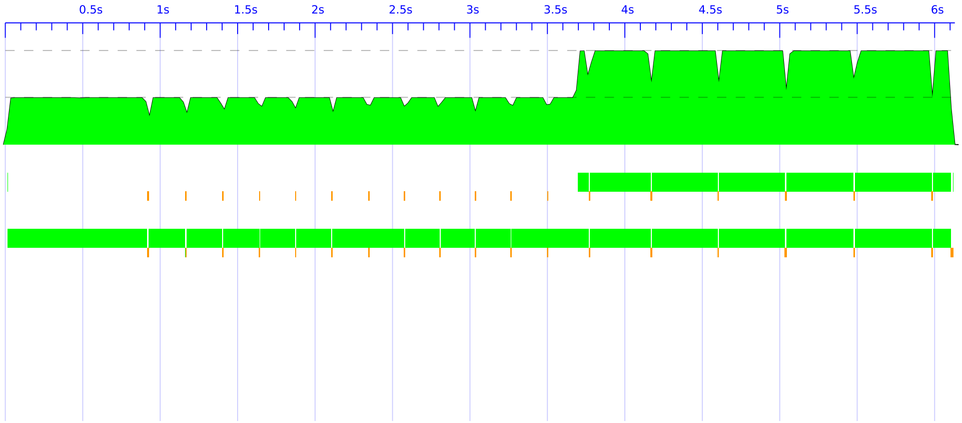

FIGURE B.46: RSA Encryption of 50K Plain-text over 2 Cores - Evaluation Strategies

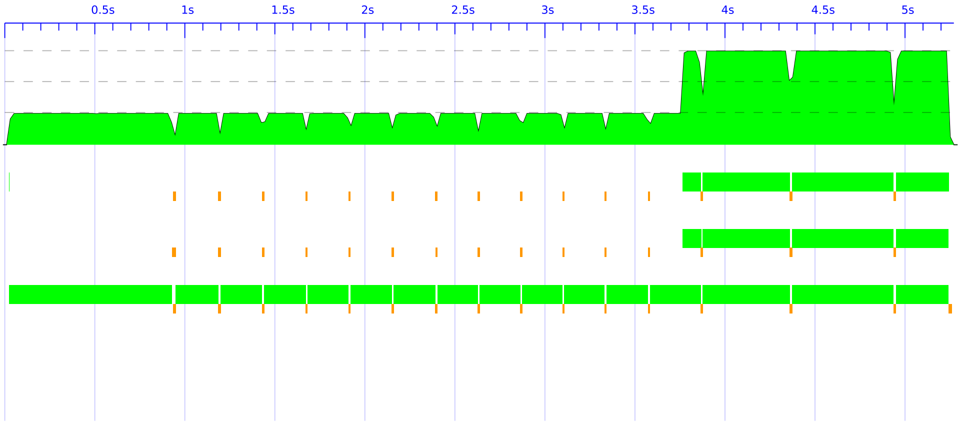

FIGURE B.47: RSA Encryption of 50K Plain-text over 3 Cores - Evaluation Strategies

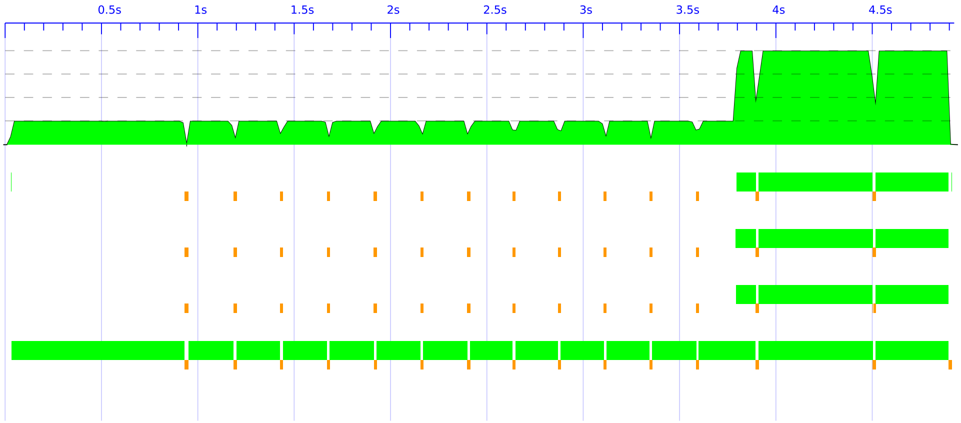

FIGURE B.48: RSA Encryption of 50K Plain-text over 4 Cores - Evaluation Strategies

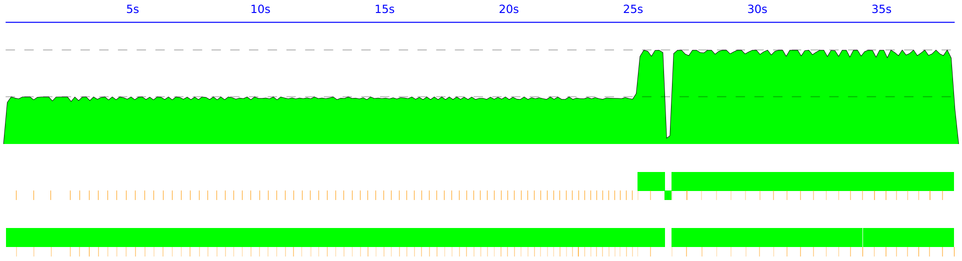

FIGURE B.49: RSA Decryption of 125K Cipher-text over 2 Cores - Original Strategies

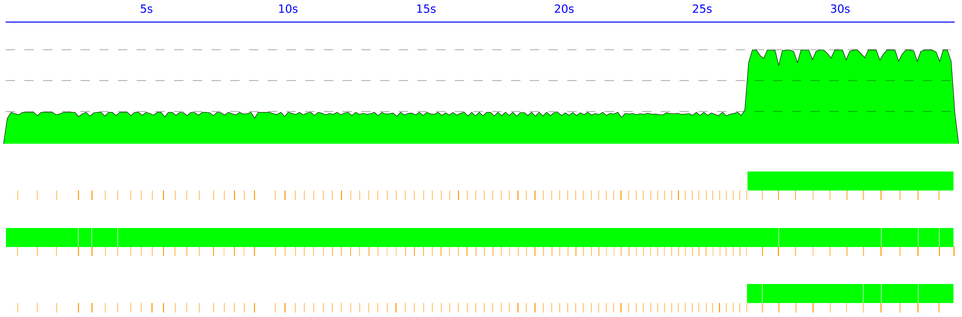

FIGURE B.50: RSA Decryption of 125K Cipher-text over 3 Cores - Original Strategies

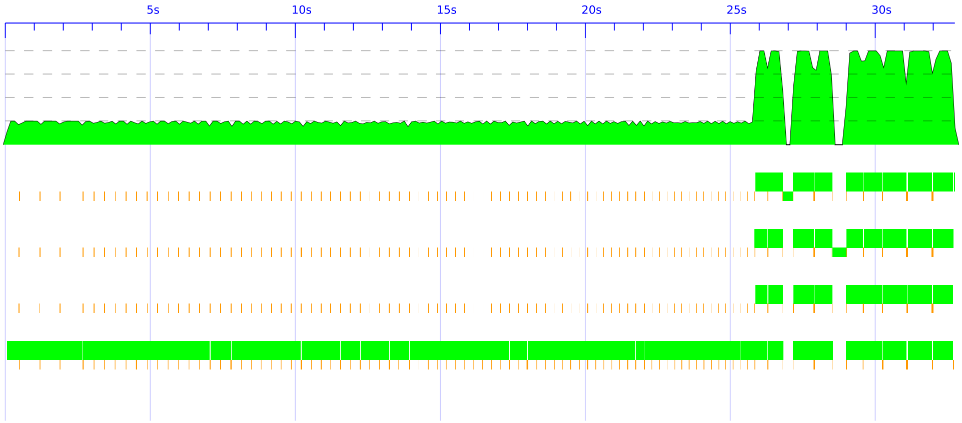

FIGURE B.51: RSA Decryption of 125K Cipher-text over 4 Cores - Original Strategies

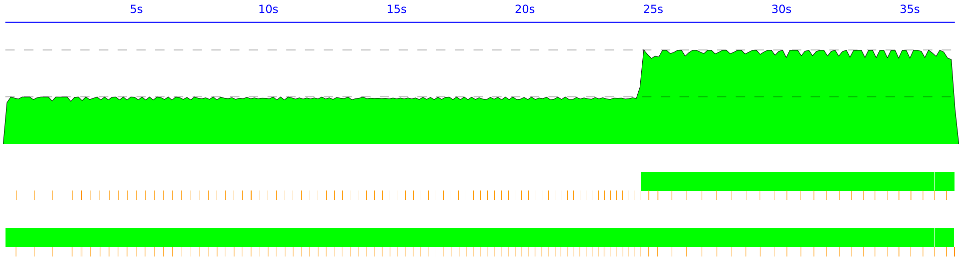

FIGURE B.52: RSA Decryption of 125K Cipher-text over 2 Cores - Evaluation Strategies

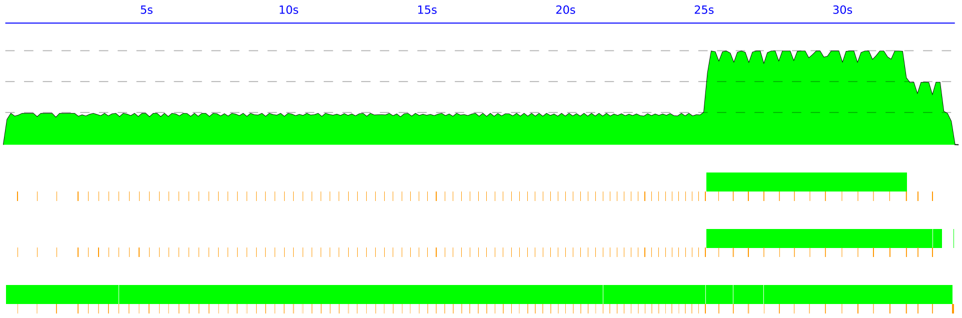

FIGURE B.53: RSA Decryption of 125K Cipher-text over 3 Cores - Evaluation Strategies

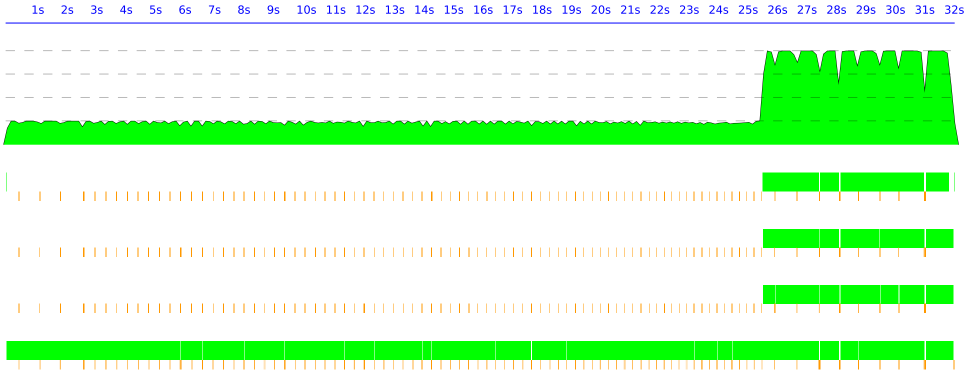

FIGURE B.54: RSA Decryption of 125K Cipher-text over 4 Cores - Evaluation Strategies

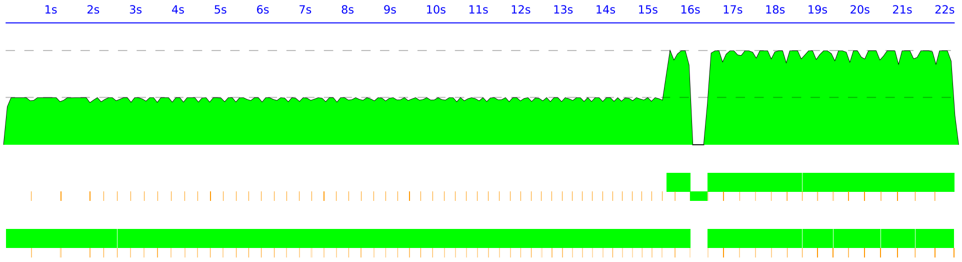

FIGURE B.55: RSA Decryption of 100K Cipher-text over 2 Cores - Original Strategies

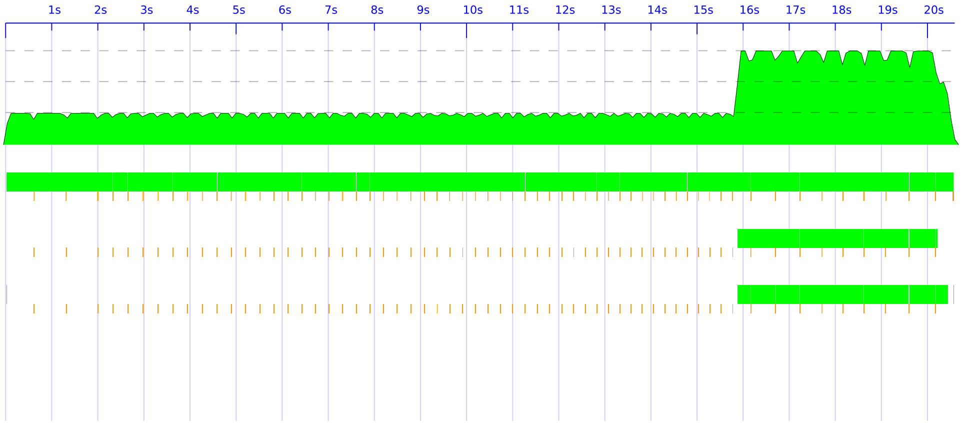

FIGURE B.56: RSA Decryption of 100K Cipher-text over 3 Cores - Original Strategies

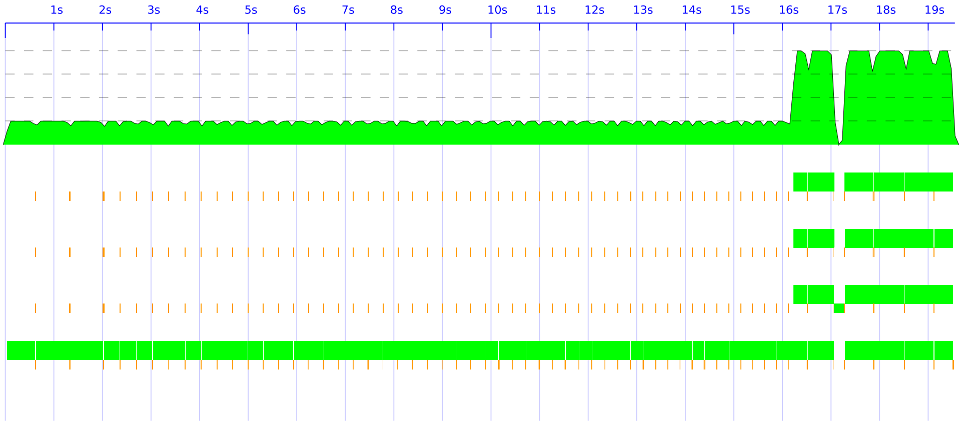

FIGURE B.57: RSA Decryption of 100K Cipher-text over 4 Cores - Original Strategies

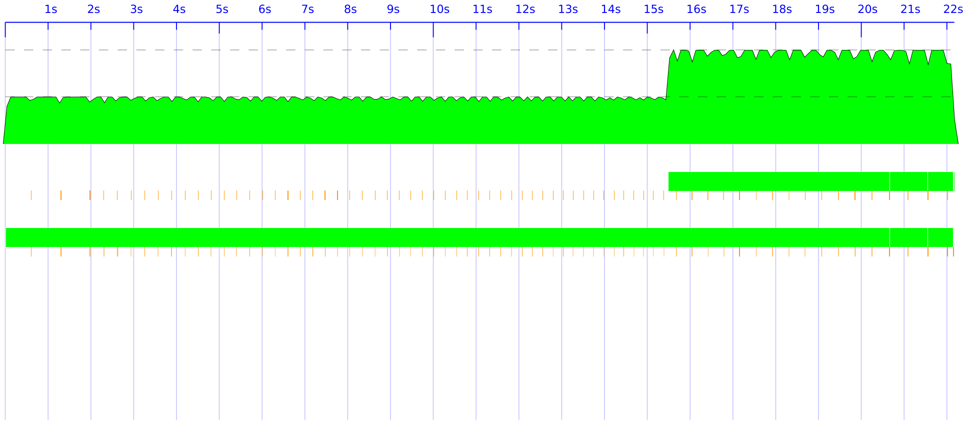

FIGURE B.58: RSA Decryption of 100K Cipher-text over 2 Cores - Evaluation Strategies

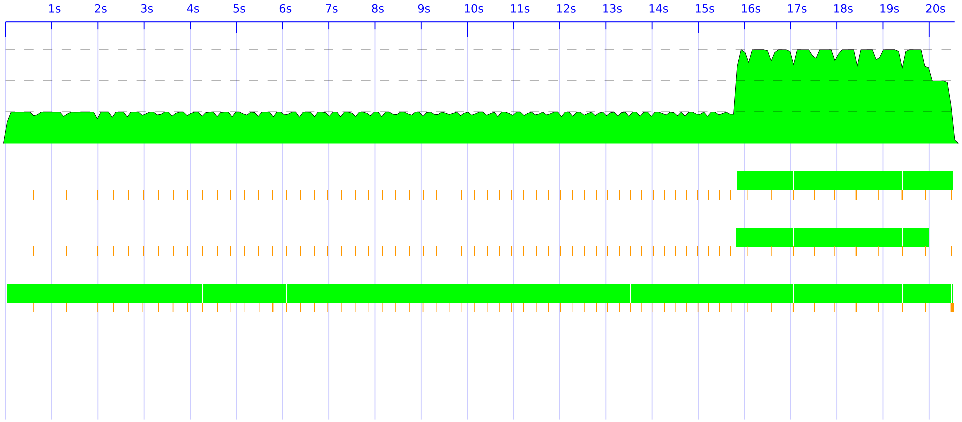

FIGURE B.59: RSA Decryption of 100K Cipher-text over 3 Cores - Evaluation Strategies

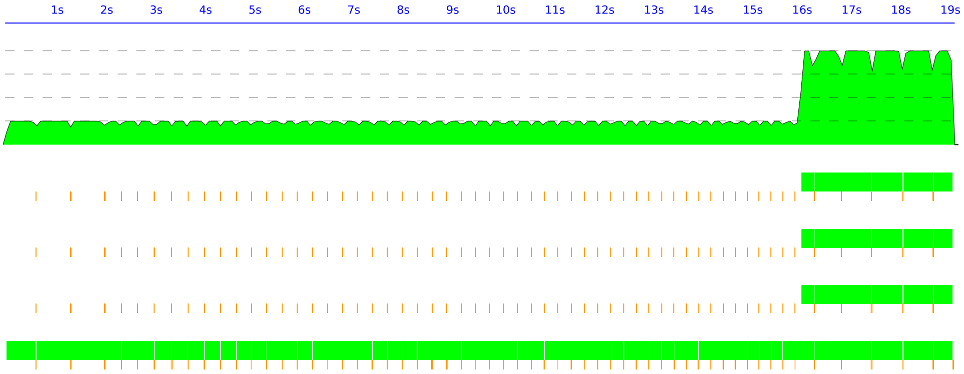

FIGURE B.60: RSA Decryption of 100K Cipher-text over 4 Cores - Evaluation Strategies

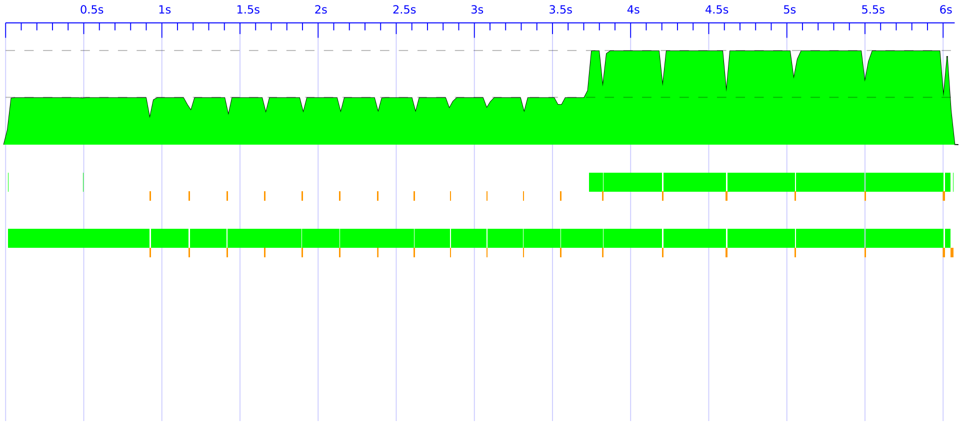

FIGURE B.61: RSA Decryption of 50K Cipher-text over 2 Cores - Original Strategies

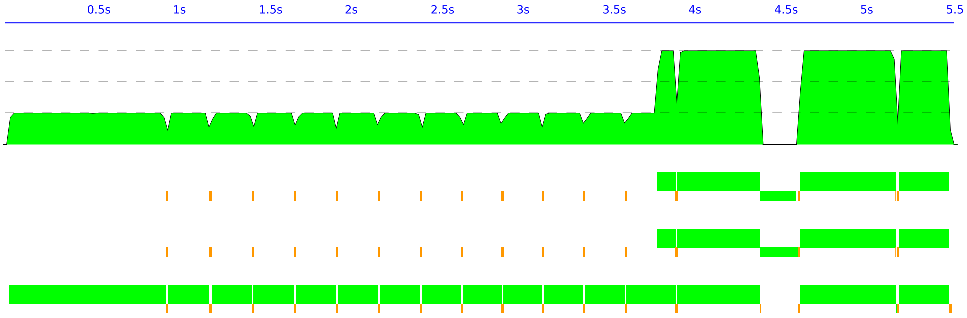

FIGURE B.62: RSA Decryption of 50K Cipher-text over 3 Cores - Original Strategies

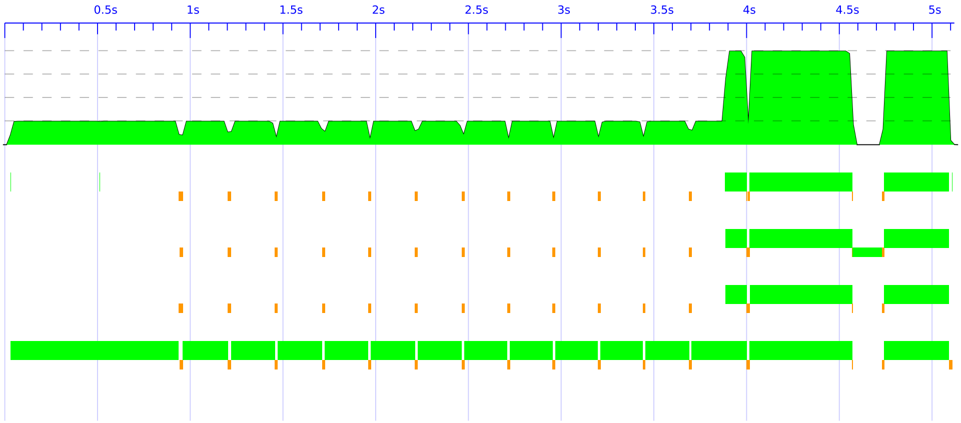

FIGURE B.63: RSA Decryption of 50K Cipher-text over 4 Cores - Original Strategies

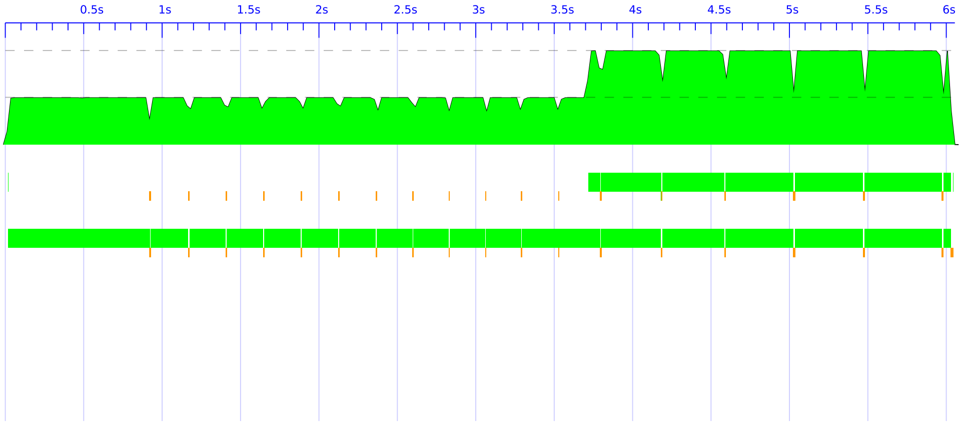

FIGURE B.64: RSA Decryption of 50K Cipher-text over 2 Cores - Evaluation Strategies

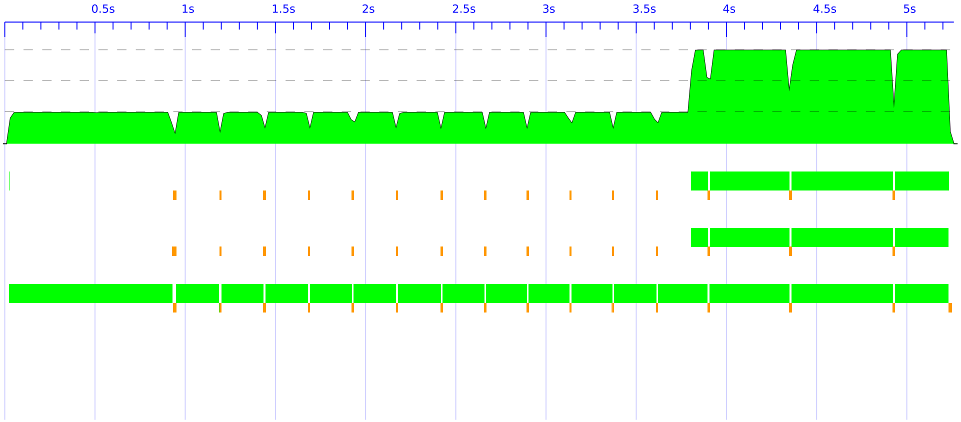

FIGURE B.65: RSA Decryption of 50K Cipher-text over 3 Cores - Evaluation Strategies

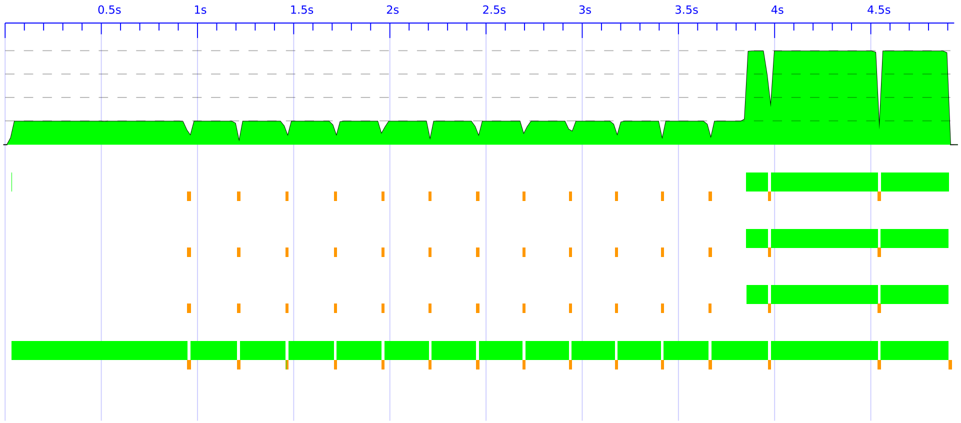

FIGURE B.66: RSA Decryption of 50K Cipher-text over 4 Cores - Evaluation Strategies

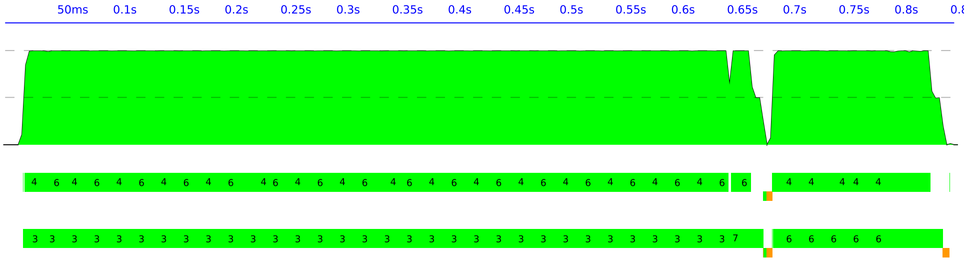

FIGURE B.67: Karatsuba Multiplications of 1K Integers over 2 Cores - Original Strategies - Depth: 16 view

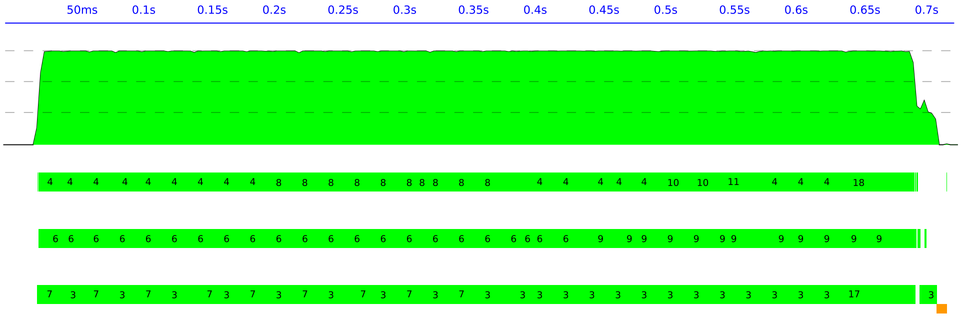

FIGURE B.68: Karatsuba Multiplications of 1K Integers over 3 Cores - Original Strategies - Depth: 8 view

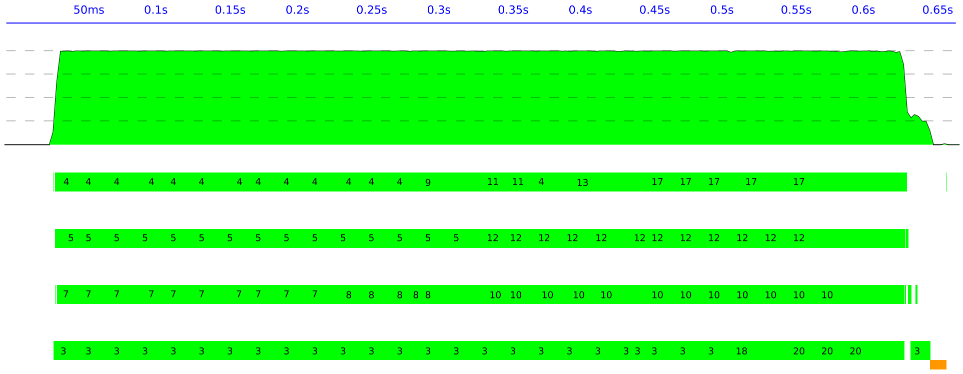

FIGURE B.69: Karatsuba Multiplications of 1K Integers over 4 Cores - Original Strategies - Depth: 8 view

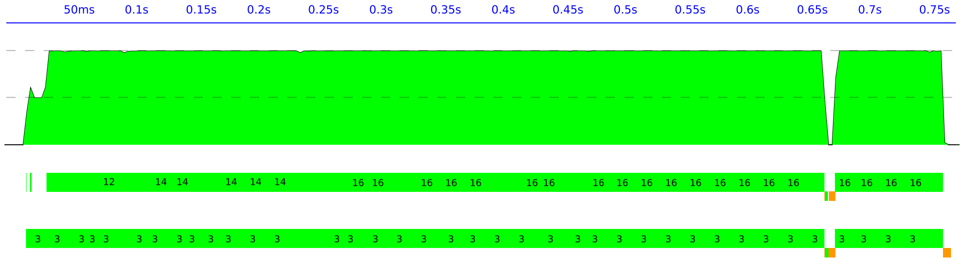

FIGURE B.70: Karatsuba Multiplications of 1K Integers over 2 Cores - Evaluation Strategies - Depth: 8 view

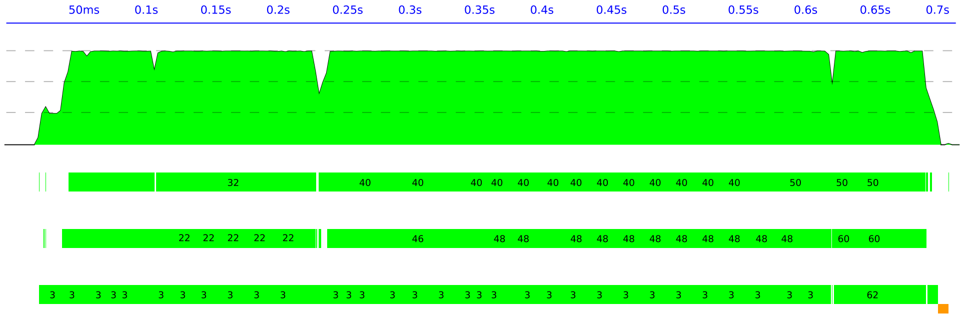

FIGURE B.71: Karatsuba Multiplications of 1K Integers over 3 Cores - Evaluation Strategies - Depth: 8 view

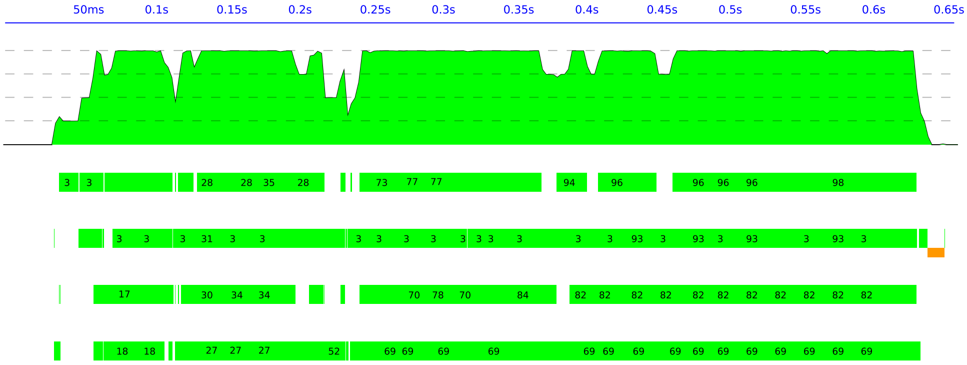

FIGURE B.72: Karatsuba Multiplications of 1K Integers over 4 Cores - Evaluation Strategies - Depth: 8 view

Bibliography

[1] J.-P. Marquis, “Category Theory,” in *The Stanford Encyclopedia of Philosophy*, Spring 2011 ed., E. N. Zalta, Ed. http://plato.stanford.edu/archives/spr2011/entries/category-theory/, 2011. [Online]. Available: http://folli.loria.fr/cds/1999/library/pdf/barrwells.pdf.

[2] A. M. Pitts and M. J. Gabbay, “A Metalanguage for Programming with Bound Names Modulo Renaming,” in *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, ser. Lecture Notes in Computer Science, R. Backhouse and J. N. Oliveira, Eds., vol. 1837. Springer-Verlag, Heidelberg, 2000, pp. 230–255.

[3] B. Plotkin, “Algebra, Categories and Databases,” in *Handbook of Algebra*, Elsevier, Ed., vol. 2, Amsterdam, 2000, p. 79–148.

[4] P. Scott, “Some Aspects of Categories in Computer Science,” in *Handbook of Algebra*, vol. Vol. 2, 2000, p. 3–77.

[5] J. Baez, J. & Dolan, “From Finite Sets to Feynman Diagrams,” in *Mathematics Unlimited – 2001 and Beyond*, B. Springer, Ed., 2001, p. 29–50.

[6] R. Kroemer, *Tool and Object: A History and Philosophy of Category Theory*. Birkhauser, Berlin, Basel and Boston, 2007. [Online]. Available: http://tau.ac.il/~corry/publications/reviews/pdf/kromer.pdf

[7] M. Lipovača, “Learn You a Haskell for Great Good!” http://learnyouahaskell.com/, 2010. [Online]. Available: http://learnyouahaskell.com/

[8] “The Haskell Programming Language,” 2011. [Online]. Available: http://haskell.org/haskellwiki/Haskell

[9] S. Klinger, "The Haskell Programmer's Guide to the IO Monad - Don't Panic," Centre for Telematics and Information Technology (CTIT), Tech. Rep. 2005-5, 2005. [Online]. Available: http://doc.utwente.nl/53411/

[10] J. Berryman, *The State Monad: a tutorial for the confused?*, 10 2009. [Online]. Available: http://coder.bsimmons.name/blog/2009/10/the-state-monad-a-tutorial-for-the-confused/

[11] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder, "Seq no more: Better Strategies for Parallel Haskell," in *Haskell '10: Proceedings of the Third ACM SIGPLAN Symposium on Haskell.* ACM, 2010. [Online]. Available: http://community.haskell.org/~simonmar/papers/strategies.pdf

[12] S. Marlow, *Parallel and Concurrent Programming in Haskell version 1.1*, Microsoft Research Ltd., Cambridge, U.K., September 12 2011.

[13] H.-W. Loidl, P. Trinder, K. Hammond, S. Junaidu, R. Morgan, and S. P. Jones., "Engineering Parallel Symbolic Programs in GPH," *Concurrency - Practice and Experience*, vol. 11, pp. 701–752, 1999.

[14] J. D. Jones, S. Marlow, and S. Singh, "Parallel Performance Tuning for Haskell," in *Haskell '09: Proceedings of the Second ACM SIGPLAN Symposium on Haskell.* ACM, 2009. [Online]. Available: http://community.haskell.org/~simonmar/papers/threadscope.pdf

[15] M. B. Charles Wells, *Category Theory Lecture Notes for ESSLLI*, http://folli.loria.fr/cds/1999/library/pdf/barrwells.pdf., Ed. Citeseer, 1999, vol. 99. [Online]. Available: http://folli.loria.fr/cds/1999/library/pdf/barrwells.pdf.

[16] J. Longley, "Formal Programming Language Semantics note 1," Laboratory for Foundations of Computer Science, School of Informatics, Tech. Rep., 08.10.03. [Online]. Available: http://www.inf.ed.ac.uk/teaching/courses/fpls/note1.pdf

[17] J. Deng, "The Formal Semantics of Programming Languages." Shanghai Jiaotong University, Tech. Rep., 16.06.2010. [Online]. Available: http://202.120.38.217/~yuxin/teaching/Semantics/sem.pdf

[18] Y. Litus, "Operational Semantics," Simon Fraser University, School of Computing Science, Tech. Rep., Fall, 2009. [Online]. Available: http://www.sfu.ca/~ylitus/courses/cmpt383/OperationalSemantics.pdf

[19] F. Barbanera, C. . T. Genovese, and M. Salfi, "Short Introduction to the Lambda-calculus," DIPARTIMENTO DI MATEMATICA E INFORMATICA, Università di Catania, Viale A. Doria, 6 I-95125 Catania, Italy, Tech. Rep., 11 2005. [Online]. Available: http://www.dmi.unict.it/~barba/LinguaggiII.html/READING_MATERIAL/LAMBDACALCULUS/ShortIntroductiontotheLambdaCalculus.PDF

[20] P. Sestoft, "Demonstrating Lambda Calculus Reduction," in *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, number 2566 in Lecture Notes in Computer Science*. Springer-Verlag, 2002, pp. 420–435.

[21] U. Reddy, "Handout 2: Lambda Calculus Examples," The University of Birmingham, School of Computer Science, Tech. Rep., 2009-10. [Online]. Available: http://www.cs.bham.ac.uk/~udr/popl/handout2.pdf

[22] B. Hardekopfh, "Lambda Calculus Lecture Notes," Department of Computer Science, University of California, Santa Barbara, Tech. Rep., 2011. [Online]. Available: http://www.cs.ucsb.edu/~benh/cs162/slides/03-lambda-calculus.pdf

[23] Y. Huang, "Lambda Calculus and Computability," Department of Computer Science, University of Virginia, Tech. Rep., 2008. [Online]. Available: http://www.cs.virginia.edu/cs302/classes/class20.pdf

[24] A. Loeh, *Tutorial: Deterministic Parallel Programming in Haskell.*, Well-Typed LLP, 10 2011. [Online]. Available: http://www.well-typed.com/Hal6/Presentation.pdf

[25] S. Marlow, S. P. Jones, and S. Singh, "Runtime Support for Multicore Haskell," in *ICFP '09: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming*, August 2009.

[26] P. Trinder, K. Hammond, H. W. Loidl, and S. P. Jones, "Algorithm + Strategy = Parallelism," *Journal of Functional Programming*, vol. 8, pp. 23–60, 1998. [Online]. Available: http://research.microsoft.com/Users/simonpj/Papers/strategies.ps.gz.

[27] D. Gray, "Implementing Public-Key Cryptography in Haskell," School of Computer Applications Dublin City University, Tech. Rep., November, 12 2001. [Online]. Available: http://www.citidel.org/bitstream/10117/120/13/paper.pdf