THEORETICAL COMPUTER SCIENCE for the WORKING CATEGORY THEORIST

NOSON S. YANOFSKY

Blurb for back of the book:
Using basic category theory (category, functor, natural transformation, etc.), this short book describes all the central concepts and proves the main theorems of theoretical computer science. Category theory, which works with functions, processes, and structures, is uniquely qualified to present the fundamental results of theoretical computer science. In this text, we meet some of the deepest ideas and theorems of modern computers and mathematics, e.g., Turing machines, unsolvable problems, the P=NP question, Kurt Gödel's incompleteness theorem, intractable problems, cryptographic protocols, Alan Turing's Halting proble, and much more. The concepts come alive with many examples and exercises. This short text covers the usual material taught in a year-long course.

Three sentence blurb:
With just the basics of category theory, this text presents the usual material taught in a year-long course of theoretical computer science. We cover models of computation, computability theory, and complexity theory. We also describe the essentials of formal language theory, cryptography, Kolmogorov complexity theory, and much more.

Dedicated to the memory
of my brother
Eli

Every language is a world. Without translation, we would inhabit parishes bordering on silence.
George Steiner

It is a deep result in the sense that it provides a method for handling, with elegance and intellectual economy, constructions that would otherwise require extensive, complex treatment.
Hartley Rogers, Jr.
Page 179 of [64]

It appeared to me that the time is ripe to try to give the subject a coherent mathematical presentation that will bring out its intrinsic aesthetic qualities and bring to the surface many deep results which merit becoming part of mathematics, regardless of any external motivation.
Samuel Eilenberg
Page xiii of [23]

Noson S. Yanofsky

Preface

When I was studying for my PhD, I took something called "the theory qualifier," which I can now definitively say was the second *worst thing in my life after chemotherapy.*
Randy Pausch
The Last Lecture [58], Page 24.

"Unsolvable problems," "The P=NP question," "Alan Turing's Halting proble," "Kurt Gödel's incompleteness theorem," "Intractable problems," etc. These are just some of the many buzzwords that one hears about in the current technology media. These concepts are all associated with theoretical computer science. This field touches on some of the deepest and most profound parts of contemporary science and mathematics. The issues discussed in this book emerge in almost every area of computers, theoretical physics, and mathematics. In this short text you will learn the main ideas and theorems of theoretical computer science.

While there are many textbooks to study theoretical computer science, this book is unique. This is the first text that translates the ideas of theoretical computer science into the basic language of category theory. With some of the simple concepts of category theory in hand, the reader will be able to understand all the ideas and theorems that are taught in a year-long course of theoretical computer science.

Why is it better to learn theoretical computer science with category theory?

- It is easy! Most of the ideas and theorems of theoretical computer science are consequences of the categorical definitions. Once the categories and functors of the different models of computation are set up,

the concepts of theoretical computer science simply emerge. Many theorems are a straightforward consequence of composition and functoriality.

- It is short! The powerful language of category theory ensures us that what can usually be taught in a huge textbook can be taught in these few pages.
- It gives you the right perspective! Anyone who has ever experienced any category theory knows that it makes you "zoom out" and see things from a "bird's-eye view." Rather than getting overwhelmed by all the little details so that you cannot "see the forest for the trees," category theory gets you to see the big picture.
- It gets to the core of the issues! Category theory distills the important aspects of a subject and leaves one with the main structures needed to understand what is going on.
- It connects better with other areas of science and math! Category theory has become the *lingua franca* of numerous areas of mathematics, and theoretical physics (e.g., [5, 14]). By putting the ideas of theoretical computer science in the language of category theory, connections are made to these other areas.

There are several texts and papers for the computer scientist to learn category theory [7, 77, 60, 4, 67, 75, 61]. This book is totally different. Our intended audience here is someone who already knows the basics of category theory and who wants to learn theoretical computer science.

One does not need to be a category theory expert to read this book. We do not assume that the reader knows much more than the notion of category, functor, and equivalence. Any more advanced categorical concepts are taught in Chapter 2 or "on the fly" when needed.

Organization

The book starts with a quick introduction to the major themes of theoretical computer science and explains why category theory is uniquely qualified to describe those ideas and theorems. The next chapter is a short teaching aid of some categorical structures that arise in the text and that might be unknown to the category theory novice

Since we will be discussing computation, we must fix our ideas and deal with various **models of computation**. What do we mean by a computation? The literature is full of such models. Chapter 3 classifies and

categorizes these various models while showing how they are all linked together. With the definitions and notation given in Sections 3.1 and 3.2, one can safely move on to almost any other Section of the book.

Chapters 4 and 5 are the core of the book. Chapter 4 describes which functions can be executed by a computer — and more importantly — which functions *cannot* be executed by a computer. This is called **computability theory**. We give many examples of functions that no computer can perform. Along the way, we meet Alan Turing's Halting problem and Kurt Gödel's famous incompleteness theorem. We also discuss a classification and hierarchy of all the functions that cannot be computed by computers.

We go on to discuss **complexity theory** in Chapter 5. This is where we ask and answer what is *efficiently* computable. Every computable function demands a certain amount of time or space to compute. We discuss how long and how complicated certain computations are. Along the way we meet the famous NP-complete problems and the P=NP question.

Chapter 6 is about a special type of proof that arises in many parts of theoretical computer science (and many areas of mathematics) called a **diagonal argument**. Although the diagonal arguments come in many different forms, we show that they are all instances of a single simple theorem of category theory. Along the way we meet Stephen Kleene's recursion theorem, Georg Cantor's different levels of infinity, and John von Neumann's Self-Reproducing Automata.

While computability theory and complexity theory are the main topics taught in a typical theoretical computer science class, there are many other topics that are either touched on or are taught in more advanced courses. Chapter 7 is a series of short introductions for several exciting topics:

- **Formal language theory** describes the relationship between computers and languages. The more complicated a computer is, the more complicated is the language it understands. Along the way we meet the Chomsky hierarchy and many of the important theorems of finite automata.
- **Cryptography** is about encoding and decoding secret messages. We give a simple categorical description of a structure and show that every major cryptographic protocol is an instance of this categorical structure.
- **Kolmogorov complexity theory** discusses the complexity of strings

of symbols. We talk about the compressability of strings. Along the way, we meditate on the nature of randomness.

- **Algorithms** are the core of computer science. This short section uses categories to provide a formal definition of an algorithm. This definition highlights the intimate relationship between programs, algorithms, and functions.

The last section of Chapter 7 summarizes what was learned about theoretical computer science from the categorical perspective. It lists off some common themes that were seen throughout the text. It also has a guide for readers who wish to continue on with their studies. The book ends with an Appendix that has answers to selected exercises.

Throughout the text, there are many examples and exercises. The reader is strongly urged to work on the exercises. In addition, the book is sprinkled with short "Advanced Topics" paragraphs that point out certain advanced theorems or ideas. We also direct the reader to where they can learn more about the topic.

At the end of every Chapter, the reader is directed to places where they can learn more about the topic from sources in classical theoretical computer science and in category theory.

There are, however, omissions:

We do not cover every part of theoretical computer science. For example, we will not deal with program semantics and verification, analysis of algorithms, data structures, and information theory. While all these are interesting and can be treated with a categorical perspective, we have omitted these topics because of space considerations. Even the topics that we do cover, we invariably must miss certain theorems and ideas.

This is not a textbook about the relationship between category theory and computer science. Over the past half century, category theorists have made tremendous advances in computer science by applying categorical concepts and constructions to the structures and processes of computation. The literature in this area is immense. Section 7.5 gives some texts and papers to look for more in this direction. This text will not present all these areas. Rather, it is focused on the task at hand. Here we use a novel presentation to understand all the classic parts of theoretical computer science.

This text does not stand alone. I maintain a web page for the text at

www.sci.brooklyn.cuny.edu/noson/tcstext
The web page will contain links to interesting books and articles on category theory and theoretical computer science, some solutions to exercises not solved in the text, and an erratum. The reader is encouraged to send any and all corrections and suggestions to *noson@sci.brooklyn.cuny.edu*.

Acknowledgment

There are many people who made this book possible. In 1989, as an undergraduate at Brooklyn College, I had the privilege of taking a master's level course in theoretical computer science with Rohit Parikh. This world-class expert made the entire subject come alive, and I have been hooked ever since. I have stayed by his side ever since. Although thirty years has passed, I am still amazed at how much he knows and — more importantly — how much he understands on a deeper level. He was my teacher, and now he is a colleague and a friend. I am forever grateful to him.

One of the founders of category theory was Saunders Mac Lane (1909 – 2005). Whenever we met at a conference, he was always warm and friendly. He wrote one of the "bibles" of the field, *Categories for the Working Mathematician* [51]. As the title implies, the book teaches category theory to someone who is well-grounded in mathematics. In 1999, I wrote him a letter about a book I wanted to write. The book was supposed to teach much advanced mathematics to someone who is well-grounded in basic category theory. I asked him for permission to call the book *Mathematics for the Working Category Theorist*. Saunders was encouraging and gave his imprimatur. The book you are holding in your hand is a first step towards that dream.

I would like to thank John Baez for taking an interest in this work and for his support. Over the years, I gained so much from John's amazing papers and posts. His clarity and lucidity are an inspiration. More than anyone else, John has shown how the language of category theory can be applied in so many different areas. In addition, Yuri Manin's paper [53] and book [52] have been an inspiration for this book. He put all computational processes into one category called a "computational universe." Here we take these ideas further. I thank him for his ideas and for taking an interest in my work.

Ted Brown, the former chair of the computer science department at The Graduate Center of the City University of New York, has been very kind

to me. He asked me to teach the PhD level theoretical computer science course every year from 2003 through 2013. I also taught several other advanced courses in those years. That experience has been tremendously helpful. My thanks also go out to the next chair of the department, Prof. Robert M Haralick. Their kindness is appreciated.

Brooklyn College has been my intellectual home since 1985. The entire administration, faculty, staff, and students have been encouraging and have made the environment conducive to such projects. Dean Kleanthis Psarris, Chairman Yedidyah Langsam, and former Chairman Aaron Tenenbaum were all very helpful. Over the years I also gained much from David M. Arnow, Eva Cogan, Lawrence Goetz, Ira Rudowsky, and Joseph Thurm. I thank them all.

Sergei Artemov, Gershom Bazerman, Chris Calude, John Connor, James L. Cox, Walter Dean, Scott D. Dexter, Mel Fitting, Grant Roy, Tzipora Halevi, Joel David Hamkins, Keith Harrow, Pieter Hofstra, Karen Kletter, Roman Kossak, Deric Kwok, Moshe Lach, Florian Lengyel, Armando Matos, Michael Mandel, Yuri Manin, Jean-Pierre Marquis, Robert Paré, Rohit Parikh,Vaughn Pratt, Avi Rabinowitz, Phil Scott, Robert Seely, Morgan Sinclaire, David Spivak, Alexander Sverdlov, Gerald Weiss, Paula Whitlock, Mark Zelcer, and all the members of The New York City Category Theory Seminar were very helpful with discussions and editing. I am also grateful to two anonymous reviewers for their many comments and helpful suggestions. Thank you!

This work was done with the benefit of the generous support of PSC-CUNY Award 61522-00 49.

Neither this book nor anything else could have been done without my wife Shayna Leah. Her loving help in every aspect of my life is deeply appreciated. I am grateful to my children, Hadassah, Rivka, Boruch, and Miriam for all the joy they bring me.

This book is dedicated to the memory of my brother Rabbi Eliyahu Mordechai Yanofsky Z"TL (February 19, 1964 - March 12, 2018). He was a brilliant scholar who spent his entire life studying Jewish texts and helping others. Eli had a warm personality and a clever wit that endeared him to many. He was an inspiration to hundreds of friends and students. As an older brother, he was a paragon of excellence. He left behind parents, a wife, siblings, and ten wonderful children. His loss was terribly painful. He is sorely missed.

Contents

1

Introduction

If you talk to a man in a language he understands, that goes to his head.
If you talk to him in his language, that goes to his heart.
Nelson Mandela

Theoretical computer science started before large-scale electronic computers actually existed. In the 1930's, when engineers were just beginning to work out the problems of making viable computers, Alan Turing was already exploring the limits of computation. Before physicists and engineers began struggling to create quantum computers, theoretical computer scientists designed algorithms for quantum computers and described their limitations. Even today, before there are any large-scale quantum computers, theoretical computer science is working on "post-quantum cryptography." The prescient nature of this field is a consequence of the fact that it studies only the important and foundational issues about computation. It asks and answers questions such as "What is a computation?" "What is computable?" "What is efficiently computable?" "What is information?" "What is random?" "What is an algorithm?" etc.

For us, the central role of a computer is to calculate functions. They input data of a certain type, manipulate the data, and have outputs of a certain type. In order to study computation we must look at the collection of all such functions. We must also look at all methods of computation and see how they describe functions. To every computational method, there is an associated function. What will be important is to study which functions are computed by a computational method and which are not. Which functions are easily computed and which functions need more resources? All these issues — and many more — will be dealt with in these pages.

Category theory is uniquely qualified to present the ideas of theoretical computer science. The basic language of category theory was formulated

by Samuel Eilenberg and Saunders Mac Lane in the 1940s. They wanted to classify and categorize topological objects by associating them to algebraic objects. In order to compare topological objects and algebraic objects, Eilenberg and Mac Lane had to formulate a language that was not specifically related to topology or algebra. It was abstract enough to deal with both types of objects. This is where category theory gets its power. By being about nothing in particular, or "general abstract nonsense," it is about everything. In this text, we will see categories containing various types of functions and different models of computation. There are then functors comparing these functions and computational methods.

When categories first started, the morphisms in a category were thought of as homomorphisms between algebraic structures or continuous maps between topological spaces. The morphism

$$X \xrightarrow{\quad f \quad} Y \tag{1.1}$$

might be a homomorphism f from group X to group Y or it might be a continuous map from topological space X to topological space Y. However, as time progressed, researchers realized that they might look at f as a *process* of going from X to Y. An algebraic example is when X and Y are vector spaces and f is a linear transformation, a way of going from X to Y. In logic, X and Y can be propositions and f is a way of showing that after assuming X one can imply Y. Or for logicians interested in proofs, again X and Y were propositions and f represented a proof (or a formal way of showing) that with X as an assumption, one can conclude with Y. Physicists also got into the action. For them, X and Y were states of a system and f represented a dynamic or a process of going from state X to state Y. Geometers and physicists considered X and Y to be manifolds of a certain type and f to be a larger shape whose boundaries were X and Y. Computer scientists consider X and Y to be types of data and f is a function or a computational process that takes inputs of type X to output of type Y. In this text, many of our categories will be of this form. (See [5, 14] for comparisons of many different types of processes.)

It pays to reexamine the definition of a category with the notion of a morphism as a process. When there are two composable maps

$$X \xrightarrow{\quad f \quad} Y \xrightarrow{\quad g \quad} Z \tag{1.2}$$

their composition $g \circ f$ is to be thought of as **sequential processing**, i.e., first performing the f process and then performing the g process. If X is any object, then there is an identity process $Id_X : X \longrightarrow X$ which does

nothing to X. No change is made. When any process is composed with the identity process, it is essentially the same as the original process. When there are three composable maps,

$$X \xrightarrow{\ f\ } Y \xrightarrow{\ g\ } Z \xrightarrow{\ h\ } W \tag{1.3}$$

the associativity $(h \circ g) \circ f = h \circ (g \circ f)$ essentially follows from the fact that we are doing these processes in order: first f, then g, and finally h.

In the 1960s, with the development of monoidal categories, it was realized that not only do categories contain structures, but sometimes categories themselves have structure. In certain cases objects and morphisms of categories can be "tensored" or "multiplied." For example, given X and X' in the category of groups, one can form their tensor product $X \otimes X'$ as an object of the category of groups. Or, if X and X' are objects in the category of sets, then their Cartesian product, $X \times X'$, is a also an object in the category of sets. As usual in category theory, we are concerned with morphisms. If $f : X \longrightarrow Y$ and $f' : X' \longrightarrow Y'$ are morphisms then we can not only form $X \otimes X'$ and $Y \otimes Y'$ but also $f \otimes f'$ which we can write as

$$\begin{array}{ccc} X & \xrightarrow{\ f\ } & Y \\ & \otimes & \\ X' & \xrightarrow[\ f'\ ]{} & Y' \end{array} \qquad \text{or} \qquad X \otimes X' \xrightarrow{\ f \otimes f'\ } Y \otimes Y'. \tag{1.4}$$

This should be considered as modeling **parallel processing**, i.e., performing the f and f' process independently and at the same time. When a category has this ability to tensor its objects and morphisms, the category and the extra structure form a **monidal category**. When the $f \otimes f'$ process is essentially the same as the $f' \otimes f$ process, we formalize that and come to the notion of a **symmetric monoidal category**. Most of the categories we will meet in these pages will be symmetric monoidal categories.

Let us summarize. The categories we will use have objects that are types of data that describe the inputs and outputs of functions and processes, while the morphisms are the functions or the processes. There will be two ways of composing the morphisms: (i) regular composition within the category will correspond to function composition or sequential processing, and (ii) the monoidal structure of the categories will correspond to parallel processing of functions or processes. There will be various functors between such categories which take computational procedures to functions or other computational procedures. We will ask questions such

as: when are these functors surjective? What is in their image? What is not in their image? When are they equivalences? etc.

This book will not only use the language of categories, it will also use the mindset of categories. As anyone who has studied a topic in category theory knows, this language has a unique way of seeing things. Rather than jumping in to the nitty-gritty details, categorists insist on setting up what they are talking about first. They like to examine the larger picture before getting into the details. Also, category theory has a knack for getting to the essence of a problem while leaving out all the dross that is related to a subject. Theoretical computer science stands to gain from this mindset.

Here are some of the topics that we will meet in these pages. We start off by considering different **models of computation**. These are formal, virtual machines that perform computations. We classify them into three different groupings. There are models, such as **Turing machines**, that perform symbol manipulations. Historically, this is one of the first computational models. There are models called **register machines** that perform computations through manipulating whole numbers. A hand-held calculator is a type of computer that manipulates numbers. And finally there are models that perform bit manipulations called **circuits**. Every modern electronic computer works by manipulating bits. Yet another way of describing computation is with **logical formulas**. For each of these types of computational models, there is a symmetric monoidal category that contains the models. There are functors from the categories of models to the categories of functions. There also exist functors between the categories of models. For example, your hand-held calculator appears to manipulate whole numbers. In fact, what is really happening is that there is a circuit which manipulates bits that performs the calculations. This is the essence behind a functor from the category of register machines to the category of circuits.

computability theory and complexity theory study the relationship between the syntax and the semantics of functions.

Let us look at theoretical computer science from a broadly philosophical perspective. By the "syntax" of a function we mean a description of the function such as a program that implements the function, a computer that runs the function, a logical formula that characterizes the function, a circuit that executes the function, etc. By the "semantics" of a function we mean the rule that assigns to every input an output. Computability

theory and complexity theory study the relationship between the syntax and the semantics of functions. There are many aspects to this relationship. Computability theory asks what functions are defined by syntax, and — more interestingly — what functions are *not* defined by syntax. Complexity theory asks how can we classify and characterize functions by examining their syntax.

In a categorical setting, the relationship between the syntax and semantics of functions is described by a functor from a category of syntax to a category of semantics. The functor takes a description of a function to the function it describes. Computability theory then asks what is in the image of this functor and — more interestingly — what is *not* in the image of the functor. Complexity theory tries to classify and characterize what is in the image of the functor by examining the preimage of the functor.

Let us look at some of the other topics we meet on these pages. Diagonal arguments are related to self-reference. About two and a half thousand years ago, Epimenides (a cantankerous Cretan philosopher who proclaimed Cretans always lie) taught us that language can talk about itself and is self-referential. He showed that because of this self-reference there is a certain limitation of language (some sentences are true if and only if they are false.) In the late nineteenth century, the German mathematician Georg Cantor showed that sets can be self referential and Bertrand Russell showed that one must restrict this ability or set theory will be inconsistent. In the 1930s, Kurt Gödel showed that mathematical statements can refer to themselves ("I am unprovable") and hence there is a limitation to the power of proofs. We are chiefly interested in Alan Turing showing that computers can reference themselves (after all, an operating system is a program dealing with programs). This entails a limitation of computers. Diagonal arguments are used in systems with self reference to find some limitation of the system. We describe a simple categorical theorem that models all these different examples of self-reference. Many results of theoretical computer science are shown to be instances of this theorem.

Formal language theory deals with the interplay between machines and languages. The more complicated a machine, the more complicated is the language it understands. With categories, we actually describe a functor from a category of machines to a category of languages. Machines that are weaker than Turing machines and their associated languages are explored.

Modern cryptography is about using computers to encrypt messages.

While it should be computationaly easy to encode these messages, only the intended receiver of the message should be able to decode the message with ease. Categorically, we will use some of the ideas we learned in complexity theory to discuss a subcategory of computable functions containing easily computable functions. In contradistinction, the other computable functions can be thought of as hard. A good cryptographic protocol is when the encoding and decoding are easy to compute and where the decoder is hard to find. The power of category theory will become apparent when we make a simple categorical definition and show that many modern cryptographic protocols can be seen an instance of this definition.

Kolmogorov complexity theory is a discussion about the complexity of strings. We use the Turing machines that we discussed earlier to classify how complicated strings of characters are. The section ends with a discussion of the notion of randomness.

Algorithms are at the core of computer science. From our broad prospective, algorithms are on the thin line between the syntax and semantics of computable functions. The functor from syntax to semantics factors as

$$Syntax \longrightarrow Algorithms \longrightarrow Semantics. \tag{1.5}$$

Or we can see it as,

$$Programs \longrightarrow Algorithms \longrightarrow Functions. \tag{1.6}$$

This view affords us a deeper understanding of the relationship between programming, computer science, and mathematics.

The last section of Chapter 7 is a continuation of this Introduction. It summarizes what was done in this text using technical language. We show that there are certain unifying themes and categorical structures that make theoretical computer science more understandable. We also close that section with a reader's guide on going further with their studies.

Let's roll up our sleeves and get to work!

2

Aide-Mémoire for Category Theory

He who is ignorant of foreign languages knows not his own.
Johann Wolfgang von Goethe[1]

We assume the basics of category theory. The reader is expected to know the concept of a category, functor, natural transformation, equivalence, limit, etc. There are, however, some category theory concepts employed that might not be so well known. We review these concepts here. Any other required categorical constructions will be reviewed along the way. Feel free to skip this Chapter and return as needed.

2.1 Slice Categories and Comma Categories

The slice and coslice constructions make the morphisms of one category into the objects of another category.

Definition 2.1.1. Given a category $\mathbb{A}$ and an object a of $\mathbb{A}$, the **slice category**, $\mathbb{A}/a$, is a category whose objects are pairs $(b, f : b \longrightarrow a)$ where b is an object of $\mathbb{A}$ and f is a morphism of $\mathbb{A}$ whose codomain is a. A morphism of $\mathbb{A}/a$ from $(b, f : b \longrightarrow a)$ to $(b', f' : b' \longrightarrow a)$ is a morphism $g : b \longrightarrow b'$ of $\mathbb{A}$ that makes the following triangle commute

$$\begin{array}{ccc} b & \xrightarrow{g} & b' \\ & {\scriptstyle f}\searrow \quad \swarrow{\scriptstyle f'} & \\ & a & \end{array} \tag{2.1}$$

[1] "Wer fremde Sprachen nicht kennt, weiss nichts von seiner eigenen." *Über Kunst und Altertum*, 1821.

Composition of morphisms comes from the composition in $\mathbb{A}$, and the identity morphism for the object $(b, f : b \longrightarrow a)$ is id_b. $\diamond$

Example 2.1.2. Some examples of a slice category:

- Consider the category $\mathbb{Set}$. Let $\mathscr{R}$ be the set of all real numbers. The category $\mathbb{Set}/\mathscr{R}$ is the collection of all $\mathscr{R}$-valued functions.
- Let us foreshadow by showing an example that we will see in our text. Let $\mathbb{Func}$ be the category whose objects are different types and whose morphisms are all functions between types. One of the given types is Boolean, $Bool$, which has the possible values of 0 and 1. Then $\mathbb{Func}/Bool$ is the category whose objects are functions from any type to $Bool$. Such functions are called **decision problems** and they will play a major role in the coming pages. We also have a notion of a morphism between decision problems:

$$\begin{array}{ccc} Type_1 & \xrightarrow{\quad g \quad} & Type_2 \\ & {\scriptstyle f}\searrow \quad \swarrow{\scriptstyle f'} & \\ & Bool & \end{array} \tag{2.2}$$

  where f and f' are decision problems and g is a morphism from f to f'. Such a morphism of decision problems will be called a "reduction." g is a reduction of f to f'. The idea behind a reduction is that if we know the answer to f' then we can also find the answer to f. Since the triangle commutes, to find the answer to f on an input, simply evaluate the input using g and then find the answer using f'. In other words, solving f reduces to solving f'.

$\square$

Exercise 2.1.3. Show that $id_a : a \longrightarrow a$ is the terminal object in $\mathbb{A}/a$.

Solution: For any object $f : b \longrightarrow a$ of $\mathbb{A}/a$, f is the unique morphism to the identity that makes the triangle commute.

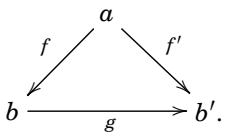
$\blacksquare$

Exercise 2.1.4. Prove that any $f : a \longrightarrow a'$ in $\mathbb{A}$ induces a functor $f_* : \mathbb{A}/a \longrightarrow \mathbb{A}/a'$ that takes any object $g : b \longrightarrow a$ of $\mathbb{A}/a$ to $f \circ g$ is in $\mathbb{A}/a'$. $\blacksquare$

The dual notion of a slice category is a coslice category:

Definition 2.1.5. Given a category $\mathbb{A}$ and an object a of $\mathbb{A}$, the **coslice category**, $a/\mathbb{A}$ is a category whose objects are pairs $(b, f : a \longrightarrow b)$ where

b is an object of $\mathbb{A}$ and f is a morphism of $\mathbb{A}$ with domain a. A morphism of $a/\mathbb{A}$ from $(b, f : a \longrightarrow b)$ to $(b', f' : a \longrightarrow b')$ is a morphism $g : b \longrightarrow b'$ of $\mathbb{A}$ that makes the following triangle commute



(2.3)

Composition of morphisms comes from the composition in $\mathbb{A}$, and the identity morphism for the object $(b, f : b \longrightarrow a)$ is id_b. ◇

Example 2.1.6. Some examples of a coslice category:

- Consider the one element set $\{*\}$. The category $\{*\}/\mathbb{S}\mathbb{e}\mathbb{t}$ has objects that are sets with a function that picks out a distinguished element of the set. An object is a pair (S, s_0) where S is a set and s_0 is a distinguished element of that set. The morphisms from (S, s_0) to (T, t_0) are set functions that preserve the distinguished element. That is, $f : S \longrightarrow T$ such that $f(s_0) = t_0$. This is the category of pointed sets.
- When we talk about finite automata, we will generalize the above example. Let $\mathbb{F}\mathrm{in}\mathbb{G}\mathrm{raph}$ be the category of finite graphs and graph homomorphisms. Let $\mathbf{2}_0$ be the graph with just two vertices (named s and t for "source" and "target.") Then $\mathbf{2}_0/\mathbb{F}\mathrm{in}\mathbb{G}\mathrm{raph}$ is the category of finite graphs with a distinguished vertex for s and a distinguished vertex for t. (They might be the same vertex.)
- Let $\mathbb{P}\mathrm{rop}$ be the category of propositions with at most one morphism from a to b if a entails b. Let $p \in \mathbb{P}\mathrm{rop}$ be a particular proposition. Then $p/\mathbb{P}\mathrm{rop}$ is the category of all propositions that are implied by p.

□

Exercise 2.1.7. Show that $id_a : a \longrightarrow a$ is the initial object in $a/\mathbb{A}$.

Solution: This is similar to the solution to Exercise 2.1.3.

■

Definition 2.1.8. Given functors $F : \mathbb{A} \longrightarrow \mathbb{C}$ and $G : \mathbb{B} \longrightarrow \mathbb{C}$, one can form the **comma category** (F, G), sometimes written $F \downarrow G$. The objects of this category are triples (a, f, b) where a is an object of $\mathbb{A}$, b is an object of $\mathbb{B}$ and, $f : F(a) \longrightarrow G(b)$ is a morphism in $\mathbb{C}$. A morphism from (a, f, b) to (a', f', b') in (F, G) consists of a pair of morphisms (g, h) where $g : a \longrightarrow a'$ is in $\mathbb{A}$ and $h : b \longrightarrow b'$ is in $\mathbb{B}$ such that the following diagram

commutes:

$$\begin{array}{ccc} F(a) & \xrightarrow{f} & G(b) \\ {\scriptstyle F(g)}\downarrow & & \downarrow{\scriptstyle G(h)} \\ F(a') & \xrightarrow[f']{} & G(b') \end{array} \quad . \tag{2.4}$$

Composition of morphisms comes from the fact that one commuting square on top of another ensures that the whole diagram commutes. The identity morphisms are obvious. ◇

Example 2.1.9. Some examples of comma categories are familiar already.

- If $\mathbb{B} = \mathbf{1}$ and $G\colon \mathbf{1} \longrightarrow \mathbb{C}$ picks out the object c_0 and $\mathbb{A} = \mathbb{C}$ with $F = Id_{\mathbb{C}}$, then the comma category (F,G) is the slice category $\mathbb{C}/c_0$.
- If $\mathbb{A} = \mathbf{1}$ and $F\colon \mathbf{1} \longrightarrow \mathbb{C}$ picks out the object c_0 and $\mathbb{B} = \mathbb{C}$ with $G = Id_{\mathbb{C}}$, then the comma category (F,G) is the coslice category $c_0/\mathbb{C}$.
- In this text, we will use a comma category construction with F being an inclusion functor. In this case the comma category will have objects of a certain type and morphisms from the subcategory. This will be important for reductions of one problem to another.

□

There are forgetful functors $U_1\colon (F,G) \longrightarrow \mathbb{A}$ and $U_2\colon (F,G) \longrightarrow \mathbb{B}$ which are defined on objects as follows: U_1 takes (a,f,b) to a and U_2 takes (a,f,b) to b.

Exercise 2.1.10. Show that if the following two triangles of categories and functors commute

$$\begin{array}{ccccc} & & \mathbb{C}' & & \\ & {\scriptstyle F'}\nearrow & \uparrow & \nwarrow{\scriptstyle G'} & \\ \mathbb{A} & & & & \mathbb{B} \\ & {\scriptstyle F}\searrow & \downarrow & \swarrow{\scriptstyle G} & \\ & & \mathbb{C}, & & \end{array} \tag{2.5}$$

where $\mathbb{C} \hookrightarrow C'$ is an inclusion functor, then $(F,G) \hookrightarrow (F',G')$. ■

2.2 Symmetric Monoidal Categories

Many of our categories are collections of processes, where besides sequentially composing processes, we can compose processes in parallel. Such categories are formulated as categories with extra structure.

Definition 2.2.1. A **strictly associative monoidal category** or a **strict monoidal category** is a triple $(\mathbb{A}, I, \otimes)$ where $\mathbb{A}$ is a category, I is an object in $\mathbb{A}$ called a "unit," and $\otimes$ is a bifunctor $\otimes\colon \mathbb{A} \times \mathbb{A} \longrightarrow \mathbb{A}$ called a "tensor" which satisfies the following axioms:

- $\otimes$ is strictly associative, i.e., for all objects a, b and c in $\mathbb{A}$, $(a \otimes b) \otimes c = a \otimes (b \otimes c)$, and
- I acts like a unit, i.e., for all objects a, $a \otimes I = a = I \otimes a$.

◇

Example 2.2.2. Any monoid (including $\mathcal{N}$, the natural numbers) can be thought of as a strictly associative monoidal category. The elements of the monoid form a discrete category and the unit of the monoid becomes the unit of the symmetric monoidal category. The multiplication in the monoid becomes the tensor of the category. □

In the literature, there is a weaker definition of a monoidal category. This is a category where there is an isomorphism between$(a \otimes b) \otimes c$ and $a \otimes (b \otimes c)$. This isomorphism must satisfy a higher dimensional axiom called a "coherence condition." While this concept is very important in many areas of mathematics, physics, and computer science, it will not arise in our presentation. There is a theorem of Saunders Mac Lane that says that every monoidal category is equivalent (in a very strong way) to a strict monoidal category. The categories that we will be dealing with will have the natural numbers or sequences of types as objects and will be strict monoidal categories.

Before we go on to the structure, we need the following functor. For any category $\mathbb{A}$, there is a functor

$$tw\colon A \times \mathbb{A} \longrightarrow \mathbb{A} \times \mathbb{A} \tag{2.6}$$

that is defined for two elements a and b as $tw(a,b) = (b,a)$. tw is defined similarly for morphisms.

Definition 2.2.3. A **symmetric monoidal category** is a quadruple $(\mathbb{A}, I, \otimes, \gamma)$ where $(\mathbb{A}, I, \otimes)$ is a monoidal category and γ is a natural transformation that is an isomorphism $\gamma\colon \otimes \longrightarrow (\otimes \circ tw)$, that is, for all objects

a, b in $\mathbb{A}$, there is an isomorphism

$$a \otimes b \xrightarrow{\gamma_{a,b}} b \otimes a \tag{2.7}$$

which is natural in a and b. This γ is called a "braiding" and must satisfy the following axioms:

- the symmetry axiom,

$$\gamma_{b,a} \circ \gamma_{a,b} = Id_{a \otimes b} \tag{2.8}$$

- the braiding axiom,

$$\begin{array}{ccc} a \otimes b \otimes c & \xrightarrow{\gamma_{a,b \otimes c}} & b \otimes c \otimes a \\ & {\scriptstyle \gamma_{a,b} \otimes Id_c} \searrow \quad \nearrow {\scriptstyle Id_b \otimes \gamma_{a,c}} & \\ & b \otimes a \otimes c & \end{array} \tag{2.9}$$

◇

Of course we are not only interested in such symmetric monoidal categories, but in the way they relate to each other.

Definition 2.2.4. A **strong symmetric monoidal functor** from $(\mathbb{A}, I, \otimes, \gamma)$ to $(\mathbb{A}', I', \otimes', \gamma')$ is a triple (F, ι, ∇) where $F \colon \mathbb{A} \longrightarrow \mathbb{A}'$ is a functor, $\iota \colon I' \longrightarrow F(I)$ is an isomorphism in $\mathbb{A}'$, and

$$\nabla \colon (\otimes' \circ (F \times F)) \longrightarrow (F \circ \otimes) \tag{2.10}$$

is a natural transformation which is an isomorphism, that is, for every a, a' in $\mathbb{A}$, there is an isomorphism

$$\nabla_{a,a'} \colon F(a) \otimes' F(a') \longrightarrow F(a \otimes a') \tag{2.11}$$

natural in a and a'. These natural transformations must satisfy the following coherence rules.

- ∇ must cohere with the braidings

$$\begin{array}{ccc} F(a) \otimes' F(a') & \xrightarrow{\nabla_{a,a'}} & F(a \otimes a') \\ {\scriptstyle \gamma'_{F(a),F(a')}} \downarrow & & \downarrow {\scriptstyle F(\gamma_{a,a'})} \\ F(a') \otimes' F(a) & \xrightarrow[\nabla_{a',a}]{} & F(a' \otimes a) \end{array} \tag{2.12}$$

- ι must cohere with ∇

$$\begin{array}{ccc} I' \otimes' F(a) & \xrightarrow{\iota \otimes Id} & F(I) \otimes' F(a) \\ \| & & \downarrow{\scriptstyle \nabla_{I,a}} \\ F(a) & = & F(I \otimes a). \end{array} \tag{2.13}$$

(There is a similar coherence requirement with ι and $\nabla_{a,I}$.)

- ∇ must cohere with itself, i.e., it is associative

$$\begin{array}{ccc} F(a) \otimes' F(a') \otimes' F(a'') & \xrightarrow{Id \otimes' \nabla_{a',a''}} & F(a) \otimes' F(a' \otimes a'') \\ \downarrow{\scriptstyle \nabla_{a,a'} \otimes' Id} & & \downarrow{\scriptstyle \nabla_{a,a' \otimes a''}} \\ F(a \otimes a') \otimes' F(a'') & \xrightarrow[\nabla_{a \otimes a',a''}]{} & F(a \otimes a' \otimes a''). \end{array} \tag{2.14}$$

◇

We will also need a stronger notion of a functor.

Definition 2.2.5. A **symmetric monoidal strict functor** from $(\mathbb{A}, I, \otimes, \gamma)$ to $(\mathbb{A}', I', \otimes', \gamma')$ is a functor $F\colon \mathbb{A} \longrightarrow \mathbb{A}'$ where $F(I) = I'$ and $F(a) \otimes' F(a') = F(a \otimes a')$, i.e., the ι and ∇ are the identity. ◇

Definition 2.2.6. A **symmetric monoidal natural transformation** from (F, ι, ∇) to (F', ι', ∇') is a natural transformation $\mu\colon F \longrightarrow F'$, i.e., for every a in $\mathbb{A}$, a natural morphism $\mu_a : F(a) \longrightarrow F(a')$ which must satisfy the following coherence axioms

$$\begin{array}{ccc} & I' & \\ {\scriptstyle \iota}\swarrow & & \searrow{\scriptstyle \iota'} \\ F(I) & \xrightarrow[\mu_I]{} & F'(I) \end{array} \qquad \begin{array}{ccc} F(a) \otimes' F(a') & \xrightarrow{\mu_a \otimes' \mu_{a'}} & F'(a) \otimes' F'(a') \\ \downarrow{\scriptstyle \nabla_{a,a'}} & & \downarrow{\scriptstyle \nabla'_{a,a'}} \\ F(a \otimes a') & \xrightarrow[\mu_{a \otimes a'}]{} & F'(a \otimes a'). \end{array} \tag{2.15}$$

Some of these axioms will easily be satisfied because we are dealing with strict functors. ◇

Given the notion of a symmetric monoidal natural transformation, we can easily describe what it means for two symmetric monoidal categories to be equivalent.

Definition 2.2.7. A **symmetric monoidal equivalence** between $(\mathbb{A}, I, \otimes, \gamma)$ and $(\mathbb{A}', I', \otimes', \gamma')$ means there is a symmetric monoidal functor (F, ι, ∇) from $\mathbb{A}$ to $\mathbb{A}'$ and a symmetric monoidal functor (F', ι', ∇') from $\mathbb{A}'$

to $\mathbb{A}$ with symmetric monoidal natural transformations which is an isomorphism $\mu\colon Id_{\mathbb{A}} \longrightarrow F' \circ F$ and $\mu'\colon F \circ F' \longrightarrow Id_{\mathbb{A}'}$. As usual, an equivalence in this case means that the functor is full, faithful, and essentially surjective. ◇

Further Reading

The notions of a slice and comma categories can be found in Section II.6 of [51] and pages 3, 13, and 47 of [6]. Symmetric monoidal categories can be found in Chapter XI of [51] and Chapters XI-XIV of [37].

3

Models of Computation

I'm just sitting here watching the wheels go round and round
I really love to watch them roll
No longer riding on the merry-go-round
I just had to let it go.
John Lennon
Watching the Wheels, 1981

Since we are going to deal with the questions "Which functions are computable?" and "Which functions are efficiently computable?" we better first deal with the question "What is a computation?" We all have a pretty good intuition that a computation is a process that a computer performs. Computer scientists have given other, more formal, definitions of a computation. They have described different models where computations occur. These models are virtual computers that are exact and have a few simple rules. In most textbooks, one computational model is employed. As category theorists, we have to look at several models and see how they are related. This affords us a more global view of computation.

3.1 The Big Picture

We have united all the different models that we will deal with in Figure 3.1. Warning: Figure 3.1 can look quite intimidating the first time your see it. Fear not dear reader! We will spend Sections 3.1, 3.2, 3.3, and 3.4 explaining this diagram and making it digestible. We call the diagram "The Big Picture." It has a center and three spokes coming out of it. The spokes will correspond to different ways of viewing computations. Some models perform computations by manipulating strings of charac-

ters, while other models manipulate natural numbers, and still other models work like real computers and manipulate bits. Most of the rest of this text will concentrate on the top spoke which is concerned with performing computations by manipulating strings.

Here is some orientation around The Big Picture so that it is less intimidating. All the categories are symmetric monoidal categories. All the functors are symmetric monoidal functors. All the equivalences use symmetrical monoidal natural transformations. All horizontal lines are inclusion functors which are the identity on objects. Almost every category comes in two forms: (i) all the possible morphisms which include morphisms that represent partial functions, that is, functions that have inputs that do not have outputs, and (ii) the subcategory of total morphisms, i.e., morphisms where every input has an output. The diagram has a central line that consists of different types of functions that our models try to compute. This line will be the central focus of this text. There are three spokes pointing to that line. These correspond to three types of models of computation: (i) the top spoke corresponds to models that manipulate strings; (ii) the lower right spoke corresponds to models that manipulate natural numbers; and (iii) the lower left spoke corresponds to models that manipulate bits.

In all these categories, the composition corresponds to sequential processing (that is, performing one process after another). The monoidal structure corresponds to parallel processing. The symmetric monoidal structure corresponds to the fact that the order of parallel processing is easily interchanged.

We need a little discussion about types. Every function and every computational device takes inputs and returns outputs. In order to keep track of all the different functions we usually describe the input and output as types. There are **basic types** such as Nat, Int, $Real$, $Char$, $String$, and $Bool$ which correspond to natural numbers, integers, real numbers[1], characters, strings of characters, and Booleans (0 and 1.) Using these we can describe simple functions.

Example 3.1.1. Some examples of functions with basic types.

- $\lceil - \rceil : Real \longrightarrow Int$ is the ceiling function that takes a real number and outputs the least integer equal or greater than it.

[1] Finite computers with finite memory cannot deal with arbitrary large natural numbers or integers. They also do not deal with arbitrary real numbers. We must put restrictions on the size of whole numbers we use and only work with real numbers up to a certain precision. However we will not discuss these issues here.

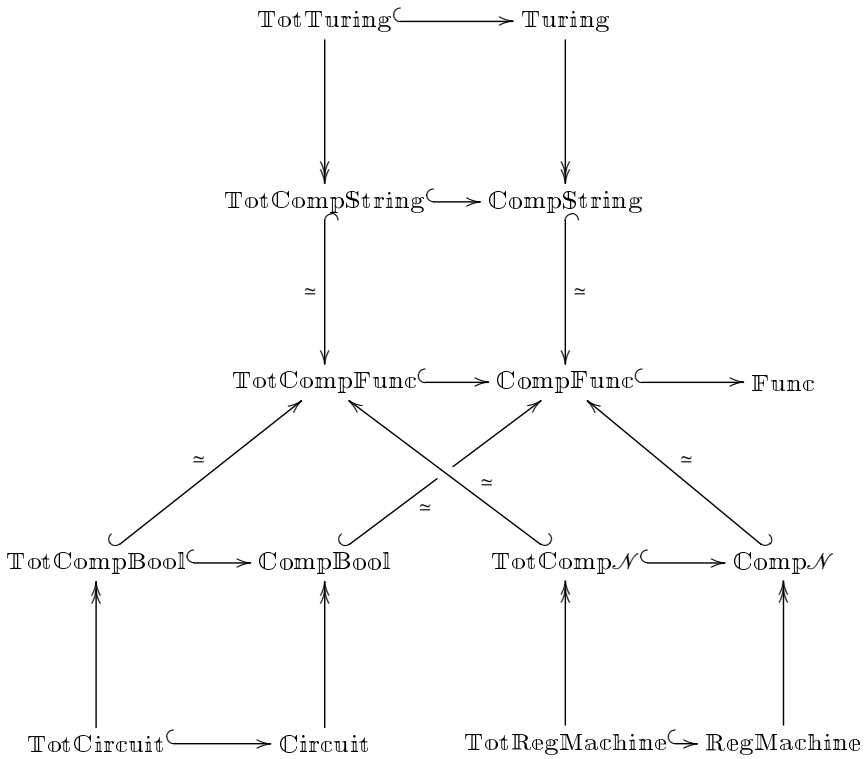


Figure 3.1 "The Big Picture" of models of computation

- $Prime\colon Nat \longrightarrow Bool$ is a function that takes a natural number and outputs a 1 if the number is a prime and 0 otherwise.
- $Beautiful\colon String \longrightarrow Nat$ that takes a string of characters to be thought of as a poem and outputs an integer from 1 to 10 depending if the poem is beautiful. This is not objective ("Beauty is in the eye of the beholder") so it is not really a function. And even if you believe that beauty is objective, it is probably not computable. You might get a computer to imitate a belief that most humans have, but that does mean the computer is judging it with a comparative aesthetic experience.

□

Exercise 3.1.2. Describe the types of the inputs and outputs of the following functions.

- A function that determines the length of a string of characters.

- A function that takes a natural number and outputs a written description of that number. For example $42 \longmapsto$ "forty-two".
- A function that takes a computer program and determines how many lines of code there are in the program.

Solution:

- $Length : String \longrightarrow Nat$.
- $Desc : Nat \longrightarrow String$.
- $Lines : String \longrightarrow Nat$.

■

Basic types are not enough. We need operations on types to form other types.

- **List types**: Given a type T, we can form T^* which corresponds to finite lists of elements of type T. $Bool^*$ is the type of lists of Booleans and Nat^* corresponds to tuples of natural numbers.
- **Function types**: Given types T_1 and T_2, we form type $Hom(T_1, T_2) = T_2^{T_1}$. This type corresponds to functions that takes inputs of type T_1 and output T_2 types.
- **Product types**: Given types T_1 and T_2, we form type $T_1 \times T_2$. This type will correspond to pairs of elements, the first of type T_1 and the second of type T_2. When there is a tuple of types, we will call them a **sequence of types**. For example $Seq = Nat \times String \times Bool$ or $Seq' = Float \times Char^{Nat} \times Nat^* \times Int$. Given two sequences of types, we can concatenate them. For example the concatenation of Seq and Seq' is

$$Seq \times Seq' = (Nat \times String \times Bool) \times (Float \times Char^{Nat} \times Nat^* \times Int) \tag{3.1}$$

$$= Nat \times String \times Bool \times Float \times Char^{Nat} \times Nat^* \times Int \tag{3.2}$$

  For a type T, we shorten $T \times T$ as T^2 and $T \times T \times T$ as T^3. In general T^n will be a sequence of n Ts. T^0 will correspond to $*$ which is called the terminal type. It is used to pick out an element of another type. A function $f : * \longrightarrow Seq$ picks out an element of type Seq.

Example 3.1.3. Here are a few functions that use compound types:

- $DayOfWeek : String \times Nat \times Nat \longrightarrow Nat$ is a function that takes a date as the name of the month, the day of the month, and the year. The function returns the day of the week (1,2,...,7) for that date.

- $Comp\colon T_3^{T_2} \times T_2^{T_1} \longrightarrow T_3^{T_1}$ is a function that corresponds to function composition. It takes an $f\colon T_1 \longrightarrow T_2$ and a $g\colon T_2 \longrightarrow T_3$ and outputs $g \circ f\colon T_1 \longrightarrow T_3$.
- $Inverse\colon T_2^{T_1} \longrightarrow T_1^{T_2}$ takes a function and returns the inverse of the function (if it exists). This arises in our discussion of cryptography.

□

Exercise 3.1.4. Describe the following as functions from sequences of types to sequences of types.

- A function that takes a list of natural numbers and returns the maximum of the numbers, the mean of the numbers, and the minimum of the numbers.
- A function that takes (i) a function from the natural numbers to the natural numbers, and (ii) a natural number. The function should evaluate the given function on the number and determine what the output is.
- A function that takes two natural numbers and determines if they are the last twin primes. That is, (i) both numbers are prime, (ii) they are of the form n and $n+2$, and (iii) and there are no greater twin primes.

Solution:

- $MaxMeanMin\colon Nat^* \longrightarrow Nat \times Real \times Nat$.
- $Eval\colon Nat^{Nat} \times Nat \longrightarrow Nat$.
- $LastTwinPrimes\colon Nat \times Nat \longrightarrow Bool$. The interesting part of this problem is that it is unknown if there is a last twin prime or if there are an infinite number of them.

■

There are much more complicated types that are needed to describe all types of functions. There are "coproduct types" and things called "dependent types." However, we will not use them in our presentation.

Advanced Topic 3.1.5. These types are the beginning of several areas of theoretical computer science. **Type theory** studies complicated types and how they relate to computation. Category theorists have been making categories of types and looking at all of their possible structures (see

[4, 7].) This is also related to λ**-calculus** or **lambda calculus** which is another way of describing computations (see [40].) The theory of types is also the beginning of a current very hot area of research called **homotopy type theory**. This has all types of connections to foundations of mathematics and proof checkers (see [62].) ○

.....

Now that we have the language of types down pat, let us move on to the most important definition in this book.

Definition 3.1.6. The category $\mathbb{F}\mathrm{unc}$ consists of all functions. The objects are sequences of types. The morphisms in $\mathbb{F}\mathrm{unc}$ from Seq to Seq' are all functions that have inputs from type Seq and outputs of type Seq'. We permit all functions including partial functions and functions that computers cannot compute. The identity functions are obvious. Composition in the category is simply function composition. For example, if $f: Seq_1 \longrightarrow Seq_2$ and $g: Seq_2 \longrightarrow Seq_3$ are two functions, then $g \circ f: Seq_1 \longrightarrow Seq_3$ is the usual composition. The monoidal structure on objects is concatenation of sequences of types. Given $f: Seq_1 \longrightarrow Seq_2$ and $g: Seq_3 \longrightarrow Seq_4$, their tensor product is

$$f \otimes g: (Seq_1 \times Seq_3) \longrightarrow (Seq_2 \times Seq_4) \tag{3.3}$$

which corresponds to performing both functions in parallel. The symmetric monoidal structure comes from the trivial function that swaps sequences of types, i.e., $tw: Seq \times Seq' \longrightarrow Seq' \times Seq$. We leave the details for the reader. ◇

Exercise 3.1.7. What is the unit of the symmetric monoidal category structure?

Solution: The terminal type $*$ because $T \times * = T = * \times T$.

■

Exercise 3.1.8. Show that the function tw satisfies the axioms making $\mathbb{F}\mathrm{unc}$ into a symmetric monoidal category. ■

Definition 3.1.9. The category $\mathbb{C}\mathrm{omp}\mathbb{F}\mathrm{unc}$ is a subcategory of $\mathbb{F}\mathrm{unc}$ which has the same objects. The morphisms of this subcategory are functions (including partial functions) that a computer can compute. A computer can compute a partial function if for any input for which there is an