Tool to explore finite categories

Pradnesh Sanderan

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

This skeleton demonstrates how to use the `infthesis` style for undergraduate dissertations in the School of Informatics. It also emphasises the page limit, and that you must not deviate from the required style. The file `skeleton.tex` generates this document and should be used as a starting point for your thesis. Replace this abstract text with a concise summary of your report.

Research Ethics Approval

Instructions: *Agree with your supervisor which statement you need to include. Then delete the statement that you are not using, and the instructions in italics.*

Either complete and include this statement:

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: ???
Date when approval was obtained: YYYY-MM-DD
[If the project required human participants, edit as appropriate, otherwise delete:]
The participants' information sheet and a consent form are included in the appendix.

Or include this statement:

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Pradnesh Sanderan*)

Acknowledgements

I would like to thank red bull and coffee for keeping me awake, my legs for supporting me, my arms for always being by my side, the gym for preventing me from killing myself and manchester united for wanting to kill myself.

Table of Contents

Chapter 1

Introduction

The preliminary material of your report should contain:

- The title page.
- An abstract page.
- Declaration of ethics and own work.
- Optionally an acknowledgements page.
- The table of contents.

As in this example `skeleton.tex`, the above material should be included between:

```
\begin{preliminary}
    ...
\end{preliminary}
```

This style file uses roman numeral page numbers for the preliminary material.

The main content of the dissertation, starting with the first chapter, starts with page 1. ***The main content must not go beyond page 40.***

The report then contains a bibliography and any appendices, which may go beyond page 40. The appendices are only for any supporting material that's important to go on record. However, you cannot assume markers of dissertations will read them.

You may not change the dissertation format (e.g., reduce the font size, change the margins, or reduce the line spacing from the default single spacing). Be careful if you copy-paste packages into your document preamble from elsewhere. Some LaTeX packages, such as `fullpage` or `savetrees`, change the margins of your document. Do not include them!

Over-length or incorrectly-formatted dissertations will not be accepted and you would have to modify your dissertation and resubmit. You cannot assume we will check your submission before the final deadline and if it requires resubmission after the deadline to conform to the page and style requirements you will be subject to the usual late penalties based on your final submission time.

1.1 Using Sections

Divide your chapters into sub-parts as appropriate.

1.2 Citations

Citations (such as [1] or [2]) can be generated using `BibTeX`. For more advanced usage, we recommend using the `natbib` package or the newer `biblatex` system.

These examples use a numerical citation style. You may use any consistent reference style that you prefer, including "(Author, Year)" citations.

Chapter 2

Background

The study of finite monoidal categories has been a fascinating area of research that has seen significant attention in recent years. This chapter aims to provide an overview of the background and related work in the field, and discussing the key concepts and ideas that underpin the study of finite monoidal categories. The chapter begins with an introduction to the basic concepts of category theory, including categories, functors, and natural transformations. From there, we delve into the definition and properties of monoidal categories, exploring their connections to algebraic structures such as monoids .The next section focuses on the special case of finite monoidal categories, which are defined by giving their multiplication table.The chapter concludes with a survey of previous work in the field, including existing algorithms and tools for checking the legality of monoidal categories. The chapter sets the stage for the subsequent chapters, which will detail the implementation and evaluation of the tool for exploring finite monoidal categories.

2.1 Categories

2.1.1 Categoreis In a nutshell

Category theory is based on the abstraction of the arrow,
ADD BETTER DESCIPTION OF FORMAL CATEGORIES HERE. ACCORDING TO SIMEN

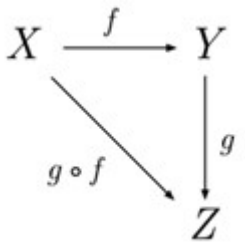
$$f: A \rightarrow B$$

In category theory, A and B are called objects and the arrow relating these 2 objects is called a morphism. In this case, the source of the morphism is object A and the target is object B. We have seen similar directional structures before as they occur widely in other mathematical concepts and topics, for example, set theory, algebra, topology and logic. For example in set theory, A and B may be sets and f could be a function with A as the domain and B as the codomain, and in logic, A and B may be propositions and f would be a proof for $A \vdash B$.

2.1.2 Categories as graphs

In category theory, a category is often described as a graph, mainly a directed graph(directed multigraph) and for this project, it would be easier to view the categories as said graphs. Based on this, we are able to give a formal definition of a category.

Definition 1: A graph is a pair N, E of classes (of nodes and edges) together with a pair of mappings s, t: E $\rightarrow$ N called source and target respectively. We write f: A $\rightarrow$ B when f is in E and s(f) = A and t(f) = B. An example of a graph is shown below:



Definition 2: A category is a graph (O, M, s,t) whose nodes O we call objects and whose edges M we call morphisms. Just like the graphs, these morphisms have sources and targets. For example in f: A $\rightarrow$ B, f would represent the morphism f, the source of the morphism is object A and the target of the morphism is object B.
Although we often represent the graphs as categories in category theory, not all directed graphs can be considered categories. A graph would need an additional structure which is the collection of objects and the collection of morphisms where each morphism would require a well-defined source and target object. Furthermore, to be considered a valid category, the directed graph would need to satisfy certain properties:

- P1: For every object,A, in the category, there must be a morphism IDA which has object A as the source and target object (it maps the category from object A to object A), is both left-associative and right-associative and obeys the unitality condition where for any morphism f, composing it with the identity morphism does nothing:

  $$f \circ IDA = f = IDA \circ f$$

- P2: For any 3 morphisms, h,g,f in the category, (h $\circ$ g) $\circ$ f = h $\circ$ (g $\circ$ f) whenever either side is defined.
- P3: For any 3 Object A, B,C, if there is a morphism f, from A to B (f: A $\rightarrow$B) and a morphism g, from B to C (g: B $\rightarrow$ C) then there must also exist a morphism h, from A to C (h: A $\rightarrow$ C)
- P4: If f : A $\rightarrow$ B, then f $\circ$ id_A = idB $\circ$ f = f.

2.1.3 Terminology

Now that we have a basic understanding of categories, before we move any further we will set some terminology that will be used throughout the dissertation to prevent any further confusion. Given a category C, the term ob(C) refers to a class whose elements are objects in C and hom(C) refers to the class of morphisms in C. we will refer to the

mappings between objects as morphisms rather than functions or arrows. We cannot always consider morphisms as functions as if we were to look at a monoid as a category, then the elements in the category would be the morphism and not a function. Each morphism will have a domain and codomain rather than source and target as this seems to suit the concept of categories more. It allows us to distinguish between categories and non-category-directed graphs which use source and target.
We will denote Objects in the category as uppercase letters and morphisms as lowercase letters. For example, as shown below, f is the morphism with object A as the domain and object B as the codomain.

$$f: A \rightarrow B$$

If we have a composition of morphisms, we would write both morphisms together and only state the domain and codomain of the composition. For example if, f: A $\rightarrow$B and g: B$\rightarrow$ C then fg: A $\rightarrow$ C. Lastly, HomC (A, B) or Hom(A, B) is the notation used to represent the set of all Morphisms in a category C that has A as their domain and B as their codomain. If the category is obvious from the context, Hom(A, B) can be used instead.

2.1.4 Morphisms

As said, above, a category also consists of a composition of morphisms. This means that 2 or more morphisms can be joined to relate 3 or more objects. The relations between morphisms are depicted using commutative diagrams which we will look into later to represent the coherence conditions for the monoidal categories ().
As mentioned, morphisms are used to represent the relations between objects in a category. Given a morphism F: A $\rightarrow$B, it can have some of these properties:

- Monomorphism: a monomorphism is a type of morphism between two objects that behaves like an injective function in set theory (it is left cancellative) whenever two morphisms composed with the monomorphism result in the same morphism, those two morphisms themselves must have been the same.

  If f $\circ$ g = f $\circ$ h, then g = g for all morphisms g,h: X $\rightarrow$ A.

- Epimorphism: an epimorphism is a type of morphism between two objects that behaves like a surjective function in set theory. Similar to monomorphism but is instead right cancellative.

  If g $\circ$ f = h $\circ$ f, then g = h for all morphisms g, h: B $\rightarrow$ X.

- Bimorphism: a morphism that has the properties of a monomorphism and an endomorphism is a bimorphism.

- Isomorphism: an isomorphism is a type of morphism between two objects that behaves like a bijective function in set theory. If f is an isomorphism then there exists a morphism g (g: B-¿ A) that is the inverse of f.

  f $\circ$ g = idA(identity morphism on A) and g $\circ$ f = idB (identity morphism on B)

- Endomorphism: An endomorphism that has an Object X as the domain and Codomain. Then f: X $\rightarrow$ X. the identity morphism is a type of endomorphism. The significance of endomorphisms in category theory lies in their ability to define fundamental algebraic structures like monoids, groups, and rings. Moreover, the set of all endomorphisms of an object X in category C forms a monoid under the composition of morphisms, which is referred to as the endomorphism monoid of X. The study of endomorphism monoids provides insights into the internal structure of the category and can help classify objects based on their endomorphism monoids.

2.1.5 Functors

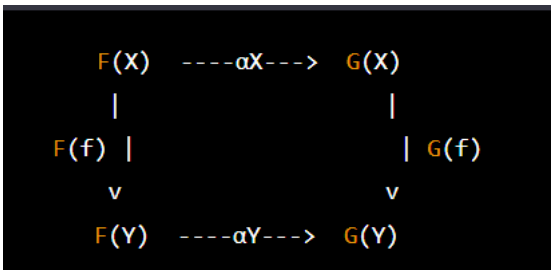
A functor is a mapping between categories that preserves the structure of the categories. Given a Functor F in Category C, where F: C $\rightarrow$ D, it assigns to each object X in C, an object F(X) in category D. It also assigns to each morphism g:X $\rightarrow$ Y in C,
F(g): F(X) $\rightarrow$ F(Y) in such a way that :

- The functor preserves the composition: for any composable pair of morphisms f:X $\rightarrow$ Y and g: Y $\rightarrow$ Z in C, we have F(g $\circ$ f) = F(g) $\circ$ F(f) in D.
- F preserves the identity morphisms. For any object X in C, we have F(idX) = idF(X)

The concept of functors plays a big role in category theory as they provide a way of comparing and relating different categories, and they allow us to import techniques and concepts from one category to another. This concept will be revisited later in the chapter on bifunctors.

2.1.6 Natural transformations

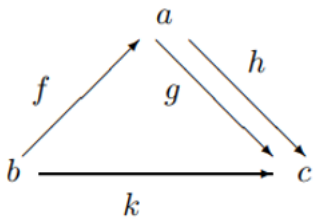
In category theory, natural transformation is a relation between two functors. It is a way of transforming one functor into another while respecting the structure of the categories involved. Supposed we have two functors F and G between the categories C and D. the natural transformation a where a: F $\rightarrow$ G is a family of morphisms
aX: F(X) $\rightarrow$ G(X), one for each object X in C such that for every morphism f, in C where f : X $\rightarrow$ Y, the following diagram commutes.



Natural transformations allow us to compare and relate different functors and it provides a way of measuring how similar or different two functors are.

2.1.7 Finite vs infinite categories

A category is considered to be finite if it has a finite number of objects and morphisms. To be more precise, a category is finite if there exists a finite set of objects Ob(C) and a finite set of morphisms Hom(C) such that the composition of any two morphisms in Hom(C) is again a morphism in Hom(C). An example of a finite category is given below.



Here a,b and c are objects and f, g and h are morphisms with domains and codomains as depicted and the existence of an identity morphism for each object is assumed.
For this project, we will be only working with finite categories as the category would need to be finite to be able to generate a valid multiplication table. Otherwise, we would be left with an infinite multiplication table.

2.1.8 Strict vs Free categories

In category theory, a category can either be strict or free. A strict category is a category in which the composition of morphisms satisfies the associative and unitality properties stated above in (Categories as a graph). A free category on the other hand Is a category built from a set of generators (the elements of a given set S, which is usually referred to as the set of generating objects or generating morphisms) and relations, subject to the requirements of composition and identity. For example, the free category F(S, R) with a set S of generators and a set R of relations, has one object whose morphisms are formed by concatenating generators from S and the inverses of generators from S, subject to their relations in R. (WHY WE ARE ONLY USING STRICT CATEGORIES HERE. The strictification theorem and the strictification process)

2.2 Monoidal Categories

Since the project revolves around monoidal categories, finite strict monoidal categories to be precise, in this chapter, we will be looking into the formal definition of a monoidal category, its properties and how it can be represented as a multiplication table.

2.2.1 Monoids

A monoid is a set with a binary operation and an identity element that satisfies certain axioms. The binary operation needs to be associative (the order of applying the operation does not matter) and the identity element, which is an element, when combined with any other element using the binary operation must return the other element unchanged. A more formal definition would be that a monoid is a triple (M,op,e) where M is a

set, op is the binary operation and e is the identity element, such that it satisfies these properties:

- Associativity: For all a,b and c in M, (a op b) op c = a op (b op c)
- Identity: there exists an element e such that for all a in M e op a = a = a op e.

In Haskell, a monoid can be expressed as a type class as follows:

```
class Monoid m where
    mappend :: m -> m -> m
    mempty  :: m
```

```
mappend :: m -> (m -> m)
```

in Haskell, the definition of mappend is curried. It can be interpreted as the mapping of every element of m to a function as shown above. The definition of a monoid as a single-object category mentioned in (), where the elements of the monoid are represented by endomorphisms (m $\rightarrow$ m), arises from this interpretation.

2.2.2 Formal Definition

A Monoidal Category is a Category C, equipped with a monoidal structure. The monoidal structure consists of a bifunctor called the tensor product, an object I called the monoidal unit and three natural isomorphisms. A deeper explanation of the monoidal structure and its properties is explained in the next few subsections

2.2.3 Tensor Product

A bifunctor is a functor whose domain is t a product category. We refer to them as the tensor product in monoidal categories and in monoidal categories, they take 2 arguments each from a different category and produce a result as shown :

$$\otimes: C * D \rightarrow E$$

Specifically, C and D are categories and E is a monoidal category. We use the infix notation so that the value at (A,B) is written as A $\otimes$ B.

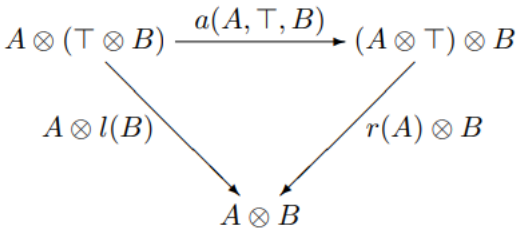
2.2.4 Natural Isomorphisms

A monoidal category must have 3 natural isomorphisms a,r and l for each object A, B, and C, which are subject to certain coherence conditions. These conditions ensure that the tensor operation:

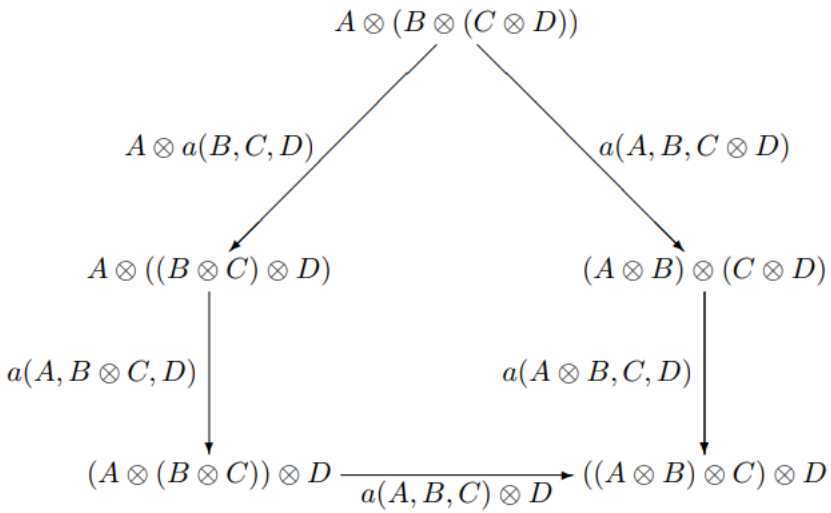
- Is Associative. There exists a natural isomorphism a, called the associator whereby a(A, B, C): A $\otimes$ (B $\otimes$ C) $\rightarrow$ (A $\otimes$ B) $\otimes$ C;
- Has I as left and right identity: there are natural isomorphisms r and l which are right and left unitors respectively and satisfy:
    1. rA : A $\otimes$ I $\rightarrow$ A

2. lA : I $\otimes$ A $\rightarrow$ A

If the monoidal category had such natural isomorphisms and they satisfy the said conditions, then they must make the following diagrams commute. CHANGE T TO I !!!!!!!!!!



The triangle diagram depicts the concept of the unit law, which declares the existence of certain objects (represented by I) that serve as identity elements for the $\otimes$ operation. It consists of three vertices, each of which is labelled with an object of the category, and three edges labelled with $\otimes$ operations. One of the vertices is marked with the distinctive object I, and the other two vertices are linked to it by edges labelled with $\otimes$ operations. The diagram affirms that if we begin at any vertex and follow the edges towards the vertex marked with I, we obtain the same result.



The pentagram diagram illustrates the principle of associativity, which asserts that the order in which we group three elements using the $\otimes$ operation is immaterial. It comprises of five vertices, each of which denotes an object of the category, and ten edges labelled with $\otimes$ operations. These edges combine to form a pentagon, where each vertex is connected to its adjacent vertices by edges labelled with $\otimes$ operations. The diagram affirms that if we traverse the edges around the pentagon in either direction starting from any vertex, we obtain the same outcome. These diagrams provide a way

to visualize and reason about the coherence conditions that must hold in a monoidal category.

Chapter 3

Design

The design chapter of this dissertation aims to outline the process and decisions made in the development of this project. This chapter will detail the various design considerations, such as the choice of programming language, style of programming, and algorithms used in the implementation of the tool. The chapter will begin with an overview of the project requirements and the challenges and limitations that were encountered during the design process. It will then delve into the specific design choices made and the reasons behind them. Overall, the design chapter aims to provide a comprehensive account of the design process, from initial concept to final implementation, and to justify the choices that were made at each stage of development.

3.1 Requirements

The requirements section of this dissertation outlines the key features and functionality that are essential for the successful implementation of the tool. In this section, we will discuss the technical requirements, and design requirements that must be taken into account when developing the tool.

3.2 Design Choices

In this section, decisions made before the start of implementation are explained

3.2.1 Programming Language

When starting the project, I contemplated using Java, Python and Haskell as the main programming language for the background. I had chosen these languages because of the previous work done in these languages that are related to this project. In the end, I chose Java because of certain reasons. Firstly, Java is an object-oriented programming language (OOP) which is more well-suited for building complex programs. A more in-depth reasoning for choosing OOP-style programming is given in the next chapter. Another reason for choosing Java was because the project could then be compiled and run on a Java Virtual Machine(JVM). Although it was not used, the usage of the

JVM was considered an important factor because of its cross-platform capabilities, its excellent memory management through garbage collection to reduce the risk of memory-related errors and its usage of a just-in-time compiler which would optimize the performance of the program at runtime. Lastly, the large developer community and extensive libraries were a big factor in choosing it. The various Graphical user interface libraries such as Java Swing were a big factor in choosing it as it would make building the frontend part of the project simpler. Furthermore, the large community would mean that for any problem I run into, there would probably be a blog post or discussion on it already. I did not choose python because, it is an interpreted language, which means that it may be slower than a compiled language like Java for performing complex computations. Additionally, Python's dynamic typing may make it harder to catch errors in the code. Although functional programming languages such as Haskell have strong support for algebraic data types and pattern matching, which can be useful for representing the structure of monoidal categories, I am not very familiar with the language and it seemed to be a waste of time, trying to learn a new language when I am already fairly proficient in one. JavaScript and C++ were also considered for their performance and popularity but were rejected for the same reasons as Haskell.

3.2.2 Programming Style

Functional programming had a couple of advantages for this project that we considered at the start when designing this program. The advantages mainly were that there already exists a partial tool, written in Haskell that we could build on, that developing the program may be simpler if done in a functional programming language as most books and research papers on category theories and monoidal categories had explained them in terms of Haskell code. Finally, it was that functional programming languages have strong support for algebraic data types and pattern matching which would make it easier to represent the monoidal categories. However, we decided to choose OOP because of a variety of reasons. The first reason is that it provides us with a way to organize and encapsulate the data and functionality associated with monoidal categories. It seemed logical to use OOP to create objects to represent the elements of the monoidal categories (multiplication table, morphisms, tensor products) and to define their behaviour through methods. Encapsulating these elements in objects, it provides us with a way to organize and manage the complexity of the monoidal categories in a modular fashion, making it easier to maintain and modify. Another reason is that OOP allows us to reuse code and implementations through inheritance. This would become useful in cases when more than 1 category or morphism is involved. We chose not to use functional programming because firstly it is immutable. Data structures created with functional programming cannot be modified later which makes it less suitable for usage with monoidal categories which often require elements to be modified. Next, Functional programming languages tend to favour pure functions and immutable data structures, which can sometimes result in slower performance compared to languages like Java, which are designed for performance-critical applications. Lastly, the type system is more complex and would require a significant amount of time and effort to learn. It seemed more logical to use Java and channel that time and effort into improving the program

3.2.3 Usage of Previous Work

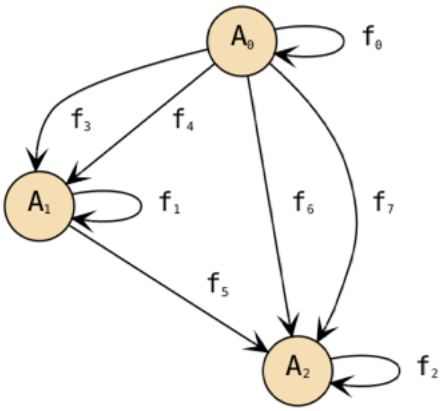
I chose not to use the previous work done (DiscoPy and the partial tool done in Haskell) mainly because I preferred building the tool from scratch as I would then be more familiar with it and easier to implement. I had chosen not to use DiscoPy toolbox because firstly, it only works with free monoidal categories but we would be working with strict ones. Next, the current toolbox does not have any relation to multiplication tables or validating categories hence I would have to implement a significant majority of the code myself. I chose not to build upon the current tool that was built in Haskell with a SAT solver firstly because I am not familiar with Haskell or functional programming as mentioned above. Next, as mentioned above, I believed that an OOP approach would be most suitable for this project rather than a functional programming approach.

3.3 First Implementation

Since it was our first time working with categories and it was also the first time categories were being represented as multiplication tables, we found it easier to build a program that would take a category as input and then check if it is a valid category. If it is then it would output the various multiplication tables that could be formed from this one valid category. We found that doing this reverse-engineering approach first would help build our understanding of the multiplication table and how categories can be checked to see if they are valid or not. A more detailed explanation of this implementation is given in section ().

3.3.1 Input CSV

The program would take a CSV file as input and this file would have all the information required to represent the category. The first line in the file would be the name of all the objects in the category and the following lines would be the names of each morphism and their respective domain and codomain objects. For example for the following category, the CSV file is listed below it.




```
A0,A1,A2
f0,A0,A0
f1,A1,A1
f2,A2,A2
f3,A0,A1
f4,A0,A1
f5,A1,A2
f6,A0,A2
f7,A0,A2
```

3.3.2 File Reader

The file reader was designed to read the CSV file, and save each object as a State type and each morphism as a Morphisms type. It would also check each morphism and determine if it is an identity morphism or not. The pseudocode for the file reader can be found below: ADD PSEUDOCODE HERE!!

3.3.3 Validating the Category

3.3.3.1 Checking identity morphisms

The method used to check the identity morphisms was designed such that it would check if the identity morphisms identified by the file reader possess the properties of an identity morphism in a valid category which is that it has the same object as its domain and codomain, and it satisfies the unitality condition. If a morphism does not satisfy these properties, we do not throw an error. Instead, it just removes the morphism from the list of identity morphisms.

3.3.3.2 Checking composition

The program was designed to fill up the multiplication table with morphisms while checking its composition. As the method traverses the multiplication table, it fills it up with the appropriate morphism or fills it up with "-" if the composition does not exist. While filling up the table, the method checks If the domain of the first morphism, the codomain of the first morphism and the codomain of the second morphism is different (the codomain of the first morphism and the domain of the second morphism must be the same for there to be a composition). If they are different then it means that there are 3 different objects involved and the following property must hold:

$$\text{If } f\text{: } A \to B \text{ and } g\text{: } B \to C \text{ then there must be } h\text{: } A \to C$$

The method then checks every morphism until it finds a morphism with object A as the domain and object C as the codomain. If it does not find such a morphism. It prints out "this is not a valid category" and the program ends as the category does not satisfy the composition of morphisms property.

3.3.3.3 Checking associativity

If the category passes the composition property check then, it checks the associativity property. For this check, the method takes 3 morphisms and checks if the composition exists. If it does then the following property must hold:

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

For any morphism, if the property does not hold then it returns that it is not a valid category.

3.3.3.4 Outputting the various tables

While filling up the table, the method keeps track of all the entries in the table that could be replaced by a different morphism (this happens when there is more than 1 morphism with the same domain and codomain). The method printTable then prints the original table the previous method had filled and also different variations of the table with different morphisms where applicable. Hence it outputs every possible multiplication table that the category could produce.

3.4 Second Implementation

In the second implementation, we changed the methods to better suit the project whereby in this implementation, the program would read a multiplication table, rebuild the table in the program, create the appropriate morphisms and objects and then check the properties of the category followed by checking the monoidal properties of the category.

3.4.1 Input CSV

The program was designed to take 2 CSV files; one to represent the multiplication table of the category and another to represent the multiplication table of the tensor product. The CSV file is structured such that it reflects a multiplication table with columns and rows. An example of a category and tensor product multiplication table is given below along with a screenshot of the CSV file.

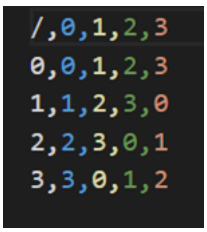
Composition table:

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

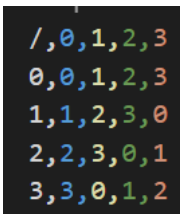
Tensor table:

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Category CSV:



Tensor Product CSV



3.4.2 File Reader

Unlike in the first implementation where the file reader would read the files and format the morphisms and objects, this implementation was designed to just read the CSV files. The file reader was designed to read the CSV files and then store each line in the readlines variable. It would also store the row titles and column titles for the table. The same file reader would be used to read the category CSV file and for reading the tensor product CSV File. The only difference would be that for the tensor product, each line would be stored in the readTensor list. Once the lines were read, a different method would format the morphisms and objects and recreate the category table and tensor product table in the program.

3.4.3 Validating the category

Using the knowledge gained from the first implementation, designing the methods used to validate the categories was simple as it was the same properties that needed to be checked as before; the presence of valid identity morphisms, the composition of morphisms and associativity of morphisms. The methods were designed to make use of the reconstructed category multiplication table created which would be more efficient than cycling through each morphism as what was done in the previous implementation. The pseudocode for each check can be found below.: ADD PSEUDOCODE HERE

3.4.4 Validating the Tensor Product

To check if the category is a valid monoidal category, there are 6 properties that the category must satisfy. For that, we created 6 different methods that would each check one of those 6 properties. The pseudocodes for the methods designed are shown below: ADD PSEUDOCODE HERE

Chapter 4

Implementation

4.1 Program Overview

4.2 Multiplication Tables

- how is it formed
- how we get information about a category from it
- reference the code used

4.3 Code Architecture

4.4 Validating the Category

- reading the files
- check format identites
- format create morphisms
- validating the category

4.5 Validating the Monoidal Properties

- read and validate tensor
- run checks on all properties
- if pass, return pass, else show where failed

4.6 Testing

-unit tests
- code coverage
- integrated testing
- overall testing

- speed
- accuracy

Chapter 5

Results and Evaluation

Chapter 6

Conclusions

6.1 Final Reminder

The body of your dissertation, before the references and any appendices, *must* finish by page 40. The introduction, after preliminary material, should have started on page 1.

You may not change the dissertation format (e.g., reduce the font size, change the margins, or reduce the line spacing from the default single spacing). Be careful if you copy-paste packages into your document preamble from elsewhere. Some LaTeX packages, such as `fullpage` or `savetrees`, change the margins of your document. Do not include them!

Over-length or incorrectly-formatted dissertations will not be accepted and you would have to modify your dissertation and resubmit. You cannot assume we will check your submission before the final deadline and if it requires resubmission after the deadline to conform to the page and style requirements you will be subject to the usual late penalties based on your final submission time.

Bibliography

[1] Hiroki Arimura. Learning acyclic first-order horn sentences from entailment. In *Proc. of the 8th Intl. Conf. on Algorithmic Learning Theory, ALT '97*, pages 432–445, 1997.

[2] Chen-Chung Chang and H. Jerome Keisler. *Model Theory*. North-Holland, third edition, 1990.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration. This information is often a copy of a consent form.