Why did I choose Java:

* Object-Oriented Programming: Java is an object-oriented programming language, which makes it well-suited for building complex systems. Monoidal categories are complex objects that can be represented as collections of objects and morphisms, which can be encapsulated in classes in an object-oriented system.

Why do we use Object oriented programming for this project?

Object-oriented programming (OOP) is well-suited for this project because it provides a way to organize and encapsulate the data and functionality associated with finite monoidal categories.

In OOP, objects are created to represent real-world entities, and their behavior is defined through methods. In the case of monoidal categories, objects can represent the elements of the category (e.g., objects, morphisms, tensor products), and their behavior can be defined through methods that implement the monoidal properties, such as associativity and unit objects.

By encapsulating the data and functionality of a monoidal category in objects, the complexity of the category can be managed and organized in a modular fashion, making it easier to maintain and modify. Additionally, OOP provides the ability to reuse code through inheritance and polymorphism, which can be useful in cases where similar monoidal categories need to be analyzed.

Overall, OOP provides a way to model complex systems in a way that is organized, modular, and reusable, which makes it well-suited for building a tool that explores finite monoidal categories.

* Cross-Platform: Java code can be compiled to run on any platform with a Java Virtual Machine (JVM), which makes it a versatile language that can run on a variety of systems.
* Large Developer Community: Java has a large developer community, which means that there are many resources available for learning and troubleshooting issues. This can make it easier to find help if you encounter problems while building your project.
* Extensive Libraries: Java has an extensive set of libraries that can be used to build graphical user interfaces, handle input/output operations, and perform mathematical computations, which can be useful in building a tool with a graphical user interface that checks for the completeness and legality of monoidal categories.
* Mature Language: Java is a mature language that has been around for several decades. As a result, it is a stable language with a well-established syntax and best practices, which makes it easier to write maintainable and robust code.

Why not choose other languages?

* Python: Python is a popular language for scientific computing and has a large number of libraries that can be used for mathematical computations. However, 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.
* C++: C++ is a compiled language that is known for its performance and low-level control. However, it has a steeper learning curve than Java, and its memory management features, such as pointers and memory allocation, can be difficult to manage and prone to errors.
* Haskell: Haskell is a functional programming language that has strong support for algebraic data types and pattern matching, which can be useful for representing the structure of monoidal categories. However, its syntax can be challenging for beginners, and its type system may require a significant amount of time to learn.

Why isn’t functional programming preferred for this project?

* Immutability: Functional programming emphasizes immutable data structures, meaning that once a data structure is created, it cannot be modified. While this is useful for avoiding unintended side effects, it may be less suitable for working with monoidal categories, which often involve modifying and combining elements.
* Type System: Functional programming languages often have powerful type systems that can help ensure correctness and eliminate certain classes of bugs. However, these type systems may also be more complex and require more time to learn than the type system of a language like Java. (I am more comfortable in Java)
* Performance: Functional programming languages tend to favor 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.
* Modularity: While functional programming can also be modular, it may be less well-suited for building complex object-oriented systems like finite monoidal categories, which can involve multiple layers of abstraction and complex relationships between objects.
* JavaScript: JavaScript is a popular language for building web applications and has a large number of libraries for building user interfaces. However, its syntax and semantics can be confusing, especially for beginners, and it may not be the best choice for complex mathematical computations.

Why is this project useful?

* Research in Category Theory: Monoidal categories are an important area of study within category theory, and finite monoidal categories have been studied extensively due to their connection with finite-dimensional algebras. A tool that can explore finite monoidal categories could be useful for researchers in category theory who are interested in studying these structures and their properties.
* Cryptography: Monoidal categories have been used in the development of quantum-resistant cryptography, and a tool that can explore finite monoidal categories could be useful for researchers and practitioners in this field. For example, it could be used to generate examples of finite monoidal categories with certain properties that are relevant to cryptography.
* Computer Science: Monoidal categories have also been used in computer science, particularly in the area of programming language design. A tool that can explore finite monoidal categories could be useful for researchers and practitioners who are interested in exploring the use of these structures in programming language design.
* Education: A tool that can explore finite monoidal categories could also be useful for students who are learning about category theory and related topics. It could provide examples and visualizations that help students understand the concepts and structures involved.
* Advancing our understanding of finite monoidal categories: By exploring which multiplication tables are possible for finite monoidal categories, this project could help us better understand the properties and structures of these categories. This could lead to new insights and discoveries in the field of category theory.
* Providing a tool for researchers and practitioners: A tool that can check whether a given multiplication table defines a legal monoidal category or can be completed to a legal monoidal category would be useful for researchers and practitioners in fields like cryptography and programming language design. It could help them quickly verify whether a particular structure meets certain requirements and avoid errors that could compromise the security or correctness of their systems.
* Enabling new features: By implementing features like checking whether two elements of a given multiplication table are adjoint, the tool could help researchers and practitioners explore new properties and structures of finite monoidal categories.
* Encouraging further research: By making it easier to explore and analyze finite monoidal categories, this project could encourage further research in this area and help generate new ideas and insights.

Software Architecture Description:

In this section, I will provide a description of the software architecture of the Program. I will provide an explanation for why I had chosen certain Java Classes as the right one for my implementation .

* 1. Reasoning behind classes
  2. Class Documentation

2.0

1. Introduction
2. Background
3. Design
4. Implementation
5. Evaluation
6. Conclusions

INDEX:

* CATEGORY:
  + What is a category:
    - Relation to graphs
    - Functions
  + Mophisms
    - Homomorphims
    - Identity morphims
    - Monomorphism
    - Isomorphisms
    - Endomorphisms
  + Functors
    - Normal functors
  + Normal transformations
  + Finite categories
  + Strict categories
    - Strict vs free
    - Strict monoidal categories
  + Finite categories
* Monoidal categories:
  + Formal definitions
    - Bifunctors
  + Properties
    - Natural isomorphisms
    - Coherence conditions
  + Triangle and pentagram diagram
    - Explanations
* Previous work done
* Implementations
  + Code architecture
  + requirements
  + Code algorithms
  + Multiplication table
    - How can a category be represented
    - How can we get information about a category just from the table
      * States
      * Identities
      * Morphism
      * Validity
    - How the tables are saved
    - How the tables are read
    - How the category part is validated
      * Identities
        + Properties and ways its validated
      * Associativity
        + Properties and ways its validated
      * Composition
        + Properties and ways its validated
    - How the tensor product is read and validated and then made into a table
    - If valid then we check the monoidal properties
      * State every property
      * Explain every property
      * State the obvious way of checking and the clever way that we use
      * State how it is checked
      * State how the results affect the outcome
  + Issues faced
* Results:
  + How we tested the program
    - Test data
  + Time and space complexity
  + Runtime (average)
  + Accuracy of the program
* Conclusions
  + Interpretation of the results
  + Ways to improve
  + Did we achieve the goal
  + Any further extensions that can be done

Design:

* Design choices:
  + Why was java chosen
  + Why was oop chosen?
  + Usage of previous work (DiscoPy and the SAT Solver)
* 1st Implemetation
  + Reverse engineering process
  + Format of files
  + File reader
  + Validating the category
    - Checking identities
    - Checking composition
    - Checking associativity
  + Outputting various combinations
* 2nd implementation
  + Proper process
  + File format
  + File reader
  + Validating the category
  + Validating the tensor
  + Validating monoidal properties
* Requirements
  + Requirement for each part of the program.

Background:

In this chapter we develop some basic ideas in category theory through examples and theorems. This will help us understand the goals of the projects better and how it was designed and developed.

Categories:

Categories in a nutshell

Category theory is based on the abstraction of the arrow,

f : a->b

In category theory, a and b are called objects and the arrow relating these 2 objects are called morphisms. 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 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 |- b.

Categories as graphs:

In category theory, a category is often described as graphs, mainly a directed graph(directed multigraph) and for this project it would be easier to view the categories as said graphs. Based upon 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 → N called source and target respectively. We write f : a → b when f is in E and s(f) = a and

t(f) = b. An example of a graph is shown below.

Shape

Description automatically generated with low confidence

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 source and targets. For example in f : A → 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.idA =f = idA.f

P2: for any 3 morphisms, h,g,f in the category, (h ◦ g) ◦ f = h ◦ (g ◦ 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 →B) and a morphism g, from B to C (g: B → C) then there must also exist a morphism h, from A to C ( h : A → C)

P4: If f : A −→ B, then f ◦ idA = idB ◦ f = f.

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 and 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 →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 →B and g : B→ C then fg : A → 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 have A as their domain and B as their codomain. If the category is obvious from the context, then Hom(A, B) can be used instead.

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 →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.g1 = f.g2, then g1=g2 for all morphisms g1,g2: X-> 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 g1.f = g2.f, then g1=g2 for all morphisms g1,g2: B-> 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.g = idA(identity morphism on A) and g.f = idB (identity morphism on B)

endomorphism : An endomorphism that has an Object X as the domain and Codomain. Then f: X-> 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 a category C forms a monoid under 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.

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-> D, it assigns to each object X in C, an object F(X) in category D. It also assigns to each morphism g:X-> Y in C, F(g) : F(X) -> F(Y) in such a way that :

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

The concept of functors play a bit 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.

Natural transformations:

In category theory, natural transformation is a relations 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-> G is a family of morphisms aX:F(X) -> G(X), one for each object X in C such that for every morphism f, in C where f:X->Y, the following diagram commutes.

Text

Description automatically generated with medium confidence

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

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.

A picture containing diagram

Description automatically generated

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.

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 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)

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.

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:

* 1. Associativity: For all a,b and c in M, (a op b) op c = a op (b op c)
  2. 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 typeclass as follows:

Chart

Description automatically generated with low confidence

A picture containing chart

Description automatically generated

in Haskell, the definition of mappend is curried. It can be interpreted as 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 -> m), arises from this interpretation.

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.

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 produces a result as shown :

⊗: C \* D -> 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 ⊗ B.

Natural Isomorphisms:

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

* + 1. Is Associative. There exists a natural isomorphism a, called the associator where by a(A, B, C) : A ⊗ (B ⊗ C) −→ (A ⊗ B) ⊗ C;
    2. 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 ⊗ *I* → A;
       2. lA : *I* ⊗ A −→ 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*

Diagram

Description automatically generated

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 ⊗ operation. It consists of three vertices, each of which is labeled with an object of the category, and three edges labeled with ⊗ operations. One of the vertices is marked with the distinctive object I, and the other two vertices are linked to it by edges labeled with ⊗ 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.

Diagram

Description automatically generated

The pentagram diagram illustrates the principle of associativity, which asserts that the order in which we group three elements using the ⊗ operation is immaterial. It comprises of five vertices, each of which denotes an object of the category, and ten edges labeled with ⊗ operations. These edges combine to form a pentagon, where each vertex is connected to its adjacent vertices by edges labeled with ⊗ 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.