A Computer-Checked Library of Category Theory

Universal Properties of Category Theory in Functional Programming

Markus Orav

Supervisors: Benedikt Ahrens, Lucas Escot

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Markus Orav
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Lucas Escot, Kaitai Liang

Abstract

Category theory is a branch of mathematics that is used to abstract and generalize other mathematical concepts. Its core idea is to take the emphasis off the details of the elements of these concepts and put it on the relationships between them instead. The elements can then be characterized in terms of their relationships using various universal properties. The goal of this project was to implement a pedagogical library of category theory in the computer proof assistant Lean, a software tool for formalizing mathematics, and provide a different perspective on various functional programming concepts by finding parallels between them and the universal properties of category theory.

1 Introduction

Mathematics is all about abstracting the world around us. It essentially provides frameworks and systems for a vast variety of concepts so that we could study, analyse and compare relevant features of physical objects. Eugenia Cheng has said that "Category theory is the mathematics of mathematics. Whatever mathematics does for the world, category theory does for mathematics" [1, §9]. Indeed, category theory is all about abstracting mathematical concepts, in order to study, analyse and compare structures and their relationships in various branches of mathematics.

The core idea of category theory is taking the emphasis off the internal details of the elements and focusing on their relationships instead. This is where the universal properties come in. By characterizing the objects in terms of their relationships to other objects, universal properties capture the common patterns across various fields, including mathematics, computer science, biology and more, and allow us to reason about them more generally.

Computer proof assistants are software tools that allow programmers to formally define and verify mathematical structures, theorems, proofs and concepts in general. Implementing these elements in proof assistants gives us complete confidence in their correctness.

As with any other mathematical concept, category theory can also be formalized in computer proof assistants and is already implemented in various languages, for example Lean [2], Agda [3] and Coq [4]. These libraries, however, are all quite advanced. They include a lot of complicated definitions and documentation which can be very overwhelming for beginners. Furthermore, the lack of concrete examples and instructions makes it difficult for novices to get started with using the aforementioned libraries.

To tackle these problems, I, along with my project group members Ciprian Stanciu, Csanád Farkas, Pedro Brandão de Araujo and Rado Todorov, implemented a pedagogical library of category theory in Lean. We implemented all the core definitions of category theory as well as a few advanced features while focusing on writing clear and understandable code, many examples, beginner-friendly documentation and instructions for using the library so that people who are new to the concepts of category theory and formal verification could use it as a learning tool.

The aim of this report is to explain the implementation of the universal properties of category theory in the pedagogical library. More specifically, this research only focused on four fundamental universal properties, namely initial objects, terminal objects, binary products and binary coproducts. Furthermore, due to the lack of papers highlighting the properties' connection to functional programming, the research also aims to answer the following question: *Which parallels can be drawn between the universal properties of category theory and functional programming?* All found parallels are also exemplified with code snippets from the functional programming language Haskell.

In the upcoming sections, I will first introduce the relevant concepts of category theory and computer proof assistants, after which the methodology for implementing the library will be discussed. Then I will explain the relevant concepts of category theory in more detail along with their implementation in the library and answer the research question. I will also critically reflect on the ethical aspects and reproducibility of the research, discuss the advantages and shortcomings of the library compared to other solutions as well as some possible future improvements and, finally, conclude the report.

2 Background

In this section, I will provide some important background information that is necessary for understanding the rest of the paper. I will give a very high-level explanation of the relevant concepts of category theory, namely categories and universal properties, and what exactly are computer proof assistants. I will also explain what is Lean and why we chose it for implementing the library.

2.1 Category Theory

Category theory is a branch of mathematics that was developed in the 1940s by mathematicians Samuel Eilenberg and Saunders Mac Lane and was properly introduced in a 1945 paper [5] by the same authors. It provides a general, high-level framework for studying mathematical structures and relationships between them by capturing the essential features and common patterns across all areas of mathematics and can, therefore, be used to abstract any mathematical concept.

Nowadays, category theory has found many applications in various fields beyond mathematics, such as physics [6][7], biology [8] and even philosophy [9], just to name a few. Furthermore, it has proven to be particularly useful in computer science as it provides a foundation for studying and designing programming langagues.

Categories

As the name might suggest, category theory is mostly concerned with studying categories. These are essentially just structures, consisting of objects and morphisms (relationships) between them, that enable abstracting concepts from various fields into a common framework. The exact definition of a category along with its implementation is explained in Section 4.1.

Universal properties

The main idea of category theory is abstracting away the specific, often messy and irrelevant details of the objects themselves and focusing on their relations to other objects within the category. Instead of studying the objects in isolation, it puts the emphasis on the interactions between them. This is where universal properties come in.

Universal properties characterize the objects in a category in a unique way, in terms of their relations to other objects. They are useful because they capture common patterns and parallels across various fields and, therefore, allow us to study and reason about them in a unified way while avoiding repetition by enabling the reuse of the same (or at least similar) proofs for different categories.

Although there are many papers and books that explain the connection between category theory and functional programming [10][11][12], they usually focus on other elements of category theory, mostly functors and monads, as well as lack concrete examples. However, there are many parallels that can be drawn between the universal properties of category theory and various functional programming concepts and understanding them can give many insights into the nature of these concepts.

Category theory includes several universal properties, some more common and useful than others. This report will only describe the four universal properties introduced in "Category Theory for Programming" [13, §3], namely initial objects, terminal objects, binary products and binary coproducts, as these are arguably the most fundamental ones. Their definitions and implementations are explained in Section 4.2.

2.2 Computer Proof Assistants

Proving mathematical concepts by hand can be a very time-consuming, repetitive and error-prone process. Handwritten definitions, notations and logical steps can also lack the necessary rigor and precision, making the proofs vulnerable to misinterpretation and invalid reasoning. To tackle these problems, several systems and concepts, most notably Automath and Martin-löf type theory, started to arise during the late 1960s and early 1970s [14, §2] which laid the foundation for the computer proof assistants as we know them today.

Computer proof assistants, also known as interactive theorem provers, are software tools for formalizing mathematical concepts. They make use of dependent type theory, an extension of traditional type theory where the types are allowed to depend on some values, to type check all terms in the code which provides a high degree of confidence in the correctness of the mathematical statements.

There are many different computer proof assistants available, each one with their own advantages and disadvantages. Some of the most popular theorem provers are Agda, Coq, Isabelle, HOL and, our choice, Lean.

Lean

Lean is a functional programming language and theorem prover. It is an open source project launched in 2013 as an effort to help mathematicians solve and formalize complex math problems. Since then, it has been in constant development thanks to the contributions of a large community of students, professors and mathematicians. The community has already formalized over half of the undergraduate mathematics curriculum in Mathlib [2], the unified library of mathematics in Lean, and is aiming to finish digitizing the entire curriculum in the next 5 years [15].

Lean is one of many proof assistants that is based on the Curry-Howard correspondence, a framework for representing proofs in formal logic as computer programs. More specifically, it states that propositions in formal logic can be interpreted as types in programming languages and each instance of the type as its proof [16, §5]. Therefore, constructing a proof for a proposition is equivalent to defining a term of the corresponding type. Finally, type-checking the code ensures that the terms are well-typed and, therefore, satisfy the logical rules and constraints encoded in them.

We chose to implement the library in Lean as this is the proof assistant of choice of the project's responsible professor. We also considered using Agda because that is what our supervisor is the most familiar with but ended up deciding for Lean for two reasons. Firstly, Lean was designed with simplicity in mind and has a more intuitive syntax which makes it easier to learn for beginners. Secondly, Lean offers many automation features which make the proofs less cluttered and reduce manual effort by automating routine proof steps.

We opted for Lean 3 instead of Lean 4 as it is the current stable version. It has been extensively used and tested by the community, has a large ecosystem of libraries and tools as well as documentation, tutorials and other learning resources. Although the community is currently switching to Lean 4, we prioritized the abundance of learning materials and existing libraries of Lean 3 over the improvements that Lean 4 offers due to the very limited time frame of roughly 9 weeks for the whole project. Furthermore, unlike for Lean 3, there is no Docker image for Lean 4 yet, so using the older version allowed us to get started more quickly.

3 Methodology

The core of the project was implementing a pedagogical library of category theory in the computer proof assistant Lean. The insights gained while learning about category theory and implementing the library could then be used to answer the research question. With that in mind, the project was split into two phases.

During the first phase, we worked together with the project group to familiarize ourselves with both category theory and Lean and then implement the core features of the library. In addition to some features that are irrelevant for this report, such as functors, these include the definition and examples for a category which were necessary for moving on to the next phase.

During the second phase, each of us extended the library individually by focusing on a different specific area of category theory, implementing relevant definitions and examples and answering their research question. My task was to implement the universal properties of category theory.

Although the codebase was developed on TU Delft's self-managed Gitlab instance, the repository was made publicly available on Github and can be accessed here. To guaran-

tee the correctness of the implemented definitions throughout the project and when merging branches, we set up a very basic pipeline that simply checks whether the code compiles. Since the successful compilation means that the definitions are mathematically solid, we could be certain of the correctness of the code. However, whether the design choices we made are the best ones for the purpose of a learning tool, remained our concern.

In the repository, we also included a folder called *doc* that contains some explanations regarding the less intuitive definitions as well as used resources to make using the library as convenient and informative as possible for beginners. Furthermore, in the *readme* file, there are instructions for installing the library.

4 Implementation

In this section, I will give definitions of a category and the four universal properties mentioned in Section 2.1 as well as explain how they were implemented in Lean. Although the section does include high-level explanations of the explicit Lean code snippets, Lean's reference manual [17] can be consulted to better understand the exact syntax of the code.

4.1 Category

This section explains how a category is defined, how we implemented it in Lean as well as an example of a category, namely the category of sets.

Definition

Each category consists of the following four components:

1. *Objects*. Objects are the fundamental entities of a category and can represent any kind of mathematical structures, such as sets, groups and even categories themselves. The collection of objects as a whole is usually denoted by C_0 and individual objects within this collection are often denoted by uppercase letters, most typically A, B, X, Y and Z.
2. *Morphisms*. Morphisms are relationships or mappings between objects. Morphisms are targeted which means they have a source and a target object, basically indicating that applying the morphism to its source results in its target. The most intuitive example of morphisms are simply functions between the objects but in some categories they are not so straight-forward. The collection of morphisms for any pair of objects $X, Y \in C_0$ is called a *hom-set* and is usually denoted as either $hom(X, Y)$ or $X \to Y$.
3. *Identity morphisms*. For each object, there must exist an identity morphism which simply transforms the object into itself. An identity morphism for an object $X \in C_0$ is usually denoted as Id_X. Note that $Id_X \in hom(X, X)$.
4. *Composition function*. This is a binary operation that combines two morphisms into a new one that goes from the source of one to the target of the other. More specifically, given objects $X, Y, Z \in C_0$ and morphisms $f \in hom(X, Y)$ and $g \in hom(Y, Z)$, the composition operation, denoted by $\circ$, returns a new morphism $g \circ f \in hom(X, Z)$. Intuitively, it first applies f to go from X to Y and then g to go from Y to Z, resulting in a morphism that goes straight from X to Z. Note that the morphism that is applied first (f in that case) is the second argument of the operation.

Furthermore, in order for the category to be valid, these components must also satisfy three properties:

1. *Left unit law*: for any morphism $f \in hom(X, Y)$, it must hold that
$$f \circ Id_X = f$$
2. *Right unit law*. for any morphism $f \in hom(X, Y)$, it must hold that
$$Id_Y \circ f = f$$
3. *Associative law*. for any morphisms $f \in hom(X, Y)$, $g \in hom(Y, Z)$ and $h \in hom(Z, W)$, it must hold that
$$(h \circ g) \circ f = h \circ (g \circ f)$$

Categories are often visualized by directed graphs where the vertices and edges represent the objects and morphisms, respectively. For example, a simple category with objects $X, Y, Z \in C_0$ and morphisms $f \in hom(X, Y)$ and $g \in hom(Y, Z)$ can be represented by the graph shown in Figure 1. Graphs are a very intuitive way for representing categories and will prove useful for explaining the universal properties of category theory.

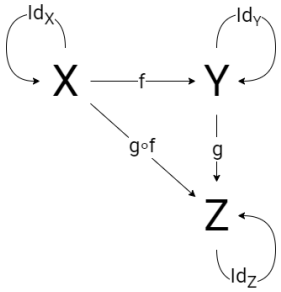

Figure 1: A category with objects X, Y, Z and morphisms f and g represented as a graph. The composition $g \circ f$ is also included.

Implementation

In Lean, we defined the category as a structure with fields that represent the components and axioms described in the previous subsection. The following code segment can be found in the file *category.lean*.:

```
structure category :=
  --attributes
  (C₀      : Sort u)
  (hom     : Π (X Y : C₀), Sort v)
  (id      : Π (X : C₀), hom X X)
  (compose : Π {X Y Z : C₀} (g : hom Y Z) (f : hom X Y), hom X Z)
  --axioms
  (left_id  : ∀ {X Y : C₀} (f : hom X Y), compose f (id X) = f)
  (right_id : ∀ {X Y : C₀} (f : hom X Y), compose (id Y) f = f)
  (assoc    : ∀ {X Y Z W : C₀}
    (f : hom X Y) (g : hom Y Z) (h : hom Z W),
      compose h (compose g f) = compose (compose h g) f)
```

C_0 represents the type of objects in the category but can be thought of as a collection of objects (collection of elements of the given type). $Sort\ u$ basically denotes any type universe so C_0 can basically be any type.

hom is a function that assigns a hom-set for each pair of objects $X, Y \in C_0$. Since the hom-set is of type $Sort\ v$, the morphisms can also be of any type.

id is a function that assigns an identity function to each object $X \in C_0$.

$compose$ is a function that takes two morphisms $g \in hom(Y, Z)$ and $f \in hom(X, Y)$ and returns their composition. Note the curly brackets around X, Y and Z: these are implicit arguments which means that Lean is able to deduce their values based on the context so they do not have to be passed to the function call.

In order to verify that the components do indeed satisfy the *Left unit law*, *Right unit law* and *Associative law*, the structure also includes the corresponding axioms. When defining a concrete instance of a category, these axioms need to be initialized with the proofs of these properties.

Example

As an example, I will explain how we defined the category of sets. The code segment can be found in the file *Set_category.lean* or Appendix A.

The objects in the Set category are sets which can be represented by $Type\ 0$ in Lean.

The morphisms in the Set category are simply functions between sets. Therefore, given sets $X, Y \in C_0$, the morphisms from X to Y are defined as $X \to Y$, the functions between them.

The identity function takes a set $X \in C_0$ as input and returns its trivial identity morphism: a function that leaves the input element unchanged.

The composition function takes two functions $g \in Y \to Z$ and $f \in X \to Y$ and returns their trivial composition: a function that first applies f and then g.

Finally, the three laws need to be proven. Since all of them state a property for all objects and morphisms in the category, they can all be proven by first assuming a random value for each implicit and explicit parameter and then proving the property. Since all three laws are already proven for functions, we made use of these proofs to prove the properties.

4.2 Universal Properties

This section gives definitions of the following universal properties: initial objects, terminal objects, binary products and binary coproducts. Furthermore, it explains how I implemented each property in the library as well as give an example of it in the category of sets. All the code can be found in the folder *universal_properties*.

Initial objects

An object $A \in C_0$ is initial if there is a unique morphism from A to every object $B \in C_0$. This is illustrated in Figure 2.

In the library, the property of initiality is checked by the $is\_initial$ function and the structure $initial\_object$ represents an initial object in the given category:

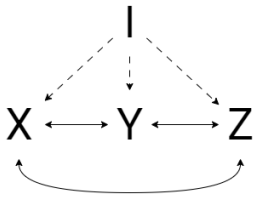

Figure 2: A category with objects X, Y, Z and initial object I. A dashed arrow represents a unique morphism and double-sided arrows represent (possibly zero or many) morphisms in both directions. Identity morphisms are hidden in the graph.

```
def is_initial (C : category) (A : C.C₀) : Prop :=
  ∀ (B : C.C₀) (f g : C.hom A B), f = g

structure initial_object (C : category) :=
  (object : C.C₀)
  (property : is_initial C object)
```

$is\_initial$ takes as inputs the category C and object $A \in C_0$ and returns a proposition indicating whether A is initial in C. A is initial if for any object $B \in C_0$ and any morphisms $f, g \in hom(A, B)$, f and g are equal, meaning there is a unique morphism from A to each B.

$initial\_object$ takes as input a category C and has two fields: $object$ and $property$. $object$ must be initialized with an object in the given category and $property$ with a proof that $object$ is indeed initial in the given category.

Example:

The initial object in the Set category is the empty set ($\emptyset$) because for any set S, there is exactly one function from $\emptyset$ to S, namely the empty function.

To define the empty set as an initial element in Set, $is\_initial\ Set\ empty$ needs to be proven. Since the property states that $f = g$ must hold for all B, f and g, a random value is assumed for all of them. Next, to prove that $f = g$, $f\ x = g\ x$ needs to be proven for all values in x. Since x is the empty set, however, there is nothing left to prove. The code segment doing all of this can be found in Appendix B.

Terminal objects

An object $B \in C_0$ is terminal if there is a unique morphism from every object $A \in C_0$ to B. This is illustrated in Figure 3.

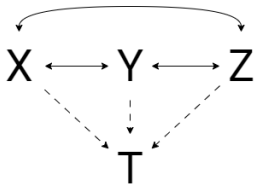

Figure 3: A category with objects X, Y, Z and terminal object T. A dashed arrow represents a unique morphism and double-sided arrows represent (possibly zero or many) morphisms in both directions. Identity morphisms are hidden in the graph.

The property of terminality is implemented analoguously to initiality:

```
def is_terminal (C : category) (B : C.C₀) : Prop :=
  ∀ (A : C.C₀) (f g : C.hom A B), f = g

structure terminal_object (C : category) :=
  (object : C.C₀)
  (property : is_terminal C object)
```

Compared to $is\_initial$, the roles of A and B are switched in $is\_terminal$. Now B is passed as an argument and the property must hold for all A. Note that the morphisms f and g still go from A to B, so that the property would check if there is a unique morphism going from each A to B.

Example:
In the category of sets, every singleton set is a terminal object because for any set S, there is exactly one function from S to any one-element set $\{x\}$, namely a function that maps every element in S to x.

To define a singleton set as a terminal object in Set, $is\_terminal\ Set\ unit$ needs to be proven. The proof starts analogously to the one in the previous section regarding initial objects. However, given that the codomain only has one element, $f\ x = g\ x$ can be proved easily by using the $subsingleton.elim$ property. The code segment doing all of this can be found in Appendix C.

Binary products
Given objects $A, B \in C_0$, a triple

$$(P \in C_0, \pi_1 \in hom(P, A), \pi_2 \in hom(P, B))$$

is called a product of A and B if for any triple

$$(Q \in C_0, q_1 \in hom(Q, A), q_2 \in hom(Q, B))$$

there is a unique morphism $f \in hom(Q, P)$ such that $\pi_1 \circ f = q_1$ and $\pi_2 \circ f = q_2$. This is illustrated in Figure 4.

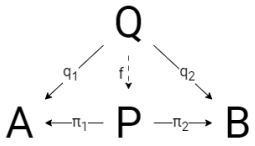

Figure 4: The relationship between the binary product (P, π_1, π_2) and any triple (Q, q_1, q_2) in the category. The dashed arrow represents a unique morphism.

In the library, the property of binary products is checked by the $is\_binary\_product$ function and the structure $binary\_product$ represents a product in the given category:

```
def is_binary_product (C : category) {A B : C.C₀}
    (P : C.C₀) (π₁ : C.hom P A) (π₂ : C.hom P B) : Prop :=
        ∀ (Q : C.C₀) (q₁ : C.hom Q A) (q₂ : C.hom Q B),
            ∃! (f : C.hom Q P),
                C.compose π₁ f = q₁ ∧ C.compose π₂ f = q₂

structure binary_product (C : category) (A B : C.C₀) :=
  (P : C.C₀)
  (π₁ : C.hom P A)
  (π₂ : C.hom P B)
  (property : is_binary_product C P π₁ π₂)
```

$is\_binary\_product$ takes as input the category C, objects $A, B \in C_0$ (implicitly, meaning they do not need to be passed to the function call), object $P \in C_0$ representing the product of A and B, and projection functions $\pi_1 \in hom(P, A)$ and $\pi_2 \in hom(P, B)$ and returns a proposition indicating whether the triple (P, π_1, π_2) is a binary product in C. The triple is a binary product if for all $Q \in C_0$, $q_1 \in hom(Q, A)$ and $q_2 \in hom(Q, B)$, there exists exactly one morphism $f \in hom(Q, P)$ such that $\pi_1 \circ f = q_1$ and $\pi_2 \circ f = q_2$.

$binary\_product$ takes as input a category C and objects $A, B \in C_0$. It has three fields $P \in C_0$, $\pi_1 \in hom(P, A)$ and $\pi_2 \in hom(P, B)$ representing the triple and a $property$ field for validating that the triple is indeed a binary product.

Example:
The binary product in the category of sets is simply the cartesian product along with the trivial projection functions. For sets A and B, the cartesian product $A \times B$ is equal to $\{\langle a, b\rangle | a \in A, b \in B\}$ and the projection functions π_1 and π_2 are defined as a and b for each $\langle a, b\rangle \in A \times B$, respectively. These elements form a binary product because for any set Q and functions q_1 and q_2 from Q to A and B, respectively, there is a unique function from Q to $A \times B$, namely $f = \langle q_1, q_2\rangle$, such that $\pi_1 \circ f = q_1$ and $\pi_2 \circ f = q_2$.

To define the cartesian product of two sets as a binary product in Set, $is\_binary\_product\ Set\ (A \times B)\ (\lambda p, p.1)\ (\lambda p, p.2)$ needs to be proven. This can be done by assuming random values for Q, q_1 and q_2, defining the product morphism f and, finally, proving that f is indeed a valid product morphism. This can be done in two parts. Firstly, f must satisfy the property of binary products ($\pi_1 \circ f = q_1 \wedge \pi_2 \circ f = q_2$). Secondly, f must be a unique function or, in other words, any morphism that satisfies the property of binary products must be equal to f. The code doing all of this can be found in Appendix D.

Binary coproducts
Given objects $A, B \in C_0$, a triple

$$(C_p \in C_0, \iota_1 \in hom(A, C_p), \iota_2 \in hom(B, C_p))$$

is called a coproduct of A and B if for any triple

$$(D \in C_0, i_1 : hom(A, D), i_2 : hom(B, D)$$

there is a unique morphism $f \in hom(C_p, D)$ such that $f \circ \iota_1 = i_1$ and $f \circ \iota_2 = i_2$. This is illustrated in Figure 5.

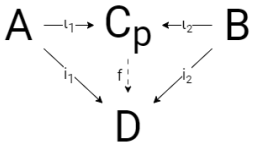

Figure 5: The relationship between the binary coproduct (C_p, ι_1, ι_2) and any triple (D, i_1, i_2) in the category. The dashed arrow represents a unique morphism.

Binary coproduct is implemented analoguously to binary product:

```
def is_binary_coproduct (C : category) {A B : C.C₀}
    (Cₚ : C.C₀) (ι₁ : C.hom A Cₚ) (ι₂ : C.hom B Cₚ) : Prop :=
        ∀ (D : C.C₀) (i₁ : C.hom A D) (i₂ : C.hom B D),
            ∃! (f : C.hom Cₚ D),
                C.compose f ι₁ = i₁ ∧ C.compose f ι₂ = i₂
```

```
structure binary_coproduct (C : category) (A B : C.C₀) :=
  (Cₚ : C.C₀)
  (ι₁ : C.hom A Cₚ)
  (ι₂ : C.hom B Cₚ)
  (property : is_binary_coproduct C Cₚ ι₁ ι₂)
```

Compared to $is\_binary\_product$, the variable names are different in $is\_binary\_coproduct$ in order to match the notation in the definition of binary coproducts. Since A and B are now part of the functions' domain, the directions of the morphisms have also changed. Otherwise, the definitions of binary products and coproducts are pretty similar.

Example:
The binary coproduct in the category of sets is the disjoint union along with its inclusion maps. For sets A and B, the disjoint union $A \oplus B$ is equal to $\{\langle a, 0\rangle | a \in A\} \cup \{\langle b, 1\rangle | b \in B\}$ and the inclusion maps ι_1 and ι_2 are defined as $\langle a, 0\rangle$ for each $a \in A$ and $\langle b, 1\rangle$ for each $b \in B$, respectively. These elements form a binary coproduct because for any set D and functions i_1 and i_2 from A and B to D, respectively, there is a unique function from $A \oplus B$ to D, namely the function f that is defined as $f(\langle a, 0\rangle) = i_1(a)$ and $f(\langle b, 1\rangle) = i_2(b)$, such that $f \circ \iota_1 = i_1$ and $f \circ \iota_2 = i_2$.

To define the disjoint union of two sets as the binary coproduct in Set, $is\_binary\_coproduct\ Set\ (A \oplus B)\ sum.inl\ sum.inr$ needs to be proven. This can be done by assuming random values for D, i_1 and i_2, defining a coproduct morphism f and finally proving that f is indeed a valid coproduct morphism. This can be done in two parts. Firstly, f must satisfy the property of binary coproducts ($f \circ \iota_1 = i_1 \wedge f \circ \iota_2 = i_2$). Secondly, f must be a unique function or, in other words, any morphism that satisfies the property of binary coproducts must be equal to f. The code doing all of this can be found in Appendix E.

5 Results

Learning about and defining the universal properties of category theory in the pedagogical library gave me a much deeper understanding of some of the underlying concepts of functional programming languages as there are many parallels that can be drawn between the two concepts. Understanding these analogues allows us to reason about various functional programming concepts in a categorical way by using these properties which can lead to the design of more modular and maintainable code in the future.

In this section, I will explain how the implemented universal properties relate to various functional programming concepts and exemplify the observations with Haskell code snippets. Although the examples are fairly simple, the book "Programming in Haskell" by G. Hutton [18] can be consulted to better understand functional programming and Haskell syntax.

5.1 Algebraic Data Types

Most functional programming languages make use of algebraic data types, which are composed of several distinct constructors, each of which can possibly have some arguments. Here is a simple example in Haskell:

```
data Shape = Circle Float | Rectangle Float Float
```

The type $Shape$ has two constructors: $Circle$ and $Rectangle$. To construct an element of this type, one has to create either a circle by calling the corresponding constructor with one argument of type $Float$ (representing the radius) or a rectangle by calling the corresponding constructor with two arguments of type $Float$ (representing the width and height).

Empty and unit types
Empty type is a type with no values. For example, the built-in empty type in Haskell is called $Void$ and is defined simply by $data\ Void$. Since it has no constructors, it cannot have any values either.

Unit type is a type with only one possible value. For example, the built-in unit type in Haskell is called $()$ and is defined simply by $data\ () = ()$. Since it only has one constructor without any arguments, it only has one possible value too.

Similarly to the category of sets where the initial and terminal objects are the empty and unit sets, respectively, empty and unit types can be represented as initial and terminal objects, respectively, in the category of data types. This allows us to reason about them in a categorical way. For example, the property of initial objects being essentially unique within a category gives us the insight that all empty types are also equivalent in practice. The same goes for terminal objects and unit types.

Both initial objects and empty types capture the notion of a starting point. Therefore, it is not possible to define a (terminating) function with an empty return type. However, since there is a unique morphism from the initial object to any object in the category, there is exactly one possible function (the empty function) from the empty type to any other type. In Haskell, this function is called $absurd$. It is the logical reasoning tool of the principle of explosion which basically states that any proposition can be proven from a contradiction.

Both terminal objects and unit types capture the notion of an ending point. Therefore, there is no point in defining functions with arguments of unit types since they would always act the same way. Terminal objects and unit types are also similar in the sense that they do not really contain any information other than the indication that they are terminal. Therefore, unit types are often used in functional programming, including in Haskell, as the return type of functions that perform side-effects (instead of returning an informative result) to indicate the completion and termination of the evaluation.

Product and sum types
Algebraic data types are typically split into product and sum types but are often a combination of both.

Product types contain one or more fields of possibly different types to combine simpler values into more complex ones. For example, tuples and records are product types.

Sum types, on the other hand, have values of different variants, each of which having its own separate constructor. For example, the $Shape$ type introduced earlier is a sum type of two variants: $Circle$ and $Rectangle$. Since both constructors include at least one field, they also represent two separate product types.

Many parallels can be drawn between binary products in category theory and product types in functional program-

ming. Both of them combine multiple objects or values, respectively, to construct new ones. Similarly to the category of sets where the binary product of two sets is their cartesian product, the set of all possible values of a product type is the cartesian product of the sets of all possible values of its field types. Furthermore, since field accessors (for example, functions fst and snd for tuples in Haskell) provide a way to extract the individual values from the composite type, they represent the projection morphisms. Therefore, product types are a natural analogue to products in category theory and the two concepts can be reasoned about in a similar manner.

Likewise, sum types are a natural analogue to coproducts. Both of them essentially provide a way to construct an object or value, respectively, by choosing one of the options. Similarly to the category of sets where the binary coproduct of two sets is their disjoint union, the set of all possible values of a sum type is the disjoint union of the sets of all possible values of its variants. Furthermore, the constructors can be seen as the injection morphisms that transform the (possibly empty or composite) types of their fields into a new value of the corresponding sum type. Taking the $Shape$ type as an example again, the constructors $Circle$ and $Rectangle$ basically transform a $Float$ and a pair of $Float$s, respectively, into another type called $Shape$.

5.2 Pattern matching

Pattern matching is an important feature in functional programming languages as it allows to destructure data and execute specific code blocks according to the matched patterns in the structure. For example, a function in Haskell that takes one argument of type $Shape$ and calculates its area could be defined by using pattern matching as follows:

```
area :: Shape -> Float
area (Circle r)      = r * r * pi
area (Rectangle a b) = a * b
```

If the provided $Shape$ is a $Circle$, the first pattern is matched and the area is calculated by the first formula. If the $Shape$ is a $Rectangle$, the second formula is used. Furthermore, the individual arguments of each constructor are extracted, in order to use them in the respective code blocks.

Binary products involve the idea of composing elements into more complex structures while still having access to their individual components through projection functions. This idea corresponds really well to the extraction of the arguments of product types in pattern matching.

Likewise, binary coproducts capture the notions of choice and alternatives which correspond really well to pattern matching handling different cases of sum types separately.

5.3 Recursion

The universal property of initiality is a powerful tool for reasoning about both recursive data types and functions.

Recursive data types

Values of recursive data types are built using constructors for the base and recursive case(s). A classic example of a recursive data type in Haskell is $List$:

```
data List a = Nil | Cons a (List a)
```

The base case for $List$ is Nil which represents an empty list and the recursive case is $Cons$ which constructs a new list by adding an element to the start of an existing one.

Initial objects capture the notions of emptiness and starting points. Similarly, the base case(s) in recursive data types represent the simplest and often emptiest (as is the case with $List$) values of the type as well as the starting point(s) for constructing more complex values of the same type. Therefore, there are many parallels between these two concepts and various properties of initial objects also apply for the base cases of recursive data types. For example, from the property of initial objects having a unique morphism to every object within the category, one can reason that each value of a recursive data type is constructed and represented uniquely by starting from (one of) the base case(s) and applying a specific combination of other constructors to it.

Recursive functions

Since the structure of recursive functions usually mimics that of the data types they operate on, their base case(s) also have many parallels with initial objects.

Recursive functions work by breaking down its arguments into simpler terms until reaching the base case, assigning a value to it and then doing some operations (defined in the recursive cases) on this value to get the final result. For example, we can take the $length$ method in Haskell that returns the number of elements in a list:

```
length :: List a -> a
length Nil = 0
length (Cons x xs) = 1 + length xs
```

It breaks down the provided list until reaching the base case (empty list), assigns the value 0 to it and adds 1 for each element in the list. This again coincides with the essence of an initial object: it is the empty starting point from which the final result is constructed.

6 Responsible Research

This section describes the ethical aspects as well as reproducibility of the research.

6.1 Ethical Aspects

Since the project was mostly about developing a library, it did not directly involve any human subjects. Therefore, the research did not pose any risks that are often associated with studies involving human participants such as informed consent, privacy and data protection.

It was, however, important to acknowledge the potential for indirect implications, such as the impact on future users and developers of the library. We added clear and comprehensive documentation, including guidelines for using the library as well as warnings about potential limitations. Furthermore, we did our best to follow the principles of universal design [19, §4] to ensure that the library would be accessible for a wide variety of users regardless of their experience in the domain.

6.2 Reproducibility

The library's codebase, including its documentation, is publicly accessible on Github. I, along with the rest of my project

group, have documented the important design decisions, external dependencies and guidelines for setting up the environment in either the repository, our individual papers or both so that other researchers could examine and reproduce our work with ease. Since we utilized Git to keep track of the changes in the codebase, they can also access previous versions, in order to better understand the evolution of the library.

7 Discussion

In this section, the implemented library will be compared to other, already existing alternatives. Additionally, possible future improvements will be listed and explained.

7.1 Comparison to Related Work

Defining the concept of category theory in computer proof assistants, even Lean, is not a new idea. There are already existing libraries for it in various languages, for example Lean [2], Agda [3] and Coq [4]. However, all of these are quite advanced and built for experts whereas our library aims to act as a learning tool for beginners. With that in mind, this section lists the advantages and disadvantages of our library compared to the other, aforementioned category theory libraries.

Advantages

1. Since we focused on defining the elements as simply and understandably as possible and did not make things too complicated by over-generalizing things, our solution is *more intuitive* and *easier to use* than the alternatives.
2. *Concrete examples* go a long way in learning new concepts which is why we added at least one for each implemented element, contrary to most other libraries.
3. To make the library more usable for beginners, we included many *comments* and much *documentation* about both category theory and Lean.
4. Because of all the aforementioned reasons, our library is more *beginner-friendly* than its alternatives.

Disadvantages

1. Our library is *not as general and abstract* as its alternatives and might not be advanced enough for defining some more complicated instances.
2. The *number of implemented features* in our library is *significantly smaller* than in other libraries.

7.2 Possible Future Improvements

Due to time constraints, the implemented library only includes the most important elements of category theory. This section lists and elaborates on some possible future improvements and additions related to universal properties.

Add more universal properties

Currently, the library only includes definitions of four universal properties: initial and terminal objects, products and coproducts. Although these are arguably the most common and useful ones, there are also other properties that could be added into the library in the future. These include but are not limited to *exponential objects*, *equalizers* and *coequalizers* as well as *pullbacks* and *pushouts*.

Add more examples of categories

The library currently includes three examples of concrete category instances: *Set*, *Pos* (partially ordered sets) and *Monoid*. However, there exist many more categories that could be defined in the library in the future. The most common ones that should, therefore, be implemented first are:

1. Top - category of topological spaces as objects and continuous maps as morphisms
2. Grp - category of groups as objects and group homomorphisms as morphisms
3. $Vect$ - category of vector spaces as objects and linear transformations as morphisms

Add more examples of universal properties

Even though the library already includes three aforementioned category instances, the implemented universal properties are currently only defined for one of them, the category of sets. It would be beneficial to also define them (as well as new ones in the future) for other categories.

Prove theorems about universal properties

The library currently only includes definitions and examples of universal properties but no theorems related to them. To deepen the insights into these properties further, various theorems (for example, the fact that initial and terminal objects are isomorphic) could be defined and proven about them.

8 Conclusion

The main goal of this research project was to implement a pedagogical library of category theory in the computer proof assistant Lean and this report focused, more specifically, on the implementation of universal properties. During the project, four universal properties, namely initial and terminal objects as well as binary products and coproducts, were defined along with their examples in the category of sets. By including many examples and documentation, the library was implemented in such a way that it would be beneficial for beginners who want to start learning about category theory.

Furthermore, while implementing the library, this research also tried to find parallels between the universal properties of category theory and various functional programming concepts, in order to give programmers a different, categorical perspective for reasoning about these concepts. The most significant parallels found were between initial objects and empty types, initial objects and base cases in recursion, terminal objects and unit types, binary products and product types as well as binary coproducts and sum types. These parallels allow us to leverage the abstract nature of category theory to reason about functional programs and type systems which can lead to the design of more modular and maintainable code in the long run.

A Set Category

```
def Set : category :=
{
  -- Type 0 in Lean is essentialy a set.
  C₀ := Type 0,

  -- A morphism between two sets maps the elements from one set
  -- to the other, same as what a function between types does.
  hom := λ X Y, X → Y,

  -- The identity morphism maps each element to itself.
  id  := λ X, λ (x : X), x,

  -- Each morphism is a function, so morphism composition is the
  -- same as composition of the underlying functions.
  compose := λ {X Y Z} (g : Y → Z) (f : X → Y), g ∘ f,

  -- We can use the proofs from function.comp.
  left_id :=
    begin
      intros,
      apply function.comp.right_id,
    end,
  right_id :=
    begin
      intros,
      apply function.comp.left_id,
    end,
  assoc :=
    begin
      intros,
      apply function.comp.assoc,
    end,
}
```

B Empty Set as Initial Object in Set

```
lemma empty_set_initial_in_Set : is_initial Set empty :=
begin
  intros B f g,
  funext x,
  cases x, -- There are no elements in the empty set, so we can
      use cases to handle all possible cases (i.e., none)
end

def initial_object_in_Set : initial_object Set :=
{
  object := empty,
  property := empty_set_initial_in_Set
}
```

C Singleton Set as Terminal Object in Set

```
lemma singleton_set_terminal_in_Set : is_terminal Set unit :=
begin
  intros A f g,
  funext x,
  apply subsingleton.elim (f x) (g x), -- Using the subsingleton
      property to conclude f x = g x
end

def terminal_object_in_Set : terminal_object Set :=
{
  object := unit,
  property := singleton_set_terminal_in_Set,
}
```

D Binary Product in Set

```
lemma cartesian_product_binary_product_in_Set (A B : Set.C₀) :
      is_binary_product Set (A × B) (λ p, p.1) (λ p, p.2) :=
begin
  intros Q q₁ q₂,

  -- Define a product morphism f
  let f : Q → A × B := λ x, (q₁ x, q₂ x),

  -- Define and prove lemmas proj₁ and proj₂
  -- They state that for any x ∈ Q, applying the projection
     functions to (f x) yield (q₁ x) and (q₂ x), respectively
  -- This is true because (f x) is defined as (q₁ x, q₂ x)
  have proj₁ : ∀ (x : Q), (λ p : A × B, p.1) (f x) = q₁ x,
  {
    intro x,
    simp [f],
  },
  have proj₂ : ∀ (x : Q), (λ p : A × B, p.2) (f x) = q₂ x,
  {
    intro x,
    simp [f],
  },

  -- Prove that f is the unique product morphism
  -- This can be done by splitting the goal into two subgoals and
     proving them
  -- The first subgoal states that f is indeed a valid product
     morphism
  -- The second subgoal states that f is the only valid product
     morphism
  existsi f,
  split,

  -- Prove the first subgoal
  {
    split,

    -- Prove that the diagram commutes to the left
    -- That is, applying the projection function π₁ to f results
      in q₁
    {
      apply funext,
      intro x,
      exact (proj₁ x),
    },

    -- Prove that the diagram commutes to the right
    -- That is, applying the projection function π₂ to f results
      in q₂
    {
      apply funext,
      intro x,
      exact (proj₂ x),
    },
  },

  -- Prove the second subgoal
  {
    intros g H,
    apply funext,
    intro x,
    cases H with H₁ H₂,

    -- Define and prove lemmas H₁' and H₂'
    -- They state that for any x ∈ Q, applying the projection
      functions to (g x) yield (q₁ x) and (q₂ x), respectively
    have H₁' : (λ (p : A × B), p.1) (g x) = q₁ x,
    {
      rw ←H₁,
      refl,
    },
    have H₂' : (λ (p : A × B), p.2) (g x) = q₂ x,
    {
      rw ←H₂,
      refl,
    },

    -- Use H₁' and H₂' to finish the proof
    simp [f],
    rw ←H₁',
    rw ←H₂',
    simp [H₁', H₂'],
  },
end
```

```
def binary_product_in_Set (A B : Set.C₀) : binary_product Set A B
    :=
{
  P := A × B,
  π₁ := λ p, p.1,
  π₂ := λ p, p.2,
  property := cartesian_product_binary_product_in_Set A B,
}
```

E Binary Coproduct in Set

```
lemma disjoint_union_binary_coproduct_in_Set (A B : Set.C₀) :
    is_binary_coproduct Set (A ⨁ B) sum.inl sum.inr :=
begin
  intros D i₁ i₂,

  -- Define the coproduct morphism
  let f : (A ⨁ B) → D := λ x, sum.cases_on x i₁ i₂,

  -- Define and prove lemmas inl and inr
  -- They state that for any x ∈ A, applying f to the inclusion
     functions applied to x yield (i₁ x) and (i₂ x), respectively
  -- This is true because (f x) maps (sum.inl x) to i₁ x and
     (sum.inr x) to i₂ x
  have inl : ∀ (x : A), f (sum.inl x) = i₁ x,
  {
    intro x,
    simp [f],
  },
  have inr : ∀ (x : B), f (sum.inr x) = i₂ x,
  {
    intro x,
    simp [f],
  },

  -- Prove that f is the unique coproduct morphism
  -- This can be done by splitting the goal into two subgoals and
     proving them
  -- The first subgoal states that f is indeed a valid coproduct
     morphism
  -- The second subgoal states that f is the only valid coproduct
     morphism
  existsi f,
  split,

  -- Prove the first subgoal
  {
    split,

    -- Prove that the diagram commutes to the left
    -- That is, applying f to the inclusion function ι₁ results
     in i₁
    {
      apply funext,
      intro x,
      exact (inl x),
    },

    -- Prove that the diagram commutes to the right
    -- That is, applying f to the inclusion function ι₂ results
     in i₂
    {
      apply funext,
      intro x,
      exact (inr x),
    },
  },

  -- Prove the second subgoal
  {
    intros g H,
    apply funext,
    intro x,
    cases x,

    -- Prove the case where x = sum.inl x
    {
      change (g (sum.inl x)) with ((λ (p : A ⨁ B), g p) (sum.inl
     x)),

      -- Define and prove lemma H₁'
      -- It states that for any x ∈ A, applying g to (sum.inl x)
     yields (i₁ x)
      have H₁' : (λ (p : A ⨁ B), g p) (sum.inl x) = i₁ x,
      {
        rw ←H.1,
        refl,
      },
      rw H₁',
    },

    -- Prove the case where x = sum.inr x
    {
      change (g (sum.inr x)) with ((λ (p : A ⨁ B), g p) (sum.inr
     x)),

      -- Define and prove lemma H₂'
      -- It states that for any x ∈ A, applying g to (sum.inr x)
     yields (i₂ x)
      have H₂' : (λ (p : A ⨁ B), g p) (sum.inr x) = i₂ x,
      {
        rw ←H.2,
        refl,
      },
      rw H₂',
    },
  },
end

def binary_coproduct_in_Set (A B : Set.C₀) : binary_coproduct Set
    A B :=
{
  C_p := A ⨁ B,
  ι₁ := sum.inl,
  ι₂ := sum.inr,
  property := disjoint_union_binary_coproduct_in_Set A B,
}
```

References

[1] E. Cheng, *How to bake pi: An edible exploration of the mathematics of mathematics*. Basic Books, 2015.

[2] The mathlib community, "The Lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, 2020, pp. 367–381. [Online]. Available: https://doi.org/10.1145/3372885.3373824

[3] J. Z. S. Hu and J. Carette, "Formalizing category theory in agda," in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 327–342. [Online]. Available: https://doi.org/10.1145/3437992.3439922

[4] A. Timany and B. Jacobs, "Category Theory in Coq 8.5," in *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), D. Kesner and B. Pientka, Eds., vol. 52. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 30:1–30:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6000

[5] S. MacLane and S. Eilenberg, "General theory of natural equivalences," *Transactions of the American Mathematical Society*, pp. 231–294, 1945.

[6] B. Coecke, "Introducing categories to the practicing physicist," in *What is category theory*. Polimetrica Monza, 2006, pp. 45–74.

[7] S. Majid, "Some physical applications of category theory," in *Differential Geometric Methods in Theoretical Physics: Proceedings of the 19th International Conference Held in Rapallo, Italy 19–24 June 1990*. Springer, 1991, pp. 131–142.

[8] T. Haruna, "An application of category theory to the study of complex networks," *Int. J. Comp. Anti. Sys*, vol. 23, pp. 146–157, 2011.

[9] Z. Król, "Category theory and philosophy," in *Category theory in physics, mathematics, and philosophy*. Springer, 2019, pp. 21–32.

[10] B. C. Pierce, "A taste of category theory for computer scientists," 2 2011. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/A_taste_of_category_theory_for_computer_scientists/6602756

[11] S. C. Gammon, "Notions of category theory in functional programming," Ph.D. dissertation, University of British Columbia, 2007.

[12] D. Borsatti, W. Cerroni, and S. Clayman, "From category theory to functional programming: A formal representation of intent," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. IEEE, 2022, pp. 31–36.

[13] B. Ahrens and K. Wullaert, "Category theory for programming," *arXiv preprint arXiv:2209.01259*, 2022.

[14] H. Geuvers, "Proof assistants: History, ideas and future," *Sadhana*, vol. 34, pp. 3–25, 2009.

[15] M. Research. (2023) What is lean? [Online]. Available: https://www.microsoft.com/en-us/research/project/lean/

[16] S. Chakraborty, "Curry-howard-lambek correspondence," *University of Calgary: Calgary, AB, Canada*, 2011.

[17] J. Avigad, L. De Moura, and S. Kong, "Theorem proving in lean," *Online: https://leanprover. github. io/theorem_ proving_ in_ lean/theorem_ proving_ in_ lean. pdf*, 2021.

[18] G. Hutton, *Programming in haskell*. Cambridge University Press, 2016.

[19] M. F. Story, "Principles of universal design," *Universal design handbook*, vol. 2, 2001.