A Computer-Checked Library of Category Theory

Functors and F-Coalgebras

Pedro Henrique Brandão de Araujo[1]

Supervisor(s): Benedikt Ahrens[1]**, Lucas Escot**[1]

[1]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: Pedro Henrique Brandão de Araujo
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Lucas Escot, Kaitai Liang

Abstract

This research project aims to develop a computer-checked library of category theory within the Lean proof assistant, with a specific emphasis on concepts and examples relevant to functional programming. Category theory offers a robust mathematical framework that allows for the abstraction and comprehension of concepts across diverse fields, including computer science. Additionally, the project will explore the application of final coalgebras, particularly in the context of understanding infinite data structures. By creating a formalized category theory library within Lean, our objective is to enhance our understanding of functional programming and programming language concepts. Moreover, we aim to facilitate the study of these topics by providing a comprehensible library and a rigorous foundation for program reasoning, thereby benefiting researchers and practitioners in the field.

1 Introduction

Category theory and functional programming are two interconnected fields that have gained significant attention in computer science and mathematics [1]. Category theory provides a mathematical language for comprehending and abstracting concepts, allowing the study of relationships and structures across diverse domains, including computer science. On the other hand, functional programming offers a paradigm that emphasizes the use of pure functions and immutable data structures to develop reliable and modular software.

Computer proof assistants offer a formal environment where we can interactively construct and verify definitions, theorems, proofs, and examples based on rigorous mathematical foundations. By formalizing category theory concepts in a computer-checked environment, we enhance the reliability and correctness of our implementations, thereby mitigating the risk of introducing errors [2].

This paper presents a research project focused on creating a computer-checked library of category theory within the Lean proof assistant, with a specific emphasis on concepts and examples relevant to functional programming. In this particular paper, we will place a specific focus on coalgebras and their application to infinite data structures. Coalgebras provide a way to model and reason about potentially infinite sequences or structures [3]. By exploring coalgebras within the context of category theory and functional programming, we aim to shed light on their practical significance and implications.

The key question we aim to address is ”How can define streams in a computer-checked library of category theory in the Lean proof assistant?” With that, we must also answer the following sub-questions:

- What are the necessary formalizations, definitions, and proofs required to establish a solid foundation for category theory, and coalgebra, within the Lean environment?
- How can the library facilitate the exploration and understanding of category theory concepts and their practical implications in the context of functional programming?
- How does the library compare to existing works regarding comprehensiveness, correctness, and usability?

The primary objective of this project is to provide a comprehensive understanding of category theory concepts and their application in functional programming to the participating students. By developing a library that formalizes these concepts, the project also aims to create a valuable educational resource. This resource will empower students and researchers to explore the depths of category theory and its practical significance in the realm of functional programming.

While this work may not be considered groundbreaking, it contributes to an existing body of libraries that serve a similar purpose. However, our library distinguishes itself by prioritizing understandability and educational value over sheer generality and completeness. Unlike many other libraries that aim to cover a wide range of topics, our primary focus is on the practical application of category theory to functional programming. By emphasizing clarity and educational value, our library aims to provide a resource that facilitates a deeper understanding of the relationship between category theory and functional programming, making it accessible and valuable to students and researchers in the field.

The subsequent sections of this paper will outline the research questions, methodology, and expected contributions of this project. In Chapter 2 we will present the method we chose for implementing our library as well as how the research group collaborated and contributed. Chapter 3 will delve into the theory behind categories and their implementation in Lean. Chapter 4 will discuss functors and their implementation, and provide examples of functional programming. Chapter 5 will explore coalgebras and how they can be used to understand infinite data structures. The order of these chapters is essential, as each concept builds upon the previous one. Additionally, we will include a chapter discussing responsible research practices and another chapter addressing the challenges encountered during the project and how our library compares with prior work. Finally, we will present a comprehensive conclusion summarizing the entirety of our work and reflecting on the library we have constructed.

2 Methodology

The methodology employed for this research project involves a systematic approach to the development of a computer-checked library of category theory in the Lean proof assistant, with a specific emphasis on concepts and examples relevant to functional programming.

The project began with an extensive review by the students of the existing literature on category theory, functional programming, and their intersection, in order to learn and understand the concepts of the field. This served as the foundation for identifying the specific areas of category theory to be covered in the library, focusing on their relevance to functional programming and potential impact in the field. Concurrently we also studied and experimented with Lean, which is a language we didn't have previous experience with.

2.1 Lean

Computer proof assistants play a crucial role in enabling the formal verification of mathematical theorems and ensuring the correctness of complex systems [2]. These software tools provide an environment where users can interactively construct and verify proofs, define formal specifications, and establish the properties of mathematical objects or programs. By leveraging proof assistants, researchers can achieve a higher level of confidence in the accuracy and validity of their work.

In the context of this project, a proof assistant is an indispensable tool. It allows us to construct and verify proofs within a formal framework, providing a solid foundation for our exploration of category theory and functional programming concepts. By using a proof assistant, we can ensure that our proofs adhere to the rules of inference and type checking, minimizing the risk of errors and logical inconsistencies. This level of rigor is essential for building a reliable and trustworthy library of category theory in the context of functional programming.

Lean, as a well-established and actively maintained proof assistant, offers a robust platform for our project. It has been developed over many years and benefits from extensive community support [4; 5]. Lean provides a high-level language, interactive proof development, a powerful type system, and a wide range of libraries. These features make Lean an excellent choice for formalizing category theory and functional programming concepts, as it facilitates rigorous verification and enables insightful exploration of the subject matter.

Furthermore, it is worth noting that numerous category theory concepts have already been extensively explored in other proof assistants, such as Agda [6; 7; 8]. These explorations are not limited to research papers and formalized developments but also encompass educational resources, such as lecture notes and videos. In the case of Lean, there is even an official category theory library available[9]. However, it is important to recognize that the focus of this library is primarily on providing a practical tool for experienced users, rather than serving as an educational resource for learning category theory from scratch. As a result, individuals who already possess a solid understanding of category theory concepts can effectively leverage Lean's library, but it may not be the most suitable resource for newcomers aiming to learn category theory.

Overall, the choice of Lean as our proof assistant provides us with a powerful and well-supported tool for formalizing category theory in the context of functional programming. Its robust features and extensive libraries, combined with the rigorous verification capabilities, ensure the reliability and soundness of our work, leading to a valuable resource for researchers, students, and practitioners in the field.

2.2 Reproducing our work

This section outlines the steps required to reproduce our work and provides information on accessing and compiling our code base.

To reproduce our work locally, please follow the instructions below:

- Obtain the source code: The source code for our library is hosted on a public Git repository. You can clone the repository using the following Git command: `git clone https://github.com/sgciprian/ct`
- Install Lean: To compile and run the code, you will need to have the Lean proof assistant installed on your machine, we used Lean 3 for this project[1]. You'll also need to install *leanproject*[1].
- Build the code: Once Lean is installed, navigate to the cloned repository directory and use the leanproject build command to build the library. This command will only return ok if all code in the library is error-free. `leanproect build`

In addition, we have implemented a pipeline on Gitlab to ensure that our code base consistently compiles. This pipeline automates the compilation process and helps maintain the integrity of the library. You can access and review the pipeline configuration files in the repository to gain insights into our continuous integration setup.

If you wish to test a specific part of our code, Lean offers an online platform called the Lean Web Editor[2] that can be used for testing and experimenting with our library without the need for local installation. However, it's important to note that the availability and functionality of the Lean Web Editor may vary, and it is not guaranteed to support all the features and dependencies required by our library.

2.3 Two phases

The library development process is divided into two phases. The first phase involves creating a common "core" of the library that establishes a solid basis for subsequent developments. The team collaboratively implements the fundamental concepts, as well as examples, of category theory in the Lean proof assistant, ensuring correctness and verification through the formal methods provided by the tool. Regular meetings and discussions are conducted to address any challenges, make decisions, and coordinate the development efforts.

In the second phase, each team member selects a specific area of category theory, such as universal properties, functors and algebras, functors and coalgebras, monads, or adjunctions. They undertake an in-depth exploration of their chosen area, studying relevant literature and identifying key definitions, theorems, and examples to be included in the library. The implementation and formalization of these selected concepts are carried out in collaboration with the team, ensuring coherence and compatibility with the existing core library. In this paper, the second phase will focus on the examination of functors and final coalgebras.

Throughout the project, effective coordination and communication play a crucial role. Regular meetings and discussions are held to facilitate information exchange, foster collaboration, and ensure alignment among team members. Online collaboration tools and version control systems are utilized

[1] https://leanprover-community.github.io/get_started.html

[2] https://leanprover-community.github.io/lean-web-editor/

to manage the development process and track progress efficiently. Clear communication channels are established to address any challenges, provide support, and ensure a cohesive and synchronized development effort.

Evaluation and iteration are essential components of the project. The library's correctness, usability, and practical relevance to functional programming are regularly evaluated and tested. Feedback from team members and our supervisors is actively sought and incorporated into the library's development, allowing for iterative improvements and refinements.

By following this methodology, the research project aims to achieve its objectives of creating a computer-checked library of category theory in the Lean proof assistant. The collaborative nature of the project, coupled with the rigorous verification provided by the proof assistant, ensures the reliability and quality of the library, thereby providing a valuable resource for researchers, practitioners, and students interested in exploring the connections between category theory and functional programming.

3 Category Theory

Category theory is a branch of mathematics that provides a powerful framework for understanding and abstracting concepts, relationships, and structures across various domains. It has found significant applications in computer science, physics, biology, and other disciplines as a unifying language for describing and studying phenomena.[10; 11; 12]

One of the key strengths of category theory lies in its ability to capture common patterns and structures across different domains. By focusing on the relationships between objects rather than their internal details, category theory provides a high-level perspective that enables us to identify and analyze universal properties and concepts.

3.1 Definition

At the heart of category theory is the notion of a category, which consists of objects and arrows (also known as morphisms) that represent relationships between objects. The objects can be anything from sets, types, or mathematical structures, while the arrows capture transformations or mappings between these objects.

In a category, there are two fundamental operations: composition and identity. Composition allows us to combine arrows, representing the successive application of transformations. Identity arrows provide a notion of self-transformation for each object in the category. These operations satisfy certain axioms, such as associativity and identity laws, which ensure the coherence and consistency of the category structure.

Formally, a category $\mathcal{C}$ can be defined as follows, inspired by the definition used in Leinster's book "Basic Category Theory" [13]:
A category $\mathcal{C}$ consists of:

- A collection C_0 of objects.
- For every pair of objects $X, Y \in C_0$, a collection $C_0(X, Y)$ (also denoted as $Hom(X, Y)$) of morphisms from X to Y.
- For every triple of objects $X, Y, Z \in C_0$, a composition operation $\circ : C_0(Y, Z) \circ C_0(X, Y) \to C_0(X, Z)$ that allows the composition of morphisms.
- For each object $X \in C_0$, an identity morphism $1_X \in C_0(X, X)$ that serves as the neutral element for composition.

The category $\mathcal{C}$ must satisfy the following axioms:

- Associativity: For any morphisms $f \in C_0(X, Y)$, $g \in C_0(Y, Z)$, and $h \in C_0(Z, W)$, the composition $(h \circ g) \circ f$ is equivalent to $h \circ (g \circ f)$.
- Identity Laws: For any morphism $f \in C_0(X, Y)$, the compositions $f \circ 1_X$ and $1_Y \circ f$ are equal to f.

In Lean we offered the following definition for categories[3]:

```
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)
```

3.2 Examples

One notable example of a category is the Set category, where the objects are sets and the morphisms are functions between sets. Leinste [13] describes it as follows.

> There is a category Set described as follows. Its objects are sets. Given sets A and B, a map from A to B in the category Set is exactly what is ordinarily called a map (or mapping, or function) from A to B. Composition in the category is ordinary composition of functions, and the identity maps are again what you would expect.

However, in the context of functional programming, a relevant category is the category of the programming language itself, where objects are data types and morphisms are functions that convert between types. The composition works as a composition of functions and the identity is a function that does nothing. Barr and Wells define the Functional Programming Category as follows.

> [10] A functional programming language L has a category structure C(L) for which:
>
> - FPC–1 The types of L are the objects of C(L).
> - FPC–2 The operations (primitive and derived) of L are the arrows of C(L).

[3] https://github.com/sgciprian/ct/blob/main/src/category/category.lean

- FPC–3 The source and target of an arrow are the input and output types of the corresponding operation.
- FPC–4 Composition is given by the composition constructor, written in the reverse order.
- FPC–5 The identity arrows are the do-nothing operations.

We have implemented the category of sets in the file Set_category.lean[4]. And because Lean types are treated the same way as sets we will use the Set category for our future constructions and definitions of types.

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

  -- 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,
}
```

4 Functors

Since categories are also a mathematical structure we can construct the category of categories. Its objects are, obviously, the categories and the morphisms are the *functors*. A functor is a mapping between categories that preserves the structure and relationships of the objects and morphisms. It provides a way to transform categories, allowing us to analyze and compare them in a systematic manner.

Functors play a crucial role in relating different categories and studying their properties. They allow us to translate concepts and results from one category to another, uncovering hidden connections and similarities. In the context of functional programming, functors often capture the notion of structure-preserving mappings between different types.

4.1 Definition

Let C_1 denote a generic morphism $C_0(X, Y)$ in C. Formally, a functor $F : \mathcal{C} \to \mathcal{D}$ between two categories $\mathcal{C}$ and $\mathcal{D}$ consists of the following components[14]:

- A mapping $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$ that assigns each object $X \in \mathcal{C}$ to an object $F_0(X) \in \mathcal{D}$.
- A mapping $F_1 : \mathcal{C}_1 \to \mathcal{D}_1$ that assigns each morphism $f : X \to Y$ in $\mathcal{C}$ to a morphism $F_1(f) : F_0(X) \to F_0(Y)$ in $\mathcal{D}$.

To be a valid functor, the following conditions must be satisfied:

- Preservation of identities: For every object $X \in \mathcal{C}$, the identity morphism $1_X : X \to X$ in $\mathcal{C}$ is mapped to the identity morphism $1_{F_0(X)} : F_0(X) \to F_0(X)$ in $\mathcal{D}$.
- Preservation of composition: For any pair of morphisms $f : X \to Y$ and $g : Y \to Z$ in $\mathcal{C}$, the composition $g \circ f : X \to Z$ is mapped to the composition $F_1(g) \circ F_1(f) : F_0(X) \to F_0(Z)$ in $\mathcal{D}$.

In Lean, we offered the following definition for functors[5]:

```
structure functor (C D : category) :=
  (map_obj : C → D)
  (map_hom : Π {X Y : C} (f : C.hom X Y),
        D.hom (map_obj X) (map_obj Y))
  (id : ∀ (X : C), map_hom (C.id X)
        = D.id (map_obj X))
  (comp : ∀ {X Y Z : C} (f : C.hom X Y)
        (g : C.hom Y Z), map_hom (C.compose g f)
        = D.compose (map_hom g) (map_hom f))
```

4.2 Examples

Functors play a fundamental role in functional programming, enabling the transformation and manipulation of data structures while preserving the structure and relationships between them. They provide a powerful abstraction that allows programmers to write generic code and express computations in a composable and reusable manner. In this section, we will discuss specific examples and use cases of functors, highlighting their practical significance and implications in functional programming.

In programming, we have what are called type constructors, things that take types and return new types. The most used example is *Lists*, they are a constructor because a List

[4] https://github.com/sgciprian/ct/blob/main/src/instances/Set_category.lean

[5] https://github.com/sgciprian/ct/blob/main/src/functors/functor.lean

doesn't exist as a type by itself, but given another type, such as *Int*, it can construct the *List_of_Int* type.[15]

The List Functor: One of the most commonly encountered functors in functional programming is the List functor. In this case, the functor maps objects of type X $\mapsto$ List(X) and maps the morphisms using what most languages call the *maplist* (or sometimes just *map*) function. It applies a function over each element of a list, producing a new list as a result. This allows for transformations and operations to be applied uniformly to every element in the list, promoting code reusability and modularity.

Here is our code for the definitions of *List*, *maplist* (called List.fmap) and the List functor:[6]

```
inductive List (α: Type) : Type
  | nil : List
  | cons (head: α) (tail: List) : List

def List.fmap {α β: Type} (f: α → β) :
                        List α → List β
  | List.nil := List.nil
  | (List.cons head tail) :=
       List.cons (f head) (List.fmap tail)

def List.functor : Set  Set :=
{
  map_obj := λ A, List A,
  map_hom := λ _ _ f, List.fmap f,
}
```

Objects are mapped to Lists and morphisms are mapped using induction over the input List. In the base case, we return Nil, and when mapping Cons (head, tail) we apply the given function to the head element and recursively continue with the tail. For the proofs of the identity and composition preservation rules refer to our file in the repository, but in both cases, a proof by induction is used to leverage the fact List is an inductive type.

The Maybe Functor: Another important functor in functional programming is the Maybe functor, also known as the Option functor. It is used to handle optional values or computations that may fail to produce a result. The Maybe functor wraps a value, and applying a function to it results in a new wrapped value. This allows for concise and safe handling of optional values, eliminating the need for explicit null checks or error-prone code. The Maybe functor is particularly useful for writing robust and error-resistant code in functional programming.

Here is our code for the definitions of *Maybe*, Maybe.fmap and the Maybe functor:[7]

```
inductive Maybe (α : Type*)
  | none : Maybe
  | some : α → Maybe

def Maybe.fmap {α β : Type*} (f : α → β) :
                        Maybe α → Maybe β
  | Maybe.none := Maybe.none
  | (Maybe.some x) := Maybe.some (f x)

def Maybe.functor : Set  Set :=
{
  -- Objects are mapped to the Maybe type.
  map_obj := λ A, Maybe A,
  -- Morphisms are mapped by chosing between two cases.
  -- 1) Given input None, None is returned.
  -- 2) Given input Some a, Some (f a) is returned.
  map_hom := λ _ _ f, Maybe.fmap f,
}
```

Objects are mapped to the Maybe type and morphisms are mapped by choosing between two cases, given input None, None is returned, and given input Some a, Some (f a) is returned. Again, for the proofs of the identity and composition preservation rules refer to our file in the repository, but once more in both cases, a proof by induction is used to leverage the fact Maybe is an inductive type, both the base case and the inductive case have straight forward solutions and Lean solves them by itself.

These are just a few examples of how functors are used in functional programming. They provide a mechanism for encapsulating and transforming data, allowing for the creation of generic and reusable code. Functors enable programmers to write code that is concise, expressive, and modular, promoting the principles of functional programming such as immutability, purity, and composability.

The practical significance of functors in functional programming lies in their ability to abstract away the specific details of data structures and computations, focusing on the underlying patterns and transformations. By leveraging the power of functors, programmers can write code that is easier to understand, test, and maintain. Additionally, the use of functors can lead to more efficient and optimized code, as they allow for high-level optimizations and transformations to be applied uniformly across different data structures.

By incorporating the concept of functors into our computer-checked library of category theory in Lean, we provide a formalized and rigorously verified foundation for reasoning about functors in functional programming. This enables programmers and researchers to explore the practical implications of functors, understand their underlying principles, and leverage them effectively in their software development processes.

5 F-coalgebras

In category theory, F-coalgebras provide a dual perspective to F-algebras (this concept will be explained below), focusing on the generation and observation of structures rather than their construction and manipulation. A coalgebra represents a system or process that produces an infinite or potentially infinite sequence of states or observations.

The formalism of F-coalgebras provides a rigorous foundation for studying infinite or potentially infinite data structures, enabling reasoning and manipulation of such structures

[6] https://github.com/sgciprian/ct/blob/main/src/instances/functors/List_functor.lean

[7] https://github.com/sgciprian/ct/blob/main/src/instances/functors/Maybe_functor.lean

in a principled manner. By incorporating the concept of F-coalgebras into our computer-checked library of category theory in Lean, we aim to provide a solid framework for modeling and reasoning about infinite data structures and systems in functional programming.

5.1 A quick note on F-algebras

Here we will give a simple explanation of F-algebras in order to properly explain, its dual, the F-coalgebras, which is our main object of study. A more detailed work on F-algebras is being produced concurrently by my colleague Rado Todorov[16]. Rutten [3] presents a good summary of algebras and F-algebras as follows.

Algebras are presented as sets with operations. An example is the algebra (N, 0, succ), where N = 0, 1, 2, . . ., (0 $\in$ N) and (succ(n) = n + 1), the algebra of natural numbers can be presented as [zero, succ] : (1 + N) $\to$ N. Where 1 = is the singleton set with element , where 1 + N is the coproduct, that is, the disjoint union, of 1 and N; and where zero : 1 $\to$ N, zero() = 0, succ : N $\to$ N, succ(n) = n + 1

Presented in this manner, the algebra of natural numbers becomes an instance of the F-algebra categorical definition. Let $\mathcal{F}$: $\mathcal{C}$ $\to$ $\mathcal{C}$ be a functor from a category $\mathcal{C}$ to itself. An F -algebra is a pair (A, α) consisting of an object A and an arrow α : $\mathcal{F}$ (A) $\to$ A. We call F the type, A the carrier, and α the structure map of the algebra (A, α).

Defining the functor N : Set $\to$ Set, for every set X, by N(X) = 1 + X, we observe that (N, [zero, succ]) is an N-algebra.

5.2 Definition

As we said, F-coalgebras are the dual concept of F-algebras. Rutten [3] defines duality as "the elementary process of 'reversing the arrows' in a diagram. If this diagram was used to give a definition or to express a property, then reversing the arrows leads to a new definition or a new property, which is called the dual of the original one."

A F-coalgebra (C, ϕ) also consists of a carrier object C, and a structure-preserving morphism that describes the transition ϕ : C $\to$ $\mathcal{F}$ (C). Note how the morphism is "flipped" in relation to the algebra morphism. The morphism of an F-algebra can be understood as a mechanism for combining elements. Similarly, in the context of an F-coalgebra (C, ϕ), the structure map ϕ provides insights into how we can break down or unfold the elements of the coalgebra.

We also have a way to define morphisms between F-coalgebras, these are called coalgebra homomorphisms. Let $\mathcal{F}$: $\mathcal{C}$ $\to$ $\mathcal{C}$ be a functor and let (C, ϕ) and (D, ψ) be F-coalgebras. A homomorphism from (C, ϕ) to (D, ψ) is a morphism f : C $\to$ D such that $\psi \circ$ f = $\mathcal{F}$(f) $\circ$ ϕ

In our library we have defined F-coalgebras and their homomorphisms as:[8]

```
structure coalgebra {C: category}
        (F : functor C C) :=
    (object : C.C₀)
    (morphism : object → (F.map_obj object))
```

[8]https://github.com/sgciprian/ct/blob/main/src/coalgebras/coalgebra.lean

```
structure f_coalgebra_homomorphism
        {C : category} {F : functor C C}
        (A B : coalgebra F) :=
  (morphism : C.hom A.object B.object)
  (proof : B.morphism ∘ morphism
        = (F.map_hom morphism) ∘ A.morphism)
```

5.3 Examples

Coalgebras find practical applications in various areas, including the modeling of infinite data structures, stream processing, probabilistic systems, and reactive systems[3]. In functional programming, coalgebras provide a powerful tool for reasoning about and manipulating infinite or potentially infinite data.

In the context of functional programming, coalgebras are particularly relevant for understanding and working with infinite data structures. Examples of infinite data structures include streams, lazy lists, and coinductive types. By modeling these structures as coalgebras, we can capture their generation processes, observe their elements or states, and define operations on them in a compositional and modular manner.

For instance, consider the coalgebraic representation of an infinite stream of integers. The carrier object is the set of all streams of integers, and the transition function describes how to generate the next element of the stream given the current state. By working with coalgebras, we can define operations on streams, such as unfolding, mapping, ziping and filtering.

Unfortunately, coinductive data types[17] and corecursion are not natively supported in Lean and this presented a significant challenge when it came to defining and reasoning about coinductive data types such as streams. Since Lean primarily focuses on inductive reasoning, which is well-suited for reasoning about finite structures, we had to find alternative approaches to work with potentially infinite objects. For stream, we were able to define it using natural numbers, where given the number n the stream would return the nth element. Unfortunately, for trees, a similar approach would not be possible therefore we were not able to extend our proofs for the data structure, however, there is a file illustrating how a binary tree could be defined if we had coinductive types defined[9].

Our definitions of the stream data type, and its auxiliary functions, can be found in the file stream_dt.lean[10]

```
def stream (α : Type*) := nat → α

def cons {α : Type*} (a : α) (s : stream α)
        : stream α
| nat.zero       := a
| (nat.succ n) := s n

def head {α : Type*} (s : stream α) : α := s 0

def tail {α : Type*} (s : stream α) : stream α
    := λ i, s (nat.succ i)
```

[9]https://github.com/sgciprian/ct/blob/main/src/coalgebras/trees.lean

[10]https://github.com/sgciprian/ct/blob/main/src/coalgebras/stream_dt.lean

Streams are coalgebras of the functor $X \to (\mathcal{A} \times X)$, so to define the coalgebra we need to first define this functor, which can be found in the file stream_functor.lean[11]. The composition and identity preservation proofs are left out of the paper but can be found in the file, the proofs are made simple by splitting the cases of the coproduct.

```
def stream_functor (α : Type)
        :functor Types Types :=
{
  map_obj := λ X, α × X,
  map_hom := λ X Y f, λ p, (p.fst, f p.snd),
}
```

Our definitions of streams as a coalgebra can be found in the stream.lean file[12]

```
def stream_coalgebra (α : Type)
        : coalgebra (stream_functor α) :=
{
  object := stream α,
  morphism := λ s, (head s, tail s)
}
```

An important homomorphism of the stream coalgebra is the unfold higher-order function, which generates a stream from a seed element and a step function. Defined as

```
def unfolds {α β : Type*}
        : (α → β × α) → (α) → stream β
| f a nat.zero := (f a).1
| f a (nat.succ n) := unfolds f (f a).2 (n)
```

The proof that unfold is a homomorphism of any coalgebra of the functor $X \to (\mathcal{A} \times X)$ to the stream coalgebra that can be found in the stream.lean file[12] but it is too big to be added here. We started by simplifying the compositions on both sides of the proposition then applying the unfold definition on both sides and finally using the definition of tail to prove that the proposition holds.

5.4 Terminal F-coalgebras

One key concept related to coalgebras is the notion of final (or terminal) coalgebras [18]. In fact, the reason why we are interested in streams is that it is a final coalgebra, that's why we can write so many functions and operations that hold for all instances of the coinductive data type. To define final coalgebras we must first define the category of coalgebras.

We call $CoAlg(\mathcal{F})$ the category of all coalgebras of the functor $\mathcal{F}$. The objects are the F-coalgebras and the morphisms are the homomorphisms between them. It is important to note that not all $CoAlg(\mathcal{F})$ will have a terminal object, but for the case of the functor $X \to (\mathcal{A} \times X)$, we do. Here is our definition of the $CoAlg(\mathcal{F})$.[13]

```
def coalgebra_category {C: category}
    (F : functor C C) : category :=
{ C₀ := coalgebra F,
  hom := λ A B, f_coalgebra_homomorphism A B,
  id := λ A, {
    morphism := C.id A.object,
    proof := begin
      intros,
      simp [F.id, C.right_id, C.left_id],
    end
  },
  compose := λ X Y Z g f, {
    morphism := g.morphism ∘ f.morphism,
    proof := begin
      intros,
      rw C.assoc,
      rw g.proof,
      rw ← C.assoc,
      rw f.proof,
      rw C.assoc,
      repeat { rw f.proof },
      rw functor.comp,
    end
  },

  left_id :=
    begin
      intros,
      simp [functor.id, C.left_id],
      cases f,
      refl,
    end,
  right_id :=
    begin
      intros,
      simp [functor.id, C.right_id],
      cases f,
      refl,
    end,
  assoc :=
    begin
      intros,
      simp [C.assoc],
    end,
}
```

With the category $CoAlg(\mathcal{F})$ we define the final coalgebra as the coalgebra (ν, out), such that for any coalgebra (A, α) there is one, and only one, homomorphism (A, α) $\to$ (ν, out), and this unique morphism is called *anamorphism*

So, in Lean, we define a final coalgebra as a collection of F-coalgebra, the anamorphism and a proof that this anamorphism is unique.[14]

```
structure final_coalgebra {C : category}
        (F : functor C C) :=
  (obj : coalgebra F)
  (anamorphism : Π (A : coalgebra F),
            (coalgebra_category F).hom A obj)
  (unique :  ∀ {A : (coalgebra_category F).C₀}
```

[11] https://github.com/sgciprian/ct/blob/main/src/coalgebras/stream_functor.lean

[12] https://github.com/sgciprian/ct/blob/main/src/coalgebras/stream.lean

[13] https://github.com/sgciprian/ct/blob/main/src/coalgebras/coalgebras_category.lean

[14] https://github.com/sgciprian/ct/blob/main/src/coalgebras/final_coalgebra.lean

```
    (f : (coalgebra_category F).hom A obj),
      f = anamorphism A)
```

Finally, we tried to prove that stream is indeed a final coalgebra of the category $CoAlg(X \to (\mathcal{A} \times X))$ with the unfold homomorphism being the anamorphism. However, we were not able to finish this proof in time. We had to prove that any homomorphism applied to a generic coalgebra would be equal to applying the anamorphism (in this case unfold using the coalgebra morphism). We used induction on the stream and for the base case we rewrote the general homomorphism and applied the proof that the unfold homomorphism commutes. For the inductive step we tried a similar approach and after doing the substitutions we end with

```
⊢ f_morphism (A.morphism x).snd n
    = unfolds A.morphism (A.morphism x).snd n
```

Although it seems obvious that this should be the inductive hypothesis we were not able to make lean induce that way, it applies induction only to n, so our hypothesis is

```
IH : f_morphism x n = unfolds A.morphism x n
```

Not being able to reconcile these I had to leave the proof undone. The tactics used can be seen below or in the streams.lean file.[12]

```
def proof_stream_is_final {α : Types.C₀}
    : final_coalgebra (stream_functor α) :=
{
  obj := stream_coalgebra α,
  anamorphism := unfold_homomorphism,
  unique :=
begin
  intros A f,
  rw unfold_homomorphism,
  cases f,

  have h : f_morphism = unfolds A.morphism
  :=
  begin
    funext x,
    funext n,

    induction n with n ih,

    case nat.zero{ -- Case n = 0
      rw unfolds,
      have h : f_morphism x 0 =
        (Types.compose prod.fst (Types.compose
            (stream_coalgebra α).morphism
                f_morphism)) x := by refl,
      simp [h],
      rw [f_proof],
      refl,
    },
    case nat.succ{ -- Case n > 0
      rw [unfolds],
      -- unfold unfolds at IH,
      -- rw [IH],

      have h : f_morphism x n.succ =
        (Types.compose prod.snd (Types.compose
            (stream_coalgebra α).morphism
                f_morphism)) x n := by refl,
      simp [h],
      rw [f_proof],

      have h : Types.compose prod.snd
        (Types.compose (
            (stream_functor α).map_hom
            f_morphism
        ) A.morphism) x n
        = f_morphism (A.morphism x).snd n
        := by refl,
      simp [h],

      cases (A.morphism x) with a s,
      simp,

      -- exact inductive_hypothesis,
      sorry,
    },
  end,
  simp [h],
end
}
```

5.5 Anamorphism examples

We also provided practical examples of anamorphisms.

The first one is the function nats : $N \to \text{Stream}(N)$ which returns the stream of all natural numbers starting with the number given as the argument. First we define a function succ_pair : $N \to N \times N$ that takes a number n and pairs it with its successor succ_pair(n) = (n, n+1). With this we can define the nats coalgebra which is a F-coalgebra of the $X \to (N \times X)$ functor, with N as the carrier object and succ_pair as the morphism. Now we define the nats homomorphism as the unfold function using the succ_pair function as the step function. It is interesting to note how it corroborates with our definition of unfold as a general homomorphism for any coalgebra to the stream coalgebra. We don't need to prove this homomorphism is unique since all homomorphisms that go to the stream coalgebra are unique since it is a terminal coalgebra. Therefore unfold composed with succ_pair is the anamorphism that satisfies the above description.[15]

```
def succ_pair : N → N × N := λ n, (n, n + 1)

def nat_coalgebra
    : coalgebra (stream_functor N) :=
{
  object := N,
  morphism := succ_pair
}

def nats_homomorphism
    : f_coalgebra_homomorphism nat_coalgebra
      (stream_coalgebra N) :=
```

[15] https://github.com/sgciprian/ct/blob/main/src/coalgebras/nats_stream.lean

```
{
  morphism := unfolds succ_pair,
  proof := by refl,
}
```

The next one is the function zip : Stream(A)×Stream(B) → Stream(A × B) that zips the argument streams together. First we define a function pair_head_tail : (stream α × stream β) → (α × β) × (stream α × stream β) that takes a pair of streams s and returns a product of products of the heads and tails of the streams pair_head_tail(s) = ((head s.1, head s.2), (tail s.1, tail s.2)). With this we can define the zip coalgebra which is a F-coalgebra of the X → ((α × β) × X) functor, with the product of streams (stream α × stream β) as the carrier object and pair_head_tail as the morphism. Now we define the zip homomorphism as the unfold function using the pair_head_tail function as the step function. It is interesting to note how it again corroborates with our definition of unfold as a general homomorphism for any coalgebra to the stream coalgebra. We don't need to prove this homomorphism is unique since all homomorphisms that go to the stream coalgebra are unique since it is a terminal coalgebra. Therefore unfold composed with pair_head_tail is the anamorphism that satisfies the above description.[16]

```
def pair_head_tail {α β : Type}
    : (stream α × stream β) →
        (α × β) × (stream α × stream β) :=
λ s, ((head s.1, head s.2),
    (tail s.1, tail s.2))

def zip_coalgebra (α β : Type)
    : coalgebra (stream_functor (α × β)) :=
{
  object := stream α × stream β,
  morphism := λ x, pair_head_tail x,
}

def zips_homomorphism (α β : Type)
    : f_coalgebra_homomorphism
        (zip_coalgebra α β)
        (stream_coalgebra (α × β)) :=
{
  morphism := unfolds pair_head_tail,
  proof := by refl,
}
```

6 Responsible Research

In this section, we discuss the responsible research practices taken into account throughout the project to uphold high standards of integrity, transparency, and ethical considerations.

6.1 Ethical Considerations

The nature of this research primarily revolves around logic and mathematical methods, devoid of any direct involvement with human subjects or sensitive data. As such, there are no significant ethical issues associated with this research. All the results presented in this paper are derived solely from the code developed and the principles of category theory and functional programming.

Furthermore, we have taken additional measures to ensure the transparency and integrity of our work. The entire code base, including the implemented library, is openly available for public scrutiny and audit. This accessibility allows for independent verification and verification of the absence of any malicious elements.

Throughout the research process, we have prioritized adherence to the principles of responsible research and conducted our work in a manner consistent with the highest ethical standards. Although the research does not raise any significant ethical concerns, we remain committed to upholding responsible research practices and contributing to the advancement of the field of category theory and functional programming.

6.2 Use of LLMs/AI tools

As part of our commitment to responsible research practices, I must clarify I used LLM (Language Model) tools to aid in the writing process of this paper. These tools were employed to enhance the clarity, coherence, and overall quality of the written content. By utilizing LLM tools, we were able to benefit from their language generation capabilities, including grammar and style suggestions, ensuring that the text is well-structured and effectively communicates our research findings.

It is important to note that while LLM tools provide valuable assistance in the writing process, the content and ideas presented in this paper are the results of research and critical thinking. The tools served as an aid to improve the writing, but the responsibility for the accuracy and validity of the research rests solely with the authors.

By leveraging LLM tools responsibly and in conjunction with our domain knowledge, we aimed to optimize the clarity and effectiveness of our written communication, ultimately enhancing the overall quality of this research paper.

7 Results and Discussion

In this section, we present the results of our research project, compare them to previous work, and provide a broader context for their significance. We reflect on the conclusions drawn from the results and discuss the methodology employed. Additionally, we attempt to provide possible explanations for the observed outcomes.

7.1 Results

Our research project aimed to create a computer-checked library of category theory within the Lean proof assistant, with a specific focus on concepts and examples relevant to functional programming. We successfully developed and formalized the core concepts of category theory, including categories, functors, and coalgebras, within the Lean environment.

The library we constructed demonstrates the application of category theory to functional programming by providing formal definitions, theorems, and examples. We implemented

[16] https://github.com/sgciprian/ct/blob/main/src/coalgebras/zip_ana.lean

key operations and properties associated with category theory concepts, such as composition of morphisms, functorial mapping, and unfolding of coalgebras.

7.2 Comparison to Previous Work

To evaluate the contributions of our library, we compared it to existing works in the field of category theory and functional programming. We examined other libraries and resources that cover similar topics and serve a similar purpose. While our library may not be the first of its kind, it differs by emphasizing understandability and educational value over generality and completeness.

The most popular libraries publicly available are agda-categories[7] and the category theory part of the official Lean library mathlib[9]. They, as most libraries, aim to provide a broad coverage of category theory concepts, catering to advanced users already familiar with the subject. They are very difficult to understand from just reading the code, for someone who has not studied categories yet, and are not a viable resource for starting out. In contrast, our library prioritizes clarity and pedagogy, making it accessible to students and researchers who are learning category theory and its applications to functional programming.

Furthermore, on the topic of coalgebras, it is very difficult to find a library that covers it, in fact, agda-categories[7] is the only one I could find doing so. Even in the literature, many authors dismiss the concepts of coalgebra just as the dual of algebra, without developing their concepts further[19; 20], Meijer et al.[20] for example, explain that if you consider the CPO (complete partial order) category instead of the set category you can use the same carriers for initial algebra and and terminal coalgebras, in programming this translates to how many languages deal with lazy evaluation.

Overall, our library compares favorably to existing works in terms of its focus on educational value, correctness, and usability. While it may not provide the same breadth of coverage as some comprehensive libraries, it offers a unique perspective that facilitates learning and understanding of category theory concepts in the context of functional programming.

8 Conclusions

In this paper, we presented a research project focused on the formalization of category theory concepts within the Lean proof assistant, with a specific emphasis on their application to functional programming. We developed a computer-checked library that provides formal definitions, theorems, and examples, enabling students and programmers to explore the connections between category theory and functional programming.

Throughout the project, we addressed research questions related to the necessary formalizations, definitions, and proofs required to establish a solid foundation for category theory within the Lean environment. We also explored how the library facilitates the understanding and exploration of category theory concepts and their practical implications in the context of functional programming. By comparing our library to existing works, we highlighted its unique focus on understandability and educational value.

In this particular paper, we wanted to learn how streams can be defined in the computer-checked library. We successfully defined and formalized streams within our library. We utilized the concept of coalgebras to model streams as final coalgebras in the category of sets. By defining the necessary structure maps and proving the associated properties, we established a solid foundation for representing and manipulating streams within the Lean proof assistant

In order to establish a solid foundation for category theory and coalgebra within Lean, we formalized and defined several key concepts. This included formalizing the notion of categories and their associated operations such as composition and identity morphisms. We also defined functors, which are mappings between categories, and studied their properties. Additionally, we introduced the concept of coalgebras as the dual of algebras and demonstrated their relevance in modeling infinite data structures.

Our library could play a role in facilitating the exploration and understanding of category theory concepts in the context of functional programming. By providing formal definitions, theorems, and examples focused on programming, the library serves as an educational resource that enables programmers and students to delve into category theory. The use of the Lean proof assistant ensures that the definitions and theorems adhere to rigorous standards, enhancing confidence in their accuracy and validity. Moreover, the library's focus on understandability and pedagogy makes it accessible to individuals who are learning category theory and its practical implications.

In conclusion, this research project has successfully achieved its objectives by creating a computer-checked library of category theory within the Lean proof assistant, focusing on its application to functional programming. The results contribute to the dissemination of knowledge and provide a solid foundation for further exploration and research in this field, while also serving as an educational resource for students and researchers. The library's success in addressing the research questions validates the methodology employed and reinforces the significance of the research conducted.

References

[1] B. C. Pierce, "A taste of category theory for computer scientists," 2 2011.

[2] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl, and J. Limperg, "The hitchhiker's guide to logical verification," 2020.

[3] J. Rutten, "The method of coalgebra: exercises in coinduction," 2019.

[4] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *Automated Deduction - CADE-25* (A. P. Felty and A. Middeldorp, eds.), (Cham), pp. 378–388, Springer International Publishing, 2015.

[5] 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.

[6] 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*, CPP 2021, (New York, NY, USA), p. 327–342, Association for Computing Machinery, 2021.

[7] J. Z. S. Hu and J. Carette, "agda-categories." https://github.com/agda/agda-categories, 2021.

[8] D. Peebles, J. Deikun, U. Norell, D. Doel, D. Jahandarie, and J. Cook, "categories," *GitHub repository*, 2019.

[9] T. mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, (New York, NY, USA), pp. 367–381, Association for Computing Machinery, 2020.

[10] M. Barr and C. Wells, *Category theory for computing science*, vol. 1. Prentice Hall New York, 1990.

[11] B. Coecke and É. Paquette, *Categories for the Practising Physicist*, pp. 173–286. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[12] R. Mary, "Memory evolutive systems: Hierarchy, emergence, cognition," in *Memory Evolutive Systems* (A. C. Ehresmann and J.-P. Vanbremeersch, eds.), vol. 4 of *Studies in Multidisciplinarity*, pp. 1–386, Elsevier, 2007.

[13] T. Leinster, *Basic category theory*, vol. 143. Cambridge University Press, 2014.

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

[15] D. E. Rydeheard, "Functors and natural transformations," in *Category Theory and Computer Programming: Tutorial and Workshop, Guildford, UK September 16–20, 1985 Proceedings*, pp. 43–50, Springer, 2005.

[16] R. Todorov, "A computer-checked library of category theory: Defining functors and their algebras." 2023.

[17] T. Hagino, "Codatatypes in ml," *Journal of symbolic computation*, vol. 8, no. 6, pp. 629–650, 1989.

[18] P. Aczel and N. Mendler, "A final coalgebra theorem," in *Category Theory and Computer Science: Manchester, UK, September 5–8, 1989 Proceedings*, pp. 357–365, Springer, 2005.

[19] G. Malcolm, "Data structures and program transformation," *Science of computer programming*, vol. 14, no. 2-3, pp. 255–279, 1990.

[20] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Conference on functional programming languages and computer architecture*, pp. 124–144, Springer, 1991.