A computer-checked library of category theory

Defining functors and their algebras

Rado Todorov

Supervisor(s): 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: Rado Todorov
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Lucas Escot, Kaitai Liang

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Category theory is a branch of abstract mathematics that aims to give a high-level overview of relations between objects. Proof assistants are tools that aid in verifying the correctness of mathematical proofs. To reason about category theory using such assistants, fundamental notions have to be defined. Computer-checked libraries contain all relevant structures and theorems in an accessible way for end users. However, current libraries of category theory are not welcoming to people without in-depth domain knowledge. This paper introduces a library of category theory tailored towards newcomers to the field as well as the learning journey of the authors. We describe the project's structure, design choices and provide examples of the main features. Moreover, a detailed overview is provided of F-algebras and their relation with inductive data types found in functional programming languages. Construction and evaluation of types like lists and binary trees can be defined in terms of algebras. They provide a general framework for recursion over these types which allow us to reason about them with simple functions.

1 Introduction

Category theory addresses mathematical structures and allows us to formally describe their relations. Definitions of such structures are very convenient for capturing a specific behaviour without worrying about the concrete implementation. This high level of abstraction allows for reasoning about related concepts as a single entity by exploring their common root.

Notions from category theory have been applied in computer science to formally describe and reason about problems like network management [1]. In functional programming, languages interacting with encapsulated objects can be done by incorporating Monads [2, Section 2.7]. Utilizing category theory has also been shown to produce concise syntax that can capture desired properties in domain-specific languages [3].

Reasoning about mathematics is a tedious process, prone to human errors. Proof assistants are software tools that aid with rigorously proving mathematical theories [4, Section 3]. They rely on a type theory to enable the encoding of mathematical proofs and laws. The correctness of proofs is ensured as the assistant verifies that each step is sound. Thus, a library of category theory implemented in a proof assistant, containing fundamental structures and theorems, can ease reasoning about category theory.

Previous work which addresses the problem has been done by researchers like J.Z. Hu et al. in the language Agda [5] and F. Genovese et al. in the language Idris [6]. Defining categories and morphisms in code to express most of the theoretical constructs, while also ensuring respectable performance is a difficult task. Design suggestions have been presented by J. Gross et al. [7]. However, due to the abstract nature of the topic, most such libraries are not welcoming for newcomers as their construction tends to be too complex and lacks worked-out examples.

This paper presents an effort to create an accessible library of category theory in the proof assistant Lean [8]. The library contains fundamental notions of category theory supported by examples related to computer science. Moreover, the design choices and development process are reported.

In order to achieve these goals this paper is structured as follows. A description of the research process and the resulting library is presented in Section 2. Section 3 provides background information about Lean. Details about the basic constructs in the library are shared in Section 4. The definition of Functor Algebras is the focal point of Section 5. Design choices are discussed in Section 6. Section 7 focuses on the reproducibility of the project and the integrity of the research. Finally, section 8 will summarize the conducted research work and touch upon possible future improvements.

2 Methods and Structure

The construction of complex notions in category theory relies on simpler ones. Due to this, the research process has been split into two parts. The fundamental structures of category theory have been implemented as a group endeavor among the 5 group members of the research group during part one. They include the definitions of category, functor, and natural transformations, with examples like the category of Sets and the Maybe functor. For the latter part of the project different areas have been explored individually. This paper presents notions related to F-algebras - their theoretical definition, implementation and examples.

Creating the library has been an iterative process. The development process can be summarized by a cycle of acquiring knowledge in a specific topic via literature reviews (e.g. definition of a category), writing a proper definition and proving interesting examples. The correctness of the final product has been ensured by employing a pipeline that performs an inspection of the code base.

The library is divided into folders by topics, due to broader concepts consisting of multiple structures and supporting theorems. Default files containing imports from each topic are positioned in the root of the file structure to ease the importing of code structures when developing. A folder containing the keyboard commands for the used notations and valuable resources for learning category theory is also part of the repository.

3 Introduction to Lean

Proof assistants like Lean are built based on the Curry–Howard correspondence which is a direct relation between programs and mathematical proofs. Therefore, they allow us to present proofs using programs and the compilation of the code can be interpreted as successful proofs. Calculus of construction [9] is an extension to the Curry-Howard correspondence that introduces reasoning about quantified statements like "for all" and "some". This is the basis of Lean which is extended by including inductive types.

This paper presents code snippets of the implemented library. Thus, we now introduce some basic notations to alleviate confusion.

Arguments - In Lean explicit arguments are defined using (). However, when working with parameterized structures, we can also define implicit arguments using {}. The example below shows a function which takes another function f as input. Since f is a function, we need its input and return type. We can use implicit arguments for those and lean will infer them when applying an argument.

```
def input_function {A B : Type} (f : A → B)
    (X : A) : B := f X
```

Quantified statements - In lean we can use ∀ to form proofs which hold for all instances of a given argument. The Π notation shares this behavior and the two are interchangeable. The library presented in this paper uses ∀ for stating theorems and Π for attributes.

Universes - The type system in lean begins at $Type\ 0$ and can grow endlessly. Each consecutive $Type$ contains the previous as an element. The number in this series is also known as the *universe*. $Sort$ is essentially the same notion with the difference that $Type\ 0 = Sort\ 1$. $Sort\ 0$ is reserved for predicates. Due to this, our library makes use of $Sort$ to define types and to make our structures abstract enough, we have defined them for arbitrary universes.

Tactics - Proofs in lean are usually wrapped by $begin$ and end keywords. Defined below are some of the tactics used in our proofs and part of this paper:

`intro(s)` - introduces hypotheses to the proof

`rw` - attempts to rewrite the current goal with a provided equality.

`simp` - recursively attempts to rewrite the goal by using either a provided equality or theorems embedded in Lean.

`cases` - decomposes disjunctions

4 Basic constructs

The library aims to provide the tools necessary for most fundamental structures in category theory. Categorizing the notions has been ruled by their importance for more complex definitions, which have been researched individually. This section displays some of the relevant structures defined in our library.

4.1 Category Definition

The category as a structure is the basis of all other concepts. It represents a construct of data, containing:

- A set of objects - C_0
- A collection of structure-preserving maps between the objects of the category, also called *morphisms* and usually displayed as $X \to Y$
- An identity morphism for each object in the category, mapping it to itself
$$X \to X$$
- A composition of morphisms (depicted as $\circ$), which is a binary operator that given a morphism $g : Y \to Z$ and $f : X \to Y$ maps to $h : X \to Z$.

In addition, the following properties should be satisfied:

- Associativity - For any three morphisms f: $X \to Y$, g: $Y \to Z$ and h: $Z \to W$ we have:
$$h \circ (g \circ f) = (h \circ g) \circ f$$
- Left unit law - For any morphism f: $X \to Y$, where 1_X is the identity morphism of X we have that:
$$f \circ 1_X = f$$
- Right unit law - For any morphism f: $X \to Y$, where 1_Y is the identity morphism of Y we have that:
$$1_Y \circ f = f$$

The notion of a category is fundamental as it allows us to group related objects and describe how they can be mapped to one another. Our library implements it as a structure having all properties and data as fields. The set of objects in the category is defined as an arbitrary type and so is the set of morphisms (also known as hom-set). As stated in section 3, the attributes of the category are defined using the Π notation to be differentiated from the properties.

```
structure category :=
  (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)

  (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)
```

4.2 Set category

The category of sets is a famous category that is addressed as **Set**. To implement this category in Lean, we have taken advantage of Lean's type system, which begins at 0 and can be incremented to infinity. Lean's types can be also used as objects in functions and other expressions. Essentially, the initial type - $Type\ 0$ can be interpreted as the type containing all sets.

To prove that **Set** is indeed a category, we need to provide the appropriate data and prove that all properties of a category hold. This proof pattern is how all examples of structure definitions are proved in our library. The data of **Set** consists of:

- The set of objects for this category is the set of sets.
- The morphisms of this category represent all functions between two sets
- The identity morphism is a function, which simply maps the input to itself.
- Composition of morphisms in **Set** can be depicted as applying a function to another:

$$g \circ f = g(f)$$

Having defined the data in **Set**, we can now prove that the properties of a category hold:

- Associativity can be proved by showing

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

  Since morphisms in the category are simply functions, we use their associativity directly. This means that

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

- Left unit law can be proven by using the fact that the identity morphism is a function, which returns its input, meaning that if we apply a function to the result of the identity morphism it would be as if we apply that function directly to the input element.
- The proof to the right unit law is similar, with the difference being that the identity morphism will take and return the value, after having a function f applied.

4.3 Functor Definition

Functors are structure-preserving mappings which enable us to depict relations between categories. They are a fundamental structure in category theory, which finds many applications in computer science and specifically functional programming. For instance, in Haskell functors are used as a wrapper class:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

data List a = Nil | Cons a (List a)
instance Functor List where
    fmap f Nil = Nil
    fmap f (Cons x xs) = Cons (f a) (fmap f a)

-- Example application
fmap (+1) (Cons 1 (Cons 2 Nil))
==> Cons 2 (Cons 3 Nil)
```

Formally, a functor from category C to category D holds the following data:

- A function that maps an object in C to an object in D
- A function that maps a morphisms in C to morphisms in D

Additionally, the functor has to preserve:

- Composition - Applying a functor after a composition of functions should be the same as applying the functor mapping before the composition.

  $$F(g \circ f) = F(g) \circ F(f)$$

- Identity morphisms - For each X $\in C.C_0$ we have that applying a functor mapping to the identity morphism of X is equivalent to applying the identity morphism of the element in category D, which X is mapped to:

  $$F(1_X) = 1_{F(X)}$$

In the library, the implementation of the functor is a parameterized structure that is defined by two categories. Unlike morphisms in the definition of a category, object mapping is defined as the built-in function type as no matter the categories, its definition is unchanged.

```
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))
  ...
```

4.4 The $1 + (A \times X)$ Functor

This functor maps an element X to $1 + (A \times X)$ in the category **Set**. Breaking down the notation, 1 stands for the singleton set (the set containing only one object) and $(A \times X)$ is a binary product of an element A and X, for simplicity this can be also interpreted as a pair of the two objects. Lastly, $+$ represents a binary coproduct, also known as *Either* in Haskell. The definition of (co)products in category theory is more complex than the presented simplified explanation, a thorough description can be found in [10, Sections 5.1 and 5.2].

Functors that map object from one category back to itself are also known as *endofunctors*. It is also important to note A is taken as input when defining the functor, making it a parametarized functor. We can summarize how the function operates by presenting its mappings:

```
def list_algebra_functor (A : Set.C₀) : functor
    Set Set :=
{
  -- Objects are mapped to 1 + (A × X)
  map_obj := λ X, Either Singleton (Pair A X),
  -- Morphisms are mapped based on 2 cases:
  -- 1) The Singleton element is unchanged,
  -- 2) (A × X) is mapped to ( A × f (X) )
  map_hom :=
  begin
    intros x y f E,
    cases E,
    case Either.left : a {
      exact Either.left a
    },
    case Either.right : b {
      exact Either.right ⟨fst b, f (snd b)⟩
    }
  end,
  ...
```

The proofs for composition and identity preservation have been omitted, but readers are encouraged to inspect them in the library's files. Interestingly, this functor is at the root of how lists as a recursive type are defined. Section 5 examines this relation as well as how other similar types can be constructed. Other examples part of the library include the $1 + X$

and the $A + X \times X$ functors, which define natural numbers and binary trees respectively. The ideas and proofs regarding them are similar to those of the presented $1 + (A \times X)$ functor.

5 F-Algebras

Another definition, part of our library is the algebra of a given functor. Essentially, an algebra allows us to create expressions and do calculations with them (i.e. evaluate them). For instance, inductive data types as lists, binary trees and natural numbers, which are an essential part of all functional programming languages, can be depicted as algebras. This section aims to present the formal definition of an F-Algebra and provide intriguing instances.

5.1 Definitions

F-Algebra

An ***algebra*** of a given *endofunctor* $F : C \to C$ consists of the following data:

- An object in category C, also known as the carrier of the algebra - $X \in C.C_0$
- A moprhism - $\phi : F(X) \to X$

Due to this definition, our library defines the F-algebra as a parameterized structure

```
structure Falgebra {C:category} (F:functor C C) :=
  (object : C.C₀)
  (function : C.hom (F.map_obj object) object)
```

An algebra can be depicted as (X, ϕ), where X is the carrier and ϕ the morphism. Using a diagram, we can represent an algebra with

$$F(X) \xrightarrow{\phi} X$$

Multiple algebras exist for a given endofunctor, therefore, we can reason about algebras as objects in the category of all algebras, defined by a given endofunctor. The construction of this category requires the definition of a suitable morphism.

Homomorphism

An algebra ***homomorphism*** is the structure that properly depicts the relation between two algebras. Formally, it is a morphism between the carriers X and Y of algebras (X, ϕ) and (Y, ψ) such that the following diagram commutes:

$$\begin{array}{ccc} F(X) & \xrightarrow{\phi} & X \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle f} \\ F(Y) & \xrightarrow[\psi]{} & Y \end{array}$$

For the diagram to commute, we expect that the paths from $F(X)$ to Y are equivalent

$$f \circ \phi = \psi \circ F(f)$$

Our library implements this structure using two fields, the morphism and a proof that it commutes the above diagram, represented as the equality of the two paths.

```
structure Fhomomorphism {C:category} {F:functor C
    C} (A B : Falgebra F) :=
(morph : C.hom A.object B.object)
(square_proof :
    C.compose morph (A.function) =
    C.compose B.function (F.map_hom morph))
```

Algebra category

The category of algebras $Alg(F)$ can be implemented using the previous definitions. Here, objects are algebras and morphisms are homomorphisms. The category is parameterized by an endofunctor F just like the algebras. An important property of this category which finds practical applications is the composition, further discussion of this property will be given in section 5.3.

```
def AlgebraCategory {C:category} (F:functor C C) :
    category := {
  C₀  := Falgebra F,
  hom := λ A B, Fhomomorphism A B,
  ...
```

5.2 "List" Algebras

This is the algebra of the $1 + A \times X$ endofunctor in **Set** that can be labelled as L. Algebras in $Alg(L)$ allows us to reason about various operations applied to a list or an intermediate state. For instance, we can depict the intermediate state of applying the function *length* by the algebra

$$1 + (A \times \mathbb{N}) \xrightarrow{[0,(1+)]} \mathbb{N}$$

Here, the morphism $F(X) \to X$ maps the singleton set to 0 and the pair P to $1 +$ the second element of P. The morphism of the algebra is a simple function, which is both easy to implement and reason about. The general framework for the recursion of the types is given by the *initial* algebra in the category. This algebra has a carrier of type List A, due to which we have labeled the algebras under the "List" umbrella. The algebra with carrier List A is defined by the morphism $[nil, cons]$. Given 1, the morphism maps it to nil, while $A \times ListA$ is mapped to $cons\ A\ (ListA)$. Using nil and $cons$ as constructors is the exact way we can define a List in functional programming languages.

An object I is ***initial*** if there exists a morphism from I to any other object in the category and it is unique. For initial algebrs, these unique morphisms are named *catamorphisms* (depicted in isolation as $(\!|_|\!)$). They take ψ : the $F(X) \to X$ morphism of other algebras as input and are then presented as $(\!|\psi|\!)$. Due to this, they are a powerful tool that allows us to reason about relations between algebra carriers in terms of simple non-recursive functions. This abstracting power finds an application in all functional programming languages. For instance, when it comes to the algebra with carrier List A this unique morphism turns out to be the well-known ***fold*** function. Therefore, fold commutes the diagram

$$
\begin{array}{ccc}
1 + (A \times ListA) & \xrightarrow{[nil, cons]} & ListA \\
\Big\downarrow {\scriptstyle F(fold(\psi))} & & \Big\downarrow {\scriptstyle fold(\psi)} \\
1 + (A \times Y) & \xrightarrow[\psi = [\psi_a, \psi_b]]{} & Y
\end{array}
$$

In programming languages like Haskell, fold is written as

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

From the diagram, we can see that the morphism ψ can be split into two, just like the morphism of the initial algebra. These cases fit exactly with the definition of fold since the morphism that maps the singleton set can be used for an accumulator, and the one which handles the paired input, can be used as the function f. This is equivalent to using ψ as a single function, which expects arguments in the form $1 + A \times X$. Our library adheres to the latter specification when defining fold as it closely represents the definition of a *catamorphism*. Moreover, fold being the unique morphism to an algebra B is proven by utilizing the fact it commutes the square diagram between the two algebras and uses B's morphism recursively.

5.3 Fusion Property

The Fusion Property states that given a catamorphism ϕ between an initial algebra A and another algebra B, a morphism f between the carriers of B and the carrier of another algebra C, if we can provide a proof that their square diagram commutes, then we can prove that the composition of ϕ and f is equal to the catamorphism ψ from A to C. For instance, when working with lists we can prove that

$$map\ (g) \circ map\ (f) = map\ (g \circ f)$$

Functional programming languages can benefit from fusion by allowing users to write more expressive code whilst preserving computation speed. Code can be safely rewritten during compilation using the fusion property to remove intermediate products. When working with large amounts of data or input which requires complicated processing, both memory usage and performance are optimized. The example of map composition can be visualised by the following diagram, where it is clear to see how the intermediate result can be safely omitted.

$$
\begin{array}{ccc}
1 + (A \times ListA) & \xrightarrow{[nil, cons]} & ListA \\
\Big\downarrow {\scriptstyle F(map(\phi))} & & \Big\downarrow {\scriptstyle map(\phi)} \\
1 + (A \times ListB) & \xrightarrow[\phi]{} & ListB \\
\Big\downarrow {\scriptstyle F(map(\psi))} & & \Big\downarrow {\scriptstyle map(\psi)} \\
1 + (A \times ListC) & \xrightarrow[\psi \circ \phi]{} & ListC
\end{array}
$$

with outer arrows $F(map(\psi \circ \phi)) : 1 + (A \times ListA) \to 1 + (A \times ListC)$ and $map(\psi \circ \phi) : ListA \to ListC$.

The power of fusion extends beyond the structure of lists. It can be applied to inductive types using a generalized fold in Haskell [11, Section 4].

Our library is equipped with a proof of the fusion property, supported by examples of map composition and composition of the functions length and filter. The proof of fusion in our library utilizes the composition property of the category of algebras, resulting in a concise definition. It is also defined as a function in order to be easier to use in further proofs.

```
def fusion {ℂ : category} {F : functor ℂ ℂ}
(A : initial (AlgebraCategory F))
(B C : Falgebra F)
(f : ℂ.hom B.object C.object)
(square_proof : ℂ.compose f B.function =
ℂ.compose C.function (F.map_hom f)) :
ℂ.compose f (A.exist_morph B).morph =
    (A.exist_morph C).morph :=

begin
  -- We can utilize the Algebra category's
    composition of Homomorphisms to create one
    from A to C.
  -- Then, we apply the fact that A is an initial
    object and it's morphism is unique.
  have h := A.is_unique ((AlgebraCategory
    F).compose {morph := f, square_proof:=
    square_proof} (A.exist_morph B)),
  -- Since the 2 homomorphisms are equal, this
    implies that the underlying morphisms are also
    equal.
  cases (A.exist_morph C),
  cases h,
  simp [h],
end
```

5.4 Lambek's Theorem

The catamorphism is the unique morphism from the carrier of an initial algebra to the carrier of another algebra. However, knowing that it exists does not help us with its construction. Lambek's theorem provides a solution. The theorem states that the morphism $\phi : F(X) \to X$ of the initial algebra in the category of algebras defined by the endofunctor F is an *isomorphism*.

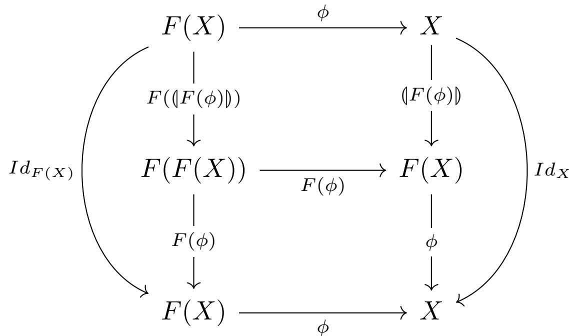
An isomorphism between objects N and M is a morphism that has a unique inverse. Moreover, their composition has to equal the identity morphism. Our library contains the definition of an isomorphism, which is an object with a morphism $N \to M$ as an argument, as well as the proof for the uniqueness of the inverse. The design decision to divide the property of composition into two fields has been made with convenience in mind because one would usually use them separately in proof steps.

```
structure isomorphism {C : category} {N M : C.C₀}
    (hom : C.hom N M) :=
  (inverse : C.hom M N)
  (idl : C.compose hom inverse = C.id M)
  (idr : C.compose inverse hom = C.id N)
```

Additionally, Lambek's theorem states that the inverse of ϕ is the catamorphism $(\!|F(\phi)|\!)$.

$$F(X) \underset{(|F(\phi)|)}{\overset{\phi}{\rightleftarrows}} X$$

The proof of Lambek's theorem involves the use of the Fusion property as well as the uniqueness of the inverse in the definition of an isomorphism. The idea of the proof revolves around showing that $(|F(\phi)|)$ is the inverse in an isomorphism with ϕ and that such inverse is unique. To construct the isomorphism, the Fusion property can be employed to prove that $\phi \circ (|F(\phi)|) = Id_X$ and $(|F(\phi)|) \circ \phi = Id_{F(X)}$. It can be visualised with this diagram



Knowing this, we are now able to give a general definition of how to construct a catamorphism. Applying Lambek's theorem to the diagram of a catamorphism, we can see that there are two paths from the carrier of the initial algebra to the carrier of any other algebra.

$$\begin{array}{ccc} F(X) & \underset{(|F(\phi)|)}{\overset{\phi}{\rightleftarrows}} & X \\ \downarrow F((|\psi|)) & & \downarrow (|\psi|) \\ F(Y) & \xrightarrow[\psi]{} & Y \end{array}$$

Therefore, a catamorphism $(|\psi|)$ can be defined as

$$\psi \circ F((|\psi|)) \circ (|F(\phi)|)$$

The proof for this has the following steps :

1. Apply the fact that $(|\psi|)$ commutes the diagram

$$\psi \circ F((|\psi|)) = (|\psi|) \circ \phi$$

2. Since ϕ and $(|F(\phi)|)$ form an isomorphmism, we can apply

$$\phi \circ (|F(\phi)|) = Id_X$$

3. Lastly, we use the category's left unit law to remove the identity morphism

$$Id_X \circ (|F(\phi)|) = (|F(\phi)|)$$

This means that given an initial algebra, we can obtain the recipe to construct the catamorphism to any algebra that is defined by the same endofunctor. This allows us to abstract the recursion and utilize non-recursive algebra morphisms that are a lot easier to reason about.

6 Discussion

A major design decision has been to utilize as many fields as possible in the structures that have been defined in the library. Analyzing other category theory libraries, this approach has been also chosen in other proof-assistants like Coq [7]. The main advantage over prioritizing arguments is that it encourages users to define instances outside their proof resulting in them being more concise and understandable. Another benefit is the shortened type signature. Lean's Infoview is a panel which aids users by informing them about the current goal of the proof and providing available data such as lemmas and variables which have been defined and can be used in the current proof. Short type signatures are essential for maintaining a clear view of the panel. Moreover, this allows for emphasis on the arguments when working with parameterized types. For instance, the current category structure has no arguments, and turning the attributes into arguments will cause a massive increase in the type signature, which will then be propagated to the signatures of other types

```
structure category :=
  (C₀      : Sort u)
  (hom     : Π (X Y : C₀), Sort v)
  (id      : Π (X : C₀), hom X X)
  ...

structure category (C₀:Sort u) (hom:Π (X Y : C₀),
    Sort v)   (id:Π (X : C₀), hom X X):=
  ...

Effect on functor signature

structure functor (C D : category)
  ...
structure functor
(C₀ : Sort u) (homC : Π (X Y : C₀), Sort v)
(idC : Π (X : C₀), homC X X)
(D₀ : Sort w) (homD : Π (X Y : D₀), Sort z)
(idD : Π (X : D₀), homD X X)
(C: category C₀ homC idC) (D: category D₀ homD idD)
```

Another important choice has been the use of structures, instead of functions, returning a boolean answer, for properties like initial or terminal objects. For instance, the initial object of a category has been defined as a structure, whereas it could have been a function, which given an object in a category returns true or false.

```
structure initial (C : category) :=
  (object : C.C₀)
  (exist_morph : Π (X : C.C₀), C.hom object X)
  (is_unique : ∀ {X : C.C₀} (f : C.hom object X),
    f = exist_morph X)

def is_initial {C : category} (object :  C.C₀) :
    Prop :=
  ...
```

This is beneficial for proofs as we can use an instance of the structure which carries all necessary data. Otherwise, the data is not bundled and cannot be used directly in proofs. For example, imagine proving that an object is initial is done with a boolean function where we prove that a unique morphism

exists. Using this in a proof, we can understand that such morphism exists, but it is unknown exactly which the desired morphism is and Lean is unable to infer this information.

Duality in category theory is the concept of *mirror images*. This means, that we could use the *opposite* category to apply duality. We have opted away from applying the duality property directly and have created the dual structures separately, similarly to [5, Section 2.3]. For instance, the dual concept of the algebra in category theory is the coalgebra, which finds its application in functional programming in the form of streams. Both have been independently implemented in the library.

Lastly, the library is intended for people without much prior domain experience. Due to this, less emphasis has been put on the automation and integration of the structures into Lean's ecosystem. This enforces users to rely on their knowledge for proofs instead of relying on the library to automate a large chunk of the work. Indubitably, it may be challenging for most newcomers to understand the theory and apply it using our library. For this reason, emphasis has been put on detailed documentation that walks through the example proofs provided by the library.

7 Responsible Research

Despite the abstract nature of category theory, in this contribution, we have aimed to present convincing evidence of the direct connection with functional programming languages. Moreover, the conducted research aims to help others have an easier entrance to this branch of mathematics. Teaching or presenting mathematics is a cornerstone of the ethics related to the field [12, Section 2].

The integrity of our work is ensured due to the nature of mathematics as the proofs provided in our library can be easily verified. Furthermore, Lean as a proof assistant aids with the prevention of incomplete or simply wrong proofs. A program that compiles can be viewed as correct. Because of this, a pipeline has been employed in the repository of the code base, which fails whenever Lean is not able to compile. Therefore, incorrect statements have been disallowed to be part of the library.

Nevertheless, the fact that a statement or proof is deemed valid does not imply that it accurately represents the objective it has been developed for. Therefore, ensuring that the definitions and related proofs are correct and abide by their purpose is a collective endeavour. Providing precise notions has been established by adhering to the definitions in [13]. Peers have reviewed all new structures before merging them with the existing collection, which has enforced quality control.

Reproducibility of the results is another fundamental characteristic of Responsible Research because transparency is essential for validating one's work. Due to this, all code has been committed to a repository hosted by the university. Thus, interested readers are encouraged to analyze or extend the provided definitions and examples.

8 Conclusions and Future Work

This paper presents the implementation of a new library of category theory in the proof assistant Lean [8]. Fundamental definitions of category and functor are defined, accompanied by examples. Following, the notion of algebra in the realm of category theory is introduced together with the category of algebras. Comparing the design choices to other implementations of libraries of category theory resulted in the discovery of many similarities.

The gap between category theory and functional programming is bridged via the introduction of a general framework of recursion over lists, known as *fold*. The paper also covers proofs regarding the removal of intermediate results when applying multiple operations to inductive data types and the construction of *fold-like* functions for arbitrary recursive types.

Due to the vast field of category theory, the library can be extended in various ways. A natural extension would be the addition of more examples like the category of categories or monoidal categories. Currently, the library prioritizes concepts closely related to computer science, but this may change in the future. Notions like groupoids or the famous Yoneda lemma can be interesting additions.

Performance and proof automation are other areas of improvement. However, this library is to be used for education purposes and these additions are not primary. Therefore, the focus will remain on providing definitions supported by examples.

References

[1] 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)*, pp. 31–36, 2022.

[2] J. M. Hill and K. Clarke, "An introduction to category theory, category theory monads, and their relationship to functional programming," tech. rep., Citeseer, 1994.

[3] J.-P. Bernardy and A. Spiwack, "Evaluating linear functions to symmetric monoidal categories," (New York, NY, USA), p. 14–26, Association for Computing Machinery, 2021.

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

[5] J. Z. Hu and J. Carette, "Formalizing category theory in agda," in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 327–342, 2021.

[6] F. Genovese, A. Gryzlov, J. Herold, A. Knispel, M. Perone, E. Post, and A. Videla, "idris-ct: A library to do category theory in idris," *arXiv preprint arXiv:1912.06191*, 2019.

[7] J. Gross, A. Chlipala, and D. I. Spivak, "Experience implementing a performant category-theory library in coq," in *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 5*, pp. 275–291, Springer, 2014.

[8] 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:*

25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25, pp. 378–388, Springer, 2015.

[9] T. Coquand and G. Huet, *The calculus of constructions*. PhD thesis, INRIA, 1986.

[10] T. Leinster, “Basic category theory,” 2016.

[11] P. Johann and N. Ghani, “Initial algebra semantics is enough!,” in *Typed Lambda Calculi and Applications: 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007. Proceedings 8*, pp. 207–222, Springer, 2007.

[12] P. Ernest, “Mathematics, ethics and purism: An application of macintyre’s virtue theory,” *Synthese*, vol. 199, no. 1-2, pp. 3137–3167, 2021.

[13] B. Ahrens and K. Wullaert, “Category theory for programming.” https://github.com/benediktahrens/CT4P.