The monad and examples from Haskell

A computer-checked library for Category Theory in Lean

Csanád Farkas[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: Csanád Farkas
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Lucas Escot, Kaitai Liang

Abstract

Category Theory is a widely used field of Mathematics. Some concepts from it are often used in functional programming. This paper will focus on the Monad and a few implementations of it from Haskell. We will also present the computer-checked library we have written to help us in this task.

Introduction

Category theory is a highly abstract field in mathematics. It first arose from algebraic topology but has found wide applicability since. For example, it is a good tool for studying functional programming languages. Turns out categories can describe types, functions between types and a lot of other concepts in functional programming.

This paper will focus specifically on monads, what they are, and what laws they must follow. Further, we will look at some examples of monads from Haskell (a functional programming language). The reason for this is that while Haskell has an interface called Monad, it has no way to ensure that types implementing it do indeed follow the monad laws.

Our group was tasked with making a computer-checked library of definitions, proofs and examples. For this, we first needed to define what a Category is, with examples. Then we were able to define Functors between Categories and some examples. Finally, we also made a definition for Natural Transformations with examples. After that, we all started to work on our individual questions. We had first to implement a definition for Monads. Then I looked at some examples of Monads used by the functional programming language Haskell and tried to implement these as examples in our library.

The paper will first discuss the technical background of the paper briefly in the section 1. This section will explain the choice of language and the details of our setup. Then the paper will describe how each of the steps from the previous paragraph was achieved. This can be found in section 2. This section will also mention the necessary background for the concepts discussed. Further, it will try to justify some of our choices. Here we also mention any difficulties that have popped up during the implementation. The section 4 will compare our library to the official one for lean.

This project has helped us understand both category theory and writing proofs in a computer-checked language better. While the library is not as readable and concise as the official one, it is still correct and useable. It could be enhanced and extended in the future.

1 Methodology

As the previous section has stated, one of the main aims of this project was to create a library of category theory. To achieve this we first needed to decide on what language we were going to use, this is detailed in subsection 1.1. After this the work was divided into two parts, the first was to establish a common base together with the whole team. This can then be used by the team members in the second part, where we each work mostly alone on our own question. Due to the topics, there still are some overlaps, which led to people still not having to work completely alone. The rest of this section will detail what tools and processes we have used to coordinate the project, this is in subsection 1.2.

1.1 Language Choice

To write we needed to find a language that supported computer-checked proofs and had a robust type system. The former means the developer can write proofs in the language that are then checked by the computer for correctness. This helps ensure that the proofs are actually sound, further, the computer may aid the proof and thereby make the process easier. A robust type system is needed to be able to correctly translate concepts from theory into code.

We had multiple options for the language that we used for the library. Mainly we considered either Agda or Lean. As we did not have much experience with either, we decided to go with Lean. This gave us one further choice to make since Lean's most recent stable version is Lean 3, but it also has a nightly version, Lean 4.

We opted to use the stable version for our development. This was done to ensure that our project will not get hung up on some problem caused by the nightly version. Furthermore, the differences between the versions are not too major [1], so adopting the library to the new version, once it is stable, will be fairly simple.

1.2 Coordination

Throughout the project, we have done most of the work remotely. This was no problem when we were studying Category Theory, but to make a coherent library we needed to coordinate the code we write.

The coordination between team members was done through multiple means throughout the project. First were multiple meetings every week, during which we discussed and wrote code together. As we could not meet every day, we also wrote code remotely. The remote code was reviewed by the rest of the team either during the next meeting, or through the merge requests on the project's Gitlab page. This allowed us to leave comments on the code directly. All these methods allowed us to work on our own, but in a coordinated and controlled manner.

2 Category Theory

This section of the paper will detail some of the decisions behind the library, as well as, the theoretical background of the concepts discussed. We will be covering four main topics within the field of Category Theory. First, subsection 2.1 will discuss how we implemented Categories. Then we will move on to Functors and some operations on functors that will be useful later in subsection 2.2. Following that subsection 2.3 will present Natural Transformations with their implementation and some relevant operators. Finally, we will close off with Monads in subsection 2.4, together with examples from Haskell. Throughout all of these sections, we will give examples to illustrate the concepts.

The library was developed on a private Gitlab repository, but all code has been moved to a public GitHub repository.

The move has conserved all commits, so the full development history can be seen. The repository can be found as `sgciprian/ct`.

2.1 Category

Categories as the name implies are one of the fundamental concepts in Category Theory. They are useful in representing the connections between 'objects'.

2.1.1 Definition

A category $\mathcal{C}$ is defined as the following [2, sec. 2.1.1] :

- A collection of objects, denoted:
$$A, B, \cdots \in \mathcal{C}$$
- A collection of arrows, between objects, denoted:
$$f : A \to B$$
- A composition operator between arrows:
$$f : A \to B, g : B \to C; g \circ f : A \to C$$
- The composition must follow two (or three, depending on one counts them) laws:
$$\textit{associativity} \qquad h \circ (g \circ f) = (h \circ g) \circ f \tag{1}$$
$$\textit{left unit} \qquad id_B \circ f = f \tag{2}$$
$$\textit{right unit} \qquad f \circ id_A = f \tag{3}$$
The last two of which are sometimes combined into one unit law. id_A is the identity arrow on object A.

```
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)
```

Listing 1: The definition of category

Our implementation can be seen in listing 1. The definition for objects (`Sort u`) just states that they can be of any type, but that type must be the same for all objects in the category. Also worth noting is that we opted to model the category based on the function $\mathcal{C}(A, B)$ that gives all arrows between objects A and B in category $\mathcal{C}$. Further, we enforce that all objects have an identity arrow and that the arrows can be composed. Finally, we also add the laws that we have listed above.

With the definition done, we have defined some examples to confirm that our definition is correct. All of the examples can be found in the folder `src/instances`, this is also true for any future examples. Some of the examples implemented include: the category of sets with functions between sets acting as arrows[1]; the category of natural numbers[2] with an arrow from x to y iff $x \leq y$; and even the category of partially ordered sets with monotone functions as arrows[3].

[1] commit `7f024038`, `src/instances/Set_category.lean`

[2] commit `7f024038`, `src/instances/Pre_Nat-le_category.lean`

[3] commit `7f024038`, `src/instances/Pos_category.lean`

2.2 Functor

Functors give us a way to map one category to another. This is useful to show that two categories have a similar structure. Further, some of these functors will come back directly in section 2.4.

2.2.1 Definition

The functor definition will once again give us some new definitions and some laws for those definitions that must hold to be a functor. The full definition is as follows [2, sec. 2.10, p. 28]:

- A functor $F : \mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}$ and $\mathcal{D}$.
- F maps each $\mathcal{C}$-object to a $\mathcal{D}$-object.
- F maps each $\mathcal{C}$-arrow to a $\mathcal{D}$-arrow.
- It must follow the laws below:
$$\textit{unit law} \qquad F(id_A) = id_{F(A)} \tag{4}$$
$$\textit{composition law} \qquad F(g \circ f) = F(g) \circ F(f) \tag{5}$$
where
$$A, B, C, D \in \mathcal{C};\ f \in \mathcal{C}(B, C);\ g \in \mathcal{C}(C, D)$$

Our implementation of this definition can be found in listing 2. Objects can be mapped with a simple function, but due to our definition of arrows, mapping arrows need a bit more effort to make work. The definition in the code is more verbose than what we have above, but this is due to the need to be more precise for the type checker. For example, in the definition above we didn't specify to which category the ids belong, but we cannot skip this in the code.

```
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))
```

Listing 2: The functor definition

2.2.2 Composition

Functors, similar to arrows can be composed. Given three categories $\mathcal{C}, \mathcal{D}, \mathcal{E}$ and two functors between them $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ there is a functor $G \circledcirc F$ from $\mathcal{C}$ to $\mathcal{E}$ that has the combined effect of applying F and then G. This functor must still satisfy the two laws we have shown in the definition above.

```
def composition_functor {𝒞 𝒟 ℰ : category} (G : 𝒟 ⇒ ℰ) (F :
   𝒞 ⇒ 𝒟) : 𝒞 ⇒ ℰ :=
{
  map_obj := λ X : 𝒞, G.map_obj (F.map_obj X),
  map_hom := λ X Y : 𝒞, λ f, G.map_hom (F.map_hom f),
  id := begin intro, rw F.id, rw G.id, end,
  comp := begin intros, rw F.comp, rw G.comp, end,
}
```

Listing 3: Functor composition definition

As can be seen in listing 3, the mappings are just defined by applying F and then G. The two proofs for the laws both use that F and G satisfy the laws. Below is an explanation of what is done in the proof for `id`. The first step is done

automatically by lean, whereas the second two steps use the `rw` command.

$$
\begin{aligned}
G \cdot F(id_A) &\stackrel{?}{=} id_{G \cdot F(A)} && \text{using } G \cdot F(B) = G(F(B)) \\
G(F(id_A)) &\stackrel{?}{=} id_{G(F(A))} && \text{using } F(id_B) = id_{F(B)} \\
G(id_{F(A)} &\stackrel{?}{=} id_{G(F(A))} && \text{using } G(id_B) = id_{G(B)} \\
id_{G(F(A))} &= id_{G(F(A))} && \text{done.}
\end{aligned}
$$

The proof for the composition law is done in a similar way. Using this definition it can be shown that functor composition is associative, meaning that the following holds:

$$
F : \mathcal{C} \to \mathcal{D}; \qquad G : \mathcal{D} \to \mathcal{E}; \qquad H : \mathcal{E} \to \mathcal{F}
$$
$$
H \cdot (G \cdot F) = (H \cdot G) \cdot F
$$

2.3 Natural Transformation

A natural transformation will be similar to what functor is to a category, but to functors. A natural transformation is between two functors that are between the same two categories. They are natural because they satisfy a naturality condition, which we will define later on.

2.3.1 Definition

As shown before, a transformation α is between two functors. Assume the functors F, G are from $\mathcal{C}$ to $\mathcal{D}$. The transformation must for each $\mathcal{C}$-object A assign a $\mathcal{D}$-arrow from $F(A)$ to $G(A)$, denoted α_A. To be natural, it must also:

$$
\forall f : A \to B, (A, B \in \mathcal{C}) \Rightarrow \alpha_B \circ F(f) = G(f) \circ \alpha_A \quad (6)
$$

Our definition in listing 4 is just a one-to-one implementation, though it once again is more verbose to satisfy the type checks.

```
structure natural_transformation {C D : category} (F G : functor
    C D) :=
  (α : Π (X : C.C₀) , D.hom (F.map_obj X) (G.map_obj X))
  (naturality_condition : ∀ {X Y : C.C₀} (f : C.hom X Y),
    D.compose (G.map_hom f) (α X)  =
    D.compose (α Y) (F.map_hom f)
  )
```

Listing 4: The definition of natural transformation

2.3.2 Composition

Similar to functors, natural transformations can also be composed. The composition is defined as follows[3]:

- Given two categories $\mathcal{C}, \mathcal{D}$.
- Given three functors F, G, H from $\mathcal{C}$ to $\mathcal{D}$.
- Given two natural transformations, $\alpha : F \stackrel{\cdot}{\to} G, \beta : G \stackrel{\cdot}{\to} H$.
- For each $\mathcal{C}$-object A, $(\beta \circledcirc \alpha)_A = \beta_A \circ \alpha_A$.

This was once again a simple definition to translate to code. The definition can be seen in listing 5. A note on notation, to shorten the definition, we use $\Rightarrow$ for functors and $\Longrightarrow$ for natural transformations.

```
def nt_composition {C D : category} {F G H : C ⇒ D}
  (τ₁ : G ⟹ H) (τ₂ : F ⟹ G) : F ⟹ H :=
  {
    α := λ X, D.compose (τ₁.α X) (τ₂.α X),
    naturality_condition := begin
      intros,
      rw D.assoc,
      rw τ₁.naturality_condition f,
      rw ←D.assoc,
      rw τ₂.naturality_condition f,
      rw D.assoc,
    end,
  }
```

Listing 5: The composition of natural transformations

2.3.3 Horizontal Composition

Natural transformations can also be composed in a different way. An illustration of this can be found in figure 1. This will be useful later when we will define the laws for the monad.

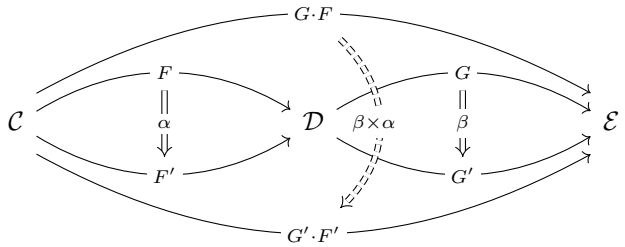

Figure 1: Horizontal Composition

The code for this is rather involved, but an explanation is provided in the repository[4]. The transformation part of the code can be seen in listing 6.

```
def bimap {C D E : category} {F F' : C ⇒ D} {G G' : D ⇒ E}
  (β : G ⟹ G') (α : F ⟹ F') : (G · F) ⟹ (G' · F') :=
  {
    α := λ X,
      E.compose (β.α (F'.map_obj X)) (G.map_hom (α.α X)),
    ...
  }
```

Listing 6: The definition for horizontal composition

2.3.4 Natural Isomorphism

A special natural transformation is a natural isomorphism. A natural isomorphism has an 'inverse', another natural transformation which has the assigned arrows reversed. It can be therefore defined in terms of the arrows that it assigns. If all arrows $f : A \to B$ it asigns are an isomorphism (meaning that there is $f^{-1} : B \to A$ such that $f^{-1} \circ f = id_A$ and $f \circ f^{-1} = id_B$), then the transformation is also iso.

To help the type system, we need to define some natural isomorphisms. The first of these is the associative transformation (later `assoc\_nt`), pointing from $(H \cdot G) \cdot F$ to $H \cdot (G \cdot F)$. As we have discussed before, the functor composition is associative (later `left/right\_unit\_nt`), so these two are just the same functor, but the type system still considers them to be different. The last two are the left and right unit transformations, pointing from $Id \cdot F$ and $F \cdot Id$ respectively to F. A

[4] commit `cdfe5b62`, `/doc/natural_transformations.md`

note on Id is the identity functor, pointing to the same category, and assigning everything to itself. With that definition, it is easy to see that combining a functor with the identity functor will not change it.

2.4 Monad

Monads are another construction from category theory that turned out to be useful for programming, especially functional programming. They turn out to be really useful when we want to compose functions that return not the result directly, but return the result in a 'container'. A nice explanation of how that leads to the definition of monads can be found in [4] and a video version can be found in [5].

2.4.1 Definition

The monad is defined by two natural transformations and some laws that must apply to said transformations:

- Given a category $\mathcal{C}$, and an endofunctor $F : \mathcal{C} \to \mathcal{C}$.
- A natural transformation μ, from $F^2 = F \cdot F$ to F.
- A natural transformation η, from Id to F.
- With laws:

$$\mu \circledcirc \mu \times ID_F = \mu \circledcirc ID_F \times \mu \tag{7}$$

$$\mu \circledcirc ID_F \times \eta = ID_F = \mu \circledcirc \eta \times ID_F \tag{8}$$

  Here ID_F is the identity natural transformation from F to F.

Implementing the monad definition in the code was again rather simple. The only caveat was the natural isomorphisms needed to help the type checker succeed.

```
structure Monad {C : category} (T : C ⇒ C) :=
  (μ : (T · T) ⟹ T)
  (η : (Id C) ⟹ T)
  (assoc : μ ⊚ μ × (ID T) = μ ⊚ (ID T) × μ ⊚ (assoc_nt T T
     T))
  (lu : μ ⊚ ID T × η = ID T ⊚ right_unit_nt T)
  (ru : μ ⊚ η × ID T = ID T ⊚ left_unit_nt T)
```

Listing 7: The monad definition

2.5 Examples from Haskell

With all those definitions done, we can finally take a look at examples from Haskell for monads. Haskell usually defines an operator `>>=`, that has type `m a → (a → m b) → m b`. But using this operator, we can define μ as the following:

```
-- μ
join               :: Monad m ⇒ m (m a) → m a
join x             =  x >>= id
```

Here `id` is the identity function that just returns the input. In this case, it will have type `m a → m a`.

2.5.1 Maybe

The Maybe is a type that allows functions to handle unexpected inputs and errors. It has two constructors, nothing/none: holds no value; just/some: holds some value. This can be used to extend what inputs a function accepts, as an example, we can implement a safe square root operation that takes one integer, and produces a Maybe integer, with the square root. When the input is negative, it can give nothing instead of breaking.

To make handling the Maybe type easier, Haskell implements for it their Monad interface. This implementation follows the monad laws, though this is not proven in Haskell, as the language is not equipped with such features. In this section, we will mirror the implementation in lean, and prove that it does indeed follow the laws.

The following is a simplified definition of monad for the `Maybe` type:

```
-- Definition
data  Maybe a  =  Nothing | Just a

-- μ
join                :: Maybe (Maybe a) → Maybe a
join x              = x >>= id
(Just x) >>= k      = k x
Nothing  >>= _      = Nothing
-- Rewriting:
join (Just x)       = x
join Nothing        = Nothing

-- η
return      :: a → Maybe a
return      = Just
```

This can be easily translated to lean, as follows:

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

notation (name := none) ∅ := Maybe.none

def Maybe.join {α : Type*} : Maybe (Maybe α) → Maybe α
| ∅ := ∅
| (Maybe.some x) := x

def Maybe.return {α : Type*} (x : α) : Maybe α := Maybe.some x
```

To be able to prove that `Maybe` with these operations does indeed follow the monad laws we need to first prove that it is an endofunctor. This step will be omitted here, but the proof can be found in `src/instances/functors/Maybe_functor.lean`. With that, we get the maybe monad definition:

```
def Maybe.monad : Monad Maybe.functor :=
{
    μ := {
      α := λ X, Maybe.join,
      naturality_condition := _,
    },
    η := {
      α := λ X, Maybe.return,
      naturality_condition := _,
    },
    ru := _,
    lu := _,
    assoc := _,
}
```

We removed the proofs here to keep the paper a manageable size, but the full proof is available in the repository.

This is not the only way to handle errors using monads, as some of the exceptions in Haskell also implement the interface, but that has a lot more complex implementation than maybe. For those reasons, and due to the short nature of this project, we will not delve into those.

2.5.2 List

Another type that is commonly used and implements the monad interface, is the List. We have also implemented this in Lean and proved that it does indeed follow the laws. The definition from Haskell simplifies to:

```
-- Definition
data List a = [] | a : List a

-- μ
join               :: [[a]] → [a]
join x             = x ≫= id
xs ≫= f             = [y | x ← xs, y ← f x]
-- Rewriting:
join xs            = [y | x ← xs, y ← x]

-- η
return      :: a → [a]
return x    = [x]
```

This can be implemented in lean as:

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

def List.merge {α : Type} : List α → List α → List α
  | List.nil ys := ys
  | (List.cons x xs) ys := List.cons x (List.merge xs ys)


def List.join {α} : List (List α) → List α
| List.nil := List.nil
| (List.cons l ls) := List.merge l (List.join ls)

def List.return {α} (a : α) : List α := List.cons a List.nil
```

The proof for this satisfying the laws can be found in the repository. It is largely the same as the proof for maybe, though perhaps more complicated, as List is an inductive type. The inductivity combined with the triply nested lists makes the proofs rather long.

3 Responsible Research

This paper has mostly only dealt with mathematical concepts and their implementations in Lean. As such, we had no concerns with collecting and storing any data. We did however strive to make our results reproducible. To achieve this we added instructions to the repository on how to verify our code. The repository is publicly available on GitHub, a link to it can be found in section 2. Further, we aimed to provide adequate documentation for the repository for any future users. For this reason, a lot of proofs have a lot of comments attached or entire files attached in the `doc` folder.

4 Discussion

There are a lot of already existing libraries for Category theory, in various computer-checked languages. Some examples include [6], [7], [8], [9], among others, but to keep this section doable, we will focus on comparing with [10]. To do this we will reference the documentation [11]. We will disregard differences in notation, as almost all libraries use different symbols for the different operations.

4.1 Category

The definition[5] is basically the same as our definition, in terms of its contents. Though it can be easily seen in the code that ours was written by some beginners with this language, the lean library is written by people more familiar with the language. The only difference that may affect the library, is that we have the objects as a field of the category, while the library takes it as a parameter. Another minor difference is in the syntax used for `hom`, but as far as we know this is purely a difference in syntax, and both are interpreted the same.

A possible major difference could have been implementing the arrows differently, as a reminder the definition we used:

```
hom     : Π (X Y : C₀), Sort v
```

Another possible implementation would have been to instead implement it with two functions:

```
hom : Sort v     -- The arrows
dom : hom → C₀ -- Domain of the arrow
cod : hom → C₀ -- Codomain of the arrow
```

This would have also worked, but the other definition felt simpler and more intuitive to us when we were writing it.

4.1.1 Functor

The functor definition[6] is largely also the same as our definition. There are no meaningful differences, only differences in notation. Their definition of the identity functor is shorter, as the laws for the functor are automatically derived, when trivial. The same is true for the implementation of functor composition. In fact, the library tries to utilise tactics and other language features to their fullest, so they don't need to write out all proofs.

4.1.2 Natural Transformation

The definition[7] again is essentially the same definition as ours. This is also the case for the identity transformation as well as the composition. However, while the documentation does mention horizontal composition being in the library, the definition is seemingly not there.

4.1.3 Monad

There are some significant differences in the monad definition[8]. They do not use the horizontal composition operator in their definition, rather they focus on operations on arrows. More importantly, instead of showing that the two transformations are equivalent, the law they choose is the equivalent statement of the two transformations having the same effect, for each object. This avoids having to define the natural isomorphisms to overcome the type checks. Further, this would have made my proofs for the examples shorter, as my first steps were exactly to focus only on μ_X and η_X instead of μ and η.

5 Conclusions and Future Work

We have now discussed what monads are, and how we have translated that definition into code. We showed for Maybe and List from Haskell, that they do indeed follow the monad laws, by reimplementing them in Lean. It would have been nice to do this for even more types, but the shortness of the project did not allow for this.

When in another course we first worked with Haskell, we have already encountered interfaces named Functor and Monad, but we did not get a thorough background on them. Doing this project has helped us get a deeper understanding

[5] Category Definition

[6] Functor Definition

[7] Natural Transformation Definiton

[8] Monad Definition

of where these concepts come from. Further, we hope we managed to convey this to the reader.

As for the future of the library, what code we have written is definitely the most elegant way of writing some of the proofs. The proofs could be made prettier and more concise in the future. Moreover, this library is by no means complete, while that is perfectly fine for the scope of the current project, it may be nice to extend it.

References

[1] "Significant Changes from Lean 3," Lean Manual. (2023), [Online]. Available: https://leanprover.github.io/lean4/doc/lean3changes.html (visited on 05/26/2023).

[2] B. C. Pierce, "A taste of category theory for computer scientists," Feb. 2011. DOI: 10.1184/R1/6602756.v1. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/A_taste_of_category_theory_for_computer_scientists/6602756.

[3] B. Milewski. "Natural transformations," Bartosz Milewski's Programming Cafe. (), [Online]. Available: https://bartoszmilewski.com/2015/04/07/natural-transformations/.

[4] B. Milewski. "Monads: Programmer's definition," Bartosz Milewski's Programming Cafe. (Nov. 2016), [Online]. Available: https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/.

[5] B. Milewski. "Category theory 10.1: Monads," YouTube. (), [Online]. Available: https://www.youtube.com/watch?v=gHiyzctYqZ0&list=PLbgaMIhjbmEnaH_LTkxLI7FMa2HsnawM_&index=19.

[6] "Agda/agda-categories: A new categories library for agda." (), [Online]. Available: https://github.com/agda/agda-categories.

[7] J. Z. Hu and J. Carette, "Formalizing category theory in agda," *CPP 2021 - Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, co-located with POPL 2021*, pp. 327–342, Jan. 2021. DOI: 10.1145/3437992.3439922. [Online]. Available: https://github.com/agda/agda-categories.

[8] "Copumpkin/categories: Categories parametrized by morphism equality, in agda." (), [Online]. Available: https://github.com/copumpkin/categories.

[9] A. Timany and B. Jacobs, "Category theory in coq 8.5," *DROPS-IDN/6000*, vol. 52, Jun. 2016, ISSN: 18688969. DOI: 10.4230/LIPICS.FSCD.2016.30.

[10] "The lean mathematical library," *CPP 2020 - Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, co-located with POPL 2020*, pp. 367–381, Jan. 2020. DOI: 10.1145/3372885.3373824. [Online]. Available: https://dl-acm-org.tudelft.idm.oclc.org/doi/10.1145/3372885.3373824.

[11] "Maths in lean: Category theory." (2020), [Online]. Available: https://leanprover-community.github.io/theories/category_theory.html (visited on 06/20/2023).