Formalizing Category Theory in Agda

Jason Z. S. Hu
zhong.s.hu@mail.mcgill.ca
McGill University
Montréal, Québec, Canada

Jacques Carette
carette@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

Abstract

The generality and pervasiveness of category theory in modern mathematics makes it a frequent and useful target of formalization. It is however quite challenging to formalize, for a variety of reasons. Agda currently (i.e. in 2020) does not have a standard, working formalization of category theory. We document our work on solving this dilemma. The formalization revealed a number of potential design choices, and we present, motivate and explain the ones we picked. In particular, we find that alternative definitions or alternative proofs from those found in standard textbooks can be advantageous, as well as "fit" Agda's type theory more smoothly. Some definitions regarded as equivalent in standard textbooks turn out to make different "universe level" assumptions, with some being more polymorphic than others. We also pay close attention to engineering issues so that the library integrates well with Agda's own standard library, as well as being compatible with as many of supported type theories in Agda as possible.

CCS Concepts: • **Theory of computation** → **Type theory**; **Logic and verification**.

Keywords: Agda, category theory, formal mathematics

ACM Reference Format:
Jason Z. S. Hu and Jacques Carette. 2021. Formalizing Category Theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3437992.3439922

1 Introduction

There have been many formalizations of category theory [7, 21] in many different proof assistants, over more than 25 years [4, 16, 18, 24, 26, 28–30, 32, 35, 37, etc.]. All of them embody *choices*; some were forced by the ambient logic of the host system, others were pragmatic decisions, some were philosophical stances, while finally others were simply design decisions.

Category theory is often picked as a challenge, as it is both be quite amenable to formalization and to involve many non-trivial decisions that can have drastic effects on the usability and effectiveness of the results [16]. With the rapid rise in the use of category theory as a tool in computer science, and with the advent of *applied* category theory, having a stable formalization in the *standard library* of one's favourite proof assistant becomes necessary.

Our journey started as the authors were trying to keep the "old" category theory library for Agda [26] alive. Unfortunately, as Agda [3] evolved, some of the features used in that library were no longer well-supported, and eventually the library simply stopped working. As it became clear that simply continuing to patch that library was no longer viable, a new version was in order.

This gave us the opportunity to revisit various design decisions of the earlier implementation — which we will document. We also wanted to preserve as much formalization effort as possible, while also use language features introduced in Agda 2.6+ like generalized variables and revise the theoretical foundation which the library relies on. This new version is then partly a "port" of the previous one to current versions of Agda, but also heavily refactored, including some large changes in design.

Our principal theoretical contribution is to show that *setoid-based proof-relevant category theory* works just as well as various other "flavours" of category theory by supporting a large number of definitions and theorems. Our main engineering contribution is a coherent set of design decisions for a widely reusable and working library of category theory in Agda, freely available[1].

This paper is structured as follows. In Section 2, we discuss our global design choices. We discuss the rationale behind non-strictness, proof-relevance, hom-setoids, universe polymorphism, (not) requiring extra laws and concepts as record types. In Section 3, we give examples on how proof-relevance drives us to find concepts in an alternative way. In Section 4, we discuss other design decisions and some efficiency issues. In Section 5, we compare category theory libraries in other systems. Finally, we conclude in Section 6.

[1] at https://github.com/agda/agda-categories

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439922

For reasons of space, we have to make some assumptions of our readership, namely that they are familiar with:

1. category theory,
2. dependent type theory,
3. formalization, and
4. proof assistants (e.g. familiarity with Agda and a passing knowledge of other systems).

2 Design Choices

Choices arise from both the system and its logic, as well as from the domain itself.

2.1 Fitting with Agda

The previous formalization [26] was done in a much older Agda, with a seriously under-developed standard library. To better fit with *modern* Agda, we choose to:

1. use dependent types,
2. be constructive,
3. re-use as much of the standard library [12] as possible,
4. use the naming convention of its standard library whenever meaningful,
5. use the `variable` generalization feature for levels and categories,
6. try to fit with as many modes of Agda as possible.

The first two requirements are natural, as choosing otherwise would create a clash of philosophy between the system and one of its libraries. The next two are just good software engineering, while the fifth is mere convenience. Note that re-using the standard library pushes us towards setoids (more on that later) as its formalization of algebra uses them extensively.

The last requirement is more subtle: we want to allow others to use alternative systems or make postulates if they wish, and still be able to use our library. This means that we need to avoid using features that are incompatible with supported systems in Agda. For example, when added to Martin-Löf Type Theory (MLTT) [23], axiom K [31], equivalent to *Uniqueness of Identity Proofs* (UIP), creates a propositionally extensional type theory incompatible with *univalence* [33]. Thus Agda has options such as `--without-K` [10] to access the intensional type theory MLTT, and conversely `--with-K` to turn on axiom K. Separately, there is cubical type theory (`--cubical`) [34] which implements a computational interpretation of homotopy type theory (HoTT) [33] and supports univalence. Intensional type theory is compatible with both options of `--with-K` and `--cubical`, and thus if we build our library using `--without-K`, it can be maximally re-used. This further implies that we have to avoid propositional equality as much as possible, as pure MLTT gives us very few tools to work with it. We additionally turn on the `--safe` option to avoid possible misuses of certain features which could lead to logical inconsistencies.

2.2 Which Category Theory?

Category theory is often presented as a single theory, but there are in fact a wealth of flavours: set-theoretic, where a category has a single hom-set equipped with source and target maps; ETCS-style [20], where there are no objects at all; dependently-typed, where hom-"sets" are parametrized by two objects; proof-irrelevant, where the associativity and identity laws are considered to be unique [4, 16, 26, 35]; setoid-based, where each category relies on a local notion of equivalence of hom-sets rather than relying on a *global* equality relation [26, 37]. There are also questions of being strict or weak, whether to do 1-categories, n-categories or even ∞-categories. What to choose?

Standard textbooks often define a category as follows:

Definition 2.1. A category C consists of the following data:

1. a collection of objects, C_0,
2. a collection of morphisms, C_1, between two objects. We use $f : A \Rightarrow B$ to denote the morphism $f \in C_1$ is between objects A and B,
3. for each object A, we have an identity morphism $1_A : A \Rightarrow A$, and
4. morphism composition $\circ$ composing two morphisms $f : B \Rightarrow C$ and $g : A \Rightarrow B$ into another morphism $f \circ g : A \Rightarrow C$.

These must satisfy the following laws:

1. identity: for any morphism $f : A \Rightarrow B$, we have $f \circ 1_A = f = 1_B \circ f$, and
2. associativity: for any three morphisms f, g and h, we have $(f \circ g) \circ h = f \circ (g \circ h)$.

Embedded in the above definition are a variety of decisions, and we will use these as a running example to explain ours.

2.2.1 Collections. The first item to notice is the use of *collection* rather than *set* or *type*. Textbooks tend to do this to side-step "size" issues, and then define various kinds of categories depending on whether each of the collections (objects, all morphisms, all morphisms given a pair of objects) is "small", i.e. a set. This matters because a number of constructions in category theory produce large results.

We define *collections of objects* to be *types*, with no further assumptions or requirements. We do know that in MLTT types are well modeled by ∞-groupoids [17, 36] — so wouldn't this higher structure be a problem? No! This is because we never look at it, i.e. we never look at the identity type (or their identity types) of objects.

The collection of morphisms is trickier, and splits into:

1. Is there a single collection of morphisms?
2. What about equality of morphisms?

The first item will be treated here, the second in subsection 2.2.3.

If we try to put all the morphisms of certain categories together in a single collection, size issues arise, but there is also another issue: if we consider composition as a function of pairs of morphisms, then this function is partial. Luckily, our dependent type theory allows one to side-step both issues at the same time: rather than a single collection of morphisms, we have a (dependently-typed) family of morphisms, one for each pair of objects. In category theory, one rarely considers the "complete collection" of all morphisms. This solves the composition problem too, as we can only compose morphisms that have the right type, leading to the following (partial) definition:

```
record Category : Set where
  field
    Obj : Set
    _⇒_ : (A B : Obj) → Set
    _∘_ : ∀ {A B C} → B ⇒ C → A ⇒ B → A ⇒ C
```

2.2.2 Strictness. Traditional textbooks tend to implicitly assume that collections are somewhat still set-like, in that *equality* is taken for granted, i.e. that it always makes sense to ask whether two items from a collection are equal. Not just that it always makes sense, but that the underlying meta-theory will always answer such queries in finite time[2].

The *Principle of Isomorphism* [22] already tells us that we should not assume that we have any relation on objects other than the one given by categorical principles (isomorphism); a related *Principle of Equivalence* [5] can be stated formally in the context of homotopy type theory. That we normally do not have, and should not assume, such a relation have motivated some to create the concept of a *strict category*, where we have given ourselves the ability to compare objects for equality. Classically, sets have equality defined as a total relation, so that this comes "for free". In other words, given two elements x, y of a set S, in set theory it always makes sense to "ask" the question $x = y$, and this has a *boolean* answer. This is one reason why it took a while for the Principle of Equivalence to emerge as meaningful. As global extensionality is hard to mechanize in MLTT, it is simplest to forgo having an equality relation on objects at all.

2.2.3 Proof-relevant Setoids. In Definition 2.1, equality of morphisms is also taken for granted. The laws use equality, blithely assume that the meta-theory defines it. In MLTT, which equality we use matters. Usually, there are three options: local equality (setoids), propositional equality in intensional type theory (_≡_), and propositional equality with UIP.

Propositional equality does not work very well in MLTT without further properties or axioms to deal with functions (e.g. function extensionality), while many categories have (structured) functions as morphisms. The third case is a plausible option, because UIP relieves us from reasoning about equality between equalities due to UIP and reduces the issue to familiar set theory. Nonetheless, the `--with-K` mode and the `--cubical` mode approach UIP in different ways and it is not immediate to us how to organize the library so that it is compatible with both. Thus this option, though very interesting, seems to clash with our original motivation.

For these reasons, we chose to work with setoids. Earlier formalizations of category theory in type theory already used setoids [2, 18, 26, 37], which associate an equivalence relation to each type. This generalizes "hom-sets" to "hom-setoids", i.e. the definition of category is augmented as follows:

```
_≈_   : ∀ {A B} → (f g : A ⇒ B) → Set
equiv : ∀ {A B} → IsEquivalence (_≈_ {A} {B})
```

In both types, `A` and `B` are objects in the current category. `IsEquivalence` is a predicate provided by the standard library that expresses that _≈_ is an equivalence relation. Furthermore, composition must respect this equivalence relation, which we can express as[3]:

```
∘-resp-≈  : f ≈ h → g ≈ i → f ∘ g ≈ h ∘ i
```

Note that _≈_ can be specialized to _≡_ to work in other settings such as cubical type theory.

We explicitly do not assume that two witnesses of _≈_ are equivalent, making our setoids *proof relevant*. Proof-relevance is a significant difference between this library and the previous one [26], which relied heavily on irrelevant arguments [1]. In particular, all of the proof obligations (for example left and right identities, and associativity in the case of a category) were marked irrelevant in [26], making these proofs "unique" by fiat. Thus two categories that differed only in their proofs were automatically regarded as (definitionally) equal. Ignoring the details of proofs is convenient — but unfortunately irrelevant arguments are not part of MLTT. Worse yet, they are not a stable, well-maintained feature in Agda, so we refrained from using this feature in our library.

We gain other improvements over the previous library by having hom-setoids proof-relevant. In [26], due to irrelevance, the content of _≈_ is ignored. However, this is not necessarily coherent under all settings. For example, when defining the (large) category of all categories, with proof relevance, we can use natural isomorphisms as equivalence between functors. In other words, in our setting, the "natural" definition of the (large) category of all categories is a category, we do not need to move up to 2−categories. The previous library, contrarily, must use heterogeneous equality for equivalence between functors, which subsequently required axiom K for elimination and restricted the possible choice of foundations. In this case, making setoids proof-relevant actually allowed us to internalize more category theory into itself.

[2]That we should not ask whether two objects are equal is an issue well described at the *Principle of Equivalence* page of the nLab. https://ncatlab.org/nlab/show/principle+of+equivalence

[3]We use variable generalization to leave implicit variables out and let Agda infer them, so we will omit unnecessary type ascriptions provided an unambiguous context..

Libraries formalizing category theory based on HoTT [4, 16] restricts hom-sets to be hSets by requiring an additional law which states the contractibility of equality proofs between equalities in hom-sets. Our library implements a settings which allow richer structures in the hom-setoids.

2.2.4 Explicit Universe Level. In Agda, users are exposed to the explicit handling of universe levels (i.e. of type `Level`). Some find it cumbersome, but we have found it quite useful. To help with reuse, we make our definitions universe-polymorphic by parameterizing them by `Level`s. For example, a `Category` is refined as follows:

```
record Category (o ℓ e : Level)
    : Set (suc (o ⊔ ℓ ⊔ e)) where
  field
    Obj : Set o
    _⇒_ : (A B : Obj) → Set ℓ
    _≈_ : ∀ {A B} → (f g : A ⇒ B) → Set e
  -- other fields omitted
```

Since the definition of `Category` contains three `Set`s representing objects, morphisms and the equivalence relations respectively, it can be indexed by three `Level`s and thus live at least one level above their supremum.

One significant advantage of a level-parametric definition is that it simplifies the formalization of concepts such as the category of categories, or that of functors. We do not have to duplicate definitions, nor do we have to sprinkle various size constraints about (such as a category being "locally small") to avoid set-theoretic troubles.

With explicit `Level`s, new phenomena become visible. In set-based category theory, one might be tempted to talk about the (large) category of all sets or all setoids. In Agda, we can only talk about the category of all `Setoids` with particular `Level`s:

```
Setoids : ∀ c ℓ →
  Category (suc (c ⊔ ℓ)) (c ⊔ ℓ) (c ⊔ ℓ)
Setoids c ℓ = record
  { Obj       = Setoid c ℓ
    -- ... other fields omitted.
  }
```

Here `c` and `ℓ` are the `Level`s of the carrier and the equivalence of a `Setoid c ℓ`, respectively. We can clearly see the ensuing size issue. The definition *must* be indexed by `Level`s, as there is no term *in* the type theory in which all `Setoids` (for example) exist. The set of types `Set ℓ` is somewhat analogous to a *Grothendieck universe* which provides a way to resolve Russell-style paradox in set theory, as it is closed under similar operations, but not unrestricted unions, where one must then move to a larger universe. `Set (suc ℓ)` is indeed sometimes called a Russell-style universe.

However, universes in Agda are non-cumulative by default. Combined with explicit `Level`s, this leads to other issues. With cumulative universes, a type in one universe automatically inhabits all larger universes. In Agda, one must explicitly lift terms to larger levels, which adds a certain amount of "noise" to some code. For example, consider two categories of `Setoids`, `Setoids 0 1` and `Setoids 1 1`, differing only in their first indices. With cumulative universes, even though we still need to apply a lifting functor to embed `Setoids 0 1` in `Setoids 1 1`, the functor is trivially defined:

```
liftF = record  -- we are defining a functor
  { F₀ = λ x → x
    -- other fields are omitted
  }
```

With cumulativity, the second `x` has a larger universe than the first one. Without cumulativity, explicit calls to `lift` must be inserted:

```
liftF = record  -- we are defining a functor
  { F₀ = λ x → lift x
    -- other fields are omitted
  }
```

We noticed that when handling some classical definitions or results involving sets, like adjoint functors and the Yoneda lemma, we often need to postcompose with a lifting functor in order to achieve the most general statements. For example, the Yoneda lemma involves the natural isomorphism in X:

$$Nat[yX, F] \simeq FX$$

where $F : C^{op} \Rightarrow Set$ for some category C and $X \in C$ is an object. In the actual formalization, assuming C has type `Category o ℓ e` and F maps to `Setoids ℓ e`, then by some calculation, we see that $Nat[yX, F]$ actually maps to `Setoids (o ⊔ ℓ ⊔ e) (o ⊔ ℓ ⊔ e)`, because the `Setoids` must be large enough to contain F. Thus we cannot create this natural isomorphism without lifting the universe on the right hand side to the correct level. Explicit universe `lift`ing and `lower`ing are then required in subsequent equational reasoning, which quickly become rather annoying.

Since 2.6.1, Agda has an experimental feature of cumulative universes. We hope that this feature may help us remove some clutter in our statements and proofs. However, at present, cumulativity is not deemed `--safe`. Furthermore, we encountered issues with the level constraint solver when we experimented with adapting our library to that environment.

2.3 Duality

In category theory, duality is omnipresent. However, in type theory and in formalized mathematics, subtleties arise. Some are due to proof relevance, while others are usability issues, which we discuss here.

Additional Laws for Duality. In category theory, there is a very precise sense in which, if a theorem holds, then its dual statement also holds. Thus, in theory, we obtain two theorems by proving one. This is the *Principle of Duality* [7], which we would like to exploit.

But first, we need to make sure that the most basic duality, that of forming the opposite category, should be involutive. We can easily prove that the double-opposite of a category C is equivalent to C. This equivalence is true *definitionally* with proof-irrelevant definitions in [26]. Can we recover this here as well? Yes – we can follow [16] and require two (symmetric) proofs of associativity of composition in the definition of a `Category`:

```
assoc     : (h ∘ g) ∘ f ≈ h ∘ (g ∘ f)
sym-assoc : h ∘ (g ∘ f) ≈ (h ∘ g) ∘ f
```

Specifically, with `sym-assoc`, we can define its opposite category as follows:

```
op : Category o ℓ e
op = record
  { assoc     = sym-assoc
  ; sym-assoc = assoc
  -- other fields omitted
  }
```

Otherwise, without `sym-assoc`, we would have to use the symmetry of `_≈_`:

```
assoc = sym assoc
```

But now, applying duality twice gives `sym (sym assoc)` for the associativity proof, which is not definitionally equal to `assoc`. This makes the properties of an opposite category less useful than ones of the original one. For example, we might want to prove some properties about coproducts by proving the dual properties about products in the opposite category. Without involution of `op`, we would have to argue the properties still hold if we swap to another associativity proof, which defeats the usefulness of the Principle of Duality.

Another convenient law to add is

```
identity² : id ∘ id ≈ id
```

This law can be proved by taking `f` as `id` in either the left identity or right identity law:

```
identityˡ : id ∘ f ≈ f
identityʳ : f ∘ id ≈ f
```

We add this additional law for the following reasons:

1. When proving `id ∘ id ≈ id`, we need to choose between `identityˡ` and `identityʳ`, while there is no particular reason to prefer one to another. Adding this law neutralizes the need to make this choice.
2. In the implementation, we sometimes rely on constant functors, which ignore the domain categories and constantly return fixed objects in the codomain categories and their identity morphisms. Since the domain categories are completely ignored, these functors are intuitively "the same" as their duals. `identity²` allows constant functors to be definitionally equal to their duals even with proof-relevance.

Independent Definitions of Dual Concepts. In other libraries [4, 16, 32, 35], it is typical to define one concept and use duality to obtain the opposite one. For example, we could define the initial object of C, `Initial` C as usual, and then define the terminal object by taking the opposite as follows:

```
Terminal′ : ∀ {o ℓ e} (C : Category o ℓ e) →
              Set _
Terminal′ C = Initial (Category.op C)
```

However, we do not take this approach. Instead, we define concepts explicitly in terms of data and laws and define conversions between duals in modules of the form `*.Duality`. This has the following advantages:

1. when constructing or using the concepts, the names of the fields are more familiar;
2. theorems relating redundant definitions increase our confidence that our definitions are correctly formulated;
3. the redundancy helps maintain the Principle of Duality.

Expanding on this third point: like with `sym-assoc`, we want duality to be a definitional involution for a number of concepts. We were able to identify a number of concepts which require additional laws to achieve this goal, which we detail next.

Duality-Completeness of Laws. Ensuring the involution of duality turns out to be a very general design principle. We sometimes obtain it for free, e.g. `Functor` and `Adjoint`. In other cases, we need to supply a symmetric version of a law. For example, `Category`, `NaturalTransformation`, `Dinatural` (transformation) and `Monad` all need some extra laws. As a rule of thumb, if a conversion to the dual concept requires equational reasoning, even as simple as applying `sym` to `assoc`, then we need to add that equation as a law. In other words, our laws should either be self-dual, or come in dual pairs (quite reminiscent of work on reversible computation [9] where the same property is desirable). We ensure this principle by proving theorems of the following form:

```
op-involutive : Category.op C.op ≡ C
op-involutive = ≡.refl
```

Here `C` is a `Category`. We also supply similar proofs for conversions between dual concepts, e.g.:

```
op⊤⇔⊥ : (⊥ : Initial) →
  op⊤⇒⊥ (⊥⇒op⊤ ⊥) ≡ ⊥
op⊤⇔⊥ _ = ≡.refl
```

`⊥⇒op⊤` converts an initial object to a terminal object in the opposite category and `op⊤⇒⊥` does the inverse. We put these theorems in `private` blocks so they are only type checked. These theorems must be proved precisely by reflexivity. This ensures that our definitions are duality-friendly.

Once we get the definition right, we also provide a helper constructor without the additional laws, so that defining these self-dual versions are not more cumbersome than their classical counterpart. Constructions defined through the helpers still enjoy the principle of duality. Consider an application of the helper for `Category`, which effectively proves `sym-assoc` by applying symmetricity:

```
Some-Cat = record {
  -- other fields ignored
  assoc = some-proof ; sym-assoc = sym some-proof
}
```

Notice that `Category.op (Category.op Some-Cat)` remains definitionally equal to `Some-Cat`. In general, we found that the addition of these extra laws were beneficial in the setting of 1-category theory. The situation becomes more complex when we move to the `Bicategory` setting, as we must consider higher structures. Exactly how to modify the definitions of higher structures to obtain similar good behaviour with respect to definitional equalities is left as future work.

2.4 Encodings as Records

Another important design decision is how to encode definitions. Generally, two different styles are used: records [16, 37] or nested Σ types [4, 35]. In the latter style, developers typically need to write a certain amount of boilerplate accessor code. In Agda it is more natural to use record definitions:

1. It aligns very well with the design principle of the standard library,
2. Records allow various *syntactic sugar*, as well as having good IDE (via Emacs) support,
3. Most importantly records also behave as modules. That is, we can export symbols to the current context from a record when it is unambiguous to do so.

The record module feature enables some structural benefits as well. Consider the following definition of a `Monad` over a category:

```
record Monad {o ℓ e} (C : Category o ℓ e)
    : Set (o ⊔ ℓ ⊔ e) where
  field
    F : Endofunctor C
    η : NaturalTransformation idF F
    μ : NaturalTransformation (F ∘F F) F
  -- ... laws are omitted
```

We often need to refer to components of the `Functor F` or the `NaturalTransformations` η or μ when working with a `Monad`. By adding the following module definitions to the `Monad` record, we can use dot accessors to access deeper fields:

```
  module F = Functor F
  module η = NaturalTransformation η
  module μ = NaturalTransformation μ
```

For example, if we have two `Monads` `M` and `N` in scope, we can declare `module M = Monad M` and `module N = Monad N`, and get the following convenient nested dot accessors:

```
  M.F.₀  -- the mapping of objects of F of M
  N.F.₁  -- the mapping of morphisms of F of N
  M.μ.η X
    -- the component of the NaturalTransformation
    -- μ of M at object X
  N.η.commute f  -- the naturality square of the
                 -- NaturalTransformation η of N
                 -- at morphism f
```

The original syntax is more verbose, so the module syntax is significantly more convenient:

```
  Functor.F₀ (Monad.F M)
  Functor.F₁ (Monad.F N)
  NaturalTransformation.η (Monad.μ M) X
  NaturalTransformation.commute (Monad.η N) f
```

Another frequent style is to open a module with renaming:

```
  open NaturalTransformation (Monad.μ M)
    renaming (η to α)
  open NaturalTransformation (Monad.η N)
    renaming (η to β)
```

Then we use α and β to refer to the component maps of the corresponding natural transformations. Unfortunately such setup code is ad-hoc and inconsistent across files.

We use the accessor module style throughout the code base, as it feels more elegant and readable to us than other styles.

3 Formalization and Definitions

While implementing the library, we noticed several times that "standard" definitions needed to be adjusted, for technical reasons. Certain direct translations of concepts from classical category theory are not even well-typed! Proof-relevance also forces us to pay close attention to the laws embedded in each concept, to obtain more definitional equalities, rather than relying on extensional behavior for "sameness". The resulting formalization is more robust, and it also eases type checking.

Various categorical concepts are well-known to have multiple, equivalent definitions. We have found that, although classically equivalent, some turn out to be technically superior for our formalization. We are sometimes even forced to introduce new ones. Here we discuss the choices we made when defining concepts related to closed monoidal categories and finite categories in detail, focusing on the underlying rationale.

3.1 Adjoint Functors

Adjoint functors are frequently regarded as one of the most fundamental concepts in category theory and play a critical part in the definition of closed monoidal categories. The following two definitions of adjoint functors are equivalent in classical category theory.

Definition 3.1. Functors $F : \mathcal{C} \Rightarrow \mathcal{D}$ and $G : \mathcal{D} \Rightarrow \mathcal{C}$ are adjoint, $F \dashv G$, if there is a natural isomorphism $Hom(FX, Y) \simeq Hom(X, GY)$ in X and Y.

Definition 3.2. Functors $F : \mathcal{C} \Rightarrow \mathcal{D}$ and $G : \mathcal{D} \Rightarrow \mathcal{C}$ are adjoint, $F \dashv G$, if there exist two natural transformation, unit $\eta : 1_{\mathcal{C}} \Rightarrow GF$ and counit $\epsilon : FG \Rightarrow 1_{\mathcal{D}}$, so that the triangle identities below hold:

1. $\epsilon F \circ F\eta = 1_F$
2. $G\epsilon \circ \eta G = 1_G$

These two definitions are classically equivalent. Definition 3.1 is typically very easy to use in classical category theory, as it it is about *hom-sets*, and so partly set-theoretic in its formulation. However, this definition is not natural in Agda, especially in the presence of non-cumulative universes and level-polymorphic morphisms (Section 2.2.4), so that the morphisms of $\mathcal{C}$ and $\mathcal{D}$ do not always live in the same universe level. Thus $Hom(FX, Y) \simeq Hom(X, GY)$ is not well-typed as is. Instead, $Hom(FX, Y)$ and $Hom(X, GY)$ need to be precomposed by lifting functors, which lift both hom-setoids to the universe at their supremum level. One might think that this technicality is classically not present – but that is because many textbooks make the blanket assumption that all their categories are locally small. It corresponds to assuming that the morphisms of $\mathcal{C}$ and $\mathcal{D}$ live at the same (lowest!) universe level. In that case, we indeed do not need the lifting functors. This "technical noise" add by the lifts get rid of this problem, but set theory has no means to express size polymorphism (as in set, proper class, superclass, etc). However, such coercions are neither intuitive nor easy to work with.

Definition 3.2, on the other hand, has no such problem. Both natural transformations and triangle identities involve no explicit universe level management. For this reason, we choose Definition 3.2 as our primary definition of adjoint functors and have Definition 3.1 as a theorem. The added polymorphism of the unit-counit definition makes it more suitable when working in type theory.

3.2 Monoidal Category

A monoidal category can be understood as a generalization of a monoid to the categorical setting. Classically, a monoidal category has the following definition [19]:

Definition 3.3. A category $\mathcal{C}$ is monoidal with the following data:

1. a unit object u,
2. a bifunctor $\otimes$,
3. for any object X, a natural isomorphism λ of $u \otimes X \simeq X$,
4. for any object X, a natural isomorphism ρ of $X \otimes u \simeq X$, and
5. for any objects X, Y and Z, a natural isomorphism α of $(X \otimes Y) \otimes Z \simeq X \otimes (Y \otimes Z)$.

They satisfy the following diagrams for any objects X, Y, Z and W:

$$\begin{tikzcd}
(X \otimes u) \otimes Y \arrow[r, "\rho \otimes 1_Y"] \arrow[d, "\alpha"'] & X \otimes Y \\
X \otimes (u \otimes Y) \arrow[ur, "1_X \otimes \lambda"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
(X \otimes Y) \otimes (Z \otimes W) \arrow[dr, "\alpha"] & \\
((X \otimes Y) \otimes Z) \otimes W \arrow[u, "\alpha"] \arrow[d, "\alpha \otimes 1_W"] & X \otimes (Y \otimes (Z \otimes W)) \\
(X \otimes (Y \otimes Z)) \otimes W \arrow[r, "\alpha"] & X \otimes ((Y \otimes Z) \otimes W) \arrow[u, "1_X \otimes \alpha"']
\end{tikzcd}$$

The associativity of the natural isomorphism α is problematic as $(X \otimes Y) \otimes Z$ has type `Functor` $((\mathcal{C} \times \mathcal{C}) \times \mathcal{C})\ \ \mathcal{C}$, while $X \otimes (Y \otimes Z)$ has type `Functor` $(\mathcal{C} \times (\mathcal{C} \times \mathcal{C}))\ \ \mathcal{C}$. As the domains are not definitionally equal, there cannot be a natural isomorphism between them. For type correctness, one possible solution is to precompose the first functor with an associator from $(\mathcal{C} \times \mathcal{C}) \times \mathcal{C}$ to $\mathcal{C} \times (\mathcal{C} \times \mathcal{C})$. This is not mere pedantry: we know that "one level up", this is an unavoidable issue. In other words, some issues that show up as type-checking problems in 1-category theory are actually previews of 2-categorical subtleties "peeking through", that can be ignored in paper-math. Our definition instead asks for the following data:

1. an isomorphism between $(X \otimes Y) \otimes Z$ and $X \otimes (Y \otimes Z)$, for any objects X, Y and Z, and
2. two naturality squares to complement the missing laws so that the isomorphism above is natural.

This leads to a definition that is easier to use, and the required natural isomorphism becomes a theorem.

3.3 Closed Monoidal Category

Intuitively, a closed monoidal category is a category possessing both a closed and a monoidal structure, in a compatible way. In the literature, we can find various definitions of a closed monoidal category:

1. (a monoidal category with an added closed structure): given a monoidal category (with bifunctor $\otimes$), there is also a family of functors $[X, -]$ for each object X, such that $- \otimes X \dashv [X, -]$. The closed bifunctor (or inner hom) $[-, -]$ is then induced uniquely up to natural isomorphism.
2. (a closed category with an added monoidal structure): given a closed category with bifunctor $[-, -]$, it is additionally equipped with a family of functors $- \otimes X$ for each object X, such that $- \otimes X \dashv [X, -]$. The monoidal bifunctor $\otimes$ is then induced uniquely up to natural isomorphism.

3. (via a natural isomorphism of hom-sets): given a category, for each object X, there are two families of functors $- \otimes X$ and $[X, -]$, such that the isomorphism $Hom(Y \otimes X, Z) \simeq Hom(Y, [X, Z])$ is natural in X, Y and Z. Both bifunctors $\otimes$ and $[-, -]$ are then induced uniquely up to natural isomorphism.

Note that the third definition above is not biased towards either the closed or monoidal structure. All three can be shown equivalent (classically). But in the proof-relevant setting, problems arise. One problem that all three definitions share is that they all induce at least one bifunctor from a *family* of functors. For example, in the first definition, the closed bifunctor $[-, -]$ is the result of a theorem; two different instances of $[-, -]$ (which might potentially differ in their proofs) can only be related by a natural isomorphism, which is often too weak. In other words, we want *both* bifunctors $\otimes$ and $[-, -]$ to be part of the definition so that they can be constructed elsewhere and they are related by other laws. None of the three definitions above satisy this requirement. We thus arrive at the following definition, which is the one we use:

Definition 3.4. A closed monoidal category is a category with two bifunctors $\otimes$ and $[-, -]$, so that

1. $\otimes$ satisfies the laws of a monoidal category,
2. $- \otimes X \dashv [X, -]$ for each object X, and
3. for a morphism $f : X \Rightarrow Y$, the induced natural transformations $\alpha_f : - \otimes X \Rightarrow - \otimes Y$ and $\beta_f : [Y, -] \Rightarrow [X, -]$ form a mate (or a conjugate in the sense of [21]) for the two adjunctions, $- \otimes X \dashv [X, -]$ and $- \otimes Y \dashv [Y, -]$, formed by previous constraint.

This definition is *better*, in the sense that it is 1) unbiased, 2) incremental (it simply adds more constraints on both bifunctors). Further note that both bifunctors are given as part of the data, rather than derived, which allows us to consistently refer to both uniquely. The following theorem strengthens our confidence:

Theorem 3.5. *A closed monoidal category according to Definition 3.4 is a closed category.*

In addition, the closed bifunctor $[-, -]$ from the closed category in this theorem is definitionally the same one given in Definition 3.4. This allows closed monoidal categories to inherit all properties of closed categories as they are talking about precisely the same $[-, -]$.

A potential downside of this definition is that it depends on *mates* which are not present in previous definitions. Though this seems to add complexity, we argue that the benefit is worth the effort. We now discuss mates in order to justify that this new definition is equivalent to the previous three.

3.4 Mate

Mates express naturality between adjunctions. They are typically defined by two natural isomorphisms between hom-sets as follows:

Definition 3.6. For functors $F, F' : C \Rightarrow \mathcal{D}$ and $G, G' : \mathcal{D} \Rightarrow C$, two natural transformations $\alpha : F \Rightarrow F'$ and $\beta : G' \Rightarrow G$ form a mate for two pairs of adjunctions $F \dashv G$ and $F' \dashv G'$, if the following diagram commutes:

$$\begin{tikzcd} Hom(F'X, Y) \arrow[r, dotted, "\simeq"] \arrow[d, "Hom(\alpha_X, Y)"'] & Hom(X, G'Y) \arrow[d, "Hom(X, \beta_Y)"] \\ Hom(FX, Y) \arrow[r, dotted, "\simeq"] & Hom(X, GY) \end{tikzcd}$$

This definition is not very convenient because it is defined via hom-set(oid)s. The situation described in Sections 2.2.4 and 3.1 recurs, and the two natural isomorphisms need to be composed by lifting functors in order to be well-typed. As before, there is another definition which does not depend on hom-sets.

Definition 3.7. For functors $F, F' : C \Rightarrow \mathcal{D}$ and $G, G' : \mathcal{D} \Rightarrow C$, two natural transformation $\alpha : F \Rightarrow F'$ and $\beta : G' \Rightarrow G$ form a mate for two pairs of adjunctions $(\eta, \epsilon) : F \dashv G$ and $(\eta', \epsilon') : F' \dashv G'$, if the following two diagrams commute:

$$\begin{tikzcd} 1_C \arrow[r, "\eta"] \arrow[d, "\eta'"'] & GF \arrow[d, "G\alpha"] \\ G'F' \arrow[r, "\beta F'"] & GF' \end{tikzcd} \qquad \begin{tikzcd} FG' \arrow[r, "\alpha G'"] \arrow[d, "F\beta"'] & F'G' \arrow[d, "\epsilon'"] \\ FG \arrow[r, "\epsilon"] & 1_{\mathcal{D}} \end{tikzcd}$$

Both definitions are equivalent [21], but Definition 3.7 is simpler to work with in our setting.

From here, it is straightforward to see that our definition of closed monoidal category is equivalent to the previous ones. We need to show Definition 3.4 is equivalent to requiring $Hom(Y \otimes X, Z) \simeq Hom(Y, [X, Z])$ to be natural in X, Y, and Z. Since we require $- \otimes X \dashv [X, -]$ for any object X, this requirement is equivalent to naturality of Y and Z. Moreover, the naturality of X is ensured by the mate condition, due to Definition 3.6.

3.5 Morphism Equality over Natural Isomorphism

Our experience with monoidal and closed monoidal categories can be generalized into a guideline. We find that in general, characterization in morphism equalities (such as triangle identities in Definition 3.2) is better than one in natural isomorphisms (such as the natural isomorphism between hom-sets in Definition 3.1 and the associativity natural isomorphism in Definition 3.3). The latter can be proved as a theorem.

We observe that natural isomorphisms tend to be more difficult to type-check, for a variety of reasons. Similar phenomena are also observed in concepts with higher structures, e.g. `Bicategory`, which we encoded directly using morphism equality to ease the type checking process.

3.6 Finite Categories

Category theorists have developed terminology to talk about the cardinalities (sizes) of components of a category. In Section 2.2.4, we use universe levels to make size issues explicit. For small categories, since we know both objects and morphisms "fit" in sets, we can use more set-theoretic language. Among these, "finiteness" is of particular importance, especially in its guise as enabling **enumeration** and its relation with topoi.

However when we attempt to define finite categories, a problem arises: MLTT does not give us primitives to count the elements of a type. For example both [32] and [38] implement finiteness as a predicate requiring an isomorphism between a type and `Fin ℕ`. We could also do this, but that approach has the drawback of (implicitly) putting a canonical order on elements, which is undesirable[4] It also forces a notion of equivalence on objects, which does not always exist for any `Set`. We do not want finiteness to force us into strictness. We instead base our definition on adjoint equivalence:

Definition 3.8. Two categories $\mathcal{C}$ and $\mathcal{D}$ are adjoint equivalent if there are two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ so that they form a pair of adjoint functors $F \dashv G$ and their unit and counit natural transformations are isomorphisms.

Then a finite category can be defined as follows:

Definition 3.9. A category $\mathcal{C}$ is finite, if it is adjoint equivalent to a finite diagram.

We could potentially use other notions of equivalence between categories, e.g. strong equivalence, but adjoint equivalence is special in its smooth interaction with (co)limits, as will be shown in Theorem 3.12. A strong equivalence only achieves this via its induced adjoint equivalence, so we chose to formulate it more directly.

We define a finite diagram using a type family `Fin : ℕ →` `Set` representing the discrete finite set of natural numbers $[0, n-1]$ defined in the standard library:

Definition 3.10. Given $n : \mathbb{N}$ as the number of objects and a function $|a, b| : \mathbb{N}$ for a, b : `Fin x`, a finite diagram is a category with

1. `Fin n` as objects, and
2. `Fin` $|a, b|$ as morphisms for `a, b : Fin n`.

if the morphisms satisfy the categorical laws of composition with propositional equality.

Intuitively, $|a, b|$ defines an enumeration of the morphisms. In this category, we make objects and morphisms discrete, so that propositional equality can be properly used.

[4]Propositional truncation could be used, if we had it, to get around this problem.

For example, as adjoint equivalence respects equivalence, a contractible groupoid is always finite. Note that this method could sometimes be challenging: coming up with such an adjoint equivalence can be difficult and, in some cases, may require the Axiom of Choice.

Nevertheless, the above definition lets use prove:

Theorem 3.11. *A category with all finite products and equalizers has all finite limits.*

The proof is constructive, i.e. an algorithm that builds a finite limit from products and equalizers given any finite diagram. In this theorem, finite limits are described by functors mapping out of special categories defined in Definition 3.10 instead of the more general Definition 3.9. This theorem at least ensures the sufficiency of Definition 3.10.

We can then move on to verifying that a finite category as per Definition 3.9 can serve as an index category for a finite limit in the general case. This can be seen from the following theorem:

Theorem 3.12. *Limits respect adjoint equivalence, i.e. if $\mathcal{J}$ is adjoint equivalent to $\mathcal{J}'$ with $F : \mathcal{J}' \to \mathcal{J}$, then for a functor $L : \mathcal{J} \to \mathcal{C}$, $\varprojlim L = \varprojlim (L \circ F)$.*

Combining the two theorems above, we can conclude that Definition 3.9 is an adequate definition of finite categories. That Definition 3.9 does not involve any explicit isomorphism between objects and some finite natural numbers is a strength. How much the choice of adjoint equivalence reveals about the inner structure of a category still remains to be investigated.

3.7 Local Cartesian Closure of Setoids

Finally we discuss a complication in proving that the category of `Setoids` is locally cartesian closed. This is an especially interesting theorem to us because base change functors in locally cartesian categories are left adjoint to the dependent product functors. That implies that `Setoids` are a model for dependently typed language. This theorem shows some typical extra considerations when proof-relevance and setoids are involved, and how much implicit equational reasoning we use in classical settings.

Definition 3.13. Given a category $\mathcal{C}$ and its object X, a slice category $\mathcal{C}/X$ has

1. (Y, f) as objects for object Y of $\mathcal{C}$ and morphism $f : Y \Rightarrow X$,
2. as a morphism $h : Y \Rightarrow Z$ between (Y, f) and (Z, g), so that $g \circ h = f$.

X is the *base* of $\mathcal{C}/X$. Given an object (Y, f) in the slice category, we often simply refer to it as f as Y can be inferred.

Definition 3.14. A category $\mathcal{C}$ is cartesian closed when it is closed monoidal with cartesian products $\times$ as $\otimes$ and a terminal object as unit. The inner hom $[X, Y]$ between objects X and Y is the exponential, which is denoted as Y^X.

Definition 3.15. A locally cartesian closed category is a category in which all its slice categories are cartesian closed.

3.7.1 Classical construction.

Classically, products in the slice category $\mathcal{S}et/A$ are pullbacks in $\mathcal{S}et$. Exponentials can be observed from the following diagram:

$$\begin{tikzcd}
 & & D \arrow[ddl, "h"] \\
B \times_A C \arrow[r, "\pi_2"] \arrow[urr, "\alpha"] \arrow[d, "\pi_1"] & C \arrow[d, "g"] & \\
B \arrow[r, "f"] & A &
\end{tikzcd}$$

where $B \times_A C$ is an object in $\mathcal{S}et$ and is a pullback of f and g. From the pullback diagram we want to get an idea of the exponential of h and g, h^g. From the diagram and that α is a slice morphism, we know

$$\begin{aligned} Hom(B \times_A C, D) = &\{\alpha : B \times_A C \to D \mid \\ &\forall (b, c) \in B \times_A C. h(\alpha(b, c)) = f(b) = g(c)\} \end{aligned}$$

If $\mathcal{S}et/A$ is cartesian closed, then we can find the exponentials via their right adjointness to pullbacks. Assuming the exponential object h^g is a morphism from X to A, adjointness insures that the isomorphism $Hom(B \times_A C, D) \simeq Hom(B, X)$ exists. If we were not working with a slice category, the left-to-right effect is simple, namely just currying,

$$b : B \mapsto c : C \mapsto \alpha(b, c)$$

However, in the slice category, we must ensure that the coherence condition holds, i.e. $h(\alpha(b, c)) = f(b) = g(c) \in A$. Thus the exponential in the slice category must carry $f(b)$ and a function, so we have

$$X = \Sigma_{a:A}(g^{-1}(a) \to h^{-1}(a))$$

That is, as a set, X is a (dependent) pair where the second component is a function from the inverse image of g of a to one of h of a. $Hom(B, X)$ is obtained from $\alpha \in Hom(B \times_A C, D)$ by:

$$b : B \mapsto (f(b) : A, c : g^{-1}(f(b)) \mapsto \alpha(b, c))$$

The presentation contains many hidden details: we can apply α to c because $g^{-1}(f(b))$ is a subset of C, and we know $\alpha(b, c)$ is in $h^{-1}(f(b))$ because $h(\alpha(b, c)) = f(b)$. Coherence conditions are elided as they can be recovered from the structure of sets.

3.7.2 In Setoids.

We cannot directly use this kind of reasoning in `Setoids`, as we handle setoid morphisms instead. Thus we need a notion of an inverse image *setoid* which respects setoid equivalence in the codomain. So for some setoid morphism f with codomain A, if we have $a \approx a' : A$, then setoids $f^{-1}(a)$ and $f^{-1}(a')$ should have the same extensional behaviours. This observation is captured by the following theorem:

```
inverseImage-transport :
  ∀ {a a′} {f : X ⟶ A} → a A.≈ a′ →
    InverseImage a f → InverseImage a′ f
```

where `f : X ⟶ A` specifies that `f` is a setoid morphism from setoid `X` to setoid `A`. `InverseImage a f` formalizes $f^{-1}(a)$ by requiring some element `x` of setoid `X` to satisfy `f x A.≈ a` in setoid `A`. Moreover, to formalize $g^{-1}(a) \to h^{-1}(a)$ in `Setoids`, it is not enough to just provide a function `i : InverseImage a g → InverseImage a h`, because `InverseImage` contains a proof of `f x A.≈ a` for some `x`. We need an extra coherence condition stating that this proof is irrelevant from `i`'s perspective. That is, given two `InverseImage`s with `x` and `y` as the underlying elements of `X`, if `x ≈ y`, then `i x ≈ i y`. These two pieces of information are bundled in `InverseImageMap a g h`, which we use to represent the map between inverse image setoids $g^{-1}(a) \to h^{-1}(a)$. Finally we need the following theorem to ensure that a `InverseImageMap` respects A's equivalence:

```
inverseImageMap-transport : ∀ {a a′}
  {g : C ⟶ A} {h : D ⟶ A} → a A.≈ a′ →
  InverseImageMap a g h →
  InverseImageMap a′ g h
```

These definitions and theorems fill in the elided coherence conditions in the classical settings. We can proceed to define an exponential of `h` and `g` in `Setoids / A` as a Σ type:

```
Σ (a : A) (InverseImageMap a g h)
```

This type does form a setoid with the corresponding setoid equivalence between `a` and the underlying map of `InverseImageMap`, which is the exponential of `Setoids / A`. By letting the identity morphism as the terminal object and pullbacks as products, we can conclude that `Setoids` is locally cartesian closed.

4 Discussion

The previous section detailed decisions that lie in the intersection of category theory and formalization in type theory, here we document software engineering decisions as well as comment on efficiency issues.

4.1 Module Structure

The previous library favoured a flat module structure, we use a deeper hierarchy, and thus fewer top-level modules. We use the following principles as a guide:

1. Important concepts have their top level modules. For example, `Category`, `Object`, `Morphism`, `Diagram`, `Functor`, `NaturalTransformation`, `Kan`, `Monad` and `Adjoint` belong to this category.
2. Different flavours of category theory are also on the top level: `Category`, `Enriched`, `Bicategory` and `Minus2-Category` contain the definitions and properties of categories, enriched categories, bicategories

Table 1. Tools and key characteristics of various libraries

libraries	proof assistants	foundation	hom-setoids	proof-relevant	LoC†
Ours	Agda 2.6.1	MLTT	✓	✓	23998
[26]	Agda 2.5.2	MLTT + K + irrelevance	✓	✗	11770
[32]	Coq 8.11.1	CIC	✗	✗	14711
[37]	Coq 8.10.2	CIC	✓	✓	23003
[18]	Coq 8.12.0	CIC	✓	✓	7879
[4, 35]	Coq 8.12.0	HoTT	✗	✓	96366
[16]	Hoq 8.12††	HoTT with HIT	✗	✓	10604
[24]	Lean	CIC	✗	✗	14975
[28–30]	Isabelle	HOL	✗	✗	82782

† The lines of code are counted by `cloc` of Al Danial and code in Isabelle is counted by `wc`, because `cloc` does not recognize Isabelle. The lines of code might include documentation text. Only folders directly related to category theory are counted.
†† Hoq is a a modified version of Coq which implements a part of HoTT.

and -2-categories, respectively. `Pseudofunctor` contains the instances of pseudofunctors.

Submodules also follow conventions so that definitions and properties are easier to locate.

1. `*.Instance` contains instances of some concept. For example, the category of all setoids is defined in `Category.Instance`. Generally, only instances that are re-used in the library itself (making them "special") are defined.
2. `*.Construction` contains instances induced from some input. The difference with `*.Instance` is that `*.Construction` takes parameters beyond just `Levels`. For example, the Kleisli category of a monad is defined in `Category.Construction`.
3. `*.Properties` contains properties of the corresponding concepts.
4. `*.Duality` contains conversions to dual concepts (see Section 2.3).

This module structure was inspired by a recent restructuring of Agda's standard library along similar lines, which we believe helps users find what they need faster.

4.2 Hierarchy of Concepts

Similar to [14–16, 27], we need to decide how concepts are organized. Unlike Coq, which many cited works are based on, Agda does not have features like canonical structures or hint based programming. But, like the standard library, we do not wish to use type classes. One reason is performance: at this moment, type classes in Agda are fairly slow (compared to, say, Coq), potentially penalizing downstream librairies and end users. Nevertheless, we still need to organize our library so that concepts can be found.

At the lowest level, we rely on records and unification. There are typically two choices to represent a concept: predicates or structures. A predicate has the data "unbundled"; it expresses an "is-a" relation. A structure on the other hand is "bundled" and expresses a "has-a" relation. The previous library, and many other implementations too, chose to either bundle or unbundle. From a type-theoretic perspective, this choice is irrelevant, but is nevertheless quite important from a usability perspective. It is even possible to automatically map from one style to another [6]; unfortunately, such mapping is meta-theoretical in current Agda. As such a choice is unforced, we decided to implement both.

Wrapping Predicates. Structures are obtained by wrapping predicates. Influenced by the previous library [26], many concepts related to `Category` are represented as predicates:

```
record Monoidal {o ℓ e} (C : Category o ℓ e)
    : Set (o ⊔ ℓ ⊔ e) where
```

It asserts that `C` *is* a monoidal category. At other times, e.g. when working with two monoidal categories, we want to represent monoidal categories as a structure. We provide definitions in both styles:

```
record MonoidalCategory o ℓ e
    : Set (suc (o ⊔ ℓ ⊔ e)) where
  field
    U        : Category o ℓ e
    monoidal : Monoidal U
```

`U` stands for "underlying". This allows us to define (lax) monoidal functors, which are functors preserving the monoidal structure:

```
record MonoidalFunctor
  (C : MonoidalCategory o ℓ e)
  (D : MonoidalCategory o′ ℓ′ e′)
  : Set (o ⊔ ℓ ⊔ e ⊔ o′ ⊔ ℓ′ ⊔ e′) where
```

The alternative formulation using the predicate representation is more verbose:

```
record MonoidalFunctor′
  {C : Category o ℓ e} {D : Category o′ ℓ′ e′}
  (MC : Monoidal C)    (MD : Monoidal D)
  : Set (o ⊔ ℓ ⊔ e ⊔ o′ ⊔ ℓ′ ⊔ e′) where
```

Table 2. Feature comparison (part 1)

Features	Ours	[26]	[32]	[37]	[18]	[4, 35]	[16]	[24]	[28–30]
basic structures:									
initial / terminal	✓	✓	✓	✓	✓	✓	✓	✓	✓
product / coproduct	✓	✓	✓		✓	✓		✓	✓
limit / colimit	✓	✓	✓	✓	✓	✓	✓	✓	✓
end / coend	✓	✓		✓		✓			
exponential	✓	✓	✓		✓	✓			
categorical structures:									
product / coproduct†	✓	✓	✓	✓	✓✗	✓✗	✓✗	✓	✓✗
comma category	✓	✓	✓	✓	✓	✓	✓	✓	
cartesian category	✓	✓	✓	✓	✓			✓	
closed category	✓								
CCC	✓	✓	✓	✓	✓			✓	
LCCC	✓		✓						
biCCC	✓			✓					
rig category	✓	✓				✓			
topos	✓		✓						
Grothendieck topos						✓			
Eilenberg Moore	✓			✓				✓	
Kleisli	✓			✓		✓		✓	
monoidal category	✓			✓		✓		✓	✓
Kelly's coherence [19]	✓								✓
closed monoidal category	✓							✓	
closed monoidal categories are closed categories	✓								
braided monoidal category	✓	✓		✓		✓		✓	
symmetric monoidal category	✓	✓		✓				✓	
traced monoidal category	✓	✓							
lax monoidal functor	✓			✓		✓		✓	✓
strong monoidal functor	✓			✓		✓			
instances:									
Cats	✓	✓	✓	✓	✓	✓	✓	✓	
Set(oid)s	✓	✓	✓	✓	✓	✓	✓	✓	✓
Setoids are complete / cocomplete	✓		✓	✓	✓				
Setoids are cartesian closed	✓		✓	✓	✓				
Setoids are locally cartesian closed	✓		✓						
simplicial set	✓					✓			
functor	✓	✓	✓	✓	✓	✓	✓	✓	✓
(co)limit functor	✓		✓			✓	✓		✓
Hom functor	✓	✓	✓	✓	✓	✓	✓	✓	✓
Hom functors preserve limits	✓		✓		✓				
T-algebra	✓	✓	✓	✓		✓			
Lambek's lemma	✓	✓				✓			
natural transformation	✓	✓	✓	✓	✓	✓	✓	✓	✓
dinatural transformation	✓	✓		✓					
enriched category	✓	✓		✓		✓			
2-category	✓	✓				✓			✓
bicategory	✓	✓		✓		✓			✓
pseudofunctor	✓	✓				✓	✓		✓
Yoneda lemma	✓	✓	✓	✓	✓	✓	✓	✓	✓

† ✓✗ indicates that these libraries only implement product categories.

Table 3. Feature comparison (part 2)

Features	Ours	[26]	[32]	[37]	[18]	[4, 35]	[16]	[24]	[28–30]
Grothendieck construction	✓	✓					✓	✗††	
presheaves	✓	✓	✓	✓		✓		✓	
are complete / cocomplete	✓		✓					✓	
are cartesian closed	✓		✓						
are topos			✓						
adjoint functors	✓	✓	✓	✓	✓	✓	✓	✓	✓
adjoint composition	✓	✓	✓	✓			✓	✓	✓
Right(left) adjoints preserve (co)limits	✓		✓		✓	✓		✓	✓
Adjoint functors induce monads	✓			✓		✓		✓	
(Co)limit functors are left(right) adjoint to diagonal functor†	✓		✓	✗		✗	✓		✓
mate (conjugate)	✓					✓			
adjoint functor theorem	✓		✓		✓				
Kan extension	✓	✓	✓	✓		✓	✓		
(Co)limit is kan	✓		✓	✓		✓			
Kan extensions are preserved by adjoint functors			✓						
Rezk completion						✓			

† ✗ indicates that these libraries only show a special case of the theorem.
†† ✗ indicates that [24] only implements the category of elements.

When working with monoidal functors, we do not mean to assert that some category is monoidal but rather want to refer to some structured category as a whole.

In general, definitions in the structure style are defined in modules of the form `*.Structure`. As the previous library used the predicate style, we started our in that style as well and then provided wrapped structure versions. As a rule of thumb, when working with one particular concept, we often use the predicate style so that the conclusions can be easily accessed by both styles. For example, we formulate theorems about monoidal categories using the predicate style.

The paper [6] further discusses (un)bundling of definitions, along with tools for moving between the two equivalent styles.

Choosing Predicates. Next we use cartesian products to illustrate how we design predicate formulations. We have the following structure-kind definition for products:

```
record Product (A B : Obj) : Set _ where
```

The record contains projections, product morphisms, and necessary laws for a product. This definition works very well when we work on *one* category. However, when we work with two categories, then we need a predicate version:

```
record IsProduct {A B P}
  (π₁ : P ⇒ A) (π₂ : P ⇒ B) : Set _ where
```

In the arguments, `P` represents the product of `A` and `B`, and π_1 and π_2 are the projections. It is possible to have a slightly different predicate definition:

```
record IsProduct′ {A B P}
  (π₁ : P ⇒ A) (π₂ : P ⇒ B)
  (⟨_,_⟩ : C ⇒ A → C ⇒ B → C ⇒ P) :
  Set _ where
```

where `⟨ f , g ⟩` denotes the product of morphism `f` and g. We did not choose this form because `⟨ f , g ⟩` is uniquely determined by π_1 and π_2! That is, even if `IsProduct` allows a "different" `⟨ f , g ⟩′`, they are provably equivalent. In general, when formulating concepts defined by universal properties, we can omit the universal part in the predicate form due to uniqueness.

4.3 Efficiency

Basic category theory typechecks very quickly, both online (via Emacs) and offline (via calling the `agda` compiler). But for "deeper" category theory, such as properties associated to the Yoneda lemma and properties of Bicategories, typechecking gets noticeably slower and memory use goes up. One of the culprits is the module style as documented in Section 2.4: such modules are copied and rechecked, which is quite inefficient. This is why when we use local modules (either `private` or in `where` clauses) we qualify them with `using` to only copy the parts we need.

Unfortunately that same trick does not work for global `open import` (for sound reasons). Agda's `.agdai` file format is very information-rich (i.e. the files are quite large), and full transitive dependencies must be read. Splitting developments into smaller files to minimize the dependency tree has lead to substantial improvements in the compilation time and memory use of the full library. The downside is that some usability features have had to be sequestered into sub-modules that are then imported on an as-needed basis.

5 Related Work

Table 1 gives a list of formalized libraries of category theory. For each we specify the proof assistant, the foundation, lines of code and whether it uses hom-setoids and is proof-relevant. In Tables 2 and 3, we compare a list of features implement by these libraries.

We have ported all definitions and theorems from [26], except those requiring UIP or axiom K. We reuse [26] as much as we can. We also extend it with many new definitions and new theorems, as shown in Tables 2 and 3 (more than twice as much material). Moreover, since we turn on the `--safe` flag, we do not have `postulates` in our code base. This helps us to avoid inheriting a postulated unsound axiom [1], which would, for example, let us incorrectly mix relevance and irrelevance, including "recovering" a relevant value from an irrelevant one.

From Table 1, we can see that much effort has been spent in Coq (or its Hoq dialect) on category theory. The reason for the multiple efforts can be seen when comparing the versions, and foundations used. These libraries also vary in their design and organization. Some believe that Coq's tactics and hint databases provide a significant boost in the productivity of formalizations. We suspect that this may be somewhat illusory, as the explicit equational proofs in n-category theory (which can be automated via tactics) tend to turn up as data in $n + 1$-category theory, and then no longer avoidable. [37] stands out by its use of other Coq mechanisms, such as type classes, rather than record or Σ types, for structuring of the development.

Like us, [18, 37] use hom-setoids and proof-relevance. Unfortunately, [37] has not been described in a paper, so we do not know what lessons the authors learned from their experience. [18] was a smaller scale but pioneering effort that taught us the basics of formalizing category theory in MLTT, but not the kinds of design decisions we faced here.

Compared to other developments in Coq, [16, 35] are special: they build category theory in HoTT. [35] focuses more on fundamental constructions. It does not use any feature beyond the primitive type constructors like Σ and Π. By contrast, [16] experiments with the use of various HoTT ideas, and therefore is more permissive. It uses extended features like records and higher inductive types (HITs). Working in HoTT has some advantages. First, if one understands hom-sets to be literally classical sets, rendered as `hSets` in HoTT, this is straightforward. In HoTT this also implies that `hSets` have unique identity proofs, which make their equational proofs proof-irrelevant, which is closer to the set-based understanding of classical category theory. Second, HoTT has a very natural way of expressing universal properties. Using Martin-Löf type theories, e.g. ours, [18, 26, 32, 37], universal properties are usually stated in two parts: a universal part returning a morphism and a uniqueness part equating morphisms from the universal part. In HoTT, this can be expressed compactly as constructing a contractible morphism. Third, since HoTT supports the univalence, one can conflate isomorphisms and equalities. In both libraries, categories are defined with an additional law stating that isomorphic objects are equal, which provides a way to handle equal objects in a category which ours does not have.

The mathematical library of Lean [13], mathlib [24], also implements some category theory[5]. As Lean has proof-irrelevance built in and mathlib uses propositional equality, its category theory library is very classical.

Category theory has also been formalized in Nuprl [11], Idris [8] and Isabelle [25]. Due to space limitation, we are not able to fully survey all of them. We refer interested readers to [16] and the Coq discourse forum[6] for a more thorough list of formalizations of category theory.

6 Conclusion and Future Work

We implemented proof-relevant category theory in Agda, successfully. The concepts covered, and the theorems proved, are quite broad. We did not find any real barrier to doing so — strictness and hom-sets are not necessary features of modern category theory. We did find that some definitions work better than others, which we have explained in detail. Comparing with other libraries, we find that ours covers quite similar grounds, and often more.

We are still actively developping this library — many theorems of classical category theory remain; both bicategory theory and enriched category theory are being built up. Some work has been done on "negative thinking" (-2-categories, etc) and should be extended. Both double categories and higher categories are still awaiting, along with multicategories, PROPs, operads and polycategories. We also intend to move parts of this library to the standard library.

Performance needs another look. Even after some optimizations were performed, it still takes more memory and time to typecheck than we would prefer. Having said that, development can easily be done on a normal laptop, so the problem is not severe, unlike with other libraries.

Acknowledgments

We would like to thank Sandro Stucki, Reed Mullanix, Nathan van Doorn, and many others for discussions and contributing to the library. We are also grateful to the anonymous reviewers for their inspirational suggestions.

This work was supported by the National Sciences and Engineering Research Council of Canada.

[5]This library is being actively developed. Our survey is valid as of mid-September 2020 and does not consider the open PRs to the main library.
[6]https://coq.discourse.group/t/survey-of-category-theory-in-coq/371/4.

References

[1] Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:29)2012

[2] Peter Aczel. 1993. Galois: a theory development project. *manuscript, University of Manchester* (1993).

[3] Agda Team. 2019. Agda 2.6.0.1.

[4] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. 2015. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science* 25, 5 (2015), 1010–1039. https://doi.org/10.1017/S0960129514000486

[5] Benedikt Ahrens and Paige Randall North. 2019. *Univalent Foundations and the Equivalence Principle.* Springer International Publishing, Cham, 137–150. https://doi.org/10.1007/978-3-030-15655-8_6

[6] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. 2019. A language feature to unbundle data at will (short paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm (Eds.). ACM, 14–19. https://doi.org/10.1145/3357765.3359523

[7] Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press, Inc., New York, NY, USA.

[8] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[9] Jacques Carette and Amr Sabry. 2016. Computing with Semirings and Weak Rig Groupoids. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 123–148. https://doi.org/10.1007/978-3-662-49498-1_6

[10] Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 257–268. https://doi.org/10.1145/2628136.2628139

[11] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

[12] Nils Anders Danielsson, Ulf Norell, Shin-Cheng Mu, Bradley Hardy, Samuel Bronson, Dan Doel, Patrik Jansson, Liang-Ting Chen, Jean-Philippe Bernardy, Andrés Sicard-Ramírez, Nicolas Pouillard, Darin Morrison, Peter Berry, Daniel Brown, Simon Foster, Dominique Devriese, Andreas Abel, Alcatel-Lucent, Eric Mertens, Joachim Breitner, Liyang Hu, Noam Zeilberger, Érdi Gergő, Stevan Andjelkovic, Helmut Grohne, Guilhem Moulin, Noriyuki Ohkawa, Evgeny Kotelnikov, James Chapman, Wen Kokke, Matthew Daggitt, Jason Hu, Sandro Stucki, Milo Turner, Zack Grannan, and Lex van der Stoep. 2019. agda-stdlib: The Agda standard library. https://github.com/agda/agda-stdlib

[13] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[14] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings.* 327–342. https://doi.org/10.1007/978-3-642-03359-9_23

[15] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. 2002. A Constructive Algebraic Hierarchy in Coq. *J. Symb. Comput.* 34, 4 (2002), 271–286. https://doi.org/10.1006/jsco.2002.0552

[16] Jason Gross, Adam Chlipala, and David I. Spivak. 2014. 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 (Lecture Notes in Computer Science, Vol. 8558)*, Gerwin Klein and Ruben Gamboa (Eds.). Springer, 275–291. https://doi.org/10.1007/978-3-319-08970-6_18

[17] Martin Hofmann and Thomas Streicher. 1996. The Groupoid Interpretation of Type Theory. In *In Venice Festschrift*. Oxford University Press, 83–111.

[18] Gérard P. Huet and Amokrane Saïbi. 2000. Constructive category theory. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 239–276.

[19] G.M Kelly. 1964. On MacLane's conditions for coherence of natural associativities, commutativities, etc. *Journal of Algebra* 1, 4 (1964), 397 – 402. https://doi.org/10.1016/0021-8693(64)90018-3

[20] F. William Lawvere. 1964. An Elementary Theory of the Category of Sets. *Proceedings of the National Academy of Sciences of the United States of America* 52, 6 (1964), 1506–1511. http://www.jstor.org/stable/72513

[21] Saunders MacLane. 1971. *Categories for the Working Mathematician.* Springer-Verlag, New York. ix+262 pages. Graduate Texts in Mathematics, Vol. 5.

[22] M. Makkai. 2017. *Towards a Categorical Foundation of Mathematics.* Cambridge University Press, 153–190. https://doi.org/10.1017/9781316716830.014

[23] Per Martin-Löf. 1984. *Intuitionistic type theory.* Studies in proof theory, Vol. 1. Bibliopolis.

[24] The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. https://doi.org/10.1145/3372885.3373824

[25] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[26] Daniel Peebles, James Deikun, Ulf Norell, Dan Doel, Darius Jahandarie, and James Cook. 2018. categories: Categories parametrized by morphism equality in Agda. https://github.com/copumpkin/categories

[27] Bas Spitters and Eelis van der Weegen. 2011. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.* 21, 4 (2011), 795–825. https://doi.org/10.1017/S0960129511000119

[28] Eugene W. Stark. 2016. Category Theory with Adjunctions and Limits. *Archive of Formal Proofs* (June 2016). http://isa-afp.org/entries/Category3.html, Formal proof development.

[29] Eugene W. Stark. 2017. Monoidal Categories. *Archive of Formal Proofs* (May 2017). http://isa-afp.org/entries/MonoidalCategory.html, Formal proof development.

[30] Eugene W. Stark. 2020. Bicategories. *Archive of Formal Proofs* (Jan. 2020). http://isa-afp.org/entries/Bicategory.html, Formal proof development.

[31] Thomas Streicher. 1993. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität* (1993).

[32] Amin Timany and Bart Jacobs. 2016. Category Theory in Coq 8.5. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal (LIPIcs, Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:18. https://doi.org/10.4230/LIPIcs.FSCD.2016.30

[33] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[34] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *PACMPL* 3, ICFP (2019), 87:1–87:29. https://doi.org/10.1145/3341691

[35] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. [n.d.]. UniMath — a computer-checked library of univalent mathematics. available at https://github.com/UniMath/UniMath. https://github.com/UniMath/UniMath

[36] Michael A Warren. 2008. *Homotopy theoretic aspects of constructive type theory*. Ph.D. Dissertation. Carnegie Mellon University.

[37] John Wiegley. 2019. category-theory: Category Theory in Coq. https://github.com/jwiegley/category-theory

[38] Brent Yorgey. 2014. *Combinatorial species and labelled structures*. Ph.D. Dissertation. University of Pennsylvania.