Category Theory in Coq 8.5: Extended Version

Amin Timany   *Bart Jacobs*

Report CW 697, April 2016

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Category Theory in Coq 8.5: Extended Version

Amin Timany *Bart Jacobs*

Report CW 697, April 2016

Department of Computer Science, KU Leuven

Abstract

We report on our experience implementing category theory in Coq 8.5. Our work formalizes most of basic category theory, including concepts not covered by existing formalizations, in a library that is fit to be used as a general-purpose category-theoretical foundation.

Our development particularly takes advantage of two features new to Coq 8.5: primitive projections for records and universe polymorphism. Primitive projections allow for well-behaved dualities while universe polymorphism provides a relative notion of largeness and smallness. The latter is one of the main contributions of this paper. It pushes the limits of the new universe polymorphism and constraint inference algorithm of Coq 8.5.

In this paper we present in detail smallness and largeness in categories and the foundation they are built on top of. We furthermore explain how we have used the universe polymorphism of Coq 8.5 to represent smallness and largeness arguments by simply ignoring them and entrusting them to the universe inference algorithm of Coq 8.5. We also discuss our experience throughout this implementation, discuss concepts formalized in this development and give a comparison with a few other developments of similar extent. Furthermore, we discuss (future) works on top of or related to this development including our experience regarding our ongoing effort of porting a version of this development on top of the HoTT library.

Category Theory in Coq 8.5: Extended Version

Amin Timany Bart Jacobs

iMinds-DistriNet – KU Leuven

firstname.lastname@cs.kuleuven.be

Abstract

We report on our experience implementing category theory in Coq 8.5. Our work formalizes most of basic category theory, including concepts not covered by existing formalizations, in a library that is fit to be used as a general-purpose category-theoretical foundation.

Our development particularly takes advantage of two features new to Coq 8.5: primitive projections for records and universe polymorphism. Primitive projections allow for well-behaved dualities while universe polymorphism provides a relative notion of largeness and smallness. The latter is one of the main contributions of this paper. It pushes the limits of the new universe polymorphism and constraint inference algorithm of Coq 8.5.

In this paper we present in detail smallness and largeness in categories and the foundation they are built on top of. We furthermore explain how we have used the universe polymorphism of Coq 8.5 to represent smallness and largeness arguments by simply ignoring them and entrusting them to the universe inference algorithm of Coq 8.5. We also discuss our experience throughout this implementation, discuss concepts formalized in this development and give a comparison with a few other developments of similar extent. Furthermore, we discuss (future) works on top of or related to this development including our experience regarding our ongoing effort of porting a version of this development on top of the HoTT library [32].

1 Introduction

A category [20, 3] consists of a collection of objects and for each pair of objects A and B a collection of morphisms (aka arrows or homomorphisms) from A to B. Moreover, for each object A we have a distinguished morphism $id_A : A \to A$. Morphisms are composable, i.e., given two morphisms $f : A \to B$ and $g : B \to C$, we can compose them to form: $g \circ f : A \to C$. Composition must satisfy the following additional conditions:

$$\forall f : A \to B.\ f \circ id_A = f = id_B \circ f$$

$$\forall f, g, h.\ (h \circ g) \circ f = h \circ (g \circ f)$$

The notion of a category can be seen as a generalization of sets. In fact sets as objects together with functions as morphisms form the important category **Set**. On the other hand, it can be seen as a generalization of the mathematical concept of a preorder. In this regard, a category can be thought of as a preorder where objects form the elements of the preorder and morphisms from A to B can be thought of as "witnesses" of the fact that $A \preceq B$. Thus, identity morphisms are witnesses of reflexivity whereas composition of morphisms forms witnesses for transitivity and the additional axioms simply spell out coherence conditions for witnesses. Put concisely, categories are preorders where the quality and nature of the relation holding between two elements is important. In this light, categories are to preorders what intuitionistic logic is to classical logic. A combination of these two interpretations of categories can provide an essential and useful intuition for understanding most, if not all, of the theory.

This generality and abstractness is what led some mathematicians to call this mathematical theory "general abstract nonsense" in its early days. However category theory, starting from this simple yet vastly abstract and general definition, encompasses most mathematical concepts and has found applications not only in mathematics but also in other disciplines, e.g, computer science.

In computer science it has been used extensively, especially in the study of semantics of programming languages [24], in particular constructing the first (non-trivial) model of the untyped lambda calculus by Dana Scott (see [28]), type systems [19], and program verification [8, 6, 7].

Given the applications of category theory and its fundamentality on the one hand and the arising trend in formalizing mathematics in proof assistants on the other, it is natural to have category theory formalized in one; in particular, a formalization that is practically useful as a category-theoretical foundation for other works. This paper is a report of our experience formalizing such a library. There already exist a relatively large number of formalizations of category theory in proof assistants, e.g. [23, 27, 18, 2, 13]. However, most of these implementations are not general purpose and rather focus on parts of the theory which are relevant to the specific application of the authors. See the bibliography of [14] for an extensive list of such developments.

Features of Coq 8.5 used: η for records and universe polymorphism

This development makes use of two features new to Coq 8.5 which as of this writing is still under development. Namely, primitive projection for records (i.e., the η rule for records) and universe polymorphism.

Following [13], we use primitive projections for records which allow for well behaved dualities in category theory. The dual (aka opposite) of a category $\mathcal{C}$ is a category $\mathcal{C}^{op}$ which has the same objects as $\mathcal{C}$ where the collection of morphisms from A to B is swapped with that from B to A. Drawing intuition from the similarity of categories and preorders, the opposite of a category (seen as a preorder) is simply a category where the order of objects is reversed. Use of duality arguments in proofs and definitions in category theory are plentiful, e.g., sums and products, limits and co-limits, etc. One particular property of duality is that it is involutive. That is, for any category $\mathcal{C}$, $(\mathcal{C}^{op})^{op} = \mathcal{C}$. The primitive projection for records simply states that two instances of a record type are definitionally equal only if their projections are. In terms of categories, two categories are definitionally equal only if their object collections are, morphism collections are and so forth. This means that we get that the equality $(\mathcal{C}^{op})^{op} = \mathcal{C}$ is definitional. Similar results hold for the duality and composition of functors, for natural transformations, etc. That is we get definitional equalities such as the following:

$$(\mathcal{F}^{op})^{op} = \mathcal{F} \qquad (\mathcal{N}^{op})^{op} = \mathcal{N}$$

$$(\mathcal{F} \circ \mathcal{G})^{op} = \mathcal{F}^{op} \circ \mathcal{G}^{op}$$

where $\mathcal{F}$ and $\mathcal{G}$ are functors and $\mathcal{N}$ is a natural transformation.

To achieve well behaved dualities, in addition to primitive projections one needs to slightly adjust the definition of a category itself. More precisely, the definition of the category must carry a symmetric form of associativity of composition. The reason being the fact that for the dual category we can simply swap the proof of associativity with its symmetric form and thus after taking the opposite twice get back the proof we started with.

In this development we have used universe polymorphism, a feature new to Coq 8.5. to represent relative smallness/largeness. In short, universe polymorphism allows for a definition to be polymorphic in its universe variables. This allows us, for instance, to construct the category of (relatively small) categories directly. That is, the category constructed is at a universe level (again polymorphic) while its objects are categories at a lower universe level. We will elaborate the use of universe polymorphism to represent relative largeness and smallness below in Section 2.

Contributions

The main contributions of this development are its extent of coverage of basic concepts in category theory and its use of the universe polymorphism of Coq 8.5 and its universe inference algorithm to represent relative smallness/largeness.

The latter, as explained below, allows us to represent smallness and largeness using universe levels by simply forgetting about them and letting Coq's universe inference algorithm take care of smallness and largeness requirements as necessary.

Also, our experience of migrating from Coq 8.5 towards the HoTT library, despite not having high code base coverage has provided interesting lessons for us, as outsiders to the HoTT and HoTT library [32] projects, which we find worth sharing.

The structure of the rest of this paper

The rest of this paper is organized as follows. Section 2 gives an explanation of smallness and largeness in category theory based on the foundation used. This is followed by a detailed explanation of our use of the new universe polymorphism and universe constraint inference algorithm of Coq 8.5 to represent relative smallness/largeness of categories. There, we also give a short comparison of the way other developments represent (relatively) large concepts.

In Section 3, we give a high-level explanation of the concepts formalized and some notable features in this work. We furthermore provide a comparison of our work with a number of other works of similar extent. We also briefly discuss the axioms that we have used throughout this development.

In Section 4 we give a general overview of homotopy type theory and the representation of category theory in it. We particularly discuss the use of axioms in HoTT that we have used throughout our work. Section 5 outlines our on-going effort of migrating to the HoTT library and in particular how we have adapted or plan to adapt the use of axioms in our work so as to conform with HoTT premises.

Section 6 describes the work that we have done or plan to do which are based on the current work as category theoretical foundation. Finally, in Section 7 we conclude with a short summary of the paper.

Development source code The repository of our development can be found at [34]. The ongoing migration on top of the HoTT library can be found at [35].

2 Universes, Smallness and Largeness

A category is usually called small if its objects and morphisms form sets and large otherwise. It is called locally small if the morphisms between any two objects form a set but objects fail to. For instance, the category **Set** of sets and functions is a locally small category as the collection of all sets does not form a set while for any two sets, there is a set of functions between them. These distinctions are important when working with categories. For instance, a category is said to be complete if it has the limit of all *small* diagrams ($\mathcal{F} : \mathcal{C} \to \mathcal{D}$ is a small diagram if $\mathcal{C}$ is a small category). For instance, **Set** is complete but does not have the cartesian product of all large families of sets.

This terminology and considerations are due to the fact that the original foundations of category theory by Eilenberg and Mac Lane were laid on top of NGB (von Neumann-Gödel-Bernays) set theory. In NBG, in addition to sets, the notion of a class (a collection of sets which it self is not *necessarily* a set) is also formalized. For any property φ, there is a class C_φ of all sets that have property φ. If the collection of sets satisfying φ forms a set then C_φ is just that set. Otherwise, C_φ is said to be a proper class. In this formalism, one can formalize large categories but cannot use them. For instance, the functor category $\mathbf{Set}^{\mathbf{Set}}$ is not defined as its objects are already proper classes and there is no class of proper classes in NBG.

The other foundation that is probably the most popular among mathematicians is that of ZF with Grothendieck's axiom of universe. Roughly speaking, a Grothendieck universe V is a set that satisfies ZF axioms, e.g., if $A \in V$ and $B \in V$ then $\{A, B\} \in V$ (axiom of pairing), if $A \in V$ then $2^A \in V$ (axiom of power set), etc. We also have if $A \in B$ and $B \in V$ then $A \in V$. *Grothendieck's axiom* says that for any set x there is a Grothendieck universe V such that $x \in V$. This also implies that for any Grothendieck universe V, there is a Grothendieck universe V' such that $V \in V'$.

Working on top of this foundation, one can talk about V-small categories and use all the set-theoretic power of ZF. The notion of completeness for a V-small category can be defined as having all V-small limits. The category of all V-small sets will be a V'-small category where $V \in V'$. It is also a V-locally-small category as its set of morphisms are V-small but its set of objects are not. For more details on foundations for category theory see chapter 12 of [22].

The type hierarchy of Coq (also known as universes), as explained below, bears a striking resemblance to Grothendieck universes just explained. In the rest of this section we discuss how Coq's new universe polymorphism feature allows us to use Coq universes, instead of Grothendieck universes in completely transparent way. That is, we never mention any universes in the whole of the development and Coq's universe inference algorithm (part of the universe polymorphism feature) infers them for us.

2.1 Coq's Universes

In higher-order dependent type theories such as that of Coq, types are also terms and themselves have types. As expected, allowing existence of a type of all types results in self-referential paradoxes, such as Girard's paradox (see [11]). Thus, to avoid such paradoxes type theories like Coq use a countably infinite hierarchy of types of types (also known as universes): $\mathtt{Type}_0 : \mathtt{Type}_1 : \mathtt{Type}_2 : \ldots$ The type system of Coq additionally has the cumulativity property, i.e., for any term $\mathtt{T} : \mathtt{Type}_n$ we also have $\mathtt{T} : \mathtt{Type}_{n+1}$.

The type system of Coq has the property of *typical ambiguity*. That is, in writing definitions, we don't have to specify universe levels and/or constraints on them. The system automatically infers the constraints necessary for the definitions to be valid. In case, the definition is such that no (consistent) set of constraints can be inferred, the system rejects it issuing a "universe inconsistency" error. It is due to this feature that throughout this development we have not had the need to specify any universe levels and/or constraints by hand.

To better understand typical ambiguity in Coq, let's consider the following definition.

```
Definition Tp := Type.
```

In this case, Coq introduces a new universe variable for the level of the type `Type`. That is, internally, the definition looks like[1]:

```
Definition Tp : Type@{i+1} := Type@{i}.
```

Note that the universe level `i` above is global universe level, i.e., it is fixed. Hence, the following definition is rejected with a universe inconsistency error.

```
Definition TpTp : Tp := Tp.
```

The problem here is that this definition requires (`Type@{i} : Type@{i}`) which requires the system to add the constraint `i` < `i` which makes the set of constraints inconsistent. Without universe polymorphism, one way to solve this problem would be to duplicate the definition of `Tp` as `Tp'` which would be internally represented as:

```
Definition Tp' : Type@{j+1} := Type@{j}.
```

Now we can define `TpTp'`:

```
Definition TpTp' : Tp' := Tp.
```

which Coq accepts and consequently adds the constraint `i` < `j` to the global set of universe constraints. As these constraints are global however, after defining `TpTp'` we can't define `Tp'Tp`

```
Definition Tp'Tp : Tp := Tp'.
```

This is rejected with a universe inconsistency error as it requires `j` < `i` to be added to the global set of constraints which makes it inconsistent as it already contains `i` < `j` from `TpTp'`.

2.2 Universe Polymorphism

Coq has recently been extended (see [31]) to support universe polymorphism. This feature is now included in the recently released Coq 8.5. When enabled, universe levels of a definition are bound at the level of that definition. Also, any universe constraints needed for the definition to be well-defined are local to that definition. That is the definition of `Tp` defined above is represented internally as:

```
Definition Tp@{i} : Type@{i+1} := Type@{i}. (* Constraints: *)
```

Note that the universe level `i` here is local to the definition. Hence, `Tp` can be instantiated at different universe levels. As a result, the definition of `TpTp` above is no longer rejected and is represented internally as:

```
Definition TpTp@{i j} : Tp@{j} := Tp@{i}. (* Constraints: i < j *)
```

That is, the two times `Tp` is mentioned, two different instances of it are considered at two different universe levels `i` and `j` resulting in the constraint `i` < `j` for the definition to be well-defined.

Note the resemblance between universes in Coq and Grothendieck universes. E.g., the fact that if `A : Type@{i}` and `B : Type@{i}` then `{x : Type@{i} | x = A ∨ x = B} : Type@{i}`, cumulativity, etc.

In the sequel, in some cases, we only show the internal representation of concepts formalized in Coq.

[1]`Type@{i}` is Coq's syntax for $\texttt{Type}_i$.

2.3 Smallness and Largeness

In this implementation, we use universe levels as the underlying notion of smallness/largeness. In other words, we simply ignore smallness and largeness of constructions and simply allow Coq infer necessary conditions for definitions to be well-defined. We define categories without mentioning universe levels. They are internally represented as:

```
Record Category@{i j} :=
  {
    Obj : Type@{i};
    Hom : Obj → Obj → Type@{j};
    ...
  } : Type@{max(i+1, j+1)} (* Constraints: *)
```

The category of (small) categories is internally represented as:

```
Definition Cat@{i j k l} :=
  {|
    Obj := Category@{k l};
    Hom := fun (𝒞 𝒟 : Category@{k l}) ⇒ Functor@{k l k l} 𝒞 𝒟;
    ...
  |} : Category@{i j} (* Constraints: k < i, l < i, k ≤ j, l ≤ j *)
```

That is, **Cat** has as objects categories that are small compared to itself.

Having a universe-polymorphic **Cat** means for any category $\mathcal{C}$ there is a version of **Cat** that has $\mathcal{C}$ as an object. Therefore, for example, to express the fact that two categories are isomorphic, we simply use the general definition of isomorphism in the specific category **Cat**. This means we can use all facts and lemmas proven for isomorphisms, for isomorphisms of categories with no further effort required.

The category of types (representation of **Set** in Coq) is internally represented as:

```
Definition Set@{i j} :=
  {|
    Obj := Type@{j};
    Hom := fun (A B : Type@{j}) ⇒ A → B;
    ...
  |} : Category@{i j} (* Constraints: j < i *)
```

The constraint `j` < `i` above is exactly what we expect as **Set** is locally small. The reason that Coq's universe inference algorithm produces this constraint is that the type of objects of **Set** is `Type@{j}` which itself has type `Type@{i}`. But, the homomorphisms of this category are functions between two types whose type is `Type@{j}`. Thus, the type of homomorphisms themselves is `Type@{j}`. For details of typing rules for function types see [21].

Complete Small Categories are Mere Preorder Relations! Perhaps the best showcase of using the new universe polymorphism of Coq to represent smallness/largeness can be seen in the theorem below which simply implies that any complete category is a preorder category, i.e., there is at most one morphism between any two objects.

```
Theorem Complete_Preorder (𝒞 : Category) (CC : Complete 𝒞) :
    forall x y : Obj 𝒞, Hom x y' ≃ ((Arrow 𝒞) → Hom x y)
```

where `y'` is the limit of the constant functor from the discrete category **Discr**(`Arrow` $\mathcal{C}$) that maps every object to `y`, `(Arrow` $\mathcal{C}$`)` is the type of all homomorphisms of category $\mathcal{C}$ and $\simeq$ denotes isomorphism. In other words, for any pair of objects `x` and `y` the set of functions from the set of all morphisms in $\mathcal{C}$ to the set of morphisms from `x` to `y` is isomorphic to the set of morphisms from `x` to some constant object `y'`. This though, would result in a contradiction as soon as we have two objects A and B in $\mathcal{C}$ for which the collection of morphisms from A to B has more than one element. Hence, we have effectively shown that *any* complete category is a preorder category.

This is indeed absurd as the category **Set** is complete and there are types in Coq that have more than one function between them! However, this theorem holds for small (in the conventional sense) categories. That is, any *small and complete* category is a preorder category[2].

As expected, the constraints on the universe levels of this theorem that are inferred by Coq do indeed confirm this fact. That is, this theorem is in fact only applicable to a category $\mathcal{C}$ for which the level of the type of objects is less than or equal to the level of the type of arrows. This is in direct conflict with the constraints inferred for **Set** as explained above. Hence, Coq will refuse to apply this theorem to the category **Set** with a universe inconsistency error.

2.4 Limitations Imposed by Using Universe Levels for Smallness and Largeness

The universe polymorphism of Coq, as explained in [31], treats inductive types by considering copies of them at different levels. Furthermore, if a term of a polymorphic inductive type is assumed to be of two instances of that inductive type at two different universe levels, those levels are enforced to be equal. As records are a special kind of inductive types, the same holds for them. For us, this implies that if we have $\mathcal{C}$: `Category@{i j}` and we additionally have that $\mathcal{C}$: `Category@{i' j'}`, Coq enforces `i` = `i'` and `j` = `j'`. This means, **Cat**`@{i j k l}` is in fact *not* the category of *all* smaller categories. Rather it is the category of smaller categories that are at level `k` and `l` and not any lower level.

Apart from the fact that **Cat** defined this way is not the category of all relatively small categories, these constraints on universe levels impose practical restrictions as well. For instance, looking at the fact that **Cat**`@{i j k l}` has exponentials (functor categories), we can see the constraints that `j` = `k` = `l`. Consequently, only those copies have exponentials for which this constraints holds. Looking back at **Set**, we had the constraint that the level of the type of morphisms is strictly less than that of objects. This means, there is no version of **Cat** that both has exponentials and a version of **Set** in its objects.

Moreover, we can use the Yoneda lemma to show that in any cartesian closed category, for any objects a, b and c:

$$\left(a^b\right)^c \simeq a^{b\times c} \tag{1}$$

Yet, this theorem can't be applied to **Cat**, even though it holds for **Cat**.

It is worth noting that although **Cat**`@{i j k l}` is the category of all categories `Category@{k l}` and not lower, for any lower category it contains an "isomorphic copy" of that category. That is any category $\mathcal{C}$: `Category@{k' l'}` such that `k'` $\leq$ `k` and `l'` $\leq$ `l` can be "lifted" to `Category@{k l}`. Such a lifting function can be simply written as:

```
Definition Lift (𝒞 : Category@{k' l'}) : Category@{k l} :=
  {|
    Obj := Obj 𝒞;
    Hom := Hom 𝒞;
    ...
  |}.
```

[2]This theorem and its proof are taken from [3].

and the appropriate constraints, i.e., $\texttt{k}' \leq \texttt{k}$ and $\texttt{l}' \leq \texttt{l}$ are inferred by Coq. However, working with such liftings is in practice cumbersome as in interesting cases where $\texttt{k}' < \texttt{k}$ and/or $\texttt{l}' < \texttt{l}$, we can't prove or even specify `Lift` $\mathcal{C} = \mathcal{C}$ as it is ill-typed. This means, any statement regarding $\mathcal{C}$ must be proven separately for `Lift` $\mathcal{C}$ in order for them to be useful for the lifted version.

It is possible to alleviate these problems if we have support for cumulative inductive types in Coq, as proposed in [37]. In such a system, any category $\mathcal{C}$: `Category@{i j}` will also have type `Category@{k l}` so long as the constraints $\texttt{i} \leq \texttt{k}$ and $\texttt{j} \leq \texttt{l}$ are satisfied.

However, these limitations are not much more than a small inconvenience and in practice we can work in their presence with very little extra effort. At least as far as basic category theory goes. Our development is an attestation to that.

2.5 Smallness and Largeness in Other Developments

In homotopy type theory (HoTT) [33] a category $\mathcal{C}$ has a further constraint that for any two objects A and B the set of morphisms from A to B must form an hSet (a homotopy type-theoretical concept). On the other hand, for two categories $\mathcal{C}$ and $\mathcal{D}$, the set of functors from $\mathcal{C}$ to $\mathcal{D}$ does not necessarily form an hSet. It does however when the set of objects of $\mathcal{D}$ forms an hSet. Therefore, in HoTT settings one can construct the category of small *strict* categories, i.e., small categories whose type of objects forms an hSet, and not the category of all small categories. However, the category of small strict categories itself is not strict. Hence, contrary to the category **Cat** in our development, there is no category (in the HoTT sense, i.e., one whose objects form an hSet) that has the category of small strict categories as one of its objects. In this regard, working in HoTT is similar to working in NBG rather than ZF with Grothendieck universes.

The situation regarding the category of small strict categories discussed above is due to the fact that homotopy type-theoretical levels for types (e.g., hSet) concern a notion of (homotopy theoretical) *complexity* rather than cardinality. In fact, in other situations, e.g., in defining limits of functors, where cardinality is concerned universe levels can be used to express smallness and largeness. In other words, in HoTT settings, when defining limits, one can simply not mention universe levels and let Coq infer that the definition of limit for a functor $\mathcal{F} : \mathcal{C} \to \mathcal{D}$ is well-defined whenever, $\mathcal{C}$ is relatively small compared to $\mathcal{D}$. This also means that the restrictions mentioned above are also present in HoTT settings when universe levels are used to represent smallness and largeness. For instance isomorphism 1 above can't be proven in **Cat** using the Yoneda lemma even if a, b and c are strict categories.

This is how smallness and largeness works in both Gross et al. [13] and Ahrens et al. [2]. This is also the case for our development when ported on top of the HoTT library [32]. As one consequence, contrary to what was explained above, in migrating to the HoTT library settings we can't simply consider the isomorphism of categories as the general notion of isomorphism in the specific case of **Cat**.

In Huet et al. [18], working in Coq 8.4, the authors define a duplicate definition of categories, `Category`', tailored to represent large categories. This way, they form the `Category`' of categories (`Category`) – much like we used `Tp'` above.

Peebles et al. [27] however use universe levels to represent smallness and largeness. But working in Agda which provides no typical ambiguity or cumulativity, they have to hand code all universe levels everywhere; whereas we rely on Coq's inference of constraints to do the hard work. Noteworthy is also the fact that their categories have three universe variables instead of our two. One for the level of the type of objects, one for the level of the type of morphisms and one for the level of the type of the setoid equality for their setoids of morphisms.

3 Concepts Formalized, Features and Comparison

In this development we have formalize most of the basic category theory. Here, by basic we mean not involving higher (e.g., 2-categories), monoidal or enriched categories. This spans over the simple yet important and useful basic concepts like terminal/initial objects, products/sums, equalizers/coequalizers, pullbacks/pushouts and exponentials on the one hand and adjunctions, Kan extensions, (co)limits (as (left)right local Kan extensions) and toposes on the other.

The well-behaved dualities (in the sense discussed above) allow us to simply define dual notions, just as duals of their counterparts, e.g., initial objects as terminal objects of the dual category or the local left Kan extension of $\mathcal{F}$ along $\mathcal{G}$ as the local right Kan extension of $\mathcal{F}^{op}$ along $\mathcal{G}^{op}$.

Throughout this development we have tried to formalize concepts in as general a way as possible so long as they are comfortably usable. For instance, we define (co)limits as (left)right local Kan extensions along the unique functor to the terminal category. By doing so, we can extend facts about them to (co)limits. As an example, consider (left)right adjoints preserving (co)limits and (co)limit functors being adjoint to Δ explained below.

Different versions of adjunction and Kan extensions Throughout this development we have tried to formalize concepts in as general a way as possible so long as they are comfortably usable. For instance, we define (co)limits as (left)right local Kan extensions along the unique functor to the terminal category. By doing so, we can extend facts about them to (co)limits. As an example, consider (left)right adjoints preserving (co)limits and (co)limit functors being adjoint to Δ explained below.

Different versions of adjunction and Kan extensions In this formalization, we have multiple versions of the definition of adjunctions and Kan extensions. In particular, we define unit-universal morphism property adjunction, unit-co-unit adjunction, universal morphism adjunction and hom-functor adjunction. For these different versions, we provide conversions to and from the unit-universal morphism property definition which is taken to be the main definition. This definition is also taken to be the main definition of adjunction in Awodey's book [3]. For local Kan extensions, we define them as (initial)terminal (co)cones along a functor as well as through the hom-functor. Global Kan extensions are simply defined through adjunctions.

The main reason for this diversity, aside from providing a versatile category theory library, is the fact that each of these definitions is most suitable for some specific purpose.

For instance, using the hom-functor definition of adjunctions makes it very easy to prove that isomorphic functors have the same adjoints: $\mathcal{F} \simeq \mathcal{F}' \Rightarrow \mathcal{F} \dashv \mathcal{G} \Rightarrow \mathcal{F}' \dashv \mathcal{G}$, duality of adjunction: $\mathcal{F} \dashv \mathcal{G} \Rightarrow \mathcal{G}^{op} \dashv \mathcal{F}^{op}$, and uniqueness of adjoint functors: $\mathcal{F} \dashv \mathcal{G} \Rightarrow \mathcal{F}' \dashv \mathcal{G} \Rightarrow \mathcal{F} \simeq \mathcal{F}'$. The last case simply follows from the Yoneda lemma. On the other hand, the unit-universal morphism property definition of adjunctions together with the definition of Kan extensions as cones along a functor provide an easy way to convert from local to global Kan extensions.

Universal morphism adjoints in practice express sufficient conditions for a functor to have a (left)right adjoint. That is, a functor $\mathcal{G} : \mathcal{C} \to \mathcal{D}$ is a right adjoint (has a left adjoint functor) if the comma category $(x \downarrow \mathcal{G})$ has a terminal object for any $x : \mathcal{D}$. As we will briefly discuss below, (left)right adjoint functors preserve (co)limits. Freyd's adjoint functor theorem gives an answer to the question "when is a functor that preserves all limits a right adjoint (has a left adjoint functor)". Universal morphism adjoints appear in this theorem and that's why we have included them in our formalization.

(Left)right adjoints preserve (co)limits Awodey [3] devotes a whole section to this fact with the title "RAPL" (Right Adjoints Preserve Limits). For a better understanding of this fact and perhaps the concept of adjunctions, let us draw intuition from categorical interpretations of logic. In categorical interpretations of logic, the existential and universal quantifiers are interpreted as left and right adjoints to some functor while conjunctions and disjunctions are defined as products and sums respectively which respectively are in turn limits and co-limits (see Jacobs' book [19] for details). In this particular case, RAPL and its dual boil down to: $\forall x.\ P(x) \wedge Q(x) \Leftrightarrow \forall x.\ P(x) \wedge \forall x.\ Q(x)$ and $\exists x.\ P(x) \vee Q(x) \Leftrightarrow \exists x.\ P(x) \vee \exists x.\ Q(x)$. We prove this fact in general for (left)right local Kan extensions. To this end, the unit-co-unit definition of adjunctions is the easiest to use to prove the main lemma which along with hom-functor definition of Kan extensions proves that (left)right adjunctions preserve (left)right Kan extensions. That is for an adjunction $\mathcal{L} \dashv \mathcal{R}$ where $\mathcal{R} : \mathcal{D} \to \mathcal{E}$ and $\mathcal{L} : \mathcal{E} \to \mathcal{D}$ if in the diagram on the left $\mathcal{H}$ is the local right Kan extension of $\mathcal{F}$ along $\mathcal{P}$ then in the right diagram $\mathcal{R} \circ \mathcal{H}$ is the local right Kan extension of $\mathcal{R} \circ \mathcal{F}$ along $\mathcal{P}$:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\mathcal{F}"] \arrow[d, "\mathcal{P}"'] & \mathcal{D} \\
\mathcal{C}' \arrow[ur, "\mathcal{H}"'] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\mathcal{C} \arrow[r, "\mathcal{R}\circ\mathcal{F}"] \arrow[d, "\mathcal{P}"'] & \mathcal{E} \\
\mathcal{C}' \arrow[ur, "\mathcal{R}\circ\mathcal{H}"'] &
\end{tikzcd}$$

The case of (co)limits follows immediately. In Coq we show this by constructing a local right Kan extension (using the hom-functor definition) of $\mathcal{R} \circ \mathcal{F}$ along $\mathcal{P}$ where the Kan extension functor (HLRKE) is $\mathcal{R}$ composed with the Kan extension functor of $\mathcal{F}$ along $\mathcal{P}$:

```
Definition Right_Adjoint_Preserves_Hom_Local_Right_KanExt
   {C C' : Category} (P : Functor C C') {D : Category} (F : Functor C D)
   (hlrke : Hom_Local_Right_KanExt P F)
   {E : Category} {L : Functor E D} {R : Functor D E} (adj : UCU_Adjunct L R)
: Hom_Local_Right_KanExt P (R ∘ F) :=
    {|
       HLRKE := (R ∘ (HLRKE hlrke));
       HLRKE_Iso := ...
    |}.
```

(Co)limit functors are adjoint to Δ In order to show that (co)limits are adjoint to the diagonal functor (Δ) we simply use the fact that local (left)right Kan extensions assemble together to form (left)right global Kan extensions. As global Kan extensions are defined as (left)right adjoints to the pre-composition functor, putting these two facts together, we effortlessly obtain that (co)limits form functors which are (left)right adjoint to Δ.

Cardinality restrictions We introduce the notion of cardinality restriction in the category **Set**. A cardinality restriction is a property over types (objects of **Set**) such that if it holds for some type, it must hold for any other type isomorphic (in **Set**) to it. That is, if a cardinality restriction holds for a type, it must hold for any other type with the same cardinality.

```
Record Card_Restriction : Type :=
  { Card_Rest : Type → Prop;
    Card_Rest_Respect : forall (A B : Type),
      (A ≃≃ B ::> Set) → Card_Rest A → Card_Rest B }.
```

The type (`A ≃≃ B ::>` **Set**) is the type of isomorphisms $A \simeq B$ in **Set**. As an example, the cardinality restriction corresponding to finiteness is defined as follows.

```
Definition Finite : Card_Restriction :=
  {| Card_Rest := fun A ⇒ inhabited {n : nat & (A ≃≃ {x : nat | x < n} ::> Set)}; . . . |}.
```

The definition above basically says that a type A is finite if there exists some n such that A is isomorphic to the type $\{x : \texttt{nat} \mid x < n\}$ of natural numbers less than n.

(Co)limits restricted by cardinality We use the notion of cardinality restrictions above to define (co)limits restricted by cardinality. For a cardinality restriction P, we say a category $\mathcal{C}$ has (co)limits of cardinality P ($\mathcal{C}$ is P-(co)complete) if for all functors $\mathcal{F} : \mathcal{D} \to \mathcal{C}$ such that $P(Obj_{\mathcal{D}})$ and $\forall AB \in Obj_{\mathcal{D}}, P(Hom(A, B))$, $\mathcal{C}$ has the (co)limit of $\mathcal{F}$.

```
Definition Has_Restr_Limits (𝒞 : Category) (P : Card_Restriction) :=
  forall {𝒥 : Category} (ℱ : Functor 𝒥 𝒞), P 𝒥 → P (Arrow 𝒥) → Limit ℱ.
```

We state several lemmas about cardinality restricted (co)completeness, e.g., if a category has all limits of a specific cardinality its dual has all co-limits of that cardinality.

```
Definition Has_Restr_Limits_to_Has_Restr_CoLimits_Op
        {𝒞 : Category} {P : Card_Restriction}
        (HRL : Has_Restr_Limits 𝒞 P) : Has_Restr_CoLimits (𝒞^op) P := . . .
```

This also allows us to define a topos, simply as a category that is cartesian closed, has all finite limits and a subobject classifier where finiteness is represented as a cardinality restriction.

```
Class Topos : Type :=
  { Topos_Cat : Category;
    Topos_Cat_CCC : CCC Topos_Cat;
    Topos_Cat_Fin_Limit : Has_Restr_Limits Topos_Cat Finite;
    Topos_Cat_SOC : SubObject_Classifier Topos_Cat }.
```

(Co)Limits by (Sums)Products and (Co)Equalizers A discrete category is a category where the only morphisms are identities. That is, any set can induce a discrete category by simply considering the category which has as objects members of that set and the only morphisms are identity morphisms. We define the discrete category of a type A as a category, $\mathbf{Discr}(A)$ with terms of type A as objects and the collection of morphisms from an object x to an object y are proofs of equality of $x = y$.

```
Definition Discr_Cat (A : Type) : Category := {|Obj := A; Hom := fun a b ⇒ a = b; . . . |}.
```

Similarly, a discrete functor is a functor that is induced from a mapping f from a type A to objects of a category $\mathcal{C}$:

```
Definition Discr_Func {𝒞 : Category} {A : Type} (f : A → 𝒞) : Functor (Discr_Cat A) 𝒞 :=
    {| FO := f; . . . |}.
```

We define the notion of generalized (sums)products to be that of (co)limits of functors from a discrete category.

```
Definition GenProd {A : Type} {𝒞 : Category} (f : A → 𝒞) := Limit (Discr_Func f).
```

We use these generalized (sums)products to show that any category that has all generalized (sums)products and (co)equalizers has all (co)limits. We also prove the special case of cardinality restricted (co)limits. Using the notions explained above, we show that given a cardinality restriction P if a category has (co)equalizers as well as all generalized (sums)products that satisfy P, then that category is P-(co)complete.

```
Definition Restr_GenProd_Eq_Restr_Limits
   {C : Category} (P : Card_Restriction)
   {CHRP : forall (A : Type) (f : A → C), (P A) → (GenProd f)}
   {HE : Has_Equalizers C}
: Has_Restr_Limits C P := ...
```

Categories of Presheaves To the best of our knowledge, ours is the only category theory development featuring facts about categories of presheaves such as their (co)completeness, and being a topos. The category of presheaves on $\mathcal{C}$, $(\mathbf{PSh}(\mathcal{C}))$, is a category whose objects are functors of the form $\mathcal{C}^{op} \to \mathbf{Set}$ and whose morphisms are natural transformations. In other words, a presheaf $P : \mathcal{C}^{op} \to \mathbf{Set}$ on $\mathcal{C}$ is a collection of sets indexed by objects of $\mathcal{C}$ such that for a morphism $f : A \to B$ in $\mathcal{C}$, there is a function (a conversion if you will) $P(f) : P(B) \to P(A)$ in **Set**. Presheaves being toposes, each come with their own logic. As an example, [7] shows that the logic of the category of presheaves on ω (the preorder of natural numbers considered as a category) corresponds to the step-indexing technique used in the field of programming languages and program verification.

For more details about elementary properties of categories of presheaves see [3]. There categories of presheaves are called categories of diagrams.

Comparison Figures 1 and 2 give an overall comparison of our development with select other implementations of category theory of comparable extent. These figures mention only the most notable features and concepts formalized and do not contain many notions and lemmas in these developments. In these figures, our development is the fist column. As evident from these figures, in the areas concerning the basics of category theory, i.e., not higher or enriched categories, our development has almost any notable construction the other developments do and more.

In what follows, we briefly discuss features of automation and consequences of the use of setoids of morphisms or lack thereof. Features relevant to HoTT, i.e., use of axiom of unicity of identity proofs (UIP) and Rezk completion shall be discussed in Section 4 and 5. Comparison of the way these developments represent relative smallness and largeness appears in Section 2.

3.1 Automation

We use custom Ltac tactics for automation of some of the proofs. The use of custom Ltacs in [13] is much heavier. They use automation in almost any place possible. Our use is mostly in proving simple diagrams commute. There are for instance general tactics for simplifying compositions with identity morphisms, resolving associativity, or simplifying composition of a morphism with its inverse, in case of isomorphisms.

Another use of automation in our development is in combination with the Russell system. Russell is a mechanism for writing programs and proofs in Coq by allowing the user to write a definition and leave some parts as holes. Russell type checks the definition creating proof obligations for these holes that should be proven separately for that definition to be complete. It additionally tries to solve these obligations using Coq's automation system. Furthermore, allowing the user to specify an obligation tactic which is used to solve obligations provides greater control over the way obligations are handled. For further details on the Russell system see [30].

A tactic for smart rewriting of associativity We used to have a general tactic that given two morphisms f and g tried to find $f \circ g$ in the goal or a hypothesis. This tactic applied for

Concept / Feature	[34]	[13]	[18]	[2]	[27]
Automation	partial	✓			
Based on HoTT	in [35]$^\sharp$	✓		✓	
Setoid for Morphisms			✓		✓
Assumes UIP or equivalent	few restricted cases				✓
Basic constructions:					
Terminal/Initial object	✓	✓	✓	✓	✓
Products/Sums	✓		✓	✓	✓
Equalizers/Coequalizers	✓		✓		
Pullbacks/Pushouts	✓		✓	✓	✓
Basic constructions	✓		✓		
above are (co)limits					
exponentials	✓		✓		✓
Subobject classifier	✓			✓	✓
External constructions:					
Comma categories	✓	✓	✓	✓	✓
Product category	✓	✓	✓	✓	✓
Sum category		✓			
Cat. of categories (**Cat**):	✓	✓	✓		✓
Cartesian closure	✓		✓		
Initial/terminal object	✓	✓	✓		✓
Category of sets (**Set**):	✓	✓	✓	✓	✓
Basic (co)limits	✓	init./term.	partial		
(Local†)Cartesian closure	✓		CCC		
(Co†)Completeness	✓		comp.		✓
Sub-object classifier	(`Prop` : `Type`)†				
Topos	✓†				
Hom functor	✓	✓	✓	✓	✓
Fully-faithful functors	✓	✓		✓	✓
Essentially (inj)sur-jective functors	✓	✓		✓	✓
The Yoneda lemma	✓	✓	✓	✓	✓
Monoidal Categories		partial			✓
Enriched Categories		partial			partial
2-categories					✓
Pseudo-functors		✓			✓
(Co)monads and algebras :					
(Co)Monad				✓	✓
T-(co)algebras	✓			✓	✓
(T: an endofunctor)					
Eilenberg Moore cat.					✓
Kleisli cat.					✓

†Uses the axioms of propositional extensionality and constructive indefinite description (axiom of choice).
$^\sharp$This is the version of our development we are migrating to HoTT settings, on top of HoTT library.

Figure 1: Comparison of features and concepts formalized with a few other implementations of comparable extent.

Concept / Feature	[34]	[13]	[18]	[2]	[27]
Adjunction	✓	✓	✓		✓
Unit-universal morphism adjunction	✓	✓			
Hom-functor adjunction	✓	✓	✓		
Unit-counit adjunction	✓	✓	✓	✓	✓
Universal morphism adjunction	✓	✓	✓		
Uniqueness up to natural isomorphism	✓				
Naturally isomorphic functors have the same left/right adjoints	✓				
Adjoint composition laws	✓	✓			✓
Category of adjunctions (objects: categories; morphisms: adjunctions)	✓				
Partial adjunctions		✓			
Adjoint Functor Theorem	✓				✓
Kan extensions	✓	✓			✓
Global definition	✓	✓		✓	
Local definition	✓	✓			
Through hom-functor	✓				
Through cones (along a functor)	✓				✓
Through partial adjoints		✓			
Uniqueness	✓				
Preservation by adjoint functors	✓				
Naturally isomorphic functors form the same left/right Kan extension	✓				
Pointwise kan extensions (preserved by representable functors)	✓	✓			
(Co)Limits	✓	✓	✓	✓	✓
As (left)right kan extensions	✓	✓			
As (initial)terminal (co)cones			✓	✓	✓
(Sum)Product-(co)equalizer (co)limits	✓				
(Co)Limit functor	✓	✓			
(Co)Limits functor adjoint to Δ	✓	✓			
(Co)limits restricted by cardinality	✓				
Pointwise (as kan extensions), i.e., preserved by Hom functor	✓		✓		
Category of presheaves over $\mathcal{C}$ ($\mathbf{PSh}_{\mathcal{C}}$):	✓				✓
Terminal/Initial object	✓				
Products/Sums	✓				
Equalizers/Coequalizers	✓†				
Pullbacks	✓				
Cartesian closure	✓				
Completeness/Co-completeness	✓†				
Sub-object classifier (Sieves)	✓†				
Topos	✓†				

†Uses the axioms of propositional extensionality and constructive indefinite description (axiom of choice).

Figure 2: Comparison of features and concepts formalized with a few other implementations of comparable extent (cont.).

example to a goal containing the term

$$(h_2 \circ (h_1 \circ f)) \circ ((g \circ h_3) \circ h_4)$$

would replace this term by

$$(h_2 \circ h_1) \circ ((f \circ g) \circ (h_3 \circ h_4))$$

This tactic was useful both in proofs by hand and for proof automation purposes. It was specially useful for detecting cases where f is the left inverse of g and hence $f \circ g$ can be replaced with the identity morphism, e.g., in the case isomorphisms. Unfortunately, this tactic was very inefficient. The source of this inefficiency was the use of `context` in pattern matching to find sub-terms of a goal (or hypothesis) which contained f and g which was exponential. That is, in case both f and g were present in a term where they failed to form $f \circ g$ by applying associativity law, the `context` would reattempt to form the composition on smaller sub-terms containing both f and g. As we could not fix this intractability we chose to stop using it.

Perhaps some additional options to `context` pattern matching can make such tactics more efficient. For instance, if `context` pattern matching could also return the precise term where matchee was found on. In this case we could simply try finding the other morphism in this part of the term and not in the whole term. Additionally, there should be a way to instruct the pattern matching mechanism to simply fail in case the body fails instead of trying smaller matching sub-terms.

3.2 Setoids and Co-limits/Subobject Classifier in Set and PSh($\mathcal{C}$)

Setoids (aka Bishop sets) [9] consist of a set (a type in type theoretical foundations like Coq and Agda) together with an equivalence relation usually referred to as the setoid equality. In their implementation [27] and [18] have used setoids to represent morphisms of categories. Using them comes with the cost of heavier development as e.g., every function must be shown to respect the setoid equality. On the other hand, they offer the benefit of diminishing the need for axioms such as functional extensionality and propositional extensionality.

A particular case is the case of co-limits and subobject classifier of **Set** and categories of presheaves. Co-limits, drawing intuition from set theory, are quotients of sum types with respect to an equivalence relation. Having the equivalence relation built-in, the category of setoids and functions respecting their equalities, as is taken to play the role of **Set** in [27] and [18], leaves no need for axioms to define co-limits.

We on the other hand, take the category of types in Coq and functions among them as **Set** and have resorted to axioms of propositional extensionality and constructive indefinite description (a form of axiom of choice) to define co-limits. That is, we define the (quotient representing) co-limit as an existential proposition (in `Prop`). This way, whenever two elements are related we can show that their representation in the quotient type is equal using the axiom of propositional extensionality. On the other hand, when need be, we use the axiom of constructive indefinite description to take out an element of the quotient. Similar treatment is required to show that `Prop` is the subobject classifier of **Set**.

Showing that `Prop` is the subobject classifier in the category of types of Coq shows that the type $A \to$ `Prop` which is usually taken as the subset type of type A is indeed categorically speaking the type representing its subsets.

Categories of presheaves involve sets (as objects are functors to **Set**). Hence, an argument similar to the foregoing shows that, like **Set**, we need axioms of propositional extensionality and constructive indefinite description for constructing their co-limits and subobject classifier.

We will elaborate our use of axioms in Section 5 where we discuss our migration to the HoTT library. In particular we will discuss the case of UIP and how we and [27] use it.

3.3 Category Theory: Coq vs. Pen and Paper

There is no doubt that the level of rigor of working in a proof assistant such as Coq is much higher than that in the usual mathematical practices with pen and paper.

Category theory is no exception. Yet, in our experience in case of category theory the use of Coq has indeed helped make things easier. It is our experience that using the right balance between Coq and sketches on paper is the best practice. That is, we use the general understanding of the topic on paper to divide a problem into easy to handle definitions and lemmas. Then, the use of Coq helps keep track of crucial details throughout the proof.

For instance, when a definition and/or theorem involves simultaneously a category $\mathcal{C}$ and its dual $\mathcal{C}^{op}$, if not careful enough in working with pen and paper, it is very easy to get lost and confused as to which morphism was from $\mathcal{C}$ and which from $\mathcal{C}^{op}$. In Coq on the other hand, use of simplification tactics like `cbn` and `simpl` in the context helps level the field by simplifying most dualities allowing the user to carry the proof out considering only the objects and morphism of $\mathcal{C}$.

On the other hand, although dualities behave nicely (in the aforementioned sense), working with dual definitions is not always as smooth as we would have wished. This is especially evident in rewriting equalities. In some cases one has to add the equality to the proof context (usually applied to the arguments that are difficult to match) and perform a simplification on them before they can be used with the `rewrite` tactic. In some rare extreme cases, simplifications with tactics like `cbn` and `simpl` were not enough and we had to change the goal in such a way that those lemmas can be used with, e.g., the `apply` tactic, instead of `rewrite`.

4 HoTT, Axioms and Categories

In Martin-Löf's intensional type theory (see [26]), the equality type, which we denote by $x =_A y$ for terms $x : A$, $y : A$ and $A : \texttt{Type}$, has a single formation rule:

$$\frac{A : \texttt{Type} \qquad x : A}{id(x) : x =_A x}\ (\textsf{id-formation})$$

We use $x = y$ instead of $x =_A y$ when it is clear from the context. Similar to Martin-Löf's intensional type theory in systems like Coq which feature inductive types, the equality type is usually defined as an inductive family of types with a single constructor. In particular, in Coq it is defined as follows:

```
Inductive {A : Type} (x : A) eq : A → Prop :=
eq_refl : eq x x.
```

That is, it is defined as a family of types indexed by a type `A : Type`, a term `x : A` and a term `y : A` and represents the proposition $x =_A y$. For details of inductive types and in particular the equality type in Coq, see [21].

4.1 UIP and Other Equivalent Axioms

The fact that there is only one formation rule/constructor begs the seemingly trivial question *is $id(x)$ the only proof of the identity $x = x$?* This was in fact an open problem for a while until it was settled with a negative answer by [17]. They construct a model of Martin-Löf's intensional type theory where there are multiple non-equal proofs for some identities. In the following, we write UIP(A) to say UIP holds for type A, that is, for any two elements of type A there is at most one proof of equality $x = y$.

Although the axiom of "unicity of identity proofs" (UIP for short) is not provable in Coq, it is consistent to assume it and it is provided as part of the standard library. In Agda however, due to a different form of pattern matching in that language, UIP is provable unless the pattern matching mechanism is restricted to disallow it (see [10] for details).

In [27], the authors use Agda and make explicit use of heterogeneous equality which as explained below is equivalent to UIP. We will discuss our use of UIP in more details below.

There are other axioms that are equivalent to UIP, in particular, the axiom that heterogeneous equality implies equality. Heterogeneous equality is defined similarly to the inductive type `eq` above with the difference that it expresses equality of terms of different types. In Coq it is defined as follows:

```
Inductive JMeq (A : Type) (x : A) :
  forall (B : Type), B → Prop := JMeq_refl : JMeq A x A x.
```

The axiom equivalent with UIP about heterogeneous equality implying equality is called `JMeq_eq`:

```
Axiom JMeq_eq : forall (A : Type) (x y : A), JMeq A x A y → x = y.
```

The definition of `JMeq` and the axiom above are part of the standard library of Coq. The standard library also features proofs that this axiom along with some other axioms, e.g., axiom K, are equivalent to UIP.

One particularly important point about the `JMeq_eq` axiom is that to prove it, one needs to assume UIP for equalities of the form `A = A`, i.e., UIP(`Type`). This fact, as discussed below, is of special significance when we consider UIP in a restricted form, i.e., only for some types.

4.2 HoTT and the Univalence Axiom

Inspired by [17], homotopy type theory (HoTT) [33] is an interpretation of Martin-Löf's intensional type theory using homotopy theoretic notions from algebraic geometry.

At the center of HoTT sits the univalence axiom by Voevodsky (see [4, 33]). The univalence axiom states that for types equivalence[3] is equivalent to identity:

$$\forall A, B : \texttt{Type}.\ (A = B) \simeq (A \simeq B)$$

The univalence axiom has interesting consequences. For one, it allows us to consider isomorphic structures to be equal, a common practice in mathematics on paper. On the other hand, it implies interesting axioms like functional extensionality and, as discussed below, propositional extensionality (see [33]). However, whether the univalence axiom is constructive or not is still an open problem.

The univalence axiom is in direct contradiction with UIP, however. This can be easily seen as there are types that are equivalent to themselves in more than one way. Take the type `Bool` for example. The two functions

$$\begin{aligned} f_1(x) &= x \\ f_2(x) &= \begin{cases} \texttt{false} & \text{if} \quad x = \texttt{true} \\ \texttt{true} & \text{if} \quad x = \texttt{false} \end{cases} \end{aligned}$$

are inverses to themselves both establishing equivalences which are different.

Although it is inconsistent to assume UIP in general when working in HoTT, there are types for which UIP holds. For example, any type with decidable equality, like the type of natural numbers, satisfies UIP (see [33]). In HoTT, such a type is called an hSet or a 0-Type.

[3]Equivalence can be thought of as isomorphism. That is, $A \simeq B$ if there are functions $f : A \to B$ and $g : B \to A$ such that $\forall x, f(g(x)) = x$ and $\forall x, g(f(x)) = x$. In HoTT settings the exact definition is slightly different (see [33]).

4.3 n-Types, hSets and hProps

In HoTT (see [33]), in order to generalize the notion of a set and a proposition (in the spirit of propositions as types), a hierarchy of types is devised. At the bottom of the hierarchy is the notion of a (-2)-Type (also known as a contractible type). A type A is contractible if it is inhabited by a term $center : A$ and for any element of that type $x : A$, we have $center = x$. In other words, a type is contractible if it is provably a singleton. A type A is an $(n + 1)$-Type if for any two elements $x, y : A$, the type $x = y$ is an n-Type.

In particular, for a (-1)-type (also known as an hProp) P, for any two elements $x, y : P$, the type $x = y$ is contractible and hence inhabited. That is, if P is inhabited, it is inhabited by at most one term. In other words, it satisfies the property of proof irrelevance. Hence, the name hProp. Note also that the univalence axiom for hProp gives us propositional extensionality, i.e., any two equivalent propositions are equal.

On the other hand, for a 0-Type (also known as an hSet) A, for any two elements $x, y : A$ the type $x = y$ is an hProp. That is, there is at most one proof for equality of two terms of A. In other words, we have UIP(A).

The number n in n-Type is also referred to as the homotopy level or truncation level of that type. [12] gives an excellent concise explanation of the concept of homotopy levels and the foregoing arguments about hSet and hProp.

There are many interesting and useful facts about n-Types stated and proven in [33]. Most notably, a (dependent) function type $(\forall x : A.\ B(x))$ has the truncation level n if for all $(x : A)$, the type $B(x)$ has truncation level n. Also, the product $A \times B$ and the sum $A + B$ of types have truncation level n if both A and B do. Furthermore, the truncation hierarchy is cumulative, i.e., any n-Type is also an $(n + 1)$-Type.

4.4 Categories in HoTT: Pre-Categories, Strict Categories and Categories

As discussed in [33], we must assume UIP for the morphism types of categories, as otherwise, our definition of a category will not be a simple category but rather a form of higher category. That is, we must assume UIP($A \to B$) where A and B are objects and $A \to B$ is the type of morphisms from A to B.

That is precisely why, in all formalizations of category theory it is assumed or enforced in one way or another. Those developments using setoids define custom equalities so as to circumvent addition of axioms. On the other hand, developments based on HoTT explicitly require it, by adding to the definition of a category the explicit requirement that types of morphisms should form an hSet. In our development we have assumed this and use axioms, e.g., proof irrelevance to enforce it. In moving to HoTT, we follow suit and add this additional requirement and use UIP holding for hSets instead of the axioms used. We discuss this in more detail below.

In [33] and following it in [13] and [2] the authors use the name pre-category to describe what we call a category. They call a pre-category a strict category if the type of objects of that pre-category forms an hSet. A category, in their terminology, is a pre-category for which we have that any two isomorphic objects are equal. That is, a category is a pre-category that satisfies a categorical version of the univalence axiom. For instance, the category whose objects are types that are hSets and morphisms are functions among them is a category, as opposed to a pre-category, due to the univalence axiom. Note that, as discussed, functions have the same truncation level as their codomain type and hence in this case form hSets. This category, in our development (in the HoTT version) as well as other HoTT-based developments, is precisely the category **Set**.

In [2] show that any pre-category $\mathcal{C}$ can be completed to a category $\hat{\mathcal{C}}$ what they call Rezk completion.

5 Migration to the HoTT library

In this section we discuss our ongoing effort in porting a version of our development to the HoTT setting. This is based on the HoTT library development [32].

In what follows we discuss our experience so far. We moreover outline the challenges we have faced or anticipate to face during the course of this migration and our (in some cases anticipatory) solutions to these problems.

5.1 Permissible Use of UIP

As explained in Subsection 4.4, in our development in Coq 8.5, we have enforced that the type of morphisms form an hSet by using axioms. Since, in the HoTT version, it is by definition the case that types of morphisms form hSets, use of these axioms is "permissible" and in fact provide an easy migration to the HoTT library.

In particular, we package the use of these axioms into lemmas which are used to facilitate proofs and proof automation. For instance, we have a lemma "`NatTrans_eq_simplify`" that says two natural transformations are equal if their underlying family of morphisms are. In proving this lemma, the fact that the proof of naturality diagrams of the two natural transformations are equal is simply handled using proof irrelevance.

The similar lemma for functors, "`Functor_eq_simplify`", is a bit more complicated however. It states that two functors $\mathcal{F}$ and $\mathcal{G}$ are equal if we have a proof $(H : \mathcal{F}_o = \mathcal{G}_o)$ stating their object maps are equal and we have:

$$subst(\mathcal{F}_o \rightsquigarrow \mathcal{G}_o, H, \mathcal{F}_a) = \mathcal{G}_a$$

Here, $\mathcal{F}_a$ is the morphism map of $\mathcal{F}$ and $subst(a \rightsquigarrow b, P, x)$ is the term x where in its type a is substituted with b given the proof $(P : a = b)$[4]. The fact that the proofs of functors preserving identity morphisms and composition of morphisms are equal is handled using proof irrelevance.

The benefits of these packagings in migration to HoTT are twofold. First, as the use of axioms is centralized and not scattered all over the development, in moving to HoTT we only need to change these lemmas. This we simply do by replacing the tactic that applies proof irrelevance automatically with another tactic which builds instances of UIP required from the underlying categories and applies them as appropriate.

One added challenge in working in HoTT settings comes precisely from requiring morphisms to forms an hSet. In particular, whenever we want to construct a category, we have to show that the type of morphisms of that category indeed form an hSet. This is in particular the case for comma categories, categories of functors (natural transformations as morphism), etc.

The second benefit of these packagings, is exactly in facilitating proofs that these constructs, i.e., natural transformations, functors, etc. form hSets. We do this by providing in each case, a section (i.e., right inverse) for the equality simplifier lemma making it into a retraction. Sections are known to preserve levels of truncations. That is, for types A and B where B is an n-Type, we can conclude that A is also an n-Type as soon as we provide a function $f : A \to B$ and a function $g : B \to A$ and prove that $\forall x : A.\ g(f(x)) = x$. See [33] for a proof of this fact. As an example, consider the case of natural transformations. By providing a section `NatTrans_eq_simplify_inv` we show that the level of equalities of natural transformations is the same is the level of equality

[4]In Coq, *subst* is implemented through dependent pattern matching on the provided equality proof.

of morphisms (which is hProp by definition) and hence that natural transformations form an hSet.

In our on-going migration to HoTT, we have so far successfully applied this technique to natural transformations, comma morphisms, adjunctions (in creating categories of adjunctions) and functors. The case of functors is slightly different though. As the discussion of "`Functor_eq_simplify`" above indicates, equality of functors can't be simply reduced to the equality of their morphism mappings. They also involve equality of object mappings of functors; the type of which may or may not form an hSet. However, we can construct such a required right inverse whenever the type of objects of the codomain category of the functors in question forms an hSet. This is precisely why we in our HoTT development, and also [13] in theirs, construct the category of strict categories.

5.2 Impermissible Use of UIP

In some cases, we have used UIP in a way that is not permissible in HoTT. In these cases we have used the axiom `JMeq_eq` on equalities of morphisms. Recall that using `JMeq_eq` $A\ x\ A\ y$ to prove $x = y$ requires unicity of proof of $(A = A)$ and not UIP(A) which is provided by the assumption that morphisms form hSets.

These cases all happen in proofs of equality of functors or equalities depending on them. In particular, there are three cases: proof of **Cat** having exponentials, proof of associativity of adjunct composition and some proofs of isomorphisms of categories.

Cat having exponentials The case of **Cat** having exponentials can easily be proven without using UIP in general. This is due to the fact that in **Cat**, categories are strict and hence the object maps of functors can only have trivial equalities. This allows for easy simplification of *subst* after application of `Functor_eq_simplify` lemma which before required unrestricted UIP to apply it to the type of object maps of functors.

Associativity of adjunct composition Composition of adjuncts refers to the fact that whenever we have two adjunctions $(\mathrm{adj} : \mathcal{F} \dashv \mathcal{G})$ and $(\mathrm{adj}' : \mathcal{F}' \dashv \mathcal{G}')$, we can compose them to obtain

$$\mathrm{adj} \circ \mathrm{adj}' : \mathcal{F} \circ \mathcal{F}' \dashv \mathcal{G}' \circ \mathcal{G}$$

In order to prove associativity, we need to prove that

$$(\mathrm{adj} \circ \mathrm{adj}') \circ \mathrm{adj}'' : (\mathcal{F} \circ \mathcal{F}') \circ \mathcal{F}'' \dashv \mathcal{G}'' \circ (\mathcal{G}' \circ \mathcal{G})$$

and

$$\mathrm{adj} \circ (\mathrm{adj}' \circ \mathrm{adj}'') : \mathcal{F} \circ (\mathcal{F}' \circ \mathcal{F}'') \dashv (\mathcal{G}'' \circ \mathcal{G}') \circ \mathcal{G}$$

are equal which clearly have different types. That is, we have to prove that after correction of the types by using *subst* on one side with the proof of associativity of function composition, they are equal.

In this case, we don't have the assumption that the codomain category of the functors are strict categories and so we can't assume that the functors form hSet. As a result, *subst* in this case can't be easily simplified. To circumvent this problem however, we reduce the the problem to a case where *subst* performs substitution based on the proof of equality of object maps underlying the proof of associativity of the functors. Since associativity of the functors is simply proven by `Functor_eq_simplify eq_refl eq_refl`, this reduces to *subst* with `eq_refl` which can immediately be simplified.

Isomorphism of categories In a number of cases in our development, we have proven isomorphism of categories where we have used `JMeq_eq`. These cases all lead up to two particular use cases. To define the dual of universal morphism adjoints and to show that a category with co-equalizers and generalized sums is co-complete from its dual argument. In our migration to HoTT, we have not yet arrived at a stage where we need to solve these problems. The following arguments are thus speculative.

In case of isomorphism that we need in order to define the universal morphism adjoints dually, in [13], they have slightly changed the definition of a category to have those categories be definitionally equal. We have proved a more general case. That is, we have proven that for any naturally isomorphic functors $\mathcal{F} \simeq \mathcal{F}'$, $(\mathcal{F} \downarrow \mathcal{G}) \simeq (\mathcal{F}' \downarrow \mathcal{G})$. We believe that proving this fact for the particular case that we need it, i.e., where $\mathcal{F}$ and $\mathcal{F}'$ are special functors from the terminal category, should be straightforward.

In the case of co-limits produced by generalized sums and co-equalizers however, it is our best educated guess that the lemmas necessary can't be proven in general. They can perhaps be proven only for cases where categories in question are strict.

One way that we plan to approach this problem is to prove similar results using equivalence or perhaps adjoint equivalence of categories instead of isomorphism. Two categories are equivalent if there are functors between them that compose to functors that are *naturally isomorphic* (rather than equal) to the identity functor of respective categories. An adjoint equivalence is an equivalence where the underlying functors in addition to forming an equivalence of categories are also adjoints.

In any case, as these lemmas are simply needed for some dual arguments, in the worst case, these can be abandoned for a direct proof.

5.3 Other Axioms

In our development, we have made frequent use of the axiom of functional extensionality. However, this axiom is a consequence of the univalence axiom and is in fact provided in the HoTT library and frequently used therein.

We have in particular taken advantage of two other axioms, propositional extensionality and constructive indefinite description which we have used to construct co-limits in **Set** and presheaf categories.

We plan to use higher inductive types, as explained in [33], to construct such co-limits. Higher inductive types are extensions of inductive types. In a higher inductive type definition, other than the constructors of the type being defined, one can add path constructors. Path constructors can be used to add custom equalities to the type which can in turn change the truncation level of the type being constructed.

5.4 Prop vs. hProp

In our implementation, we show that any preorder relation can be extended to a category. Following the usual practice, we defined a preorder relation as a relation in Coq `R : A → A → Prop` that is reflexive and transitive. In the category obtained this way, the type of morphisms is a type in `Prop`. This has some unwanted consequences. In particular, we can't *straightforwardly* define a functor from a preorder category defined this way to any category that is not a preorder category. This is due to Coq's restriction on elimination of terms whose type is in `Prop`. More precisely, case analysis on a term of a type that is in `Prop` is only allowed if the resulting type of the case analysis is a type which itself is in `Prop`. We faced this problem particularly when we wanted to construct functors out of the preorder category ω induced by $(\mathbb{N}, \leq)$.

However, we found out that for a particular class of types in `Prop`, we can indeed perform this elimination. These are types for which we can construct a type outside of `Prop` which is isomorphic to that type. In particular, this trick works for the relation $\leq$ of natural numbers.

This class of types corresponds to decidable hProp types. Let us consider the example of $\leq$. To eliminate $\leq$, we define the isomorphic type $\leq^\tau$ which is defined as an inductive type akin to $\leq$ with the difference that it is defined in the universe `Type` instead of `Prop`. Our ultimate goal is to establish that the two types $n \leq m$ and $n \leq^\tau m$ are isomorphic. Defining the side of isomorphism from $n \leq^\tau m$ to $n \leq m$ is straightforward. It can be done with a simple case analysis on $n \leq^\tau m$ which is allowed. To define the other side of isomorphism directly one requires to perform case analysis on $n \leq m$ to construct $n \leq^\tau m$ which is forbidden. However, we can prove $\neg(n \leq^\tau m) \to \neg(n \leq m)$ by case analysis on $n \leq m$. Notice that the result of this case analysis is supposed to construct a term of type `False` which is in `Prop`.

Now to construct a conversion from $n \leq m$ to $n \leq^\tau m$, we use decidability of $n \leq^\tau m$. Given a proof of $n \leq m$, we can simply perform the decision procedure to see whether $n \leq^\tau m$ holds or not. In case the decision procedure tells us that $n \leq^\tau m$ holds, we take the proof provided by the decision procedure as the result of conversion. In case the decision procedure tells us that $(n \leq^\tau m)$, this contradicts our assumption $n \leq m$ and hence we are done. In order to prove that the constructed conversions form an isomorphism, we simply need to show that $n \leq^\tau m$ is hProp. Which we can do. In order to eliminate a term of type $n \leq m$ then, we can simply eliminate the corresponding term in $n \leq^\tau m$.

One particular drawback of this conversion is its computational behavior. The conversion procedure simply ignores the input and performs the decision procedure. The input is solely used to invalidate the contradictory branch after the decision procedure.

Even though, this trick worked in our case, $(\mathbb{N}, \leq)$, we decided to change the definition of a preorder category (even in our development in Coq 8.5) to instead require the type of relations to be an hProp. This was in part due to the fact that these isomorphisms must be proven separately for different cases and in part due to the fact that his trick only applies to decidable relations. Anticipation of migration to HoTT was another motivation for this choice.

5.5 Tactics in the HoTT library

One rather technical observation of migrating to the HoTT library is the tactics provided in there. There are a number of discrepancies between the tactics in the HoTT library and Coq8.5. In particular, the tactics of `inversion` and `cutrewrite` simply don't work giving the error that some part of the standard library which is not available in HoTT is not found. In one particular case we had the `destruct` tactic failing to generalize the goal the way required. In this case, reverting some of the hypotheses which could not be included in the generalization by `destruct` solved the problem.

6 Future Work: Building on Categories

Our priority, at the moment, is of course to finish the migration to the HoTT library.

We believe that this development is one that provides a foundation for other works based on category-theoretical foundations. We have plans to make use of the foundation of category theory that has been laid in this work. In particular, use of this foundation for mechanization of categorical logic (see [19]) and higher order separation logic (see [6]) for the purpose of using them as foundations for mechanization of program verification. In particular, the theory of presheaves developed provides a basis for formalization of the internal logic of presheaf categories with a particular interest in the topos of trees [7].

In this regard, we have already used this development as a foundation to formalize the theory of [8]. In [8], the authors use the theory of ultra-metric spaces to build unique (up to isomorphism) fixed-points of particular category-theoretical recursive domain-theoretic equations. More precisely, they construct fixed-points of a particular class of mixed variance functors, i.e., functors of the form $\mathcal{F} : (\mathcal{C}^{op} \times \mathcal{C}) \to \mathcal{C}$. Solutions to such mixed-variance functors can for example be used to construct models for imperative programming languages.

In [8], the authors define the notion of an M-category to be a category in which the set of morphisms between any two objects form a non-empty ultra-metric space. In [36], based on a general theory of ultra-metric spaces, we define M-categories as categories in which the type of morphisms between any two objects forms an ultra-metric space, dropping the rather strong non-emptiness requirement. We instead require some weaker conditions which still allow us to form fixed-points. However, we prove the uniqueness of the fixed-points in the "inhabited" subcategory of that M-category. That is, the full subcategory where every object has a morphism from the terminal object. This treatment is similar to the other recent mechanization of this theory [29], except that they don't prove uniqueness. Successful implementation of [36] on top of our general foundation of categories, although arguably not huge, is evidence that this development is fit for being used as a general-purpose foundation.

The main difference between our work, [36], and [29] is the fact that in ours M-categories are simply categories (as defined in our foundational development) with extra requirements while they define custom M-categories and provide lemmas necessary for their theory. In contrast, our work is backed by a wealth of general formalization of category theory. This is of course besides the technical differences in representation of categories.

An interesting instance of M-categories is the presheaf topos of the preorder category of natural numbers, i.e., the topos of trees. In our development, just showing that this category qualifies as an M-category is sufficient to immediately be able to construct desired fixed-points. This is due to the fact that in the foundations provided, all necessary conditions for an M-category to allow formation of solutions, e.g., existence of limits of a particular class of functors is already established.

7 Conclusion

In summary, we presented our development of the foundations of category theory. This development features most of the category-theoretical concepts that are formalized in most other such developments and some more.

We pushed the limits of the new feature of universe polymorphism and the constraint inference algorithm of Coq 8.5 by using them to represent relative smallness/largeness. As discussed, it gives encouraging results despite the restrictions imposed by not having cumulative inductive types.

In the latter part of the paper we discussed our experience of migrating this development to HoTT settings. We hope that our report on the challenges that we have faced and the solutions that we have devised prove useful for others undertaking such migrations.

We have successfully used this implementation as the categorical foundation to build categorical metric-space theoretic fixed-points of recursive domain equations. This seems an encouraging initial indication that this work is fit to perform the important role of a general purpose category theoretical foundation for other developments to build upon.

Acknowledgements We would like to thank Matthieu Sozeau and Georges Gonthier for helpful discussions.

This work was funded by EU FP7 FET-Open project ADVENT under grant number 308830.

References

[1] Samson Abramsky and Nikos Tzevelekos. Introduction to categories and categorical logic. *CoRR*, abs/1102.1313, 2011. URL: `http://arxiv.org/abs/1102.1313`.

[2] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the rezk completion. *Mathematical Structures in Computer Science*, 25(05):1010–1039, jan 2015. URL: `https://github.com/benediktahrens/rezk_completion`, `doi:10.1017/s0960129514000486`.

[3] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[4] Steve Awodey, lvaro Pelayo, and Michael A. Warren. Voevodsky's univalence axiom in homotopy type theory, 2013. `arXiv:arXiv:1302.4731`.

[5] Michael Barr and Charles Wells. Toposes, triples and theories, 2005.

[6] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007. URL: `http://doi.acm.org/10.1145/1275497.1275499`, `doi:10.1145/1275497.1275499`.

[7] Lars Birkedal, Rasmus Ejlers Mogelberg, Jan Schwinghammer, and Kristian Stovring. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. Institute of Electrical & Electronics Engineers (IEEE), jun 2011. URL: `http://dx.doi.org/10.1109/LICS.2011.16`, `doi:10.1109/lics.2011.16`.

[8] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 411(47):4102–4122, oct 2010. URL: `http://dx.doi.org/10.1016/j.tcs.2010.07.010`, `doi:10.1016/j.tcs.2010.07.010`.

[9] E. Bishop. *Foundations of constructive analysis*. McGraw-Hill series in higher mathematics. McGraw-Hill, 1967. URL: `https://books.google.com/books?id=Z7I-AAAAIAAJ`.

[10] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern Matching Without K. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 257–268, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2628136.2628139`, `doi:10.1145/2628136.2628139`.

[11] T. Coquand. An analysis of Girard's paradox. Technical Report RR-0531, INRIA, May 1986. URL: `https://hal.inria.fr/inria-00076023`.

[12] Thierry Coquand and Nils Anders Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105–1120, nov 2013. URL: `http://dx.doi.org/10.1016/j.indag.2013.09.002`, `doi:10.1016/j.indag.2013.09.002`.

[13] Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq. In *Interactive Theorem Proving - 5th International Conference, ITP 2014. Proceedings*, pages 275–291, July 2014. `doi:10.1007/978-3-319-08970-6_18`.

[14] Jason Gross, Adam Chlipala, and David I. Spivak. Experience implementing a performant category-theory library in coq, 2014. `arXiv:arXiv:1401.7694`.

[15] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, oct 1991. URL: `http://dx.doi.org/10.1016/0304-3975(90)90108-T`, `doi:10.1016/0304-3975(90)90108-t`.

[16] Christopher Henderson. Elementary topoi: Sets, generalize. 2009.

[17] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[18] Gérard P. Huet and Amokrane Saïbi. Constructive category theory. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 239–276, 2000. URL: `http://www.lix.polytechnique.fr/coq/pylons/coq/pylons/contribs/view/ConCaT/v8.4`.

[19] Bart Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[20] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1978.

[21] The Coq development team. *Coq 8.5 Reference Manual.* Inria, 2015.

[22] Colin McLarty. *Elementary Categories, Elementary Toposes.* Oxford University Press, Oxford, UK, 1996.

[23] Adam Megacz. Category Theory in Coq. URL: `http://www.megacz.com/berkeley/coq-categories/`.

[24] John C. Mitchell. *Foundations of Programming Languages.* MIT Press, Cambridge, MA, USA, 1996.

[25] Ieke Moerdijk and Jap van Oosten. Topos theory. Lecture Notes, Department of Mathematics, Utrecht University, 2007.

[26] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-LöF's Type Theory: An Introduction.* Clarendon Press, New York, NY, USA, 1990.

[27] Daniel Peebles, James Deikun, Ulf Norell, Dan Doel, Andrea Vezzosi, Darius Jahandarie, and James Cook. copumpkin/categories. URL: `https://github.com/copumpkin/categories`.

[28] Antonino Salibra. Scott is always simple. In *Proceedings of the 37th International Conference on Mathematical Foundations of Computer Science*, MFCS'12, pages 31–45, Berlin, Heidelberg, 2012. Springer-Verlag. URL: `http://dx.doi.org/10.1007/978-3-642-32589-2_3`, `doi:10.1007/978-3-642-32589-2_3`.

[29] Filip Sieczkowski, Ale Bizjak, and Lars Birkedal. Modures: A coq library for modular reasoning about concurrent higher-order imperative programming languages. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, volume 9236 of *Lecture Notes in Computer Science*, pages 375–390. Springer International Publishing, 2015. URL: `http://dx.doi.org/10.1007/978-3-319-22102-1_25`, `doi:10.1007/978-3-319-22102-1_25`.

[30] Matthieu Sozeau. Subset Coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin Heidelberg, 2007. URL: `http://dx.doi.org/10.1007/978-3-540-74464-1_16`, `doi:10.1007/978-3-540-74464-1_16`.

[31] Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In *Interactive Theorem Proving, ITP 2014, Proceedings*, pages 499–514, July 2014.

[32] The Univalent Foundations Program. HoTT Version of Coq and Library. URL: `lhttps://github.com/HoTT/HoTT`.

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

[34] Amin Timany. Categories. URL: `https://github.com/amintimany/Categories/tree/FSCD16`, `doi:http://dx.doi.org/10.5281/zenodo.50689`.

[35] Amin Timany. Categories-HoTT. URL: `https://github.com/amintimany/Categories-HoTT`.

[36] Amin Timany. Category-theoretical Solution of Recursive Metric-Space Equations. URL: `https://github.com/amintimany/CTDT`.

[37] Amin Timany and Bart Jacobs. First Steps Towards Cumulative Inductive Types in CIC. In *12th International Colloquium on Theoretical Aspects of Computing (ICTAC 2015) 2015. Proceedings*, page to appear, Oct 2015.