Experience Implementing a Performant Category-Theory Library in Coq

Jason Gross, Adam Chlipala, and David I. Spivak

Massachusetts Institute of Technology, Cambridge MA 02139, USA,
jgross@mit.edu, adamc@csail.mit.edu, dspivak@math.mit.edu

Abstract. We describe our experience implementing a broad category-theory library in Coq. Category theory and computational performance are not usually mentioned in the same breath, but we have needed substantial engineering effort to teach Coq to cope with large categorical constructions without slowing proof script processing unacceptably. In this paper, we share the lessons we have learned about how to represent very abstract mathematical objects and arguments in Coq and how future proof assistants might be designed to better support such reasoning. One particular encoding trick to which we draw attention allows category-theoretic arguments involving *duality* to be internalized in Coq's logic with definitional equality. Ours may be the largest Coq development to date that uses the relatively new Coq version developed by homotopy type theorists, and we reflect on which new features were especially helpful.

Keywords: Coq, category theory, homotopy type theory, duality, performance

1 Introduction

Category theory [12] is a popular all-encompassing mathematical formalism that casts familiar mathematical ideas from many domains in terms of a few unifying concepts. A *category* can be described as directed graph plus algebraic laws stating equivalences between paths through the graph. Because of this spartan philosophical grounding, category theory is sometimes referred to in good humor as "formal abstract nonsense." Certainly the popular perception of category theory is quite far from pragmatic issues of implementation. This paper is an experience report on an implementation of category theory that has run squarely into issues of design and efficient implementation of type theories, proof assistants, and developments within them.

It would be reasonable to ask, what would it even mean to implement "formal abstract nonsense," and what could the answer have to do with optimized execution engines for functional programming languages? We mean to cover the whole scope of category theory, which includes many concepts that are not manifestly computational, so it does not suffice merely to employ the well-known folklore semantic connection between categories and typed functional programming [17].

Instead, a more appropriate setting is a computer proof assistant. We chose to build a library for Coq [23], a popular system based on constructive type theory.

One might presume that it is a routine exercise to transliterate categorical concepts from the whiteboard to Coq. Most category theorists would probably be surprised to learn that standard constructions "run too slowly," but in our experience that is exactly the result of experimenting with naïve first Coq implementations of categorical constructs. It is important to tune the library design to minimize the cost of manipulating terms and proving interesting theorems.

This design experience is also useful for what it reveals about the consequences of design decisions for type theories themselves. Though type theories are generally simpler than widely used general-purpose programming languages, there is still surprising subtlety behind the few choices that must be made. Homotopy type theory [24] is a popular subject of study today, where there is intense interest in designing a type theory that makes proofs about topology particularly natural, via altered treatment of equality. In this setting and others, there remain many open questions about the consequences of type theoretical features for different sorts of formalization. Category theory, said to be "notoriously hard to formalize," [6] provides a good stress test of any proof assistant, highlighting problems in usability and efficiency.

Formalizing the connection between universal morphisms and adjunctions provides a typical example of our experience with performance. A *universal morphism* is a construct in category theory generalizing extrema from calculus. An *adjunction* is a weakened notion of equivalence. In the process of rewriting our library to be compatible with homotopy type theory, we discovered that cleaning up this construction conceptually resulted in a significant slow-down, because our first attempted rewrite resulted in a leaky abstraction barrier and large goals. Plugging the holes there reduced goal sizes by two orders of magnitude[1], which led to a factor of ten speedup in that file (from 39s to 3s), but incurred a factor of three slow-down in the file where we defined the abstraction barriers (from 7s to 21s). Working around slow projections of Σ types (Subsection 2.5) and being more careful about code reuse each gave us back half of that lost time.

For reasons that we present in the course of the paper, we were unsatisfied with the feature set of released Coq version 8.4. We wound up adopting the Coq version under development by homotopy type theorists [21], making critical use of its stronger universe polymorphism (see Subsection 2.2) and higher inductive types (Section 2.4). We hope that our account here provides useful data points for proof assistant designers about which features can have serious impact on proving convenience or performance in very abstract developments. The two features we mentioned earlier in the paragraph can simplify the Coq user experience dramatically, while a number of other features, at various stages of conception or implementation by Coq team members, can make proving much easier or improve proof script performance by orders of magnitude: primitive record projections (Subsection 2.5), internalized proof irrelevance for equalities (Subsection 4.2), and η rules for records (Section 3.1) and equality proofs (Subsection 3.2).

[1] The word count of the larger of the two relevant goals went from 163,811 to 1,390.

Although pre-existing formalizations of category theory in proof assistants abound [1, 9, 13, 15, 16, 18, 20, 22], we chose to implement our library from scratch. Beginning from scratch allowed the first author to familiarize himself with both category theory and Coq, without simultaneously having to familiarize himself with a large pre-existing code base. Additionally, starting from scratch forced us to confront all of the decisions that come up in designing such a library, and gave us the confidence to change the definitions of basic concepts multiple times to try out various designs, including fully rewriting the library at least three times. Although this paper is much more about the design of category theory libraries in general than our library in particular, we include a comparison of our library [7] with selected extant category theory libraries in Section 5. At present, our library subsumes many of the constructions in most other such Coq libraries, and is not lacking any constructions in other libraries that are of a complexity requiring significant type checking time, other than monoidal categories.

We begin our discussion in Section 2 considering a mundane aspect of type definitions that has large consequences for usability and performance. With the expressive power of Coq's logic Gallina, we often face a choice of making *parameters* of a type family explicit arguments to it, which looks like universal quantification; or of including them within values of the type, which looks like existential quantification. As a general principle, we found that the universal or *outside* style improves the user experience modulo performance, while the existential or *inside* style speeds up type checking. The rule that we settled on was: *inside* definitions for pieces that are usually treated as black boxes by further constructions, and *outside* definitions for pieces whose internal structure is more important later on.

Section 3 presents one of our favorite design patterns for categorical constructions: a way of coaxing Coq's definitional equality into implementing *proof by duality*, one of the most widely known ideas in category theory. In Section 4, we describe a few other design choices that had large impacts on usability and performance, often of a few orders of magnitude. Section 5 wraps up with a grid comparison of our library with others.

2 Issues in Defining the Type of Categories

We have chosen to use the outside style when we care more about the definition of a construct than about carrying it around as an opaque blob to fit into other concepts. The first example of this choice comes up in deciding how to define categories.

2.1 Dependently Typed Morphisms

In standard mathematical practice, a category $\mathcal{C}$ can be defined [2] to consist of:

- a class $\mathrm{Ob}_{\mathcal{C}}$ of *objects*
- for all objects $a, b \in \mathrm{Ob}_{\mathcal{C}}$, a class $\mathrm{Hom}_{\mathcal{C}}(a, b)$ of *morphisms from a to b*

- for each object $x \in \mathrm{Ob}_{\mathcal{C}}$, an *identity morphism* $1_x \in \mathrm{Hom}_{\mathcal{C}}(x, x)$
- for each triple of objects $a, b, c \in \mathrm{Ob}_{\mathcal{C}}$, a *composition function* $\circ : \mathrm{Hom}_{\mathcal{C}}(b, c) \times \mathrm{Hom}_{\mathcal{C}}(a, b) \to \mathrm{Hom}_{\mathcal{C}}(a, c)$

satisfying the following axioms:

- associativity: for composable morphisms f, g, h, we have $f \circ (g \circ h) = (f \circ g) \circ h$.
- identity: for any morphism $f \in \mathrm{Hom}_{\mathcal{C}}(a, b)$, we have $1_b \circ f = f = f \circ 1_a$

Following [24], we additionally require our morphisms to be 0-truncated (to have unique identity proofs). Without this requirement, we have a standard pre–homotopy type theory definition of a category.

We might formalize the definition in Coq (if Coq had mixfix notation) as:

```
Record Category :=
  { Ob : Type;
    Hom : Ob → Ob → Type;
    _○_ : ∀ {a b c}, Hom b c → Hom a b → Hom a c;
    1 : ∀ {x}, Hom x x;
    Assoc : ∀ a b c d (f : Hom c d) (g : Hom b c) (h : Hom a b),
      f ○ (g ○ h) = (f ○ g) ○ h;
    LeftId : ∀ a b (f : Hom a b), 1 ○ f = f;
    RightId : ∀ a b (f : Hom a b), f ○ 1 = f;
    Truncated : ∀ a b (f g : Hom a b) (p q : f = g), p = q }.
```

We could just as well have replaced the classes $\mathrm{Hom}_{\mathcal{C}}(a, b)$ with a single class of morphisms $\mathrm{Hom}_{\mathcal{C}}$, together with functions defining the source and target of each morphism. Then it would be natural to define morphism composition to take a further argument, a proof of equal domain and codomain between the morphisms. Users of dependent types are aware that explicit manipulation of equality proofs can complicate code substantially, often to the point of obscuring what would be the heart of an argument on paper. For instance, the algebraic laws associated with categories must be stated with explicit computation of equality proofs, and further constructions only become more involved. Additionally, such proofs will quickly bloat the types of goals, resulting in slower type checking. For these reasons, we decided to stick with the definition of `Category` above, getting more lightweight help from the type checker in place of explicit proofs.

2.2 Complications from Categories of Categories

Some complications arise in applying the last subsection's definition of categories to the full range of common constructs in category theory. One particularly prominent example formalizes the structure of a collection of categories, showing that this collection itself may be considered as a category.

The morphisms in such a category are *functors*, maps between categories consisting of a function on objects, a function on hom-types, and proofs that these functions respect composition and identity [2, 12, 24].

The naïve concept of a "category of all categories," which includes even itself, leads into mathematical inconsistencies, which manifest as universe inconsistency errors in Coq. The standard resolution is to introduce a hierarchy of categories, where, for instance, most intuitive constructions are considered *small* categories, and then we also have *large* categories, one of which is the category of small categories. Both definitions wind up with literally the same text in Coq, giving:

```
Definition SmallCat : LargeCategory :=
  {| Ob := SmallCategory;
     Hom C D := SmallFunctor C D;
     ... |}.
```

It seems a shame to copy-and-paste this definition (and those of `Category`, `Functor`, etc.) n times to define an n-level hierarchy. Coq 8.4 and some earlier versions support a flavor of *universe polymorphism* that allows the universe of a definition to vary as a function of the universes of its arguments. Unfortunately, it is not natural to parameterize `Cat` by anything but a universe level, which does not have first-class status in Coq anyway. We found the connection between universe polymorphism and arguments to definitions to be rather inconvenient, and it forced us to modify the definition of `Category` so that the record field `Ob` changes into a parameter of the type family. Then we were able to use the following slightly awkward construction:

```
Definition Cat I ObOf (CatOf : ∀ i : I, Category (ObOf i)) :=
  {| Ob := I;
     Hom C D := Functor (CatOf C) (CatOf D);
     ... |}.
```

Now the definition is genuinely reusable for an infinite hierarchy of sorts of category, but we have paid the price of adding extra parameters to both `Category` and `Cat`, and this seemingly innocent change produces substantial blow-up in the sizes of proof goals arising during interesting constructions. So, in summary, we decided that the basic type theoretical design of Coq 8.4 did not provide very good support for pleasing definitions that can be reasoned about efficiently.

This realization (and a few more that will come up shortly) pushed us to become early adopters of the modified version of Coq developed by homotopy type theorists [21]. Here, an established kind of more general universe polymorphism [5], previously implemented only in NuPRL, is available, and the definitions we wanted from the start work as desired.

2.3 Arguments vs. Fields

Unlike most of our other choices, there is a range of possibilities in defining categories, with regards to arguments (on the outside) and fields (on the inside). At one extreme, everything can be made a field, with a type `Category` whose inhabitants are categories. At the other extreme, everything can be made an argument to a dummy function. Some authors [22] have chosen the intermediate option of making all of the computationally relevant parts (object and morphism types, composition, and the identity morphism) arguments and the irrelevant proofs (associativity and left and right identity) fields. We discussed in Subsection 2.2 the option of parameterizing on just the type of objects. We now consider pros and cons of other common options; we found no benefits to the "outside" extreme.

Relevant things on the outside One of the main benefits to making all of the relevant components arguments, and requiring all of the fields to satisfy proof irrelevance, is that it allows the use of type-class resolution without having to worry about overlapping instances. Practically, this choice means that it is easier to get Coq to infer automatically the proofs that given types and operations assemble into a category, at least when the proof is a straightforward composition of already proven lemmas. Although others [22] have found this approach useful, we have not found ourselves wishing we had type-class resolution when formalizing constructions, and there is a significant computational cost of exposing so many parameters in types.

Everything on the inside Once we moved to using the homotopy type theorists' Coq with its broader universe polymorphism, we decided to use fields for all of the components of a category. Switching from the version where the types of objects and morphisms were parameters brought a factor of three speed-up in compilation time over our whole development. The reason is that, at least in Coq, the performance of proof tree manipulations depends critically on their size. By contrast, the size of the normal form of the term does not seem to matter much in most constructions; see Section 3 for an explanation and the one exception that we found. By using fields rather than parameters for the types of objects and morphisms, the type of functors goes

$$\begin{aligned}\texttt{Functor} : \forall\ & (\texttt{ob}_\mathcal{C} : \texttt{Type})\ (\texttt{ob}_\mathcal{D} : \texttt{Type})\\ & (\texttt{hom}_\mathcal{C} : \texttt{ob}_\mathcal{C} \to \texttt{ob}_\mathcal{C} \to \texttt{Type})\ (\texttt{hom}_\mathcal{D} : \texttt{ob}_\mathcal{D} \to \texttt{ob}_\mathcal{D} \to \texttt{Type}),\\ & \texttt{Category}\ \texttt{ob}_\mathcal{C}\ \texttt{hom}_\mathcal{C} \to \texttt{Category}\ \texttt{ob}_\mathcal{D}\ \texttt{hom}_\mathcal{D} \to \texttt{Type}\end{aligned}$$

to

$$\texttt{Functor} : \texttt{Category} \to \texttt{Category} \to \texttt{Type}$$

The corresponding reduction for the type of natural transformations is even more remarkable, and with a construction that uses natural transformations multiple times, the term size blows up very quickly, even with only two parameters. If we had more parameters (for composition and identity), the term size would blow up even more quickly.

Usually, we do not care what objects and morphisms a particular category has; most of our constructions take as input arbitrary categories. Thus, there is a significant performance benefit to having all of the fields on the inside and thus hidden from most theorem statements.

2.4 Equality

Equality, which has recently become a very hot topic in type theory [24] and higher category theory [11], provides another example of a design decision where most usage is independent of the exact implementation details. Although the question of what it means for objects or morphisms to be equal does not come up much in classical 1-category theory, it is more important when formalizing category theory in a proof assistant, for reasons seemingly unrelated to its importance in higher category theory. We consider some possible notions of equality.

Setoids A setoid [4] is a carrier type equipped with an equivalence relation; a map of setoids is a function between the carrier types and a proof that the function respects the equivalence relation of its domain and codomain. Many authors [8, 10, 13, 16] choose to use a setoid of morphisms, which allows for the definition of the category of set(oid)s, as well as the category of (small) categories, without assuming functional extensionality, and allows for the definition of categories where the objects are quotient types. However, there is significant overhead associated with using setoids everywhere, which can lead to slower compile times. Every type that we talk about needs to come with a relation and a proof that this relation is an equivalence relation. Every function that we use needs to come with a proof that it sends equivalent elements to equivalent elements. Even worse, if we need an equivalence relation on the universe of "types with equivalence relations," we need to provide a transport function between equivalent types that respects the equivalence relations of those types.

Propositional Equality An alternative to setoids is propositional equality, which carries none of the overhead of setoids, but does not allow an easy formulation of quotient types, and requires assuming functional extensionality to construct the category of sets.

Intensional type theories like Coq's have a built-in notion of equality, often called definitional equality or judgmental equality, and denoted as $x \equiv y$. This notion of equality, which is generally internal to an intensional type theory and therefore cannot be explicitly reasoned about inside of that type theory, is the equality that holds between $\beta\delta\iota\zeta\eta$-convertible terms.

Coq's standard library defines what is called *propositional equality* on top of judgmental equality, denoted $x = y$. One is allowed to conclude that propositional equality holds between any judgmentally equal terms.

Using propositional equality rather than setoids is convenient because there is already significant machinery made for reasoning about propositional equalities, and there is much less overhead. However, we ran into significant trouble when attempting to prove that the category of sets has all colimits, which amounts to proving that it is closed under disjoint unions and quotienting; quotient types cannot be encoded without assuming a number of other axioms.

Higher Inductive Types The recent emergence of higher inductive types allows the best of both worlds. The idea of higher inductive types [24] is to allow inductive types to be equipped with extra proofs of equality between constructors. They originated as a way to allow homotopy type theorists to construct types with non-trivial higher paths. A very simple example is the interval type, from which functional extensionality can be proven [19]. The interval type consists of two inhabitants `zero : Interval` and `one : Interval`, and a proof `seg : zero = one`. In a type theory with higher inductive types, the type checker does the work of carrying around an equivalence relation on each type for us, and forbids users from constructing functions that do not respect the equivalence relation of any input type. For example, we can prove functional extensionality as follows:

```
Definition f_equal {A B x y} (f : A → B) : x = y → f x = f y.
Definition functional_extensionality {A B} (f g : A → B)
    : (∀ x, f x = g x) → f = g
  := λ (H : ∀ x, f x = g x)
       ⇒ f_equal (λ (i : Interval) (x : A)
                     ⇒ match i with
                         | zero ⇒ f x
                         | one  ⇒ g x
                         | seg  ⇒ H x
                       end)
                 seg.
```

Had we neglected to include the branch for `seg`, the type checker should complain about an incomplete match; the function `λ i : Interval ⇒ match i with zero ⇒ true | one ⇒ false end` of type `Interval → bool` should not typecheck for this reason.

The key insight is that most types do not need any special equivalence relation, and, moreover, if we are not explicitly dealing with a type with a special equivalence relation, then it is impossible (by parametricity) to fail to respect the equivalence relation. Said another way, the only way to construct a function that might fail to respect the equivalence relation would be by some eliminator like pattern matching, so all we have to do is guarantee that direct invocations of the eliminator result in functions that respect the equivalence relation.

As with the choice involved in defining categories, using propositional equality with higher inductive types rather than setoids derives many of its benefits from not having to deal with all of the overhead of custom equivalence relations in constructions that do not need them. In this case, we avoid the overhead by making the type checker deal with the parts we usually do not care about. Most of our definitions do not need custom equivalence relations, so the overhead of using setoids would be very large for very little gain. We plan to use higher inductive types to define quotients, which are necessary to show the existence of certain functors involving the category of sets. We also currently use higher inductive types to define propositional truncation [24], which we use to define what it means for a function to be surjective, and prove that in the category of sets, being an isomorphism (an invertible morphism) is equivalent to being injective and surjective.

2.5 Records vs. Nested Σ Types

In Coq, there are two ways to represent a data structure with one constructor and many fields: as a single inductive type with one constructor (records), or as a nested Σ type. For instance, consider a record type with two type fields A and B and a function f from A to B. A logically equivalent encoding would be $\Sigma A.\ \Sigma B.\ A \to B$. There are two important differences between these encodings in Coq.

The first is that while a theorem statement may abstract over all possible Σ types, it may not abstract over all record types, which somehow have a less first-class status. Such a limitation is inconvenient and leads to code duplication.

The far more pressing problem, overriding the previous point, is that nested Σ types have horrendous performance, and are sometimes a few orders of magnitude slower. The culprit is projections from nested Σ types, which, when unfolded (as they must be, to do computation), each take almost the entirety of the nested Σ type as type arguments, and so grow in size very quickly. Matthieu Sozeau is currently working on giving Coq primitive projections for records, which would eliminate this problem with nested Σ types (if Σ itself were defined as a record) by eliminating the arguments to the projection functions.

3 Internalizing Duality Arguments in Type Theory

In general, we have tried to design our library so that trivial proofs on paper remain trivial when formalized. One of Coq's main tools to make proofs trivial is the definitional equality, where some facts follow by computational reduction of terms. We came up with some small tweaks to core definitions that allow a common family of proofs by *duality* to follow by computation.

Proof by duality is a common idea in higher mathematics: sometimes, it is productive to flip the directions of all the arrows. For example, if some fact about least upper bounds is provable, chances are that the same kind of fact about greatest lower bounds will also be provable in roughly the same way, by replacing "greater than"s with "less than"s and vice versa.

Concretely, there is a dualizing operation on categories that inverts the directions of the morphisms:

```
Notation "C ᵒᵖ" := ({| Ob := Ob C; Hom x y := Hom C y x; ... |}).
```

Dualization can be used, roughly, for example, to turn a proof that Cartesian product is an associative operation into a proof that disjoint union is an associative operation; products are dual to disjoint unions.

One of the simplest examples of duality in category theory is initial and terminal objects. In a category $\mathcal{C}$, an initial object 0 is one that has a unique morphism $0 \to x$ to every object x in $\mathcal{C}$; a terminal object 1 is one that has a unique morphism $x \to 1$ from every object x in $\mathcal{C}$. Initial objects in $\mathcal{C}$ are terminal objects in $\mathcal{C}^{\text{op}}$. The initial object of any category is unique up to isomorphism; for any two initial objects 0 and $0'$, there is an isomorphism $0 \cong 0'$. By flipping all of the arrows around, we can prove, by duality, that the terminal object is unique up to isomorphism. More precisely, from a proof that an initial object of $\mathcal{C}^{\text{op}}$ is unique up to isomorphism, we get that any two terminal objects $1'$ and 1 in $\mathcal{C}$, which are initial in $\mathcal{C}^{\text{op}}$, are isomorphic in $\mathcal{C}^{\text{op}}$. Since an isomorphism $x \cong y$ in $\mathcal{C}^{\text{op}}$ is an isomorphism $y \cong x$ in $\mathcal{C}$, we get that 1 and $1'$ are isomorphic in $\mathcal{C}$.

It is generally straightforward to see that there is an isomorphism between a theorem and its dual, and the technique of dualization is well-known to category theorists, among others. We discovered that, by being careful about how we defined things, we could make theorems be judgmentally equal to their duals! That is, when we prove a theorem

```
initial_ob_unique : ∀ C (x y : Ob C),
                       is_initial_ob x → is_initial_ob y → x ≅ y,
```

we can define another theorem

```
terminal_ob_unique : ∀ C (x y : Ob C),
                       is_terminal_ob x → is_terminal_ob y → x ≅ y
```

as

```
terminal_ob_unique C x y H H' := initial_ob_unique Cᵒᵖ y x H' H.
```

Interestingly, we found that in proofs with sufficiently complicated types, it can take a few seconds or more for Coq to accept such a definition, which we believe is due to the fact that to unify the types, Coq needs to reduce them almost to their normal forms, resulting in a speed dependency on the size of the normal form of the type, rather than on the size of the type.

In contrast to the simplicity of witnessing the isomorphism, it takes a significant amount of care in defining concepts, often to get around deficiencies of Coq, to achieve *judgmental* duality. Even now, we were unable to achieve this ideal for some theorems. For example, category theorists typically identify the functor category $\mathcal{C}^{\text{op}} \to \mathcal{D}^{\text{op}}$ (whose objects are functors $\mathcal{C}^{\text{op}} \to \mathcal{D}^{\text{op}}$ and whose morphisms are natural transformations) with $(\mathcal{C} \to \mathcal{D})^{\text{op}}$ (whose objects are functors $\mathcal{C} \to \mathcal{D}$ and whose morphisms are flipped natural transformations). These categories are canonically isomorphic (by the dualizing natural transformations), and, with the univalence axiom [24], they are equal as categories! But we have not found a way to make them definitionally equal, much to our disappointment.

3.1 Duality Design Patterns

One of the simplest theorems about duality is that it is involutive; we have that $(\mathcal{C}^{\text{op}})^{\text{op}} = \mathcal{C}$. In order to internalize proof by duality via judgmental equality, we sometimes need this equality to be judgmental. Although it is impossible in general in Coq 8.4 (see dodging judgmental η on records below), we want at least to have it be true for any explicit category (that is, any category specified by giving its objects, morphisms, etc., rather than referred to via a local variable).

Removing symmetry Taking the dual of a category, one constructs a proof that $f \circ (g \circ h) = (f \circ g) \circ h$ from a proof that $(f \circ g) \circ h = f \circ (g \circ h)$. The standard approach is to apply symmetry. However, because applying symmetry twice results in a judgmentally different proof, we decided instead to extend the definition of `Category` to require both a proof of $f \circ (g \circ h) = (f \circ g) \circ h$ and a proof of $(f \circ g) \circ h = f \circ (g \circ h)$. Then our dualizing operation simply swaps the proofs. We added a convenience constructor for categories that asks only for one of the proofs, and applies symmetry to get the other one. Because we formalized 0-truncated category theory, where the type of morphisms is required to have unique identity proofs, asking for this other proof does not result in any coherence issues.

Dualizing the terminal category To make everything work out nicely, we needed the terminal category, which is the category with one object and only the identity morphism, to be the dual of itself. We originally had the terminal category as a special case of the discrete category on n objects. Given a type T with uniqueness of identity proofs, the discrete category on T has as objects inhabitants of T, and has as morphisms from x to y proofs that $x = y$. These categories are not judgmentally equal to their duals, because the type $x = y$ is not judgmentally the same as the type $y = x$. As a result, we instead used the indiscrete category, which has `unit` as its type of morphisms.

Which side does the identity go on? The last tricky obstacle we encountered was that when defining a functor out of the terminal category, it is necessary to pick whether to use the right identity law or the left identity law to prove that the functor preserves composition; both will prove that the identity composed with itself is the identity. The problem is that dualizing the functor leads to a road block where either concrete choice turns out to be "wrong," because the dual of the functor out of the terminal category will not be judgmentally equal to another instance of itself. To fix this problem, we further extended the definition of category to require a proof that the identity composed with itself is the identity.

Dodging judgmental η on records The last problem we ran into was the fact that sometimes, we really, really wanted judgmental η on records. The η rule for records says any application of the record constructor to all the projections of an object yields exactly that object; e.g. for pairs, $x \equiv (x_1, x_2)$ (where x_1 and x_2 are the first and second projections, respectively). For categories, the η rule says that given a category $\mathcal{C}$, for a "new" category defined by saying that its objects are the objects of $\mathcal{C}$, its morphisms are the morphisms of $\mathcal{C}$, …, the "new" category is judgmentally equal to $\mathcal{C}$.

In particular, we wanted to show that any functor out of the terminal category is the opposite of some other functor; namely, any $F : 1 \to \mathcal{C}$ should be equal to $(F^{\mathrm{op}})^{\mathrm{op}} : 1 \to (\mathcal{C}^{\mathrm{op}})^{\mathrm{op}}$. However, without the judgmental η rule for records, a local variable $\mathcal{C}$ cannot be judgmentally equal to $(\mathcal{C}^{\mathrm{op}})^{\mathrm{op}}$, which reduces to an application of the constructor for a category. To get around the problem, we made two variants of dual functors: given $F : \mathcal{C} \to \mathcal{D}$, we have $F^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$, and given $F : C^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$, we have $F^{\mathrm{op}'} : \mathcal{C} \to \mathcal{D}$. There are two other flavors of dual functors, corresponding to the other two pairings of $^{\mathrm{op}}$ with domain and codomain, but we have been glad to avoid defining them so far. As it was, we ended up having four variants of dual natural transformation, and are very glad that we did not need sixteen. We look forward to Coq 8.5, when we will hopefully only need one.

3.2 Moving Forward: Computation Rules for Pattern Matching

While we were able to work around most of the issues that we had in internalizing proof by duality, things would have been far, far nicer if we had more η rules. The η rule for records is explained above. The η rule for equality says that the identity function is judgmentally equal to the function $f : \forall x\, y, x = y \to x = y$ defined by pattern matching on the first proof of equality. Matthieu Sozeau is currently working on giving Coq judgmental η for records with one or more fields, though not for equality.

Subsection 4.1 will give more examples of the pain of manipulating pattern matching on equality. Homotopy type theory provides a framework that systematizes reasoning about proofs of equality, turning a seemingly impossible task into a manageable one. However, there is still a significant burden associated with reasoning about equalities, because so few of the rules are judgmental.

We are currently attempting to divine the appropriate computation rules for pattern matching constructs, in the hopes of making reasoning with proofs of equality more pleasant.[2]

4 Other Design Choices

A few other pervasive strategies made non-trivial differences for proof performance or simplicity.

4.1 Identities vs. Equalities; Associators

There are a number of constructions that are provably equal, but which we found more convenient to construct transformations between instead, despite the increased verbosity of definitions. For example, when constructing the Grothendieck

[2] See `https://coq.inria.fr/bugs/show_bug.cgi?id=3179` and `https://coq.inria.fr/bugs/show_bug.cgi?id=3119`.

construction (also called the category of elements) of a functor to the category of categories, we found it easier to first generalize the construction from functors to pseudofunctors. The definition of a pseudofunctor results from replacing various equalities in the definition of a functor with isomorphisms (analogous to bijections between sets or types), together with proofs that the isomorphisms obey various coherence properties. This replacement helped because there are fewer operations on isomorphisms (namely, just composition and inverting), and more operations on proofs of equality (pattern matching, or anything definable via induction); when we were forced to perform all of the operations in the same way, syntactically, it was easier to pick out the operations and reason about them.

Another example was defining the unit and counit of a composition of adjunctions, where instead of a proof of equality between functors $F \circ (G \circ H)$ and $(F \circ G) \circ H$, we used a natural transformation, a sort of coherent mapping between the actions of functors. Where equality-based constructions led to computational reduction getting stuck at casts that appeal to equality proofs, the constructions with natural transformations reduce in all of the expected contexts.

4.2 Opacity

Coq is slow at dealing with large terms. For goals around 150,000 words long, we have found that simple tactics like `apply f_equal` take around 1–2 seconds to execute, which makes interactive theorem proving very frustrating. Even more frustrating is the fact that the largest contribution to this size is arguments to irrelevant functions, i.e., functions that are provably equal to all other functions of the same type. (These are proofs related to algebraic laws like associativity, carried inside many constructions.)

Making the functions opaque helps a little; it prevents the type checker from unfolding their definitions. But the type checker still has to deal with all of the arguments to the opaque function, and it is the size of these arguments that slows down term manipulation.

It would be nice if, whenever we had a proof that all of the inhabitants of a type were equal, we could forget about terms of that type, so that their size would not impose any penalties on term manipulation. One could imagine a version of Coq's logic that knows to treat all proofs of an equality as equivalent to each other. Alternatively, there might be some way to ignore these terms when doing most computation, without changing the underlying theory. Perhaps irrelevant fields, similar to those of Agda, possibly implemented via the Implicit Calculus of Inductive Constructions [3, 14], could solve this problem.

4.3 Abstraction Barriers

In many projects, picking the right abstraction barriers is essential to reducing mistakes, improving maintainability and readability of code, and cutting down on time wasted by programmers trying to hold too many things in their heads at once. This project was no exception; we developed an allergic reaction to constructions with more than four or so arguments, after making one too many mistakes in defining limits and colimits. Limits are a generalization, to arbitrary categories, of subsets of Cartesian products. Colimits are a generalization, to arbitrary categories, of disjoint unions modulo equivalence relations.

Our original flattened definition of limits involved a single definition with 14 nested binders for types and algebraic properties. After a particularly frustrating experience hunting down a mistake in one of these components, we decided to factor the definition into a larger number of simpler definitions, including familiar categorical constructs like terminal objects and comma categories. This refactoring paid off even further when some months later we discovered the universal morphism definition of adjoint functors. With a little more abstraction, we were able to reuse the same decomposition to prove the equivalence between universal morphisms and adjoint functors, with minimal effort.

Perhaps less typical of programming experience, we found that picking the right abstraction barriers could drastically cut down on compile time by keeping details out of sight in large goal formulas. In the instance discussed in the introduction, we got a factor of ten speed-up by plugging holes in a leaky abstraction barrier![3]

5 Comparison of Category Theory Libraries

We present here a table comparing the features of various category theory libraries. Our library is the first column. Gray dashed check-marks (✓) indicate features in progress. The right-most library is in Agda; the rest are in Coq. A check-mark with n stars (*) indicates a construction taking $20n$ seconds to compile on a 64-bit server with a 2.40 GHz CPU and 16 GB of RAM.

Construction	**[7]**	[13]	[18]	[1]	[16]
Mostly automated (with custom Ltac)	✓				
Uses HoTT	✓			✓	
Uses type classes		✓			
Setoid of morphisms		✓	✓		✓
Uses higher inductive types	✓				
Assumes UIP or equivalent					✓
Category of sets	✓		✓	✓	✓
Initial/Terminal objects	✓	✓	✓	✓	✓
(co)limits	✓		✓	✓	✓
(co)limit functor	✓				
(co)limit adjoint to Δ	✓				
Fully faithful functors	✓			✓	✓
Essentially surjective functors		✓		✓	✓
Unit-Counit Adjunctions	✓	✓	✓	✓	✓
Hom Adjunctions	✓		✓		

[3] See `https://github.com/HoTT/HoTT/commit/eb0099005171` for the exact change.

Construction	**[7]**	[13]	[18]	[1]	[16]
Universal morphism adjunctions	✓		✓		
Adjoint composition laws	✓**				✓* [4]
Monoidal categories	✓	✓**********			✓*******
Enriched categories	✓	✓***			✓
2-categories					✓*
Category of (strict) categories	✓		✓		✓
Hom functor	✓	✓	✓	✓	✓
Profunctors	✓				✓
Pseudofunctors	✓*				
Kan extensions	✓			✓	✓
Pointwise Kan extensions	✓				
$\mathcal{C}^{\mathcal{D}^{\mathcal{E}}} \cong \mathcal{C}^{\mathcal{D}\times\mathcal{E}}$; $(\mathcal{C}\times\mathcal{D})^{\mathcal{E}} \cong \mathcal{C}^{\mathcal{E}}\times\mathcal{D}^{\mathcal{E}}$	✓				
Adjoint Functor Theorem			✓		
Yoneda	✓		✓	✓	✓***
dep. product (oplax lim $F : \mathcal{C} \to \mathrm{Cat}$)	✓				
dep. sum (oplax colim $F : \mathcal{C} \to \mathrm{Cat}$)	✓*				✓*
$(\_/\_)$ functor $(\mathcal{C}^{\mathcal{A}})^{\mathrm{op}} \times \mathcal{C}^{\mathcal{B}} \to \mathrm{Cat}_{/\mathcal{A}\times\mathcal{B}}$	✓******				
Rezk completion				✓	
Mean lines per file	71	126	133	305	98
Total compilation time	409s	517s	21s	129s[5]	717s
Total time w/o monoidal	409s	43s	21s	129s	579s
Median file compilation time	0.3s	0.4s	0.1s	0.7s	1.5s
Total number of files	136	36	105	19	143
Total number of definitions	522	214	995	478	396

In summary, our library includes many of the constructions from past formalizations, plus a few rather complex new ones. We test the limits of Coq by applying mostly automated Ltac proofs for these constructions, taking advantage of ideas from homotopy type theory and extensions built to support such constructions. We have summarized our observations on using new features from that extension and on other hypothetical features that could make an especially big difference in our development, and we hope these observations can help guide the conversation on the design of future versions of Coq and other proof assistants.

[4] The use of proof-irrelevant fields speeds up this construction significantly in Agda.

[5] Nearly 85% of the time in this library is spent checking simple facts about pullbacks and proving that functor composition preserves certain properties. At least a third of this time is spent in trivial lemmas made slow by Σ type projections.

References

[1] Benedikt Ahrens, Chris Kapulkin, and Michael Shulman. "Univalent categories and the Rezk completion". In: *ArXiv e-prints* (Mar. 2013).

[2] Steve Awodey. *Category theory.* Second Edition. Oxford University Press.

[3] Bruno Barras and Bruno Bernardo. "The implicit calculus of constructions as a programming language with dependent types". In: *FoSSaCS*. 2008.

[4] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill series in higher mathematics. McGraw-Hill, 1967.

[5] Robert Harper and Robert Pollack. "Type checking with universes". In: *Theoretical Computer Science* 89.1 (1991), pp. 107 –136. ISSN: 0304-3975.

[6] John Harrison. *Formalized mathematics.* TUCS technical report. Turku Centre for Computer Science, 1996. ISBN: 9789516508132.

[7] *HoTT/HoTT Categories.* URL: `https://github.com/HoTT/HoTT/tree/master/theories/categories`.

[8] Gérarde Huet and Amokrane Saïbi. "Constructive category theory". In: *Proof, language, and interaction.* MIT Press. 2000, pp. 239–275.

[9] Dexter Kozen, Christoph Kreitz, and Eva Richter. "Automating Proofs in Category Theory". In: *Automated Reasoning.* Springer, 2006, pp. 392–407.

[10] Robbert Krebbers, Bas Spitters, and Eelis van der Weegen. *Math Classes.* URL: `http://coq.inria.fr/pylons/pylons/contribs/view/MathClasses/v8.4`.

[11] Tom Leinster. *Higher Operads, Higher Categories.* Cambridge University Press, Aug. 2007. ISBN: 0521532159.

[12] Saunders Mac Lane. *Categories for the working mathematician.*

[13] Adam Megacz. *Category Theory Library for Coq.* Coq. URL: `http://www.cs.berkeley.edu/~megacz/coq-categories/`.

[14] Alexandre Miquel. "The Implicit Calculus of Constructions". In: *Typed Lambda Calculi and Applications.* Vol. 2044. Springer. 2001, p. 344.

[15] Greg O'Keefe. "Towards a readable formalisation of category theory". In: *Electronic Notes in Theoretical Computer Science* 91 (2004), pp. 212–228.

[16] Daniel Peebles, Andrea Vezzosi, and James Cook. *copumpkin/categories.* URL: `https://github.com/copumpkin/categories`.

[17] B. Pierce. *A taste of category theory for computer scientists.* Tech. rep.

[18] Amokrane Saïbi. *Constructive Category Theory.* URL: `http://coq.inria.fr/pylons/pylons/contribs/view/ConCaT/v8.4`.

[19] Michael Shulman. *An Interval Type Implies Function Extensionality.*

[20] Carlos Simpson. *CatsInZFC.* URL: `http://coq.inria.fr/pylons/pylons/contribs/view/CatsInZFC/v8.4`.

[21] Matthieu Sozeau et al. *HoTT/coq.* URL: `https://github.com/HoTT/coq`.

[22] Bas Spitters and Eelis van der Weegen. "Developing the algebraic hierarchy with type classes in Coq". In: *Interactive Theorem Proving.* Springer, 2010.

[23] *The Coq Proof Assistant.* INRIA. URL: `http://coq.inria.fr`.

[24] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* 2013.