A Circus-Ready Hoop-Jumping Rooster

or, A Case Study in Engineering a Category Theory Library in Coq

Jason Gross
MIT
jgross@mit.edu

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

David Spivak
MIT
dspivak@math.mit.edu

Abstract

We present a circus-ready rooster jumping through hoops, and the lessons we learned in training our rooster. We share what we learned about the design of circus hoops (in our case, the design of a category theory library), about jumping through hoops (the design of proof assistants in general), and about the training of roosters to jump through hoops (the suitability of Coq for formalizing category theory, and workarounds we found). We conclude by presenting one particularly shiny hoop our rooster can jump through (proof by duality can be encoded as proof by unification), which the authors did not see mention of in any other circus ad (published paper).

1. Introduction

Note: I haven't touched the old intro much yet. (I think it's still pretty good.)

Category theory [] is a popular all-encompassing mathematical formalism that casts familiar mathematical ideas from many domains in terms of a few unifying concepts. A *category* is an undirected 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 programming languages, specifically statically typed functional languages.

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 richly typed 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 []. Instead, a more appropriate setting is a computer proof assistant, which acts as a kind of IDE for stating and proving rigorous theorems. We chose to implement a library for Coq [], 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. Probably most categoricists would 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. Higher-order type theory [] 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.

2. Circus Hoops

We begin by describing various choices that came up in designing our category theory library, explaining the benefits and drawbacks of each option, and attempt to justify the choice we made.

2.1 Dependently Typed Morphisms

In standard mathematical practice, a 0-truncated 1-category $\mathcal{C}$ is typically defined[1] to consist of:

- a class $\mathrm{Ob}_{\mathcal{C}}$ of *objects*
- for each pair of objects $a, b \in \mathrm{Ob}_{\mathcal{C}}$, a 0-truncated 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 all composable morphisms f, g, and 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 [](HoTT BOOK), we require our morphisms to be 0-truncated (to have unique identity proofs).

We could just as well have replaced the classes $\mathrm{Hom}_{\mathcal{C}}(a, b)$ with a single 0-truncated class of morphisms $\mathrm{Hom}_{\mathcal{C}}$, together with functions defining the source and target of each morphism. The two formulations are mathematically equivalent.

[1] See, for example []

However, the latter formulation is significantly less convenient for formalization in Coq. If we use the formulation involving $\mathrm{Hom}_{\mathcal{C}}(a, b)$, then we can syntactically ensure that composition is only defined for morphisms which line up appropriately; attempting to compose morphisms that don't line up will simply fail to type-check. If we use the formulation with a single class $\mathrm{Hom}_{\mathcal{C}}$, then we have to manually manage the checking of whether or not two morphisms are composable; this is significantly more verbose.

2.2 Positioning your bells, whistles, and flames: Arguments vs. Fields

Once you settle what goes into making a category, you need to decide how to position the arguments. At one extreme, everything can be made a field; you have a type `Category` whose inhabitants are categories. At the other extreme, everything can be made an argument to a function `IsCategory`. Authors such as [] have chosen the intermediate option of making all of the computationally relevant parts (the types of objects, morphisms, composition, and the identity morphism) to be arguments, and the irrelevant proofs (associativity and left and right identity) to be fields. We present to pros and cons of each of these three options.

TODO(jgross): Decide whether or not to include code examples of each of these strategies, for explanatory purposes.

Everything on the outside

We have not found any significant benefits to making everything into an argument.

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. Although others have found this approach useful,[] we have not found ourselves wishing we had typeclass resolution when formalizing constructions.

There is a pragmatic advantage in currently released versions of Coq to making the type of objects a parameter rather than a field, which has to do with universe polymorphism. Talking about the (large) category of all (small) categories requires having at least two universes. If the definition of a category is not polymorphic over the universe level of its objects, then every definition and construction must be duplicated. The benefit of making the type of objects a parameter comes from the way currently released versions of Coq handle universe polymorphism: only parameters to inductive definitions (such as `Records`) are considered for universe polymorphism. However, Matthieu Sozeau is currently working on full universe polymorphism via typical ambiguity, largely following []. In the version of Coq used for homotopy type theory[2], removing the need to duplicate code when the type of objects is a field.

Everything on the inside

Once we moved to using the homotopy type theorists' Coq, we decided to use fields for all of the components of a category. This resulted in a factor of three speed-up in compilation time over the version where the types of objects and morphisms were parameters. 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 doesn't seem to matter much in most constructions; see section 5 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 from

$$\texttt{Functor} : \forall\ (\texttt{ob}_{\mathcal{C}} : \texttt{Type})\ (\texttt{ob}_{\mathcal{D}} : \texttt{Type})\ (\texttt{mor}_{\mathcal{C}} : \texttt{ob}_{\mathcal{C}} \to \texttt{ob}_{\mathcal{C}} \to \texttt{Type})\ (\texttt{mor}_{\mathcal{D}} : \texttt{ob}_{\mathcal{D}} \to \texttt{ob}_{\mathcal{D}} \to \texttt{Type}),\ \texttt{Category ob}_{\mathcal{C}}\ \texttt{mor}_{\mathcal{C}} \to \texttt{Category ob}_{\mathcal{C}}\ \texttt{mor}_{\mathcal{C}} \to \texttt{Type}$$

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 when you have a construction that uses natural transformations multiple times, the term size blows up very quickly.

2.3 When two circus hoops are the same: Equality

Equality has recently become a very hot topic in type theory.[](HoTT book) Although the question of what it means for objects or morphisms to be equal does not come up much in classical category theory, it is more important when formalizing category theory in a proof assistant. We consider three possibilities of notions of equality.

Propositional Equality

Intensional type theories, such as that of Coq, 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 a type theory and therefore cannot be explicitly reasoned about inside of the type theory, is the equality that holds between $\beta\delta\iota\zeta\eta$-convertible terms. It is called judgmental equality because the only terms which are judgmentally equal are the ones which the type-checker can automatically judge to be equal.[3]

Coq's standard library defines what's called *propositional equality* on top of judgmental equality, denoted $x = y$. One is allowed to introduce a propositional equality between judgmentally equal terms.[4]

Choosing to use propositional equality, rather than the setoids discussed below, is convenient because there is already significant machinery made for reasoning about propositional equalities. However, we ran into significant trouble when attempting to prove that the category of sets has all colimits, because quotient types can't be encoded without assuming a number of other axioms.

Setoids

The traditional fix for the problem of quotient types is to replace types with setoids, which are types equipped with an equivalence relation, and each function carry around a proof that it respects the equivalence relation of its domain and codomain.[5] Although this allows us to define quotient types very easily, there is significant overhead associated with using setoids everywhere. 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 which respects the equivalence relations of those types.

[2] Currently available at `https://github.com/HoTT/coq`

[3] For example, as of Coq 8.4, we have $(\lambda\ x,\ f\ x) \equiv f$; this is the restricted version of the η rule that Coq implements. We also have that $(\lambda\ x,\ f\ x)\ y \equiv f\ y$; this is the β rule.

[4] Defined in Coq as
`Inductive eq (`T `: Type) (`x `:` T`) :` T `→Prop:= eq_refl : eq` T x x`.`

[5] See [](PAPERS FORMALIZING CATEGORY THEORY WITH SETOIDS)

Higher Inductive Types

A third option has recently emerged, which seems to allow the best of both worlds. The idea of higher inductive types is to allow inductive types to be equipped with extra proofs of equality between constructors. A very simple example is the interval type, from which functional extensionality can be proven.[] 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 which don't respect the equivalence relation. The key insight is that most types don't need any special equivalence relation, and, moreover, if we're not explicitly dealing with a type with a special equivalence relation, then it's impossible (by parametricity) to fail to respect the equivalence relation.

Univalence

When considering higher inductive types, the question "when are two types equivalent?" arises naturally. The standard answer in the past has been "when they are syntactically equal". The result of this is that two inductive types which are defined in the same way, but with different names, will not be equal. Voevodsky's univalence principle gives a different answer: two types are equal when they are isomorphic.

Although it is natural to extend this idea to categories, and declare that two objects should be equal when they are isomorphic, we have found that basic category theory constructions work fine without univalence and without making assumptions about what it means for two objects to be equal. However, it is likely that more advanced or exploratory category theory will benefit from this univalence requirement on categories; for example, it makes it impossible to do anything evil.[6] See [](HoTT Book, Ch 9) for more details.

2.4 Hoop Organization: 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; the first author developed an allergic reaction to constructions with more than five or so arguments after making one too many mistakes in defining limits and colimits.

Perhaps less typical of programming experience, we found that picking the right abstraction barriers could drastically cut down on compile time. In one instance, we got a factor of ten speed-up by plugging holes in a leaky abstraction barrier![7] In type theory (and other purely functional languages), breaking an abstraction barrier is unfolding the function that defines the interface. Because the time it takes to manipulate a term depends on the size of the term, needless unfolding of large functions incurs a harsh penalty. A good abstraction barrier is one that hides as much complexity as possible (maximally reducing term size) and never needs to be broken by unfolding (thereby eliminating the penalty).

Finally, we found that picking the right abstraction barrier simplified proofs by forcing us to ignore useless details. A great example of this was our experience with (co)limits. After defining limits as terminal objects of the appropriate comma category, we set out to formalize the well-known theorem that if a category $\mathcal{C}$ has all $\mathcal{D}$-shaped (co)limits, then these (co)limits fit together into a (co)limit functor $\mathcal{C}^{\mathcal{D}} \to \mathcal{C}$, which is furthermore adjoint the the constant diagram functor $\Delta : \mathcal{C} \to \mathcal{C}^{\mathcal{D}}$. After some hard work, we managed to prove both of these theorems.

Some months later, the first author discovered the universal morphism definition of adjoint functors.[](Wikipedia) We found that it was very straightforward to take the proofs that (co)limits assembled into functors, and turn it into a proof that universal morphisms assemble into adjoint functors, and vice-versa. Furthermore, we found some ways to simplify the proofs, which became obvious once we were forced to abstract away the functor we were constructing an adjoint to, and that one of the categories involved was a functor category.

2.5 More Hoops or More String: 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. For example, when constructing 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. This corresponds to replacing various equalities with isomorphisms. 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.

We also chose to a natural transformation between the functors $F \circ (G \circ H)$ and $(F \circ G) \circ H$, rather than a proof of equality, when defining the unit and counit of a composition of adjunctions; the benefit of using natural transformations rather than equalities is that the proof of equality does not reduce, even when we only care about a portion of it that reduces. For example, since functors act on objects and morphisms, if we have that two functors are equal, then we have that their action on objects are equal. Even though we can prove that this proof of equality (between the action on objects), derived from the proof that functor composition is associativity, is just reflexivity, it is not judgementally so, and thus the pattern matching does not reduce. Using natural transformations, by contract, results in immediate reductions, simplifying the proofs significantly.

3. Hoop-Jumping

In addition to providing useful guiding principles of category theory library design, our case study also gave rise to suggestions for new useful features of proof assistants.

3.1 Moving Hoops: Computation Rules for Pattern Matching

In subsection 2.5, we saw some 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 also discovered another flavor of judgmental pattern matching computations rules that we wanted to have, when working on duality. Duality is the idea that, sometimes, its productive to flip the direction of all the arrows. For example, if you prove something about least upper bounds, chances are you'll be able to prove the same kind of thing about greatest lower bounds, by replacing all of your "greater than"s with "less than"s and vice versa. In category theory, this is realized by saying that for every category $\mathcal{C}$, there is an opposite category $\mathcal{C}^{\mathrm{op}}$ which has the same objects, but for every arrow $x \to y$ in $\mathcal{C}$, we get an arrow $y \to x$ in $\mathcal{C}^{\mathrm{op}}$. Clearly,

[6] A construction in category theory is *evil* if it is not invariant under isomorphisms. By widening the notion of equality to be the same as the notion of isomorphism, we get for free that every construction we do is invariant under isomorphism.

[7] See `https://github.com/HoTT/HoTT/commit/eb0099005171e642d467047933660980ddc66280` for the exact change.

when you perform this operation twice, you get back the category you started with. However, this is not true judgmentally. For example, if you apply symmetry twice to a proof of equality, you get back a judgmentally distinct proof. While we were able to work around most of these issues, as explained in subsection 4.2, things would have been far, far nicer if we had more η rules. The η rule for records says that if you apply the constructor to the projections, you get back what you started with; so for products, this says that $x \equiv (x_1, x_2)$ (where x_1 and x_2 are the first and second projections, respectively). For categories, the η rule says that if you have a category $\mathcal{C}$ and you define a new category whose objects are the objects of $\mathcal{C}$, whose morphisms are the morphisms of $\mathcal{C}$, ..., then your new category is judgmentally equal to $\mathcal{C}$. 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 equality. Matthieu Sozeau is currently working on giving Coq judgmental η for records, though not for equality.

The first author is currently attempting to divine the appropriate computation rules for pattern matching constructs, in the hopes of making reasoning with proofs of equality more pleasant.[8]

3.2 Invisible String: Higher Inductive Types

As explained at the end of subsection 2.3, higher inductive types provide a way to develop most of category theory using propositional equality, which tends to be convenient, without sacrificing the ability to define quotient types.

3.3 Invisible Hoops: Opacity

Coq is slow at dealing with large terms. When I have goals which are around 150 000 words long, I've found that tactics like `apply f_equal` take around 1–2 seconds to execute. This 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 which are provably equal to all other functions of the same type.

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 wouldn't impose any penalties on term manipulation.

The naïve way to do this is to grant equality reflection for -1-truncated types; that is, whenever we have a proof that $\forall\ x\ y\ :\ T,\ x = y$, the type-checker will reflect the propositional equality into a judgmental equality, and give us $\forall\ x\ y\ :\ T,\ x \equiv y$. However, this would make type-checking undecidable: For any $x\ :\ T$, the dependent sum $\sum_{y\,:\,T} y = x$ is contractible, because all of its inhabitants are provably equal to $(x; \mathtt{refl}_x)$ by induction. Then we would get that all inhabitants of $\sum_{y\,:\,T} y = x$ are judgmentally equal, and hence that all proofs of $y = x$ are judgmentally equal for any x and y. This is called equality reflection, and is known to make type-checking undecidable.

It's not clear what is the right way to achieve this goal. It's possible that equality reflection would not lead to undecidable type-checking if it is restricted to equalities between proofs of equality. Alternatively, there might be some way to ignore the terms when doing most computation, without changing the underlying theory.

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

4. Rooster Training

Most of the deficiencies that we found in the Coq proof assistant were not fatal. We present here the workarounds that we found for dealing with most of the problems mentioned in the previous section, as well as some problems that we believe are quirks of Coq, rather than indicative problems or desired features with all current proof assistants.

4.1 Hoop Assembly: Records vs. Nested $\sum$ 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 $\sum$ type. Records are a convenient syntactic sugar for defining inductive types with one constructor and many fields, and simultaneously defining the projections at the same time. By way of example, here is the definition of the computational parts of a category as a record, as the desugared inductive type, and as the nested $\sum$ type:

Category as a record

```
Record Category := {
  Ob : Type;
  Hom : Ob -> Ob -> Type;
  Identity : forall x, Hom x x;
  Compose : forall x y z,
    Hom y z -> Hom x y -> Hom x z
}.
```

Category as an inductive type

```
Inductive Category :=
Build_Category
 : forall
     (Ob : Type)
     (Hom : Ob -> Ob -> Type)
     (Identity : forall x, Hom x x)
     (Compose : forall x y z,
       Hom y z -> Hom x y -> Hom x z),
   Category.

Definition Ob C :=
 let (Ob, _, _, _) as C' return Type := C in Ob.
Definition Hom C :=
 let (Ob, Hom, _, _) as C'
   return Ob C' -> Ob C' -> Type
 := C in Hom.
Definition Identity C :=
  let (Ob, Hom, Identity, _) as C'
    return forall x, Hom C' x x
  := C in Identity.
Definition Compose C :=
  let (Ob, Hom, Identity, Compose) as C'
    return forall x y z,
      Hom C' y z -> Hom C' x y -> Hom C' x z
  := C in Compose.
```

Category as an nested $\sum$ type

```
Definition Category :=
  { Ob : Type &
  { Hom : Ob -> Ob -> Type &
  { Identity : forall x, Hom x x &
  { Compose : forall x y z,
       Hom y z -> Hom x y -> Hom x z &
    unit }.
Definition Ob C : Type
 := projT1 C.
```

```
Definition Hom C : Ob C -> Ob C -> Type
 := projT1 (projT2 C).
Definition Identity C : forall x, Hom C x x
 := projT1 (projT2 (projT2 C)).
Definition Compose C
   : forall x y z,
       Hom C' y z -> Hom C' x y -> Hom C' x z
 := projT1 (projT2 (projT2 (projT2 C))).
```

The differences

There are two main differences in Coq. The first is that while you can prove theorems about nested $\sum$ types in general, you can't prove theorems about records in general. This is a pain and leads to code duplication. The far more pressing problem is that nested $\sum$ types have horrendous performance, and are sometimes a few orders of magnitude slower. This comes from the fact that the projections of nested $\sum$ types, when unfolded (which they must be, to do computation), each take almost the entirety of the nested $\sum$ type as a type argument, 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 $\sum$ types (if $\sum$ itself were defined as a record) by eliminating the arguments to the projection functions.

4.2 Spinning Roosters: Tricks for Involutive Duality

Having the dual of a category be judgmentally equal to itself is very useful; see section 5 for more details. We managed to avoid the need for judgmental η for records; we only cared about dualizing categories which were already eta expanded. We found a number of tricks to get around the need for η for equality.

Removing symmetry

When you take the dual of a category, you need to construct 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 way of doing this is to apply symmetry, but we'd need judgmental η for equality for this to work. Instead, we extended 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 swapped the proofs. We added a convenience constructor for categories that asked only for one of the proofs, and applied 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 these other proofs doesn't result in and coherence issues.

Dualizing the terminal category

To make everything work out nicely, we needed the terminal category (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 $y = x$. We thus had to instead use the indiscrete category, which instead 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, you have to pick whether to prove that the functor preserves composition using the right identity law, or the left identity law; both will prove that the identity composed with itself is the identity. The problem is that when you dualize the functor, you will then discover that you picked the wrong rule, and so the dual of your 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

The last problem we ran into was the fact that sometimes, we really, really wanted judgmental η. For example, we wanted to say that you could get any functor out of the terminal category as the opposite of some other functor; namely, if you have $F : 1 \to \mathcal{C}$, then it should be equal to $(F^{\text{op}})^{\text{op}} : 1 \to (C^{\text{op}})^{\text{op}}$. To get around this, we made two variants of dual functors: given $F : \mathcal{C} \to \mathcal{D}$, we have $F^{\text{op}} : \mathcal{C}^{\text{op}} \to \mathcal{D}^{\text{op}}$, and given $F : C^{\text{op}} \to \mathcal{D}^{\text{op}}$, we have $F^{\text{op}'} : \mathcal{C} \to \mathcal{D}$. There are two other flavors of dual functors, corresponding to the other two pairings of $^{\text{op}}$ with domain and codomain, but thank goodness we didn't need them. As it was, we ended up having four variants of dual natural transformation, and are very glad that we didn't need sixteen. We look forward to Coq 8.5, when we will hopefully only need one.

4.3 When Roosters find Hoops inside Hoops: Indexing vs. Subsets

In Coq, as in mathematics, there are approximately two ways to talk about subsets $A \subseteq B$.

One way is by giving function $P : B \to \texttt{Prop}$. In this case, we identify the subset A with the sigma type $\sum_{b\,:\,B} P\ b$, which in Coq is suggestively denoted $\{b\ :\ B \mid P\ b\}$. Inhabitants of this type are dependent pairs, an element $b \in B$ and a proof of $P\ b$. Note that in classical mathematics, it is standard to identify the subsets of B with functions $B \to 2$ rather than $B \to \texttt{Prop}$; this corresponds to assuming that all propositions are decidable, and satisfy proof-irrelevance. This is most useful when you already have the type B in hand. We use this method to define subcategories.

The other way is by giving an injection $A \hookrightarrow B$. This is convenient when you don't already have the type B in hand, and due to limitations in your proof assistant (such as Coq 8.4), committing to a particular type B would also be committing to living in a particular universe. In these cases, you can instead provide a function $U : I \to \texttt{Type}$ and a function $f : forallx, U\ x$, allowing you to use a fresh `Type` variable every time you talked about such an indexed category. Before moving to the homotopy type theorists' Coq, we used this method to define the category of (small) categories.

4.4 Turning Roosters into Hoops: Reified Simplification

We implemented two simplification procedures for morphisms in categories, one based on reification using typeclasses, and the other based on reification using canonical structures, as described in []. Currently, the routines remove all compositions with identity morphisms, or functors applied to identity morphisms.

During preliminary testing, the canonical structure simplification routine is about twice as fast as the typeclass simplification routine that uses reification, which is itself about twice as fast as the typeclass simplification procedure which does not use reification, which is itself about twice as fast as repeated rewriting.

Unfortunately (depending on your perspective), this effect only becomes noticeable when the goal is already horrendously large, to the point that interactive theorem proving is unpleasant. In most cases, naïve rewriting with the handful of lemmas used by the simplification machinery takes under a second, so we have not found these simplification routines to be particularly useful.

5. One Shiny Hoop

We discovered that proof by duality can be encoded as proof by unification, an idea which we were unable to find expressed in the

literature, and which we think merits a short explanation. Proof by duality is a common idea in higher mathematics. We find it noteworthy that not only is there an isomorphism between the type of a theorem and the type of its dual, but that, if definitions are made carefully, then the types actually unify! Thus, in homotopy type theory, not only are theorems homotopic to their duals, but they can be made to be judgmentally equal to their duals.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...