# Proofs and Conversations

Talia Ringer [*]

May 17, 2024

My research is on making it easier to write formal, machine-checkable proofs using special tools called proof assistants. So of course, I love these tools, and I want everyone to have a chance to use them. I am just now noticing that I have been selling my love for these tools to mathematicians the wrong way.

The sales pitch in my field is obvious. My work is primarily in the field of formal verification: using these tools to write machine-checkable proofs about software systems. In my field, we really care about our software being correctly implemented, down to the tiniest details. Subtle mistakes in software systems can be catastrophic, expensive, and even fatal. The most powerful way to avoid these mistakes is to formally prove our software correct.

So for a long time, I told mathematicians that these tools are great because they make it possible to have full confidence in one's results. The response to this was often one of confusion. The normal way of doing math has worked pretty well for most of history (modulo the occasional surmountable crisis like Russell's Paradox). Why go formal?

I can now see the *right* sales pitch to mathematicians: formal proof has the potential to empower collaboration at a scale never before seen in mathematics. For example, mathlib [mC20], the large formal math library implemented in the proof assistant called Lean, has had 141 contributors in the past month at the time of writing. These contributors range from professional mathematicians to hobbyists and everything in between. If you want to contribute, you can, too. This is the beauty of the collision of the world of proofs with the world of software.

It will take time and patience for this to catch on. Most mathematicians I have spoken to still view the proof assistant as making it *harder* to prove their results rather than *easier*. Mathematicians who use proof assistants still find themselves having to think hard about details that they often take for granted, because nothing is really obvious to a computer. Much of the work being done right now is on building the infrastructure that will help mathematicians abstract over those details in the future.

In my research field, our formal proof infrastructure is mature enough that many of us find it *easier* to write a proof in a proof assistant. The proof assistant really *assists* us. Math will get there too, and when it does, we will find ourselves in a world where more and more people can participate in the Great Conversation of Mathematics.

## Proof Assistants

To use a proof assistant, we start by writing definitions directly inside of the proof assistant (or finding relevant definitions someone else has already written). We then state theorems about these definitions. Finally, we write proofs about these theorems interactively, with the help of the proof assistant.

Suppose we wish to prove that every natural number is even or odd. We will do so using a popular proof assistant called Coq. We can get the definitions of "even" and "odd" from Coq's standard library:

```
Definition Even n := ∃ m, n = 2*m.
Definition Odd n := ∃ m, n = 2*m + 1.
```

We can use these to state our theorem:

*The author is an Assistant Professor of Computer Science at University of Illinois, Urbana-Champaign. Their email address is tringer@illinois.edu.

```
Theorem even_or_odd:
  ∀ (n : nat), Even n \/ Odd n.
```

We can then move into the interactive proof mode by simply typing the word `Proof`. In this interactive proof mode, next to where we are typing, Coq displays our current goal to us, which at this point is just the theorem we stated. To prove this goal, we send Coq these high-level strategies called *tactics*. Here, we can use a tactic that does induction:

```
  induction n.
```

Coq responds by refining the goal into the base case and the inductive case. In the base case:

```
1 goal
______________________________________(1/1)
Even 0 \/ Odd 0
```

our goal is to show that zero is either even or odd. Zero is obviously even, but "obviously" does not really *compute*. So we provide more detail than we may be used to. Our goal `Even 0 \/ Odd 0` is a disjunction, so first, we tell Coq that we will prove the disjunction by proving its left side (`Even 0`). Then, we explicitly choose `0` for `m` (there are some ways around providing this much detail, but they have many caveats). *Then* it is obvious. In other words:

```
  - left. exists 0. auto.
```

Here, `left` refines the goal to the left side of the disjunction, `exists 0` chooses `0` for `m`, and `auto` takes care of the "obvious" part.

In the inductive case, our goal is to show that, given some natural number `n`, if `n` is either even or odd, then so is its successor (denoted `S n`):

```
1 goal
n : nat
IHn : Even n \/ Odd n
______________________________________(1/1)
Even (S n) \/ Odd (S n)
```

Note that our inductive hypothesis is given a name, `IHn`, which we can refer to explicitly. Here, we call the `destruct` tactic on it, which does case analysis, splitting into the even and odd cases:

```
  - destruct IHn.
```

In the even case:

```
1 goal
n : nat
H : Even n
______________________________________(1/1)
Even (S n) \/ Odd (S n)
```

we know the successor is odd. But we have to do more work, again, since this is a computer checking the result. So after choosing the right side of the disjunction in the goal (that is, saying the successor is odd), we use `destruct` again on `H`, the fact that `n` is even. What this does is use the definition of evenness to assert that there is some `x` for which `n = 2*m`. We can then use that same exact `x` to show that `S n` is odd by the definition of oddness:

```
    + right. destruct H. exists x. lia.
```

where `lia` invokes a simple linear arithmetic solver to prove the remaining equality in a way that satisfies Coq. The odd case is fairly similar, and then we are done, so that our proof looks like this:

```
Theorem even_or_odd:
  ∀ (n : nat), Even n \/ Odd n.
Proof.
  induction n.
  - left. exists 0. auto.
  - destruct IHn.
    + right. destruct H. exists x. lia.
    + left. destruct H. exists (S x). lia.
Qed.
```

What I call a "proof" here is really a *proof script*—a sequence of tactics that proves our goal. But Coq does not check this proof script directly. Instead, it translates the whole thing down to this low-level representation called a *proof term*. This proof term is a purely logical representation of the proof, without any abstraction, and so it is often quite large. What is important is that Coq can check this purely logical proof term against the theorem statement automatically, giving us certainty that it proves that theorem.

While this proof is a toy example, Coq and its siblings like Lean and Isabelle/HOL have been used to write proofs about both state-of-the-art mathematics and security-critical software systems.

## Choosing a Proof Assistant

There are many proof assistants to choose from. Lean is likely the most popular in mathematics in the US these days, but I would not let that stop you from exploring other proof assistants. For example, I use Coq to write proofs about software and about programming languages. I often use a fairly niche proof assistant called Cubical Agda for higher-dimensional reasoning, like to reason about homotopies or, more generally, how proofs themselves relate to each other.

There are many axes along which proof assistants vary that factor into one's choice of proof assistant. Some of these axes relate to the proof assistant's community of users: mathematicians versus computer scientists, means of interacting, axioms communally agreed upon as OK to assume,[1] and style of writing proofs. Others relate to infrastructure: libraries, frameworks, automation, user interfaces, languages, and archives. Still others relate to the guts of the proof assistants: logical foundations and expressiveness, means of achieving trustworthiness, and ways of representing proofs internally.

Mathematicians I speak to often claim that the guts of the proof assistant do not matter to them. I think they do matter, they are just abstracted away. In fact, the guts are what *allow* for abstraction to begin with. For central to these guts is a design principle that states that there ought to be separation of concerns between the thing that *produces* the proof and the thing that *checks* the proof [BB02, BW05]. The thing that checks the proof should be a small, human-readable logic checker called the kernel. The thing that produces the proof (like the proof script we saw earlier) is then free to do pretty much anything, so long as in the end it produces something that the kernel can check (the corresponding proof term). This checking happens when we write `Qed`.

This separation of concerns is what makes the proof assistant trustworthy while empowering users to build and use automation that allows for higher and higher levels of abstraction. Since we can trust lemmas and theorems once they have been proven, we can also build on previous results, just like we would in math. This makes it possible for communities of users to work in parallel on different proofs, using one another's results smoothly. This whole experience then starts to look a lot like software engineering.

[1] Yes, this is a thing, and it is one reason many mathematicians tell me they prefer Lean to Coq, even though I love Coq.

## Proof Engineering

The view of writing machine-checkable proofs through a software engineering lens is called *proof engineering* [RPS$^+$19]. Every aspect of writing formal proofs, from the community to the infrastructure to the guts, has parallels in software engineering. This is good because the software engineering community has figured out a lot about how to make things easier, and all of that becomes available to us. For example, we can use existing systems to track changes to our work-in-progress proofs with our collaborators, and we can adapt design principles from software engineering to help us write proofs collaboratively.

The benefits of viewing proofs through a software engineering lens become especially potent when it comes to empowering collaboration. One example I like of this has to do with how I interact with some of my students—by *pair proving*. I view pair proving as the proof analogue to *pair programming*. In pair programming, the *driver* writes code while the *navigator* helps steer the process and give feedback. Every so often, programmers switch roles.

When I pair prove with my students, I like to start as the navigator, and occasionally jump in as driver if something is easier to show than to explain. My most useful purpose as navigator is to help students figure out when to move between *bottom-up* and *top-down* reasoning. Bottom-up reasoning asks: given what we know, what is it we can show? Top-down reasoning asks: given what we would like to show, what do we need to know? Students are often good at both of these individually, but they tend to get stuck in one mode of reasoning when switching to the other would be more effective. As navigator, though, I can help them figure out when to switch directions. This is one small way proof engineering helps us collaborate.

## The Great Conversation

The big promise of proof engineering comes when we look at collaboration between entire communities all around the world. Then, we can see the ways these tools and principles and communities can empower and enhance the Great Conversation of Mathematics, or even reduce its barriers to entry.

My favorite example of this came from Terry Tao when he was learning how to use Lean. He shared a link to his Lean proofs inside of a GitHub repository. GitHub repositories are a common way that people store code that makes it easy to contribute. Contribution happens by way of something called a pull request—a collection of changes made locally on one's computer that are submitted to the original authors for review and eventual approval. An approved pull request becomes part of the code.

This is exactly what happened. The pull requests trickled in immediately, and I found myself in awe. I realized that, in 2024, anyone in the world can submit pull requests to collaborate with Terry Tao. He had spoken to me once about how we had entered this era of mathematics where collaborating and bridging fields are valuable skills. Watching Terry's experiences with Lean, I realized that proof assistants are a powerful tool in this era of collaboration and bridging fields. Mathematicians *need to know.*

In the futuristic world we live in, when you submit a pull request to Terry Tao's GitHub repository, you do not need to worry much about accidentally breaking his proofs—no matter who you are. Thanks to the separation of concerns between proof production and proof checking, you can just check the revised proofs locally. If you change the top-level theorem he is proving or the axioms he relies on, maybe then you should start to worry. So long as you leave those in tact, though, you might be able to help him fill in holes or improve the elegance or clarity of his proof. To be sure, you can check the result and make sure it passes Lean's proof checker. Then you can submit your pull request and suddenly you are, in some sense, collaborating with Terry Tao. He and you are linked in the Great Conversation of Mathematics.

## Broadening the Conversation

An even more exciting possibility stems from broadening the very notion of what the Great Conversation of Mathematics could be—and letting it include computers. Large language models like ChatGPT, for example, are fundamentally unreliable, but it turns out this lack of reliability does not matter if we use the language model to generate formal proofs of theorems we have already stated, since the proof assistant's kernel can check the proof in the end.

Thanks to this certainty, we can start to include computers at many points of the Great Conversation—asking and answering questions, helping with discovery, debugging faulty conjectures, dispatching proofs of lemmas, finding relevant information, discovering connections and analogies and helping you use them for your goals—all without compromising trust. I hope someday this conversation grows into a computer-aided community of mathematicians at a scale never before seen, where anyone can participate. Of course, the goal should never be to replace mathematicians—only to empower you all to explore the world of mathematics more and more, with more tools at your disposal.

## Getting Started

Want to try these proof assistants? There is a large document full of resources that we recently put together during the National Academies AI for Math Seminar.[2] The tutorials for formal proof listed in that document, like the Lean Natural Number Game, are an especially great place to start. The discussion forums listed, like the Lean Zulip, are indispensable to new and seasoned proof engineers alike. Be patient—it may take some time to get used to being explicit about things that you may take for granted in pen-and-paper mathematics, or finding the right automation to help you not need to be explicit about those things. And above all, do not be afraid to keep asking questions. People want to help. Enjoy!

[2] `https://docs.google.com/document/d/1kD7H4E28656ua8jOGZ934nbH2HcBLyxcRgFDduH5iQ0`

## References

[BB02] Henk Barendregt and Erik Barendsen, *Autarkic computations in formal proofs*, Journal of Automated Reasoning **28** (2002), no. 3, 321–336.

[BW05] Henk Barendregt and Freek Wiedijk, *The challenge of computer mathematics*, Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences **363** (2005), no. 1835, 2351–2375.

[mC20] The mathlib Community, *The lean mathematical library*, Proceedings of the 9th acm sigplan international conference on certified programs and proofs, 2020, pp. 367–381.

[RPS$^+$19] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock, *Qed at large: A survey of engineering of formally verified software*, Found. Trends Program. Lang. **5** (2019sep), no. 2–3, 102–281.