

Formalizing Mathematics with AI

Adam Fiedler
www.adamfiedler.com

Introduction

There has been a lot of talk about formalizing mathematics recently, ever since OpenAI proposed [a solution to the famous Navier-Stokes problem](#) formalized in [Lean](#). Since I did some work in mathematical logic in the past, I thought it might be useful to clarify what the talk is all about. I will only scratch the surface and not go into details about Lean. We will mainly talk about constructive mathematics, and how it maps to machine-checkable proofs. I will then briefly summarize the heated discussion about generating mathematics with AI. Note that for you to understand this blog post in full detail, you need at least a basic knowledge of logic and proofs. If you are not interested in technical details, you can skip the first sections and read [the last section](#).

To give you an introductory example, I thought about the classical proof in arithmetic that we are taught very early on. This is the proof that there are infinitely many prime numbers. The intuition is clear: the opposite would be in contradiction with the prime factorization theorem because you would not have enough primes. Without a formal proof, this is not at all so obvious, though. I realized that the proof that is often taught is actually proof of negation, not proof by contradiction, and I decided that it might be fun to formalize in Lean. If you are interested in what this all means, read on.

An Easy Proof To Formalize

Before we start, let us refresh what the prime factorization theorem is about. The prime factorization theorem says that any natural number greater than 1 is either a prime number or can be expressed as a unique product of prime numbers (obviously unique modulo the order of the multiplicands, as multiplication is commutative). In human language, it just means that any natural number can be factored into primes, i.e.,

Theorem 1 (Prime Factorization). *For every $n \in \mathbb{N} > 1$, it holds that $n = q_1^{z_1} \cdot \dots \cdot q_m^{z_m}$ for some primes $q_1, \dots, q_m$ (and powers $z_1, \dots, z_m \in \mathbb{Z}^+$).*

This theorem is the cornerstone of arithmetic, it is one of the reasons why people are so much interested in prime numbers. For us to be able to factor anything into primes, the intuition tells us that there must be *always enough* primes at our disposal. Does it really mean there are infinitely many of them? We will prove the following version of this statement¹:

¹this version is in some sense weaker, but this is out of scope now.

Theorem 2 (Statement F). *There is no greatest prime number.*

Proof. Assume, for a contradiction, that there is a greatest prime number $p_n = p$. This means that there are only finitely many prime numbers $p_1, p_2, \dots, p_n$: simply list all natural numbers up to p that pass the primality test (*). Now consider the following number:

$$x = p_1 \cdot p_2 \cdot \dots \cdot p_n + 1.$$

By construction $x > 1$ and thus by prime factorization theorem also $x = q_1^{z_1} \cdot \dots \cdot q_m^{z_m}$ for some prime numbers $q_1, \dots, q_m$. However, the primes $q_1, \dots, q_m$ are contained in $p_1, \dots, p_n$ by (*). And so by the definition of x , clearly $q_i \nmid x$ for all $1 \leq i \leq m$ (we always get a remainder of 1). Thus $q_1, \dots, q_m$ are factors of x and are not factors of x , which is a contradiction. $\square$

This proof is usually presented as one being a proof by contradiction: you assume the negation of what you want to prove, and you lead it to a logical contradiction to conclude the original. In actual fact, it is one of a special class of proofs called *proofs of negation* and it is important to understand the difference from proofs by contradiction if you want to understand the world of proof assistants better. First, let us look at proofs by contradiction.

Proofs by Contradiction

In classical mathematics, we often use the law of the excluded middle (LEM). LEM roughly says that any statement is true or not true, i.e., for any statement Q we have $Q \vee \neg Q$. This is very intuitive and is generally accepted in the wider mathematics community. LEM has its disadvantages though, one being that it can lead to proofs that are not informative. Let me elaborate.

In the world of LEM, it is perfectly ok to accept the so-called double negation elimination (DNE): if you say *I never didn't like this movie*, you practically say *I not ever did not like this movie*, which by classical logic means you always liked this movie (there was never a point in time where you did not like the movie). We can abstract this into a more formal principle, which says that from $\neg\neg L$ you can conclude L (this is DNE). It can be shown that DNE follows from LEM.

To make things simple, we will define negation $\neg L$ as syntactic sugar standing for $L \rightarrow \perp$.²³ Then DNE follows from LEM like this:

1. Assume $\neg\neg L$.
2. By LEM, we know L or $\neg L$.
3. Let's do a case analysis:
 - If L : we are done (we wanted to show L).
 - If $\neg L$: by our syntactic sugar we can unpack $\neg\neg L$ to $(\neg L) \rightarrow \perp$, so by modus ponens (MP) we have $\perp$. From $\perp$ follows anything, including L .⁴

²where $\perp$ means a contradiction, what this means formally is a technical detail

³why this syntactic sugar is usually valid is also a technical detail

⁴ex falso quodlibet

This is just a step to proofs by contradiction, why? Because proofs by contradiction are double negation elimination in disguise. You assume the opposite, derive a contradiction, just to arrive at the original conclusion. Formally this means that you show $\neg L \rightarrow \perp$ to conclude L . How do you conclude L ? Once you prove $(\neg L \rightarrow \perp)$, you can see this as $\neg\neg L$, so we simply apply DNE to arrive at L .

The typical problem with this approach is that you can derive existence of an object without actually *constructing* it. For simplicity, let us consider a trivial example. Say you want to prove *There exists a minimal element of natural numbers* $\mathbb{N}$, this would be our L . You do not need to say it is 0 and show why it is so, you can simply assume $\neg L$ and show how it leads to a contradiction with the definition of $\mathbb{N}$. This is enough to prove that there is a minimal element while it says absolutely nothing about how 0 looks.

Lean includes LEM in its standard library, but using it has certain repercussions in Lean. We cannot go into full detail here. Proving existence should require you to show how to construct the given object, which is actually quite intuitive to expect. The way we can enforce this in our proofs is by avoiding LEM. Avoiding LEM sometimes makes things quite counterintuitive, especially if you are used to them from classical mathematics. However, it poses no problem for us and our statement F , because we can still use *proofs of negation*.

Back to Our Statement F

In constructive mathematics, negating a statement is saying the statement leads to a contradiction (in the same way as we defined our syntactic sugar). Wait, what? Didn't I say using contradictions is not that useful? (I wanted to write I was contradicting myself, but that would be too many contradictions). Not at all: proving $L \rightarrow \perp$ (proving a negation) is different from proving $\neg L \rightarrow \perp$ to show L (proving by contradiction).

In classical mathematics, what really happens in proofs by contradiction is that since $\neg L \rightarrow \perp$ is nothing but $\neg\neg L$, you can simply apply DNE to get L . On the other hand, proofs of negation are simply proving $L \rightarrow \perp$, which in constructive mathematics is actually the definition of $\neg L$. And this is completely consistent with our proof of F : we want to prove that there is no greatest prime number, i.e., we want to prove the non-existence of such a number. By assuming its existence, we get a contradiction, and thus the entire proof is simply proving the implication *If there exists a greatest prime number, then we get a contradiction*.

This is not just a mathematician's fancy: you can sense how the proof of F **we saw** is "concrete", and very much mechanizable. You can tell Lean exactly what to do in this proof, each step is set in stone as a simple transformation. The major steps are as follows:

1. You assume a greatest prime number p .
2. You construct a set of all prime numbers $ps := \{p_1, p_2, \dots, p_n = p\}$ simply by listing all prime numbers up to p .⁵

⁵for people knowledgeable about Lean, checking if a number is prime is decidable and exhaustiveness is given by iterating over all naturals up to p .

3. You construct x as shown, by taking the product of p s and incrementing it.
 - You prove $x > 1$ and apply prime factorization to x .
 - You pick the first (or some) factor q_1 of x , let's call it q .
4. You introduce an in-place lemma $q \mid x \wedge q \nmid x$ (this is our contradiction).
 - You show the left side of the conjunction by pointing q to the factorization $q_1, \dots, q_m$, and the right side of the conjunction by showing that q is included in $p_1, \dots, p_n$ and then computing the remainder.

Obviously I am skipping some steps but this proof is very easily formalized. Let us now have a quick look at what proofs in Lean are.

Proofs in Lean

[Lean](#) is a programming language that can be used, among other things, to prove theorems. I want to emphasize that theorems do not need to be mathematical, they can also be properties and specifications of programs and algorithms in specification-driven development. Theorems in Lean are nothing but *types*. Think of it like this: if you have a function in Java, for example, that returns a list of top 10 artists, you are giving a *specification* that says that this function returns a list. You can go further than this, and say, for example that your function returns a non-empty list containing exactly 10 elements. You would have a type that takes a value `length`, which says how long the list is, which is called a *dependent type* (it depends on a value). Then your function returns `List<10>`. So now you have a theorem saying that your function returns a list of ten elements, and in Lean, you are able not only to program your function, but also to prove that the function really returns 10 artists according to the function definition.

You can think of proofs in Lean as functions⁶ returning the specified type, i.e., theorem. Proof checking in Lean is just type checking: you check that the proof is of the correct type. You can have a very strong type system, where you can also have variables, quantifiers, etc. Then your types can specify exactly what you mean, formally, and machine-checkable. This is why they are called theorems and why they can be used for mathematics, because the types are actually much more expressive than what you are used to from other programming languages.

The proof side of Lean stands side by side with the functional programming side of Lean that can be used for normal programming. You use the former to prove that functions typecheck with more verbose types, i.e., theorems, that say more about their behavior, and you use the latter to program normal functions that can have simpler types, like returning a list. The simpler types do not need proofs because the Lean type checker can usually deduce them automatically, the more involved types (theorems) require explicit proofs to typecheck. In other words, you have to provide a proof of your own so that Lean can typecheck the theorem.

⁶They are terms, but that's a detail.

Each major step from our proof example in [the previous section](#) can be mapped onto a Lean “tactic”. Lean tactics are nothing but intelligent constructors for proofs. Some tactics introduce your assumptions, some are doing substitution, some auto-prove equalities, etc. You can have a look at the entire formalized proof in Lean [here](#).⁷ I annotated it with human-readable comments. You do not need to understand each tactic and step, the important message here is that if you understand the gist of the constructive proof, you understand how you can arrive at the conclusion in Lean. This is even simpler nowadays with letting AI agents generate the Lean code.

The Issues With AI-Generated Math

If you put this blog post into, let’s say, Claude Code, you can easily generate the same Lean proof that I showed. Just use the following prompt:

Take this blog post {INSERT URL HERE} and generate the proof in Section 4 as instructed. Do not use the attached Lean source code. Try not to deviate from the proof in the blog post, only where necessary. Do not take any shortcuts: try to stay true to the original proof and make each one of the major steps explicit using named variables, lemmas, etc. Let me know where you deviated from the proposed proof.

If you understand the Lean version of the theorem, you can be pretty sure of the proof if everything checks out and is green. You do not need to understand the proof at all, assuming that the Lean checker does not have bugs (and no fishy axioms were added). We just formalized our proof, coming from the world of mathematics into Lean.

The issue with AI-generated math is that the direction is the opposite: the AI agents came up with the Lean proofs first and then they are translated back into a human language. The new results by OpenAI were obtained by using AI agents for a smart search of the proof “space”. Think of it like this: proofs are just sentences, and Lean gives you a way to check them against your mathematical theorem. If you had enough time, you could simply try random combination of words and check them one by one, until you arrive at the correct proof. In principle, we could (dis)prove Navier-Stokes, assuming a proof exists⁸, with a brute-force search where you generate all finite proofs first of size 1, then 2, then 3, etc. What AI agents can do is that they can search this space much faster and much more cleverly, they do not generate the sentences randomly.

So what? There are multiple consequences of this, to mention a few:

1. **No intuition.** Generated Lean proofs do not need to adhere to any human standards of elegance. They do not necessarily need to tell any story at all. This is why many mathematicians, after generating a Lean proof with AI, use different AIs to reconstruct the story that may or may not be there. Even Tristan Buckmaster

⁷Strictly speaking, this Lean proof imports classical axioms via Mathlib, which uses them freely, but it could be done without them.

⁸We have to keep in mind Gödel’s incompleteness theorems.

in a recent interview called AI generated proofs one of the ugliest proofs he'd ever seen.

2. **Backward translation might be wrong.** Translations back into English are presumably done mostly by AI, and they can contain errors, or do not need to make that much sense. This actually turned out to be true also in the Navier-Stokes problem, where the English translation of the generated proof contains errors. This does not mean the Lean proof is incorrect, as long as the Lean version was independently verified, but it can give a wrong story to humans about what is going on under the hood. Especially if the proofs are so long that you cannot go over them line by line in Lean to get a good sense of them.
3. **Skipping the journey.** In the past, mathematicians working on a problem often created whole new areas of mathematics so that they could solve their problems. New theory was often inspired by work on existing problems. This is the main concern that many prominent mathematicians such as [Terence Tao](#) are talking about these days. Delegating the proofs to an AI simply skips this effort, because the search does not need to involve new theory, AI agents are trying simply anything that works to get the proof through. Since reward hacking is a well-known issue in AI, the AI agents are actually motivated in doing so on a foundational level.
4. **Wrong theorem formalizations.** A Lean formalization of a theorem might be wrong, i.e., not corresponding to what the mathematician intended, especially if it was AI generated.

I do not know what the solution is, this is what the community will need to figure out. However, I do believe that a good sense and taste for new theories in mathematics, and what results are relevant and useful, these will be the most important skills for the working mathematician in the age of AI math. We have to set new standards, and it is possible we will have to leave the current metrics of research success (like the quantity of papers in peer-reviewed journals you publish), which is not a bad thing.

Conclusion

I showed you a glimpse of what constructive mathematics is and how it maps to Lean. Understanding Lean proofs also helps you understand classical mathematical proofs better. There is much less wiggle room, because you can see how each step maps to a formal rule. Dropping LEM can be painful in everyday mathematics, but it can lead to very productive results and areas of mathematics. If you are interested in learning more about constructive mathematics, the greatest intro I have found and read myself is Andrej Bauer's [Five Stages of Accepting Constructive Mathematics](#). I highly recommend that you give it a read.