THE ART OF FORMAL PROOF

ROBERT Y. LEWIS

1. Introduction

In my first courses as a mathematics major I was amazed at how *certain* my professors, my TAs, and some of my classmates were about their proofs—and dismayed, sometimes, at how *uncertain* I was about my own. When phrases like "it follows that" and "clearly" appeared in lecture, there was no doubt that the conclusions did follow, even if I couldn't quite fit the pieces together in real time. When I wrote them on problem sets, my bluffs were sometimes caught by the grader, but just as often they passed by undetected. I was succeeding in my courses but that success felt fraudulent.

With time, the sense that I was bluffing diminished, but never went away entirely. Through repetition and osmosis, the rules of the "proof game" took on implicit form. It wasn't until after I finished my bachelors degree that I learned these rules could be made explicit.

2. Formal logic

Logicians draw a distinction between *traditional* proofs in natural language (the vast majority of proofs in mathematics) and *formal* proofs, those written following explicit symbolic patterns. A formal logical system combines a language for forming mathematical statements with a collection of derivation rules, patterns that allow a logician to establish judgments of the form "sentence φ is derivable from a set of hypotheses Γ if sentences ψ_1 and ψ_2 are derivable from Γ." Critically, these derivation rules are *formal*, in that they depend on the *form* of the involved sentences and not their meaning.

Propositional logic is an example of such a system. Start with an infinite set of symbols $P = \{p, q, r, \ldots\}$; we call these *propositional letters*, and think of them as representing true/false statements, although that already gets us dangerously close to "meaning." A *formula* of propositional logic is either

- a propositional letter $x \in P$;
- the negation $\neg\varphi$ of a formula φ;
- the conjunction ("and") $\varphi \wedge \psi$ of formulas φ and ψ;
- the disjunction ("or") $\varphi \vee \psi$ of formulas φ and ψ; or
- the implication $\varphi \to \psi$ of formulas φ and ψ.

There is a natural tree structure to formulas of this language, in which the root nodes are propositional letters (Figure 1).

Let φ and ψ denote formulas and Γ denote a set of formulas; we use the notation $\Gamma \vdash \varphi$ to denote "φ is derivable from the set of hypotheses Γ." While the full list of derivation rules for propositional logic [1, 13] is too long for this essay, we include a few examples here.

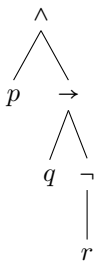

FIGURE 1. The propositional formula $p \wedge (q \to \neg r)$ represented as a tree.

(1) $\Gamma \vdash \varphi \wedge \psi$ if $\Gamma \vdash \varphi$ and $\Gamma \vdash \psi$.
(2) $\Gamma \vdash \varphi \to \psi$ if $\Gamma \cup \{\varphi\} \vdash \psi$.
(3) $\Gamma \vdash \varphi$ if $\varphi \in \Gamma$.

The symbolic representation might obscure the meanings of these rules, but in fact, they correspond to familiar patterns of reasoning.

(1) To prove an "and" statement, it suffices to prove both sides individually.
(2) To prove an implication, it suffices to suppose the left hand side and prove the right hand side.
(3) If you have supposed a statement, you can prove it.

Imagine starting with a complex formula φ and repeatedly applying these derivation rules until no goals remain. This sequence (or really, this tree) of rule applications is a *formal proof* of φ. It may take sweat and tears to find the right sequence of rule applications—after all, these proof rules insist that you check every last detail! But if you present someone with such a sequence, it should be mechanical for them to check that the rules are applied correctly.

The study of *first order* or *predicate* logic adds a number of complications to this picture of propositional logic, but the core ideas are the same. Start with a formal grammar for sentences (including "for all" and "exists" quantifiers); add a long list of derivation rules that echo how we informally reason. *Searching* for proofs of formulas in this logic becomes even harder, but *checking* proofs remains a straightforward mechanical procedure. First order set theory [2] proffers a language and deduction system strong enough to encompass most of what research mathematicians do today.

And here we see a disconnect: mathematicians, as a rule, do not write formal proofs in first order set theory. These proofs are *far* too complicated for humans to create by hand. Many mathematicians will gesture toward the existence of the formal logic as justification for their traditional proofs: with enough time and energy, theoretically, these traditional proofs could be made fully formal. The implicit rules of the proof game are rooted deep down in the formal deduction rules of first order logic. But in practice the two proof styles are far removed. And why bother trying to narrow the gap? The route of repetition and osmosis has taught many a mathematician to distinguish valid proofs from invalid proofs, and the community as a whole is close to perfect at this task (except for some rare and publicized edge cases). If easily checkable correctness is all that formal proof has to offer, it hardly seems worth the prohibitive difficulty.

3. Proof assistants as checkers

Whitehead and Russell's verbose experiments [14] and Gödel's incompleteness theorems made the idea of formal proof in practice solidly unfashionable. But a small stream of mathematicians, and later computer scientists, continued to pursue the dream of mechanically checkable proof.

Computer scientists, in particular, are used to separating high level "input languages" from low level "output languages." A program written in a modern programming language like Python is far removed from the instructions seen by the computer processor. Many layers of abstraction allow a programmer to write humanly-accessible code that can ultimately be translated to the arcane, illegible processor level. There's an analogy here with proof: a formal derivation is like processor instructions; a traditional proof is like a Python program; the unsubstantiatable gesture that a traditional proof corresponds to a formal derivation is like the translation from high level code to low level code. And this code translation is certainly practical, so why not proof translation?

One obvious difference is that the language of traditional proofs is far less controlled than that of a programming language. The gap between proof styles is much wider than that between code styles! This suggests a way to reframe the question: instead of taking traditional math proofs as "input"—written in natural language, with all its ambiguities—we should dream up a language for mathematics that looks a bit more like a programming language. And maybe first order set theory isn't the ideal target language either.

Today, instantiations of this idea are called *proof assistants* or *interactive theorem provers*. A proof assistant (broadly speaking) combines an input language for defining objects, stating theorems, and writing proofs of these theorems with an algorithm that checks whether inputs correspond to valid proofs in some underlying logic. A user takes a traditional proof, say, that $\sqrt{2}$ is irrational, and writes it in the controlled language of the proof assistant; this proof is checked for mistakes, ambiguities, and missing steps, which are reported back to the user. Under the hood, the software is producing a fully formal derivation and making sure that each rule is applied correctly. *Checking* is automatic, but *writing* is not: the ideas and arguments in the proof need to be communicated precisely by the human proof writer.

Two pioneering proof assistants, Automath [4] and Mizar [5], were developed in the 1960s and 1970s. Since then, there have been substantial mathematical developments in Isabelle [11], Coq [7], Metamath [9], and others, although computer scientists were quicker to adopt these tools. The Lean proof assistant [10], a relative newcomer to the field, has attracted a large number of mathematicians and gotten attention in the popular press [6].

Practitioners use the word "formalizing" to refer to the process of writing a proof in the language of a proof assistant. Formalizing a proof can be both frustrating and rewarding: absolutely no bluffing is allowed, and some systems require a level of detail in proofs beyond even the pickiest of human readers, but the satisfaction of convinicing the proof assistant is immense. Making the rules of the proof game fully explicit erases the uncertainty and doubt that students and professionals alike sometimes feel in their proofs.

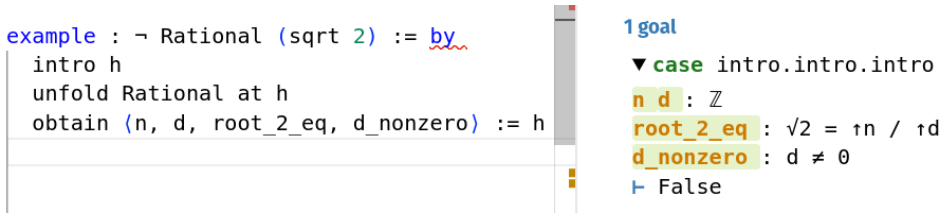

FIGURE 2. As we prove the irrationality of $\sqrt{2}$ in Lean, the system displays our context (above the $\vdash$ symbol) and goal (after the $\vdash$ symbol) on the side.

4. Proof assistants as information managers

Proof assistant input languages are still a far cry from the natural language of traditional proof. It is usually more difficult—sometimes *much* more difficult—to formalize a proof than to write it in prose. Nevertheless, a growing number of mathematicians are using these tools. Unlike derivations in formal logic, proof assistants offer something on top of correctness guarantees: a mathematician formalizes a proof incrementally, and the software reports back on the proof status at each increment. The mathematician can see what variables are in scope, what hypotheses are available, and what goals remain to be shown, and can expand definitions to see the meaning of a statement. Ideally, the proof assistant serves as an information manager helping the mathematician to construct their argument, hence the name "assistant."

Consider again formalizing a proof that $\sqrt{2}$ is not rational. I might start by supposing for the sake of contradiction that it *is* rational; the proof assistant adds a hypothesis `h` to its context asserting this, and changes my goal to showing a contradiction ("`False`"). I can then expand the definition of "rational," changing the hypothesis `h` to assert that there exist integers a and b, $b \neq 0$, such that $\sqrt{2} = \frac{a}{b}$. As I proceed with the proof, I continue to see these changes to the context and goals in real time.

Humans can only store so much information in their heads at once. At the frontiers of mathematics, keeping track of the definitions, goals, and implicit information in a proof can become challenging even for experts. In 2020 the Fields medalist Peter Scholze challenged mathematical formalizers to verify a result of his in condensed mathematics, writing of his traditional proof, "we were able to get an argument pinned down on paper, but I think nobody else has dared to look at the details of this, and so I still have some small lingering doubts." A group led by Johan Commelin completed the challenge a year later by proving Scholze's theorem in Lean, leading Scholze to remark:

> The Lean Proof Assistant was really that: An assistant in navigating through the thick jungle that this proof is. Really, one key problem I had when I was trying to find this proof was that I was essentially unable to keep all the objects in my "RAM", and I think the same problem occurs when trying to read the proof. Lean always gives you a clear formulation of the current goal, and Johan confirmed to me that when he formalized the proof of Theorem 9.4, he could — with the help of Lean — really only see one or two steps ahead,

> formalize those, and then proceed to the next step. So I think here we have witnessed an experiment where the proof assistant has actually assisted in understanding the proof.

This remarkable achievement was not a standalone effort. Commelin and collaborators worked on top of a massive library of definitions and proofs already formalized in Lean. This library, Mathlib [8], contains 1.6 million lines of code written by hundreds of contributors and is growing fast. Its contents range from the definitions of algebraic structures like rings and fields, to complicated number-theoretic objects like Witt vectors [3], to the building blocks of differential topology [12], and much more. Mathlib is a giant interconnected web of mathematical data; the proof assistant ensures that its many pieces fit together properly and that there are no mistakes or gaps in proofs, allowing for collaboration at scale. It's once again an information manager, but at the library level.

5. The art of formalizing

Learning about proof assistants for the first time, some people imagine that relying so heavily on software diminishes the creative human side of mathematics. I disagree: formalization is very much a human act, a dance between the mathematician and the proof assistant, that often teaches the formalizer mathematical lessons. There is an art to finding the perfect formal definition, just the right combination of theorems from the library, the minimal sequence of proof steps to complete a goal. These tools are far from rote proof checkers. Some day they will be indispensible tools in the mathematician's toolbox.

Kevin Buzzard's Natural Number Game[1] lets you experience this dance for yourself, and has been a jumping-off point for countless students and professional mathematicians to learn about formalization. Try it out—you might get hooked.

References

[1] Jeremy Avigad, Robert Y. Lewis, and Floris van Doorn. *Logic and Proof*. Carnegie Mellon University, 2017.

[2] Tim Button. *Set Theory: An Open Introduction*. The Open Logic Project, 2021.

[3] Johan Commelin and Robert Y. Lewis. Formalizing the ring of Witt vectors. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 264–277, New York, NY, USA, 2021. Association for Computing Machinery.

[4] N.G. de Bruijn. The mathematical language Automath, its usage, and some of its extensions. Studies in Logic and the Foundations of Mathematics, pages 29–61. Springer-Verlag, 1970.

[5] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[6] Kevin Hartnett. Building the mathematical library of the future. *Quanta Magazine*, 2020.

[7] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Nov 2020.

[8] The mathlib Community. The Lean mathematical library. In *CPP*, page 367–381, New York, NY, USA, 2020. ACM.

[9] Norman Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, 2019.

[10] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[1] `https://adam.math.hhu.de/#/g/hhu-adam/NNG4`

[12] Floris van Doorn, Patrick Massot, and Oliver Nash. Formalising the h-principle and sphere eversion. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, page 121–134, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Daniel J. Velleman. *How to Prove It: A Structured Approach*. Cambridge University Press, 3 edition, 2019.

[14] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.