Proof Assistants: A Consideration for All

Jake Weber
Louisiana State Univeristy
303 Lockett Hall
Baton Rouge, Louisiana
jwebe33@lsu.edu

ABSTRACT

The QED Manifesto challenged the scientific community to prepare for the mathematical and technological future by creating a system that can provide a foundation for all current and future mathematical results. We take a brief look at the common foundations of proof assistants, their functionalities, and discuss the advantages of Lean, a relatively new programming language and proof assistant.

1. INTRODUCTION

The QED Manifesto [3] outlines a large international mathematical project aimed at building a computer system (the QED system) that can represent all mathematical information and tools. In theory, this system would uphold the rigor of mathematics; all mathematical concepts have a formal representation where their correctness can be verified by a list of simple proof techniques.

The manifesto motivates the project with what will be presented here as four needs. One, the accumulation of mathematical knowledge over the recent centuries has made knowing and understanding all mathematical concepts and ideas nearly impossible for one person. The QED system would be a place for all mathematical knowledge to be stored, a place in which any scientist may scan for what mathematical results are presently known. Today, thousands of mathematics papers are published with no easy way of deciding whether they are completely unique from others' or free from fault. The QED system's library would allow scientists to check for such inconsistencies.

Two, the world is becoming an increasingly more complex place to be, and that world requires more complex mathematical models. This system could provide reliability for the coming complex systems to build off of. Not only does it provide for a solid foundation but it lays ground work for proficient computer program verification systems. Today, formal verification is used to verify many things including control systems, cryptography algorithms, the correctness of compilers and kernels. They are used to find satisfiability conditions (required criteria for a formula to be true), prove theorems or models, and interactively check logic validity.

Three, the QED system can be used as a means to educate all inquiring minds about mathematics. Whereas systems of education throughout the world struggle with student to teacher ratios and quality of instruction, this system would allow students to engage directly with mathematical concepts on their own. Students would be able to learn by doing through the use of interactive proof checking rather than learning from memorization/absorbing what they are told. Not only can students learn, but mathematicians may continue to improve their self-consciousness of what mathematics are necessary for proving a given result. Further, it encourages such professionals to continue to coherently generalize definitions and propositions for making work in mathematics easier.

Four, the QED system may provide an elegant and compelling representation of the natural truths that are explained through mathematics. This preserves, never to be lost regardless of the shortcomings of men, mathematical truths for the future to understand, appreciate, and build from.

With that call to action from 1994, we return our focus to mathematics and its use of computers in the world today. It is more common for computers to be used in mathematical proofs. Often, they are used to carry out simple, yet tedious, computations that surely can be completed by hand but would be more time consuming than anything else. So, that leads the mathematics community to a very important question. Sure, proofs can be verified by other mathematicians, but how can computer computation work be verified? Today, most commonly, it is accepted that if multiple independent implementations of the same algorithm can produce the same result, then the computer's computation is verified. Still, this begs the question of whether any program is actually verified or rather that the input/output relations are verified without concern of the algorithm execution being mathematically well founded. This is where proof assistant computer systems come into the picture.

For example, back in 1977, Appel and Haken [2] came up with a solution to the illusive Four Color Theorem: every loopless planar graph admits a vertex-coloring with at most four different colors. Their approach was met by some skepticism for two reasons: this was not the first time that someone had proposed they had solved the Four Color Theorem, and this proof relied on a computer to manually check that a set of 1478 graphs satisfy a specific set of properties. Robertson, Sanders, Seymour, and Thomas came along in

1997 [7] and built off of Appel and Haken's approach but did so in a way that a computer would only need to manually check a set of 633 graphs satisfying a set of properties. Still, this means a computer was left to do thousands of 'unsupervised' computations, but it nevertheless strengthens the claim that the Four Color Theorem has been solved.

In 2005, however, Georges Gonthier set out to finally write a formal proof of correctness for the Four Color Theorem. This was done using the proof assistant Coq implemented in the functional language ML [5]. Not only does this program prove the correctness of the proof given by Appel and Haken, but it provides insights into why the proof works since writing a formal proof requires every minute step be explicit and logical.

Let us now take a high-level look at a few proof assistants and their characteristics. Then, we will focus in on the Lean project and what it brings to the table as a fairly new proof-checking system.

2. PROOF ASSISTANT BACKGROUND AND DESIGN

2.1 The Core of Proof Assistants

2.1.1 Functional Programming Languages

All of the languages considered here, and many that are employed for proof assistants, are built with the functional programming paradigm; programs are constructed simply by declaring functions, composing them, and executing them. In functional languages, functions are treated as first-class citizens; they may be bound to identifiers, passed as arguments, and returned as a product of other functions, similar to any other data type.

Functional languages find their roots in lambda calculus, a formal system of mathematical logic that expresses computation based on a set of rules/reductions; expressions are subjected to rewriting rules of the lambda calculus. The lambda calculus system is Turing-complete, meaning that it is computationally universal/able to solve any computation problem given enough time, memory, instructions, etc.

This gives us a good intuition as to why functional languages are good choices for proof assistants; they are grounded in a mathematical theory that allows what can be proven, with sufficient resources, to be proven. Also, from a mathematical prospective, the syntax in which functional languages are written greatly mimic the stylistic writing of mathematicians in a natural language.

It is worth noting that many modern proof assistants may be rooted in the lambda calculus systems; however, they operate under a higher order logic (or a higher order type theory), a generalization of the former, in order to have more expressive and computational power.

2.1.2 (Dependent) Type Theory

Simple type theory stems from the association of a type with every expression. For example, the variable n can be defined as the number 1 which may be assigned the type `Nat` (natural number), `Int` (integer), or `Real` (real number). Simple type theory is powerful because it is constructive in that we can create new types out of old ones.

Now that we have introduced the notion of a type, dependent type theory is dependent because types can depend on their parameters. For example, the type `List` α depends on the argument α. For example, our α could be a `Bool` (True/False) or an `Int`, making the list type different based on what the list contains.

Why do we care about using dependent type theory? It is a useful tool when wanting to construct (or use) polymorphic functions. These are functions that accept input data of different data types. The necessity of such a functionality is apparent when considering the addition function denoted by $+$. It has many canonical meanings depending on the object operated on; for instance, we would rather not have different addition functions for integers, real numbers, matrices, functions, etc. When we allow for such functionality, we get really nice flexibility in our expressions as well a clean hierarchy of functions that do not require duplication. The different uses of the $+$ operation notation is called overloading.

2.1.3 Correctness

Nicolass Govert de Bruijn, a Dutch mathematician, had thoughts about the efficacy of theorem provers and when we should be able to trust their logical computations. Even if the logic is consistent, we must confirm that it is implemented correctly. The de Bruijn Criterion is one approach to addressing this problem. The de Bruijn Criterion is satisfied when the generation of the proof (the part of the assistant that constructs a proof) and the checking of the proof (the part of the assistant that checks the validity of the proof) are independent. Further, such a checker should be small so that anyone may verify its logic and thus verify the validity of such a proof.

Most modern proof assistants employ this approach. In fact, just to name a couple, the Coq kernel has about 30,000 lines of code, and the Lean kernel has about a 2,000 lines of code.

2.1.4 At a Glance: Properties of Proof Assistants

Let us now take a quick look at a few proof assistants and how they have implemented some of the functionality discussed in Section 2. The proof assistants below, in descending order, are written in the following languages respectively: ML, ML, Lisp, Mathematica, Pascal, ML, and Lean.

	higher order logic	first order set theory	higher order type theory	classical logic	constructive logic	untyped	decidable non-dep. types	decidable dependent types	de Bruijn criterion	proof checking	goal trans. through tactics	automated theorem proving
HOL	X			X			X		X		X	
Isabelle	X			X			X		X		X	
IMPS	X			X			X				X	
Theorema	X			X		X						X
Mizar		X		X				X		X		
Coq			X		X			X	X		X	
Lean			X		X			X	X		X	

Table 1: Wiedijk [8] breaks down how assistants (except for Lean) stack up.

2.2 Lean: The Programming Language

2.2.1 Lean Proof Assistant and `mathlib`

Lean was launched by Leonardo de Moura at Microsoft Research Redmond in 2013.

It most closely resembles the theorem prover Coq that has been already discussed; however, Lean has a few nuanced differences. Some differences come in how structures are instantiated/how they connect to other object classes. They both have libraries based on dependent type theory; this allows for some automation and convenient type inferences.

If Lean's library were based on set theory, it would have good mathematical backing but not allow for convenient automation. If it were based on simple type theory, it would be fantastic at working with structures but struggle with types that rely on parameters.

Lean's `mathlib` library is an open-source community project; it sees contributions from all levels of mathematicians, scientists, and researchers alike. Its community uses Zulip Chat to collaborate and foster progress for both beginners and masters while hosting Lean 4, the current programming language and theorem prover version, on GitHub. Lean has also seen some large verification projects in its short time [1]: The cap set problem, The continuum hypothesis, Perfectoid spaces, etc. As of 2019, `mathlib` had over 140k lines of code [1], and as of 2023, Lean has transitioned from Lean 3 to Lean 4.

Geavers [4] lays out two roles for proofs: a proof *convinces* a reader that the statement is correct and *explains* why the statement is correct. In addition, he asserts proofs generally have three stages: finding, recording, and presenting. Lean is built with all three stages in mind. Using Lean as a proof assistant addresses both roles of a proof. First, formalization is a way to prove the statement is correct, and its inference abilities help to explain why by providing the simplest of steps that explain how the proof object has changed.

The authors of *Mathematics in Lean* [6] hope for Lean to work not only as an automated reasoning system that is capable of finding proofs (no matter how small they may be) but as an interactive theorem prover that verifies proofs from an axiomatic lens.

2.2.2 Lean Specific Type Classes

Lean builds upon simple type theory by allowing types like `Nat` and `Bool` to be first-class citizens, meaning they are objects. This means they also must have a type associated with them. Since every expression has a type, `Type` itself has type `Type 1` which can be thought of as a set of ordinary types. From there, Lean builds the type of `Type 1`, called `Type 2`, such that `Type 1` is an element of `Type 2`, and it is a larger universe of types. This then continues indefinitely.

In a complex world of types, there is so much information that Lean is able to infer and type check throughout the interactive proving process. Necessary for flexible interpretation of objects of similar types, Lean's type class inference allows for coercions (for example operating on an `Int` object rather than a `Real` object or a `Walk` object rather than a `Path` object) and overloading. This allows for the user to more naturally type as they would in a natural language rather than always having to assert which type of an object or operation is being used.

2.2.3 Metaprogramming and Machine Learning

One power that Lean has over some of its competitors is that Lean itself is a programming language in addition to being a proof assistant. This sets the framework for Lean to write programs on itself. This is called metaprogramming. For example, the use of macros to replace syntax is an example of a small metaprogram; replacement syntax is generated by the program itself. Lean uses metaprogramming and elaborators to fill in implicit arguments and give power to tactics that update the internal goal/proof state in an interactive way.

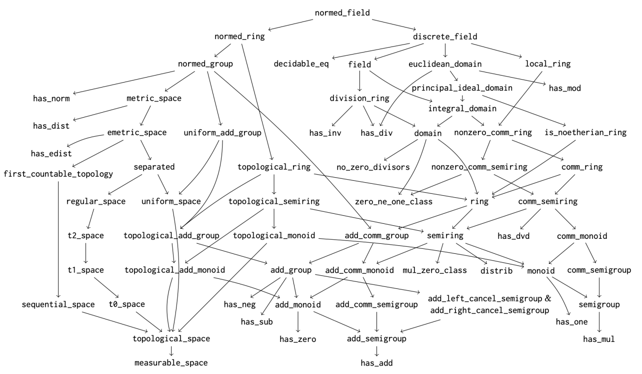

Figure 1: An example of a hierarchical structure that organizes different type classes. In Lean, there are many type classes that can be derived from a `normed_field` instance [1].

Lean also has a machine learning model used in the background when using the `library_search` help command. This puts Lean to work to find the possible results we hope to use on our current proof state in order to transition to our goal state. Lean also has commands to quickly generate auxiliary definitions based on others that have been written previously, saving time in the formalization process.

2.2.4 Tactics

In Lean, tactics are used to modify the current proof state in order to interactively obtain the goal state. Some tactics are used for introducing and eliminating logical connectives. Other tactics help automate the proof state transformations; there are tactics like `simp`, `dsimp`, `linarith`, `ring`, `norm_num`, etc. that use a prescribed list of axioms to execute tedious calculations. For example, this is helpful when we need to apply the properties of a ring in order to show the left hand side of an equation equals the right hand side. This can all be done with the `ring` tactic.

3. CONCLUSION

Proof assistants can build a foundation for a future world of mathematics and complex models. In such a short amount of time, Lean has proven to be a powerful tool. With careful design implementations, we may continue to incorporate machine learning/automation to tackle more difficult problems. With full participation from a community of enthusiastic mathematicians; scientists; and researchers, we can lay the ground work for verification of abstract programs and mystifying mathematical statements and can provide interactive education tools for the future.

4. ACKNOWLEDGMENTS

Thank you to the Simons Laufer Mathematical Sciences Institute (SLMath) for the opportunity and support in attending the 2023 Summer Graduate School in Formalization of Mathematics. Thank you to the organizers and assistants: Jeremy Avigad, Heather Macbeth, Patrick Massot, Kyle Miller, and Thomas Browning.

5. REFERENCES

[1] The lean mathematical library. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 367 – 381, 2020.

[2] K. Appel and W. Haken. Solution of the four-color-map problem. *Scientific American*, 237:108, 1977.

[3] R. B. et al. The qed manifesto. In *Lecture Notes in Artificial Intelligence*, pages 238–251. Automated Deduction - CADE 12, 1994.

[4] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana: Academy Proceedings in Engineering Sciences*, 34(1):3 – 25, 2009.

[5] G. Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382, 2008.

[6] S. K. Jeremy Avigad, Leonardo de Moura and w. c. f. t. L. C. Sebastian Ullrich. *Theorem Proving in Lean.* 2023.

[7] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. The four-colour theorem. *JOURNAL OF COMBINATORIAL THEORY SERIES B*, 70(1):2 – 44, 1997.

[8] F. Wiedijk. *Comparing Mathematical Provers.*, volume 2594 of *Lecture Notes in Computer Science. 2594.* Springer Berlin Heidelberg, 2003.