A Critique Of The Mathematical Proof: Formal Verification Of Partial Differential Equations Through The Lean Theorem Prover

Charlie Cruz
Rice University
Email: seniormars@rice.edu

***Abstract*—Rigor, correctness, and clarity are fundamental goals for mathematical theorem proving. Formal verification is a natural progression of theorem solving designed to address the limitations of traditional proof techniques and mitigate the potential for human error. This paper investigates Formal Verification as a discipline, tracing its historical development, examining the Lean Theorem Prover, and applying it to the context of Partial Differential Equations. Specifically, we formalize bump functions within the Lean framework and analyze the broader implications of Formal Verification for the practice of mathematics. The paper concludes by discussing the prospective impact of Formal Verification on the future of Mathematics.**

***Keywords*—Lean Theorem Prover, Formal Verification, Partial Differential Equations, History of Mathematics, Philosophy of Mathematics**

I. Introduction

Proofs are at the heart of mathematics—a discipline that concerns itself with the precise and rigorous development of ideas. Unfortunately, while math strives for correctness through deduction, how we construct proofs contradicts the idea that math tries to be. Succinctly, the fallacy of proofs is illustrated through the difference in how proofs are constructed in theory and practice. In theory, proofs are a series of logical statements that follow from other logical claims—a definition posed by Aristotle centuries ago—that seek to convince a reader of a claim [1]. In practice, however, proofs are written in natural language, often imprecise and ambiguous. The discrepancy between theory and practice is a fundamental problem in mathematics, especially when proofs are how mathematicians establish the validity of their claims and communicate their results to others.

In principle, the academic world has prevented this fallacy through peer review, a process that many have criticized [2] due to its various shortcomings. While peer review "adds a bit of certainty" [3] to mathematical work, it is not an end–all solution to the problem of human error. A prominent example of this flaw can be highlighted through a dynamical system theorem published in 1970 by Noel Baker. Once published, the theorem was subsequently used to prove various results in dynamical systems; in turn, these results were generalized and furthered–establishing the influence of Baker's theorem in the field. However, half a century later, a pair of mathematicians discovered a flaw in Baker's proof, a flaw so grave that it invalidated the theorem—returning its status as an open problem [4]. This example illustrates the potential for human error in mathematical proofs and the consequences of such errors. Moreover, since the 20th century, this issue has been exacerbated by the increasing complexity of mathematical proofs. Mathematics is now at a stage where several branches are so complex that they are beyond the comprehension of a single mathematician; a new proof is built upon the work of many mathematicians, each trusting the work of the other. A new approach to the mathematical proof is needed; fortunately, a growing field seeks to address these problems at once: Formal Verification.

Formal Verification emerges as a promising solution to the fallacy of mathematical proofs. It is a framework that extends the original purpose of proofs, focusing on verifying the correctness of a system or model through the use of mathematical techniques. For our purposes, formal Verification treats proofs as mathematical objects and checks their correctness through a program, offering a metamathematical approach to mathematics. To fully grasp the implications of Formal Verification, we delve into its historical development, the often misunderstood relationship between computer science and mathematics, and the pivotal role of Type Theory in Formal Verification. We then introduce the Lean Theorem Prover, a programming language that verifies the correctness of mathematical proofs. Finally, we apply the Lean Theorem Prover to the context of Partial Differential Equations, providing insights into the future of mathematics.

Paper was written for MATH 423: Partial Differential Equations

II. Preliminaries

The notion of studying proofs as mathematical objects was popularized through the concept of "Gödel Numbering" in the early 1930s. The technique provided a method to transform an object in any formal language, i.e., proofs in formal logic, to a unique natural number through a bijective function [5]. While the idea of formal methods can be attributed to Gottfried Wilhelm Leibniz—as Leibniz wished for a universal, correct, and logical language[6]—Gödel's incompleteness theorems were the true beginning of the field. These theorems explored the limitations of formal systems with notions of "numbers"; as many branches of mathematics relied on ZFC-Set Theory, Gödel's impact was profound. In short (and imprecisely), Gödel's theorems showed that any formal system powerful enough to express arithmetic was either inconsistent or incomplete. These results could be interpreted as the first major hurdle to mathematical formalization, a problem that has yet to be fully resolved. In fact, David Hilbert, a prominent figure in many fields, sought to formalize all mathematics through his project, the Foundations of Mathematics, which failed as Gödel's work showed that Hilbert's approach was doomed to fail. Gödel's work was so influential that it led to the creation of the field of Proof Theory, the study of proofs [7]. Ultimately, the first major hurdle to mathematics only emphasized the need for a new approach to proofs.

A. Where Computer Science and Mathematics Meet

For the longest time, a common misconception persists that formal verification has only been a staple in computer science—that its application to pure mathematics is a recent development. Although, the reason for this misconception is largely unknown to philosophers—this view, however, overlooks the intricate relationship between computer science and mathematics, as well as the historical trajectory of formal verification itself.

Formal verification emerged as a distinct field within computer science in the 1970s, with the primary objective of verifying software correctness [8]. This pursuit was driven by the recognition that software bugs could lead to catastrophic consequences. A stark illustration of this potential for disaster is the Therac-25 incident, where a radiation therapy machine malfunctioned due to software errors, resulting in severe radiation overdoses to patients [9]. Such incidents underscored the critical need for rigorous verification methods in software development. The importance of formal verification in computer science is further exemplified by NASA's stringent requirements. The space agency mandates that compilers used in mission-critical software development be formally verified, although this requirement does not extend to the executable code itself as it should be [10]. This practice highlights the recognition of formal verification as a crucial tool for ensuring the reliability and safety of complex systems. Clearly, the techniques of Formal Verification have been a cornerstone of computer science for decades. However, to fully grasp the significance of its application to mathematics, we rigorously explore the intricate relationship between programs and proofs—addressing the misconception that the two are distinct entities. We begin by introducing the basics of Constructive Mathematics, Type Theory, and the Curry-Howard correspondence, which form the foundation of Formal Verification.

B. Constructive Mathematics

Constructive mathematics, a branch of mathematical philosophy, challenges the traditional notions of existence and proof in mathematics. In classical mathematics, a statement is considered true if its negation leads to a contradiction, a principle known as the law of excluded middle. Constructive mathematics, however, rejects this principle [11]. Instead, it requires that to prove the existence of a mathematical object, one must explicitly construct the object. Similarly, to prove that a statement is true, one must provide a constructive method for verifying the statement.

The constructive approach to mathematics was popularized by L.E.J. Brouwer in the early 20th century as a critique of classical mathematics, particularly its reliance on non-constructive methods [11]. In constructive mathematics, proofs are not merely theoretical arguments but are tied to the idea of computation. A proof of existence must come with a method to find or approximate the object in question, and this method must be executable in a finite number of steps. To illustrate this concept rigorously, consider the following proofs of the Intermediate Value Theorem:

Theorem: Let $f : [a, b] \to \mathbb{R}$ be a continuous function such that $f(a)$ and $f(b)$ have opposite signs (i.e., $f(a) \cdot f(b) < 0$). Then, there exists a point $c \in [a, b]$ such that $f(c) = 0$.

In classical mathematics, the Intermediate Value Theorem is typically proven by contradiction: we assume that there is no such point c and then derive a contradiction from the continuity of f and the assumption that $f(a)$ and $f(b)$ have opposite signs. However, this proof is non-constructive because it does not provide a method to actually find the point c.

Constructive Proof: We first define the interval $I_0 = [a, b]$ and let $f(a) < 0$ and $f(b) > 0$. The algorithm

proceeds by recursively bisecting the interval. At each step n, we let $I_n = [a_n, b_n]$ be the current interval, and $m_n = \frac{a_n+b_n}{2}$ be the midpoint of the interval. We evaluate the function at the midpoint $f(m_n)$ and consider three cases: if $f(m_n) = 0$, then m_n is the root; if $f(m_n) > 0$, we set $b_{n+1} = m_n$ and $a_{n+1} = a_n$; if $f(m_n) < 0$, we set $a_{n+1} = m_n$ and $b_{n+1} = b_n$. By recursively bisecting the interval, we generate a sequence of nested intervals $I_0 \supset I_1 \supset I_2 \supset \ldots$ that contain the root c. Since f is continuous, the length of the intervals $|b_n - a_n|$ decreases to zero as n increases. By the construction of the algorithm, the sequence (m_n) converges to a limit point c, and by continuity, $f(c) = 0$.

This constructive proof demonstrates the existence of c, and provides a method to approximate c to any desired degree of accuracy by carrying out the bisection algorithm to a sufficient number of steps. Particularly, we make no assumptions about the law of excluded middle, which is central to classical mathematics but not required in constructive mathematics.

C. Type Theory

Type theory is a mathematical framework that serves as the foundation for many modern formal systems; it has a rich history that dates back to the early 20th century. The origins of type theory can be traced to Bertrand Russells work on the foundations of mathematics. In the early 1900s, Russell identified a paradox within the set theory of his time, known as Russell's Paradox. To resolve this paradox, Russell proposed a system of types in which objects could be classified into different levels or "types," thereby preventing self-referential definitions that led to contradictions [12].

At its core, type theory is a formal system that classifies expressions into various types, ensuring that operations and functions are applied correctly. In traditional logic, a proposition is either true or false. In type theory, however, a proposition corresponds to a type, and a proof of the proposition is an element of that type. Type theory has close ties to constructive mathematics—both emphasize construction. To illustrate the principles of type theory, let's consider a simple example: If $A \to B$ and $B \to \bot$ (the negation of B) hold, then $\neg A$ must also hold:

$$\dfrac{f : A \to B \qquad \dfrac{g : \neg B = B \to \bot \qquad \dfrac{\dfrac{\dfrac{x : A}{f(x) : B}}{g(f(x)) : \bot}}{h : A \to \bot}}{h \in \neg A}}{f \to g \to \neg A}$$

Let's break down the proof step by step to understand the logical reasoning involved:

1) Assumption $f : A \to B$: We start by assuming that f is a function that takes an input of type A and produces an output of type B. This corresponds to the premise that $A \to B$.
2) Assumption $g : \neg B = B \to \bot$: Next, we assume g, which states that $\neg B$ is equivalent to $B \to \bot$. In type theory, $\neg B$ (the negation of B) is interpreted as a function from B to $\bot$ (the type representing
3) Assumption $x : A$: We introduce a new assumption x of type A, which means we are assuming that A holds. This will help us derive a contradiction later on.
4) Deriving $f(x) : B$: Applying the function f to x, we obtain $f(x)$, which has type B. This means that under the assumption $x : A$, we can conclude B by applying f.
5) Deriving $g(f(x)) : \bot$: Next, we apply g to $f(x)$, which gives us $g(f(x)) : \bot$. This step uses the assumption g to derive a contradiction $\bot$ from the fact that $f(x)$ is of type B. The function g essentially tells us that if B holds, then a contradiction occurs.
6) Deriving $h : A \to \bot$: From the contradiction $g(f(x)) : \bot$, we can construct a function $h : A \to \bot$. This function h takes an assumption of type A and derives a contradiction. In other words, h is a proof that A implies falsity, which is the definition of $\neg A$ (the negation of A).
7) Deriving $h \in \neg A$: The function h is now seen as an inhabitant of the type $\neg A$. This means that we have shown $\neg A$ (the negation of A) under the assumptions $f : A \to B$ and $g : \neg B$.
8) Conclusion $f \to g \to \neg A$: Finally, the last step concludes the proof by showing that f (which implies $A \to B$) and g (which is equivalent to $\neg B$) together imply $\neg A$. This shows that if A implies B and B leads to a contradiction, then A must be false.

As a remark: Type theory often uses tree-style proofs, which are visual representations of logical reasoning. Each node in the tree represents a logical statement or type, and the branches show how conclusions are derived from assumptions through logical rules or function applications. To read a tree-style proof, start at the leaves (the assumptions) and follow the branches to the root (the conclusion). This method clearly shows the logical steps and dependencies, helping to ensure that each part of the proof is valid. Tree-style proofs effectively demonstrate how propositions are proven in type theory by constructing terms that inhabit types.

D. Computer Science and Mathematics

The convergence of computer science and mathematics finds one of its most profound expressions in the concept of lambda calculus. Introduced by Alonzo Church in the 1930s, lambda calculus provides a formal system for expressing computation based on function abstraction and application [13]. Initially conceived as a foundation for mathematics, lambda calculus became a cornerstone of theoretical computer science, influencing the development of programming languages and computational models. Its ability to represent all computable functions made it a powerful tool for studying the nature of computation itself. Building upon this foundation, Church later developed the simply typed lambda calculus, which introduced the concept of types to lambda calculus [14]. This extension addressed some of the paradoxes present in the untyped lambda calculus and provided a framework for ensuring certain properties of functions. In the typed lambda calculus, every term is assigned a type, and only well-typed terms are considered meaningful.

However, one of the most remarkable corollaries arising from this intersection is the Curry-Howard correspondence. The isomorphism, independently discovered by Haskell Curry and William Howard, establishes a deep correspondence between mathematical proofs and computer programs [15]. This isomorphism reveals that the relationship between a proposition and its proof in constructive logic is analogous to the relationship between a type specification and a program in typed lambda calculus. The correspondence can be illustrated through the following table:

TABLE I
Curry-Howard Correspondence

Constructive Logic	**Typed Lambda Calculus**
Proposition	Type
Proof of a Proposition	Term (Program)
Implication	Function Type
Conjunction (A & B)	Product Type (A $\times$B)
Disjunction (A or B)	Sum Type (A + B)
Negation (not A)	Function Type (A $\rightarrow$False)
Falsehood (False)	Empty Type
Universal Quantifier ($\forall$x. P(x))	Dependent Function Type
Existential Quantifier ($\exists$x. P(x))	Dependent Pair Type

This table illustrates the fundamental correspondences between logical constructs and their type-theoretic counterparts. Each row represents a pair of concepts that are isomorphic under the Curry-Howard correspondence, demonstrating the deep connection between logic and computation that this isomorphism reveals.

Given this isomorphism, the decades of research in computer science devoted to understanding and optimizing programs can be directly applied to the study of proofs. In this light, proofs and programs are not just similar; they are fundamentally the same thing. This insight has profound implications for both fields. It allows us to leverage the vast body of knowledge in computer science to study proofs in a new light and to approach programming with the rigor and formality traditionally reserved for mathematical proofs. This equivalence also paves the way for the use of type theory in Formal Verification.

III. Lean Theorem Prover

The Lean Theorem Prover, initially developed by Leonardo de Moura at Microsoft Research in 2013, is a formal proof verification system and a programming language designed to facilitate the formalization of mathematical proofs [16]. Lean is grounded in the principles of Type Theory and leverages the Curry-Howard correspondence to treat proofs as executable programs. Unlike traditional proof assistants, which may require significant effort to learn and use, Lean's design focuses on user-friendliness and accessibility, making it an increasingly popular tool in both academic and industrial settings.

At its core, Lean provides a framework for the construction of proofs that are not only rigorous but also verifiable by a computer. The language itself is based on a variant of dependent type theory known as the Calculus of Inductive Constructions (CIC). CIC serves as the underlying formal system in Lean, providing a powerful foundation for defining complex mathematical objects and reasoning about them. CIC integrates both inductive types and dependent types, allowing users to define functions and types that can depend on values, which is essential for formalizing a wide range of mathematical concepts. Moreover, one of Lean's most notable features is its emphasis on constructive mathematics. While it supports classical reasoning through the inclusion of the law of excluded middle as an axiom, Lean encourages users to provide constructive proofs whenever possible. Constructive proofs are particularly valuable in computational contexts, as they often correspond to algorithms that can be directly implemented and executed.

Finally, one of the key features of Lean is its extensive library, known as "mathlib." This community-driven library contains a vast collection of formalized mathematics, including definitions, theorems, and proofs. Mathlib serves as both a resource and a foundation for further developments in formal verification, providing users with ready-to-use tools and examples. The success of mathlib is a testament to the collaborative nature of formal verification and the growing

interest in Lean as a platform for mathematical exploration [17].

A. Lean and Formal Verification

Lean's formal verification process involves encoding a theorem as a type and then constructing a proof of that theorem by providing a term of the corresponding type. This approach is directly aligned with the Curry-Howard correspondence. By constructing a term that inhabits a type, one effectively constructs a proof of the corresponding proposition. The advantages of using Lean for formal verification are manifold. We list some of the key benefits below:

1) High level of automation: Lean provides a high level of automation through its tactic framework, which allows users to build proofs incrementally, leveraging automation to handle routine or repetitive tasks. This feature makes the process of formal verification more efficient and accessible, even for complex theorems [18].
2) Lean's interactive proof mode enables users to develop proofs in a step-by-step manner, with immediate feedback on the correctness of each step. This interactive approach not only enhances the reliability of the verification process but also serves as an educational tool, helping users to understand the underlying logic and structure of their proofs.
3) Lean's role in formal verification extends beyond pure mathematics to practical applications in software verification. By formalizing the correctness of algorithms and software systems, Lean can help prevent errors that could have significant real-world consequences, such as those seen in critical systems like aerospace or medical devices. The capacity to rigorously verify software properties before deployment underscores the importance of Lean in the broader context of Formal Verification [19].

B. Introduction to Theorem Proving in Lean

One of the reasons Lean is so popular is due to it being a functional programming language based on dependent type theory. Moreover, one of its key features is the use of tactics, which are commands that help construct proofs step-by-step. Tactics allow users to guide the proof assistant in finding a formal proof, bridging the gap between human reasoning and machine-checkable formalism. Let's consider a simple example to illustrate how proofs are constructed in Lean using tactics. We'll prove that for any propositions P and Q, if P implies Q and P is true, then Q is true. This is known as modus ponens.

```
theorem modus_ponens (P Q : Prop) (h1 : P  Q) (h2 : P) : Q := by
apply h1
exact h2
```

Let's break down this proof:

1) The **theorem** keyword declares that we're proving a theorem named `modus_ponens`.
2) **(P Q : Prop)** declares P and Q as propositions.
3) **(h1 : P Q)** is the hypothesis that P implies Q.
4) **(h2 : P)** is the hypothesis that P is true.
5) **: Q** specifies that we want to prove Q.
6) **by** begins the tactic mode, allowing us to use tactics to construct the proof.
7) **apply h1** applies the implication P Q to our goal. This changes our goal from proving Q to proving P.
8) **exact h2** proves P using our hypothesis h2.

This example demonstrates two common tactics:

- `apply`: This tactic applies a function or implication to the current goal.
- `exact`: This tactic proves the goal directly when we have a term of exactly the right type.

For a more complex example, let's look at a proof of function extensionality, which states that two functions are equal if they produce the same output for all inputs:

```
theorem funext {f f :  (x : ),  x} (h :  x, f x = f x) : f = f :=
by
show extfunApp (Quotient.mk' f) = extfunApp (Quotient.mk' f)
apply congrArg
apply Quotient.sound
exact h
```

This proof uses more advanced concepts and tactics:

- **show**: This tactic is used to restate the goal, often in a more specific form.
- **congrArg**: This tactic applies the congruence rule for function arguments.
- **Quotient.sound**: This is a theorem about quotient types, used here to relate equality of functions to equality of their quotient representations.

These examples illustrate how Lean combines a powerful type system with an expressive tactic language, allowing for the formalization of complex mathematical proofs in a way that is both rigorous and (with practice) intuitive to mathematicians.

IV. Formal Verification of Partial Differential Equations

Partial Differential Equations (PDEs) are central to many areas of mathematics and physics, modeling phenomena ranging from fluid dynamics to electromagnetism. Given the complexity and sensitivity of PDEs, ensuring the correctness of their solutions is of utmost importance. This is where Formal Verification comes into play, offering a rigorous framework for verifying the correctness of mathematical proofs related to PDEs. One of the challenges in formally verifying PDEs

is the need for a rich mathematical library that includes concepts from real and functional analysis. The Lean mathematical library, mathlib, has been developing these foundations, making it increasingly feasible to work with PDEs in a formal setting.

A. Formal Verification of Bump Functions

Bump functions play a crucial role in the study of PDEs, particularly in the construction of smooth partitions of unity and in the approximation of continuous functions by smooth functions. The formal verification of bump functions is therefore an important step towards the broader goal of verifying PDEs. Let's examine how bump functions are formalized in Lean, first by defining the structure of a bump function:

```
structure ContDiffBump (c : E) where
(rIn rOut : )
rIn_pos : 0 < rIn
rIn_lt_rOut : rIn < rOut
```

This structure defines a smooth bump function centered at a point c in a normed vector space E. The function is equal to 1 in the closed ball of radius `rIn` around c, and its support is contained in the open ball of radius `rOut`. The properties of bump functions are then formalized:

```
theorem one_of_mem_closedBall
(hx : x  closedBall c f.rIn) : f x = 1 := ...
theorem support_eq :
Function.support f = Metric.ball c f.rOut := ...
theorem contDiff :
ContDiff  n f := ...
```

These theorems establish key properties of bump functions:

- The function equals 1 in the closed ball of radius `rIn`.
- The support of the function is exactly the open ball of radius `rOut`.
- The function is infinitely differentiable (smooth).

The formal verification of these properties ensures that the bump functions behave as expected, which is crucial for their application in PDE theory. For instance, in the method of characteristics for first-order PDEs, bump functions can be used to construct local solutions that can be patched together to form global solutions. Furthermore, the formalization includes proofs of differentiability properties:

```
protected theorem _root_.ContDiffWithinAt.contDiffBump
{c g : X  E} {s : Set X} {f :  x, ContDiffBump (c x)} {x : X}
(hc : ContDiffWithinAt  n c s x)
(hr : ContDiffWithinAt  n (fun x => (f x).rIn) s x)
(hR : ContDiffWithinAt  n (fun x => (f x).rOut) s x)
(hg : ContDiffWithinAt  n g s x) :
ContDiffWithinAt  n (fun x => f x (g x)) s x := ...
```

This theorem proves that the composition of a bump function with a differentiable function is differentiable, which is essential for many applications in PDE theory, such as the construction of smooth approximations to solutions. The formal verification of bump functions lays the groundwork for more advanced topics in PDE theory. For example, one could proceed to formalize:

- Smooth partitions of unity, which are crucial in the study of manifolds and in proving existence of solutions to PDEs.
- Approximation theorems, such as the density of smooth functions in various function spaces.
- Regularization techniques for PDEs, where bump functions are used to smooth out discontinuities or singularities in solutions.

By formalizing these fundamental tools, we pave the way for the verification of more complex results in PDE theory, ultimately leading to increased confidence in the mathematical models used across various scientific disciplines.

V. Discussion

While the formalization of bump functions in Lean represents a significant achievement, it highlights the broader challenges associated with the formal verification of Partial Differential Equations. The current state of formalized mathematics in Lean, particularly within the realm of analysis, underscores several limitations that must be addressed before the formal verification of PDEs can be considered complete.

Firstly, it is important to acknowledge that while bump functions are now formalized, PDEs as a whole remain largely uncharted territory in Lean. This is not due to a lack of interest but rather the foundational nature of Lean itself. Derived from type theory, Lean has a strong focus on algebra, category theory, and topology, with these areas being well-supported within the system. However, analysis, which is crucial for the study of PDEs, is notably underdeveloped. For instance, the concept of partial derivatives, a fundamental tool in PDEs, is not standardized in Lean. This presents a significant obstacle for those seeking to formalize results in this area. The root of this issue lies in the preferences of the mathlib community, which tends to prioritize abstract and general frameworks over the concrete tools typically used in analysis. As a result, those working in the formalization of PDEs often find themselves in need of basic constructs that are either unavailable or insufficiently developed.

Moreover, Leans learning curve is another factor to consider. While Lean is incredibly powerful, it requires users to spend a significant amount of time becoming familiar with its syntax, tactics, and the underlying type theory. This can be a barrier for mathematicians who are accustomed to more traditional methods of proof construction and who may be unfamiliar with the more abstract approaches favored by Lean.

Another concern is the lack of formalized content in Lean related to undergraduate topics that are foundational for advanced studies in mathematics. For instance, many fundamental concepts in multivariable calculus, differential equations, and Fourier analysis are either not implemented or only partially developed in Lean. Key topics such as directional derivatives, Jacobian matrices, Taylors theorem, and the Fourier transform are still missing or incomplete. This gap is problematic because it limits the applicability of Lean to a broad range of mathematical problems that are central to both theoretical and applied mathematics [20].

The absence of these topics in Lean presents a significant challenge for the broader adoption of formal verification in mathematics. For mathematicians working in areas where these tools are essential, the current state of Lean may be seen as a deterrent, potentially limiting the impact of formal verification in these fields. Furthermore, without a comprehensive library that covers these foundational topics, the adoption of formal verification in educational settings, particularly at the undergraduate level, remains limited.

Despite these challenges, the Lean community is growing rapidly, and the contributions being made to mathlib are invaluable. The collaborative nature of the Lean project means that its capabilities are constantly expanding, and with time, many of the current limitations may be addressed. The potential of Lean to revolutionize mathematics is enormous. By providing a framework where proofs can be rigorously checked by a computer, Lean offers a level of certainty that traditional methods cannot match. This is particularly important in an era where mathematical results are becoming increasingly complex and difficult to verify by hand.

Looking ahead, it is not far-fetched to predict that in the coming decades, academic journals may begin to require that proofs be verified by a formal system like Lean as a condition for publication. Such a shift would represent a fundamental change in the way mathematical research is conducted, emphasizing the importance of formal verification in ensuring the correctness of mathematical results.

While the formal verification of proofs using systems like Lean presents challenges, particularly in areas such as analysis and PDEs, the benefits it offers are undeniable. As the community continues to grow and develop, the tools available in Lean will become more comprehensive, making it an increasingly valuable resource for mathematicians. The critique of traditional proofs highlighted at the beginning of this paper only underscores the need for such tools. Formal verification represents the future of mathematics, offering a path toward greater rigor, reliability, and ultimately, a deeper understanding of the mathematical universe.

VI. Conclusion

This paper has explored the emerging field of formal verification in mathematics, with a particular focus on its application to Partial Differential Equations through the Lean Theorem Prover. We have traced the historical development of formal methods, examined the theoretical foundations of constructive mathematics and type theory, and investigated the practical implementation of these concepts in Lean. The formalization of bump functions in Lean represents a significant milestone in the journey towards the comprehensive formal verification of PDEs. This achievement demonstrates the potential of formal methods to enhance the rigor and reliability of mathematical proofs, even in complex areas of analysis. However, our exploration has also revealed substantial challenges that lie ahead.

Despite these challenges, the rapid growth of the Lean community and the continuous expansion of mathlib offer reasons for optimism. The collaborative nature of formal verification projects suggests that many of the current limitations may be addressed in the coming years. As the tools become more comprehensive and accessible, we can anticipate a gradual shift in mathematical practice towards increased reliance on formal verification.

Looking to the future, we can envision a mathematics landscape where formal verification plays a central role. The day may not be far off when major mathematical journals require formal verification of proofs as a condition for publication. Such a development would represent a paradigm shift in how mathematical knowledge is created, verified, and disseminated. The ongoing critique of traditional proofs and the advancement of formal verification techniques signal a transformative period in mathematics, one that promises to enhance both the precision and the understanding of this fundamental discipline. However, it is crucial to recognize that formal verification should not be seen as a replacement for traditional mathematical thinking. Rather, it should be viewed as a powerful complementary tool that enhances the reliability and clarity of mathematical results. The creative insights and intuitive leaps that characterize mathematical discovery will always be essential, with formal verification serving to solidify and validate these insights.

Acknowledgements

Thanks to my birds for keeping me company while I wrote this paper. Also, thanks to Professor Christos Mantoulidis for being patient with me.

References

[1] Aristotle, "Prior Analytics (~350 BCE)," in *Ideas That Created the Future: Classic Papers of Computer Science*, H. R. Lewis, Ed. The MIT Press, p. 0. [Online]. Available: https://doi.org/10.7551/mitpress/12274.003.0003

[2] C. Greiffenhagen, "Checking correctness in mathematical peer review," vol. 54, no. 2, pp. 184–209. [Online]. Available: https://doi.org/10.1177/03063127231200274

[3] L. E. Andersen, "On the Nature and Role of Peer Review in Mathematics," vol. 24, no. 3, pp. 177–192.

[4] L. Rempe-Gillen and D. Sixsmith, "On Connected Preimages of Simply-Connected Domains Under Entire Functions," vol. 29, no. 5, pp. 1579–1615. [Online]. Available: https://doi.org/10.1007/s00039-019-00488-2

[5] Gödels Incompleteness Theorems > Gödel Numbering (Stanford Encyclopedia of Philosophy). [Online]. Available: https://plato.stanford.edu/entries/goedel-incompleteness/sup1.html

[6] g.-i. family=BARTOCCI, given=CLAUDIO, R. Betti, A. Guerraggio, and R. Lucchetti, *Mathematical Lives: Protagonists of the Twentieth Century From Hilbert to Wiles*. Springer Science & Business Media.

[7] S. R. Buss, *Handbook of Proof Theory*. Elsevier.

[8] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," vol. 12, no. 1, pp. 23–41. [Online]. Available: https://dl.acm.org/doi/10.1145/321250.321253

[9] Death and Denial: The Failure of the THERAC-25, A Medical Linear Accelerator. [Online]. Available: https://users.csc.calpoly.edu/~jdalbey/SWE/Papers/THERAC25.html

[10] NASA Software Safety Guidebook | Standards. [Online]. Available: https://standards.nasa.gov/standard/NASA/NASA-GB-871913

[11] D. Bridges, E. Palmgren, and H. Ishihara, "Constructive Mathematics," in *The Stanford Encyclopedia of Philosophy*, fall 2022 ed., E. N. Zalta and U. Nodelman, Eds. Metaphysics Research Lab, Stanford University. [Online]. Available: https://plato.stanford.edu/archives/fall2022/entries/mathematics-constructive/

[12] B. Russell, "Mathematical Logic as Based on the Theory of Types," vol. 30, no. 3, pp. 222–262. [Online]. Available: https://www.jstor.org/stable/2369948

[13] (Mon, 01/21/1985 - 12:00) The Calculi of Lambda-Conversion | Princeton University Press. [Online]. Available: https://press.princeton.edu/books/paperback/9780691083940/the-calculi-of-lambda-conversion

[14] Churchs Type Theory (Stanford Encyclopedia of Philosophy). [Online]. Available: https://plato.stanford.edu/entries/type-theory-church/

[15] W. A. Howard, "The Formulae-as-Types Notion of Construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, H. Curry, H. B, S. J. Roger, and P. Jonathan, Eds. Academic Press.

[16] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. Von Raumer, "The Lean Theorem Prover (System Description)," in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds. Springer International Publishing, vol. 9195, pp. 378–388. [Online]. Available: http://link.springer.com/10.1007/978-3-319-21401-6_26

[17] T. mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. Association for Computing Machinery, pp. 367–381. [Online]. Available: https://doi.org/10.1145/3372885.3373824

[18] p.-u. family=Doorn, given=Floris, G. Ebner, and R. Y. Lewis, "Maintaining a Library of Formal Mathematics," in *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 2631, 2020, Proceedings*. Springer-Verlag, pp. 251–267. [Online]. Available: https://doi.org/10.1007/978-3-030-53518-6_16

[19] Y. Hirai, "Defining the Ethereum Virtual Machine for Interactive Theorem Provers," in *Financial Cryptography and Data Security*, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Springer International Publishing, pp. 520–535.

[20] Undergrad math not in mathlib. [Online]. Available: https://leanprover-community.github.io/undergrad_todo.html

[21] J. Bayer, C. Benzmüller, K. Buzzard, M. David, L. Lampert, Y. Matiyasevich, L. Paulsen, D. Schleicher, B. Stock, and E. Zelmanov, "Mathematical Proof Between Generations," vol. 71, no. 01, p. 1. [Online]. Available: https://www.ams.org/notices/202401/rnoti-p79.pdf

[22] M. Rathi. An accessible introduction to type theory and implementing a type-checker. [Online]. Available: https://mukulrathi.com/create-your-own-programming-language/intro-to-type-checking/

[23] P. Bjesse, "What is formal verification?" vol. 35, no. 24, pp. 1–es. [Online]. Available: https://doi.org/10.1145/1113792.1113794