Formal Proof—Theory and Practice

John Harrison

A *formal proof* is a proof written in a precise artificial language that admits only a fixed repertoire of stylized steps. This formal language is usually designed so that there is a purely mechanical process by which the correctness of a proof in the language can be verified. Nowadays, there are numerous computer programs known as *proof assistants* that can check, or even partially construct, formal proofs written in their preferred proof language. These can be considered as practical, computer-based realizations of the traditional systems of formal symbolic logic and set theory proposed as foundations for mathematics.

Why should we wish to create formal proofs? Of course, one may consider it just a harmless and satisfying intellectual activity like solving crosswords or doing Sudoku puzzles and not seek a deeper justification. But we can identify two more substantial reasons:

- To establish or refute a thesis about the nature of mathematics or related questions in philosophy.
- To improve the actual precision, explicitness, and reliability of mathematics.

Philosophical goals played an important role in the development of logic and indeed of computer science too [7]. But we're more interested in the actual use of formalization in mathematics, which we think is not such a radical departure from existing practice as it might appear. In some of its most fertile periods, mathematics has been developed in speculative and imaginative ways lacking obvious logical justification. Yet many great mathematicians like Newton and Euler *were* clearly self-conscious about a lack of rigor in their work [24]. Following the waves of innovation, there have always followed corresponding periods of retrenchment, analyzing foundations and increasingly adopting a strict axiomatic-deductive style, either to resolve apparent problems or just to make the material easier to teach convincingly [11]; the "ϵ-δ" explanation of limits in calculus is a classic example. Complete formalization is a natural further step in this process of evolution towards greater clarity and precision. To be more concrete, our own hopes for formalization are focused on two specific goals:

- Supplementing, or even partly replacing, the process of peer review for mainstream mathematical papers with an objective and mechanizable criterion for the correctness of proofs.
- Extending rigorous proof from pure mathematics to the verification of computer systems (programs, hardware systems, protocols, etc.), a process that presently relies largely on testing.

It is of course debatable whether, in either case, there is a serious problem with the existing status quo and whether formal proofs can really offer a solution if so. But we will argue in this paper that the answer is a resounding *yes* in both cases. Recent decades have seen substantial advances, with proof assistants becoming easier to use and more powerful and getting applied to ever more challenging problems.

A significant early milestone in formalization of mathematics was Jutting's 1970s formalization of Landau's very detailed proof of the complete

John Harrison is principal engineer at Intel Corporation in Hillsboro, Oregon. His email address is `johnh@ichips.intel.com`.

ordered field axioms for the real numbers constructed by Dedekind cuts. Today we can point to formalizations starting from similarly basic foundations that reach nontrivial results in topology, analysis, and number theory such as the Jordan Curve Theorem, Cauchy's integral theorem, and the Prime Number Theorem. Perhaps most spectacularly, Gonthier has completely formalized the proof of the Four-Color Theorem, as described elsewhere in this issue.

Similar progress can be discerned in formal proofs of computer systems. The first proof of compiler correctness by McCarthy and Painter from 1967 was for a compiler from a simple expression language into an invented machine code with four instructions. Recently, Leroy has produced a machine-checked correctness proof for a compiler from a significant fragment of C to a real current microprocessor. In some parts of the computer industry, especially in critical areas such as avionics, formal methods are becoming an increasingly important part of the landscape.

The present author has been responsible for developing the HOL Light theorem prover, with its many special algorithms and decision procedures, and applying it to the formalization of mathematics, pure and applied. In his present role, he has been responsible at Intel for the formal verification of a number of algorithms implementing basic floating-point operations [13]. Work of this kind indicates that formalization of pure mathematics and verification applications are not separate activities, one undertaken for fun and the other for profit, but are intimately connected. For example, in order to prove quite concrete results about floating-point operations, we need nontrivial results from mainstream real analysis and number theory, even before we consider all the special properties of floating-point rounding.

Formal Symbolic Logic

The use of symbolic expressions denoting mathematical objects (numbers, sets, matrices, etc.) is well established. We normally write "$(x+y)(x-y)$" rather than "the product of, on the one hand the sum of the first unknown and the second unknown, and on the other hand the difference of the first and the second unknown". In ancient times such longwinded natural-language renderings were the norm, but over time more and more of mathematics has come to be expressed in symbolic notation. Symbolism is usually shorter, is generally clearer in complicated cases, and avoids some of the clumsiness and ambiguity inherent in natural language. Perhaps most importantly, a well-chosen notation can contribute to making mathematical reasoning itself easier, or even purely *mechanical*. The positional base-n representation of numbers is a good example: problems like finding sums and differences can then be performed using quite simple fixed procedures that require no mathematical insight or understanding and are therefore even amenable to automation in mechanical calculating machines or their modern electronic counterparts.

Symbolic logic extends the use of symbolism, featuring not only expressions called *terms* denoting mathematical *objects*, but also *formulas*, which are corresponding expressions denoting mathematical *propositions*. Just as there are operators like addition or set intersection on mathematical objects, symbolic logic uses *logical connectives* like "and" that can be considered as operators on propositions. The most important have corresponding symbolic forms; for example as we write "$x + y$" to denote the mathematical object "x plus y", we can use "$p \wedge q$" to denote the proposition "p and q". The basic logical connectives were already used by Boole, and modern symbolic logic also features the *universal quantifier* "for all" and the *existential quantifier* "there exists", whose introduction is usually credited independently to Frege, Peano, and Peirce. The following table summarizes one common notation for the logical constants, connectives and quantifiers:

English	Symbolic
false	$\bot$
true	$\top$
not p	$\neg p$
p and q	$p \wedge q$
p or q	$p \vee q$
p implies q	$p \Rightarrow q$
p iff q	$p \Leftrightarrow q$
for all x, p	$\forall x.\ p$
there exists x such that p	$\exists x.\ p$

For example, an assertion of continuity of a function $f : \mathbb{R} \to \mathbb{R}$ at a point x, which we might state in words as

For all $\epsilon > 0$, there exists a $\delta > 0$ such that for all x' with $|x - x'| < \delta$, we also have $|f(x) - f(x')| < \epsilon$

could be written as a logical formula

$$\forall \epsilon.\ \epsilon > 0 \Rightarrow \exists \delta.\ \delta > 0 \wedge \forall x'.\ |x - x'| < \delta \Rightarrow |f(x) - f(x')| < \epsilon$$

The use of logical symbolism is already beneficial for its brevity and clarity when expressing complicated assertions. For example, we can make systematic use of bracketing, e.g., to distinguish between "$p \wedge (q \vee r)$" and "$(p \wedge q) \vee r$", while indicating precedences in English is more awkward. But logical symbolism really comes into its own in concert with *formal* rules of manipulation, i.e., symbolic transformations on formulas that can be applied mechanically without returning to the underlying meanings. For example, one sees at a glance that $x = 2y$ and $x/2 = y$ are equivalent, and applies corresponding manipulations without thinking about *why*. Logical notation creates a new vista of such mechanical transformations, e.g., from $(\exists x. P(x)) \Rightarrow q$ to $\forall x. (P(x) \Rightarrow q)$. Symbolism and formal rules of manipulation:

> […] have invariably been introduced to make things easy. […] by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain. […] Civilization advances by extending the number of important operations which can be performed without thinking about them. [27]

In modern *formal logic*, the emphasis on formal, mechanical manipulation is taken to its natural extreme. We not only make use of logical symbolism, but precisely circumscribe the permissible terms and formulas and define a precise counterpart to the informal notion of *proof* based purely on formal rules. We will see more details later, but first let us see how this idea arose in relation to foundational concerns, and how it may be useful in contemporary mathematics.

The Foundations of Mathematics

Arguably, the defining characteristic of mathematics is that it is a *deductive* discipline. Reasoning proceeds from *axioms* (or *postulates*), which are either accepted as evidently true or merely adopted as hypotheses, and reaches conclusions via chains of incontrovertible logical deductions. This contrasts with the natural sciences, whose theories, while strongly mathematical in nature, tend to become accepted because of empirical evidence. (In fact, it is more characteristic of physics to start from observations and seek, by *induction* or *abduction*, the simple axioms that can explain them.) A special joy of mathematics is that one can proceed from simple and entirely plausible axioms to striking and unobvious theorems, as Hobbes memorably discovered [2]:

> Being in a Gentleman's Library, Euclid's Elements lay open, and 'twas the 47 *El. libri* 1 [Pythagoras's Theorem]. He read the proposition. *By G—*, sayd he (he would now and then sweare an emphaticall Oath by way of emphasis) *this is impossible!* So he reads the Demonstration of it, which referred him back to such a Proposition; which proposition he read. That referred him back to another, which he also read. *Et sic deinceps* [and so on] that at last he was demonstratively convinced of that trueth. This made him in love with Geometry.

This idealized style of mathematical development was already established in Euclid's *Elements of Geometry*. However, its later critical examination raised numerous philosophical difficulties. If mathematics is a purely deductive discipline, what is its relationship with empirical reality? Are the axioms actually true of the real world? Can some axioms be deduced purely logically from others, or are they all independent? Would it make sense to use different axioms that contradict the usual ones? What *are* the incontrovertible logical steps admissible in a mathematical proof, and how are they to be distinguished from the substantial mathematical assumptions that we call axioms?

Foundational questions of this sort have preoccupied philosophers for millennia. Now and again, related worries have reached a broader community, often as a reaction to disquiet at certain mathematical developments, such as irrational numbers, infinitesimal calculus, and non-Euclidean geometry. Relatively recently, foundational concerns were heightened as the theory of infinite sets began to be generalized and pursued for its own sake by Cantor, Dedekind, and others.

It was precisely to clarify basic foundational questions that Frege in 1879 introduced his *Begriffsschrift* ("concept-script" or "ideography"), perhaps the first comprehensive formal system for logic and mathematics. Frege claimed that his formal rules codified acceptable logical inference steps. On that basis, he justified his "logicist" thesis that the basic axioms for numbers and geometry themselves are, properly understood, not extralogical assumptions at all, but are derivable from purely logical principles.

However, it was later observed that right at the heart of Frege's system was a logical inconsistency now known as *Russell's paradox*. Frege's system allowed the construction of (in modern parlance) the "set of all sets that are not members of themselves", $R = \{S \mid S \notin S\}$. This immediately leads to a contradiction because $R \in R$ if and only if $R \notin R$, by definition.

Later systems for the foundations of mathematics restricted the principles of set formation so that they were still able to talk about the sets needed in mathematics without, apparently, allowing such self-contradictory collections. Two somewhat different methods were adopted, and these streams of work have led to the development of modern "type theory" and "set theory" respectively.

- Russell's system, used in the monumental *Principia Mathematica*, shared many characteristics with Frege's formal system, but introduced an explicit notion of *type*, separating mathematical objects of different kinds (natural numbers, sets of natural numbers, etc.) The original system was subsequently refined and simplified, leading to the modern system of *simple type theory* or *higher-order logic* (HOL).

- Zermelo did not adopt a formal logic, but did specify explicit axioms for set construction. For example, Zermelo's axioms imply that whenever there is a set S, there is also a set of all its subsets $\wp(S)$, and that whenever sets S and T exist, so does the union $S \cup T$. With some later additions, this has become the modern foundational system of Zermelo-Fraenkel set theory (ZF, or ZFC when including the Axiom of Choice). It can be recast as a formal system by incorporating suitable rules for formal logic.

Type-based approaches look immediately appealing, because mathematicians generally *do* make type distinctions: between points and lines, or between numbers and sets of numbers, etc. A type discipline is also consonant with the majority of modern computer programming languages, which use types to distinguish different sorts of value, mainly for conceptual clarity and the avoidance of errors, but also because it sometimes reflects implementation differences (e.g., between machine integers and floating-point numbers). On the other hand, some consider the discipline imposed by types too inflexible, just as some programmers do in computer languages. For example, an algebraist might just want to expand a field F to an algebraic extension $F(a)$ without worrying about whether its construction as a subset or quotient of $F[x]$ would have a different type from F.

In fact, the distinction between type theory and set theory is not completely clear-cut. The universe of sets in ZF set theory can be thought of as being built in levels (the Zermelo or von Neumann hierarchy), giving a sort of type distinction, though the levels are cumulative (each level includes the previous one) and continued transfinitely. And the last few decades have seen the development of formal type theories with a wider repertoire of set construction principles. The development of many recent type theories has been inspired by the *Curry-Howard correspondence*, which suggests deep connections between propositions and types and between programs and (constructive) proofs.

Formalization or Social Process?

Much of the work that we have just described was motivated by genuine conceptual worries about the foundations of mathematics: how do we know which sets or other mathematical objects exist, or which axioms are logically self-consistent? For Frege and Russell, formalization was a means to an end, a way of precisely isolating the permissible proofs and making sure that all use of axioms was explicit. *Hilbert's program* caused renewed interest in formal logic, and Brouwer even derided Hilbert's approach to mathematics as *formalism*. But Hilbert too was not really interested in actually formalizing proofs, merely in using the theoretical possibility of doing so to establish results *about* mathematics ("metamathematics").

However, some logical pioneers envisaged a much more thoroughgoing *use* of formal proofs in everyday mathematical practice. Peano, independently of Frege, introduced many of the concepts of modern formal logic, and it is a modified form of Peano's notation that still survives today. Peano was largely motivated by the need to *teach* mathematics to students in a clear and precise way. Together with his colleagues and assistants, Peano published a substantial amount of formalized mathematics: his journal *Rivista di Matematica* was published from 1891 until 1906, and polished versions were collected in various editions of the *Formulaire de Mathématique*.

What of the situation today? The use of set-theoretic language is widespread, and books sometimes describe possible foundations for mathematical structures (e.g., the real numbers as Dedekind cuts or equivalences classes of Cauchy sequences). Quite often, lip service is paid to formal logical foundations:

> …the correctness of a mathematical text is verified by comparing it, more or less explicitly, with the rules of a formalized language. [4]
>
> A mathematical proof is rigorous when it is (or could be) written out in the first-order predicate language $L(\in)$ as a sequence of inferences from the axioms ZFC, each inference made according to one of the stated rules. [19]

Yet mathematicians seldom make set-theoretic axioms explicit in their work, except for those whose results depend on more "exotic" hypotheses. And there is little use of formal proof, or even formal logical notation, in everyday mathematics; Dijkstra has remarked that "as far as the mathematical community is concerned George Boole has lived in vain". Inasmuch as the logical symbols *are* used (and one does glimpse "$\Rightarrow$" and "$\forall$" here and there), they usually play the role of ad hoc abbreviations without an associated battery of manipulative techniques. In fact, the everyday use of logical symbols we see today closely resembles an intermediate "syncopation" stage in the development of existing mathematical notation, where the symbols were essentially used for their abbreviatory role alone [26].

Moreover, the correctness of mainstream mathematical proofs is almost never established by formal means, but rather by informal discussion between mathematicians and peer review of papers. The fallibility of such a "social process" is well-known, with published results sometimes containing unsubtle errors:

> Professor Offord and I recently committed ourselves to an odd mistake (*Annals of Mathematics* (2) 49, 923, 1.5). In formulating a proof a plus sign got omitted, becoming in effect a multiplication sign. The resulting false formula got accepted as a basis for the ensuing fallacious argument. (In defense, the final result was known to be true.) [18]

The inadequacy of traditional peer review is starkly illustrated by the case of the Four-Color Theorem. The first purported proof by Kempe in 1879 was accepted for a decade before it was found to be flawed. It was not until the 1970s that a proof was widely accepted [1], and even that relied on extensive computer checking which could not feasibly be verified by hand. (Gonthier's paper in this issue describes the complete formalization of this theorem and its proof.)

A book [17] written seventy years ago gave 130 pages of errors made by major mathematicians up to 1900. To bring this up to date, we would surely need a much larger volume or even a specialist journal. Mathematics is becoming increasingly specialized, and some papers are read by few if any people other than their authors. Many results are produced by those who are not by training mathematicians, but computer scientists or engineers. Perhaps because most "easy" proofs have long ago been found, many of the most impressive results of recent years are accompanied by huge proofs: for example the proof of the graph minor theorem by Robertson and Seymour was presented in a series of twenty papers covering about 500 pages. Others, such as Wiles's proof of Fermat's Last Theorem, are not only quite large and complex in themselves but rely heavily on a daunting amount of mathematical "machinery". Still others, like the Appel-Haken proof of the Four-Color Theorem and Hales's proof of the Kepler Conjecture, rely extensively on computer checking of cases. It's not clear how to bring them within the traditional process of peer review [16], even supposing one finds the status quo otherwise satisfying.

When considering the correctness of a conventional informal proof, it's a partly subjective question what *is* to be considered an oversight rather than a permissible neglect of degenerate cases, or a gap rather than an exposition taking widely understood background for granted. Proofs depend on their power to persuade individual mathematicians, and there is no objective standard for what is considered acceptable, merely a vague community consensus. There is frequently debate over whether "proofs" from the past can be considered acceptable today. For example, the Fundamental Theorem of Algebra was "proved" by, among others, d'Alembert and, in more than one way, by Gauss. Yet opinion is divided on which, if any, of these proofs should be considered as the first acceptable by present standards. (The result is usually referred to as d'Alembert's theorem in France.) The history of Euler's theorem $V - E + F = 2$, where the letters denote the number of vertices, edges, and faces of a polyhedron, reveals a succession of concerns over whether apparent problems are errors in a "proof" or indicate unstated assumption about the class of polyhedra considered [15].

Since mathematics is supposed to be an exact science and, at least in its modern incarnation, one with a formal foundation, this situation seems thoroughly lamentable. It is hard to resist the conclusion that we should be taking the idea of formal foundations at face value and *actually* formalizing our proofs. Yet is also easy to see why mathematicians have been reluctant to do so. Formal proof is regarded as far too tedious and painstaking. Arguably formalized mathematics may be more error-prone than the usual informal kind, as formal manipulations become more complicated and the underlying intuition begins to get lost. Russell in his autobiography remarks that his intellect "never quite recovered from the strain" of writing *Principia Mathematica*, and as Bourbaki [4] notes:

> If formalized mathematics were as simple as the game of chess, then once our chosen formalized language had been described there would remain only the task of writing out our proofs in this language, [...] But the matter is far from being as simple as that, and no great experience is necessary to perceive that such a project is absolutely unrealizable: the tiniest proof at the beginning of the Theory of Sets would already require several hundreds of signs for its complete formalization. [...] formalized mathematics cannot in practice be written down in full, [...] We shall therefore very quickly abandon formalized mathematics, [...]

However, we believe that the arrival of the computer changes the situation dramatically. While perfect accuracy in formal manipulations is problematic even for trained mathematicians, checking conformance to formal rules is one of the things computers are very good at. There is also the prospect that, besides merely checking the correctness of formal arguments, the computer may be able to help in their construction: the Bourbaki claim that the transition to a completely formal text is routine seems almost an open invitation to give the task to computers. Ideally, perhaps the computer may be able to find nontrivial proofs entirely automatically. We will examine in more detail later to what extent this is true.

Like Nidditch, who complained that "in the whole literature of mathematics there is not a single valid proof in the logical sense," we welcome the prospect of formalizing mathematics. In our view, the traditional social process is an anachronism to be swept away by formalization, just as empiricism replaced a similar "social process" used by the Greeks to decide scientific questions. But we should emphasize that we aren't trying to turn mathematics into drab symbol manipulation. Traditional informal proofs bear the dual burden of compelling belief *and* conveying understanding. These are not always mutually supportive and can be antagonistic, since the former pulls in the direction of low-level details, the latter in the direction of high-level concepts. Yet a result whose proof has been formalized can be presented to others in a high-level conceptual way, taking for granted that because of full formalization there is no reasonable doubt about correctness of the details nor uncertainty about precisely what has been proved and from what assumptions. And in principle, a computer program can offer views of the same proof at different levels of detail to suit the differing needs of readers.

Formal Verification

The woolly community process by which mathematical proofs become accepted seems all the worse when one considers the fact that mathematics is applied in the real world. That bridges do not collapse and aircraft do not fall out of the sky is a direct consequence of mathematical design principles. If the underlying mathematics is in doubt, then how can we trust these engineering artifacts? One may doubt the relevance of foundational concerns to the practice of applied mathematics [8], but everyday errors in mathematical procedures, like getting the sign wrong in an algebraic calculation, can have serious engineering consequences. Even so, such errors probably happen less in practice than other problems such as mechanical defects, the inaccurate modeling of the physical world, or the failure even to perform the appropriate mathematical analysis (e.g., checking for dangerous resonances in bridges or aircraft wings). For example, the failure that Frederick II of Prussia, in a 1778 letter to Voltaire, lays at the door of Euler (and of mathematics generally) was arguably caused instead by his contractors' failure to follow Euler's advice [9]:

> I wanted to have a water jet in my garden: Euler calculated the force of the wheels necessary to raise the water to a reservoir, from where it should fall back through channels, finally spurting out in Sanssouci. My mill was carried out mathematically and could not raise a mouthful of water closer than fifty paces to the reservoir. Vanity of vanities! Vanity of mathematics!

Nowadays, there is serious concern about the correctness of computer systems, given their ubiquity in everyday life, sometimes in safety-critical systems like fly-by-wire aircraft, antilock braking systems, nuclear reactor controllers, and radiation therapy machines. Yet most large computer programs or hardware systems contain "bugs", i.e., design errors that in certain situations can cause the system to behave in unintended ways. The consequences of bugs can be quite dramatic: the recall of some early Intel® Pentium® processors owing to a bug in the floating-point division instruction [25], and the explosion of the Ariane 5 rocket on its maiden voyage as the result of a software bug, were each estimated to have cost around US$500 million. At a more mundane level, many of us who use computers in daily life are depressingly familiar with strange glitches and crashes, even though they usually cause little more than minor annoyance.

The fundamental difficulty of writing correct programs, and delivering them on time, began to be recognized almost as soon as computers became popular. By the 1970s, the general situation was often referred to as the "Software Crisis". Brooks [5], drawing on the experience of managing the design of IBM's new operating system OS/360, recounted how adding more people to foundering projects often just made things worse, drawing a striking analogy with the struggles of prehistoric creatures trapped in a tar pit:

> In the mind's eye one sees dinosaurs, mammoths, and saber-toothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks. Large-system programming has over the past decade been such a tar pit.

Why should this be? Most engineering artifacts are unfailingly reliable: collapsing buildings or exploding cars are exceptionally rare and newsworthy events. Yet in the realm of computers, unreliability sometimes seems to be the norm.

Arguably, the fundamental reason is that computers are discrete and digital. In all but the most chaotic physical systems, there are certain continuity properties meaning that small changes in the system or environment are likely to have minimal consequences; indeed, physical parameters are only ever known approximately. In a discrete system like a computer, by contrast, the state is in general much more precise and well-defined. One might think that this would make computers easier to design and more reliable, because up to a point, one escapes the approximation and estimation that is necessary in most of the physical sciences and engineering. In principle, this should be so. Yet the flipside is that discrete systems are are also much more vulnerable to design errors since the smallest possible change, a single bit, may cause a completely different behavior, such as going the other way in an "if …then …else …".

Much of the appeal of using computers is that a single algorithm is supposed to work across a range of situations. For example, a traditional program that accepts some inputs, performs a computation and produces an output, is supposed to work for *any* set of inputs, or at least for a broad and clearly defined class. The number of possible inputs is often infinite ("any finite string of alphanumeric characters"), or at least immensely large ("any two 64-bit integers"). Testing on some relatively small set of inputs can often be an effective way of finding errors, particularly if the inputs are well-chosen, e.g., to exercise all paths through the program created by conditional statements. Yet we can seldom conclude with certainty, even after extensive and elaborate testing, that there are no remaining errors. In practice, however, while programs are written with clear intellectual ideas behind them, their correctness *is* usually checked by just this kind of testing.

An assertion that a program with k inputs is correct can be considered as a universally quantified proposition $\forall n_1, \dots, n_k.\, P(n_1, \dots, n_k)$: for all k-tuples of inputs $n_1, \dots, n_k$ the program performs its intended function. For any *particular* tuple of inputs $n_1, \dots, n_k$, e.g., $(0, 1, 42)$, we can usually test whether $P(n_1, \dots, n_k)$ holds, i.e., whether the program works correctly on those inputs. Many nontrivial and/or open questions in pure mathematics can be expressed by formulas having the same characteristics. (These are roughly what logicians call Π_1^0 formulas.) For example, Goldbach's conjecture that every even integer > 2 is the sum of two primes can be expressed using quantification over natural numbers in the form $\forall n.\mathrm{even}(n) \wedge n > 2 \Rightarrow \exists p\ q.\mathrm{prime}(p) \wedge \mathrm{prime}(q) \wedge p + q = n$ (we could if we wish express the subsidiary concepts "even" and "prime" using just quantification over natural numbers and basic arithmetic). Once again, for a *specific n*, we know we can decide the body of the quantified formula, because we can restrict our search to $p, q \le n$. Less obviously, the Riemann hypothesis about zeros of the complex ζ-function can also be expressed by a formula quantifying only over the natural numbers and with the same characteristics.

In typical programming practice, as we have noted, correctness claims of the form $\forall n.\, P(n)$ are usually justified by testing on particular values of n. In mathematics, by contrast, numerical evidence of that sort may suggest conjectures, and even be subjectively compelling, yet a result is not considered firmly established until it is rigorously *proved*. There are plenty of cautionary tales to justify this attitude. For example, Fermat conjectured that all integers $2^{2^n} + 1$ were prime because this was the case for all $n = 0, \dots, 4$, yet it turned out later that even $2^{2^5} + 1$ was divisible by 641 and in fact *no* other primes of that form are currently known. Again, it is known by explicit calculations that $\pi(n) \le \mathrm{li}(n)$ holds for $n \le 10^{20}$, where $\pi(n)$ is the number of primes $\le n$ and $\mathrm{li}(n) = \int_0^n du/\log u$, yet it is known that $\pi(n) - \mathrm{li}(n)$ changes sign infinitely often. The idea of *formal verification* is to adopt the same standard of evidence in programming as in mathematics: *prove* the correctness of a program in the manner of any other mathematical theorem, rather than relying on the evidence of particular test situations.

The idea of formal verification once aroused heated controversy [3]. One criticism is that we are ultimately interested in confirming that a physical computing system satisfies real-life requirements. What we produce instead is a mathematical proof connecting abstract mathematical models of each. We can represent this situation by the diagram in Figure 1. Formal verification aims to prove that the mathematical model satisfies the mathematical specification. But one must still be cognizant of the potential gaps at the top and the bottom. How do we know that the running of the actual system conforms to the idealized mathematical model?

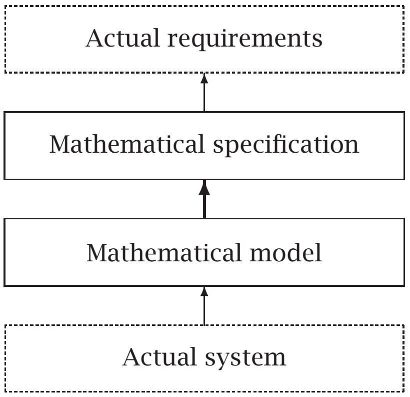

Figure 1. The role of models in formal verification.

And how do we know that our formal specification captures what we really intend?

A thorough discussion of these questions would take us too far afield, though we may note in passing that testing can suffer from analogous concerns. How do we know that the reference results we are testing against are correct? (The infamous FDIV bug was discovered at Intel by using the faulty device as a reference model against which to test the next generation.) And is our testing environment really an accurate model of the eventual deployed system? (This is particularly an issue in integrated circuit design, where the system is usually analyzed in simulation before being committed to silicon.)

We are more interested in another line of criticism, based on the claim that proofs of computer system correctness are often likely to be so long and tedious that humans cannot reasonably check them and discuss them. Even accepting that this claim is true, which some would not, we do not consider this an argument against formal verification. Rather, we think it further emphasizes the inadequacy of the traditional social process of proof and the need for a formal, computer-based replacement. Indeed McCarthy [22], one of the earliest proponents of program verification, emphasized the role of machine checking and generation of proofs. The subsequent evolution of automated reasoning has been closely intertwined with verification applications [20].

Automated Reasoning in Theory

The idea of reducing reasoning to mechanical calculation is an old dream [21]. Hobbes made explicit the analogy between reasoning and computation in his slogan "Reason […] is nothing but Reckoning". This connection was developed more explicitly by Leibniz, who emphasized that a system for reasoning by calculation must contain two essential components:

- A universal language (*characteristica universalis*) in which anything can be expressed
- A calculus of reasoning (*calculus ratiocinator*) for deciding the truth of assertions expressed in the *characteristica*.

Leibniz dreamed of a time when disputants unable to agree would not waste much time in futile argument, but would instead translate their disagreement into the *characteristica* and say to each other "*calculemus*" (let us calculate).

Leibniz was surely right to draw attention to the essential first step of developing an appropriate language. But he was far too ambitious in wanting to express all aspects of human thought. Eventual progress came rather by the gradual extension of the symbolic notations already used in mathematics, culminating in the systems of formal symbolic logic that we have already mentioned. In particular, a specific formal language called *first-order (predicate) logic* (FOL), is widely regarded as a good *characteristica*. Let us briefly sketch how we might define this precisely.

The permissible terms and formulas of FOL can be defined by *grammars* from formal language theory, similar to the BNF (Backus-Naur form) grammars often used to specify the syntax of computer programming languages. For example, given some previously-defined syntactic categories of *variables* and *functions* (more properly, variable names and function symbols), we can define the syntax of first-order terms as follows:

$$\begin{array}{lcl} \text{term} & \longrightarrow & \text{variable} \\ & | & \text{function}(\text{term}, \dots, \text{term}) \end{array}$$

meaning that a term can be constructed from variables by applying functions to other terms as arguments (we consider constants as functions with zero arguments). The class of formulas is then built up using propositional connectives and quantifiers from *atomic formulas* that apply *n*-ary *relation symbol* to *n* terms:

$$\begin{array}{lcl} \text{formula} & \longrightarrow & \bot \\ & | & \top \\ & | & \text{relation}(\text{term}, \dots, \text{term}) \\ & | & \neg\text{formula} \\ & | & \text{formula} \wedge \text{formula} \\ & | & \text{formula} \vee \text{formula} \\ & | & \text{formula} \Rightarrow \text{formula} \\ & | & \text{formula} \Leftrightarrow \text{formula} \\ & | & \forall\text{variable. formula} \\ & | & \exists\text{variable. formula} \end{array}$$

We will consider terms and formulas not as sequential strings, but as tree structures. In this tree-like *abstract syntax* we don't need bracketing to indicate precedences since the construction as a tree contains all this information. When actually writing down formulas, we may prefer a linear *concrete syntax* more like conventional notation. In this case we may again need bracketing to establish precedences, and we may prefer to use conventional infix notation for function and relation symbols, e.g., $x+y < 2$ instead of $<(+(x,y), 2())$. But we always keep in mind that the abstract syntax is what we are really talking about. In practical implementations, transforming from concrete to abstract syntax (*parsing*) and from abstract to concrete (*prettyprinting*) are well-understood tasks because of their role in compilers and other important applications.

Intuitive syntactic concepts like "the variables in a term" can be replaced by precise mathematical definitions, often by recursion on the grammar rules:

$$\begin{aligned} \mathrm{VARS}(v) &= \{v\} \\ \mathrm{VARS}(f(t_1,\ldots,t_n)) &= \bigcup_{i=1}^{n} \mathrm{VARS}(t_i) \end{aligned}$$

We say that a first-order formula ϕ is *logically valid*, and write $\models \phi$, if it *holds in any interpretation*, and *satisfiable* if it *holds in some interpretation*. An interpretation M consists of a nonempty domain D, for each n-ary function symbol f a function $f_M : D^n \to D$ and for each n-ary relation symbol a function $f_M : D^n \to \{\text{false}, \text{true}\}$. For reasons of space, we will not define precisely what it means to hold in an interpretation. But it is important to keep in mind how strong a requirement it is to hold in *all interpretations*. For example, $1 + 1 = 2$ does not hold in all interpretations, because it is perfectly permissible that an interpretation, even if it should happen to have $D = \mathbb{N}$ (which it need not), may choose to interpret the constant symbols "1" and "2" both as the number 7 and the "+" function symbol as multiplication. From abstract algebra we are already familiar with the way in which properties like $x \cdot y = y \cdot x$ may not hold in *all* groups, even if they hold in some familiar examples. For a first-order formula to be valid, it must essentially hold for *any* sensible way of interpreting the functions and relations, with only the logical constants, connectives, and quantifiers (and usually the equality relation) having a fixed meaning.

For first-order logic it is possible to define a notion of provability that is entirely formal in nature, and is *sound* (a provable formula is valid) and *complete* (a valid formula is provable). We write $\vdash \phi$ to indicate that ϕ is provable, so soundness and completeness means that for any formula ϕ we have $\vdash \phi$ if and only if $\models \phi$. There are various ways of defining a suitable notion of provability, but the usual choices are based on a set of formal *inference rules* that allow proof steps of a specific form. (To get started, we need at least one inference rule with no hypotheses, also known as an *axiom*.) For example, a typical inference rule is *modus ponens*, stating that if both p and $p \Rightarrow q$ have been proved, we can add a new step deducing q. The set of provable formulas is then generated inductively by these formal rules, and accordingly we write proof rules in the standard way for inductive definitions:

$$\frac{\vdash p \Rightarrow q \qquad \vdash p}{\vdash q}$$

It is now straightforward to define a corresponding notion of proof, such as a tree reflecting the patterns of inference, or simply a sequence of formulas with an indication of how it was derived from those earlier in the list.

For the following discussion, let us ignore the question of what a formal proof actually consists of, regarding both formulas and proofs as natural numbers and writing $\mathrm{Proves}(m, n)$ for "proof m is a valid proof of proposition n." (This is quite common when describing results of this nature, leaning on the trick of Gödel numbering, expressing a symbolic entity just as a large number. For example, one might express the symbolic expression as an ASCII string and regard the characters as base-256 digits.) Then soundness and completeness of the formal rules means that a proposition n is logically valid if and only if $\exists m.\, \mathrm{Proves}(m, n)$, i.e., if there exists a formal proof of n. From a suitably abstract point of view, the purely *formal* nature of the rules is manifested in the fact that there is a mechanical procedure, or computer program, that given any particular m and n as inputs will decide whether indeed $\mathrm{Proves}(m, n)$.

To return to a philosophical question that we raised early on, some might say that it is merely a question of terminology what we choose to call purely logical reasoning and what we consider as involving mathematical hypotheses with content going beyond pure logic. However, an important characteristic of a *proof* as traditionally understood is that even indifferent mathematicians should be able, with sufficient effort, to *check* that a long and difficult but clearly written proof really *is* a proof, even if they barely understand the subject matter and could not conceive of devising the proof themselves, just as Hobbes did for Pythagoras's theorem. The fact that for formal first-order logic, there is a proof-checking process that can be performed by machine is for many a solid reason for identifying "logical" reasoning with reasoning that is first-order valid.

As part of his foundational program, Hilbert raised further questions about logical reasoning, including the *Entscheidungsproblem* (decision problem) for first-order logic. If the binary "proof checking" relation $\mathrm{Proves}(m, n)$ is mechanically computable, what about the unary relation of provability, $\mathrm{Provable}(n) =_{def} \exists m.\, \mathrm{Proves}(m, n)$? Church and Turing showed that it is in fact uncomputable—a doubly significant step since they first needed to specify what it means to be computable. We can summarize this by saying that although formal proof checking is mechanizable, formal proof *finding* is not, even if we have an idealized digital computer without time or space limitations.

Having said that, proof finding is *semicomputable* (recursively enumerable) because we can systematically try $m = 0, 1, 2, \ldots$ in turn, testing in each case whether $\mathrm{Proves}(m, n)$. If indeed n is logically valid, we will eventually find an m that works and terminate our search with success.

However, if n turns out to be invalid, this process will continue indefinitely, and the Church-Turing result shows that the same must sometimes happen not just for this rather uninspired method, but for *any* other algorithm. Still, we might hope to come up with more intelligent programs that will find proofs of reasonably "simple" logically valid formulas relatively quickly.

We emphasized earlier the important distinction between holding in all interpretations and holding in some particular interpretation. The difficulty of the corresponding decision problems can also be very different. Consider first-order formulas built from constants 0 and 1, functions $+$, $-$ and $\cdot$ and relations $=$, $\leq$ and $<$, or as we say for brevity *arithmetic formulas*. By the results we have just described, whether a formula ϕ holds in all interpretations is *semicomputable* but not *computable* (the restriction to an arithmetic language does not affect either result). By contrast, it follows from famous undecidability results due to Gödel and Tarski that whether an arithmetic formula holds in $\mathbb{N}$ (i.e., the interpretation with domain $\mathbb{N}$ and the functions and relations interpreted in the obvious way) is *not even semicomputable*. This last result lends more support to the identification of *purely logical* with *first-order valid*, since it implies that validity for many natural extensions of first-order logic, e.g., to higher-order logic where quantification is permitted over functions and predicates, cease even to be semicomputable. This does not invalidate higher-order logic as a vehicle for formal mathematics. However it shows that for any sound formal proof system, we must reconcile ourselves to being able to prove only a proper subset of the higher-order valid formulas, or even of those first-order formulas that hold in $\mathbb{N}$.

Automated Reasoning in Practice

There are already well-established classes of computer programs that manipulate symbolic expressions, e.g., programming language compilers and computer algebra systems. The same techniques can be used to perform symbolic manipulations of the terms and formulas of formal logic. Using modern high-level languages, e.g., OCaml or Haskell, these manipulations can be expressed at a high level not far from their mathematical formulations on page 1402. For example, in OCaml we can define the first-order terms as a type of abstract syntax trees almost copying the abstract grammar:

```
type term = Var of string
          | Fn of string * term list;;
```

and express in a direct recursive way the function returning the set of variables in a term:

```
let rec vars tm =
  match tm with
    Var v -> [v]
  | Fn(f,ts) -> unions (map vars ts);;
```

The theoretical results in the last section suggest two contrasting approaches to the practical mechanization of proof. A *proof checker* expects the user to provide both the proposition n and the formal proof m, and simply checks that $\mathrm{Proves}(m, n)$. An *automated theorem prover*, by contrast, takes just the proposition n and attempts to find a suitable proof by itself. We use the broader term *proof assistant* or *interactive theorem prover* to cover the whole spectrum, including these extremes and various intermediate possibilities where the user provides hints or proof sketches to the program to direct the search.

We refer to programs that always terminate with a correct yes/no answer to a decision problem as *decision procedures*. From the Church-Turing result, we know that there is no decision procedure for first-order validity in general, but there are decision procedures for limited or modified forms of the same problem. For example, validity of purely *propositional* formulas (those without functions, variables or quantifiers) is computable, since the only predicates are nullary and therefore an interpretation simply assigns "true" or "false" to each of the relation symbols, and we can systematically try all combinations. This is the dual of the well-known propositional satisfiability problem SAT, and although it is NP-complete, there are tools that are surprisingly effective on many large problems. More generally, logical validity is computable for first-order formulas whose only quantifiers are universal and at the outside, e.g., $\forall x\ y.\ f(f(f(x)) = x \wedge f(f(f(f(f(x))))) = x \wedge f(f(x)) = y \Rightarrow x = y$ (which is valid); such methods have important applications in verification. Whether an arithmetic formula holds in $\mathbb{R}$ is also computable, albeit not very efficiently, and whether an arithmetic formula involving multiplication only by constants (a "Presburger formula") holds in $\mathbb{N}$ or $\mathbb{Z}$ is quite efficiently computable; even further restricted forms of this problem are useful in verification. There are also "combination" techniques for checking validity of formulas in languages including some symbols with a specific interpretation and others where all interpretations are permitted as in pure first-order validity. Modern SMT (satisfiability modulo theories) decision procedures are effective implementations of these methods.

In the early computer experiments in the late 1950s, most of the interest was in purely automated theorem proving. Perhaps the first theorem prover to be implemented on a computer was a decision procedure for Presburger arithmetic [6].

Subsequently, research was dominated by proof search algorithms for pure first-order logic. Decision procedures have proven useful in verification applications, while proof search has achieved some notable successes in mathematics, such as the solution by McCune [23], using the automated theorem prover EQP, of the longstanding "Robbins conjecture" concerning the axiomatization of Boolean algebra, which had resisted human mathematicians for some time. However, for some problems there seems to be no substitute for human involvement, and there was also considerable interest in proof checking at around the same time. The idea of a proof assistant that struck a balance between automation and human guidance appeared with the SAM (semi-automated mathematics) provers. Influential systems from the 1970s such as AUTOMATH, LCF, Mizar and NQTHM introduced most of the ideas that lie behind today's generation of proof assistants.

The purely automatic systems tend to adopt inference rules that are conducive to automated proof search, such as resolution, while the interactive systems adopt those considered more suitable for human beings such as natural deduction. However, in the SAM tradition, the leading interactive systems also tend to include an arsenal of decision procedures and proof search methods that can automate routine subproblems. Whatever the panoply of proof methods included, the system is *some* sort of computer program, and therefore its set of provable theorems is still at least semicomputable. However, since computer programs, especially large and complicated ones, are known to be prone to error, how can we be confident that such a system is sound, i.e., that the set of provable formulas is a subset of the set of valid ones?

Since we have been proposing theorem provers as an improvement on human fallibility and as a way of proving the correctness of other programs, this is a serious question: we seem to be in danger of an infinite regress. However, sound principles of design can provide a fairly satisfying answer. Some systems satisfy the *de Bruijn criterion*: they can output a proof that is checkable by a much simpler program. Others based on the LCF approach [10] generate all theorems internally using a small logical kernel: only this is allowed to create objects of the special type "theorem", just as only the kernel of an operating system is allowed to execute in privileged mode.

There is a fair degree of unanimity on the basic formal foundations adopted by the various proof assistants of today: most are based on either first-order logic plus set theory, or some version of simple type theory, or some constructive type theory. But the systems vary widely in other characteristics such as the level of automation and the style in which proof hints or sketches are provided [28]. One interesting dichotomy is between *procedural* and *declarative* proof styles [12]. Roughly, in a declarative proof one outlines *what* is to be proved, for example a series of intermediate assertions that act as waystations between the assumptions and conclusions. By contrast, a *procedural* proof explicitly states *how* to perform the proofs ("rewrite the second term with lemma 7 …"), and some procedural theorem provers such as those in the LCF tradition use a full programming language to choreograph the proof process.

Conclusions and Future Prospects

The use of formal proofs in mathematics is a natural continuation of existing trends towards greater rigor. Moreover, it may well be the *only* practical way of gaining confidence in proofs that are too long and complex to check in the traditional way (e.g., those in formal verification), or those that already involve ad hoc computer assistance. Hitherto, formalization has attracted little interest in the mathematical community at large because it seems too difficult. There is no escaping the fact that creating formal proofs *is* still difficult and painstaking. However, in barely fifty years, computer proof assistants have reached the stage where formalizing many nontrivial results is quite feasible, as the other papers in this issue illustrate. In our opinion, progress has come mainly through the following:

- The cumulative effects as libraries of formalized mathematics are developed and can be built upon by others without starting from scratch.
- The integration into proof assistants of more automated decision procedures, while maintaining high standards of logical rigor.
- More attention to the languages used to express proofs, and various interface questions that make the systems more convenient to use.

We believe that these trends will continue for some time, and perhaps other avenues for improvement will be more thoroughly explored. For example, computer algebra systems already feature many powerful algorithms for automating mainstream mathematics, which if incorporated in a logically principled way could be very valuable [14]. As proof assistant technology further improves, we can expect it to become increasingly accessible to mathematicians who would like to put the correctness of their proofs beyond reasonable doubt.

References

[1] K. Appel and W. Haken, Every planar map is four colorable, *Bulletin of the American Mathematical Society* **82** (1976), pages 711–712.

[2] J. Aubrey, *Brief Lives*, Clarendon Press, 1898, Edited from the author's MSS by Andrew Clark.

[3] J. Barwise, Mathematical proofs of computer correctness, *Notices of the American Mathematical Society* **7** (1989), pages 844–851.

[4] N. Bourbaki, *Theory of sets*, Elements of mathematics, Addison-Wesley, 1968, Translated from French *Théorie des ensembles* in the series "Eléments de mathématique", originally published by Hermann in 1968.

[5] F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975.

[6] M. Davis, A computer program for Presburger's algorithm, In *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University*, Institute for Defense Analyses, Princeton, NJ, 1957, pages 215–233.

[7] ______, *The Universal Computer: The Road from Leibniz to Turing*, W. W. Norton and Company, 2000; Paperback edition (2001) entitled *Engines of Logic: Mathematicians and the Origin of the Computer.*

[8] C. Diamond, editor, *Wittgenstein's Lectures on the Foundations of Mathematics, Cambridge 1939*. University of Chicago Press, 1989.

[9] M. Eckert, Water-art problems at Sanssouci—Euler's involvement in practical hydrodynamics on the eve of ideal flow theory. *Physica D: Nonlinear Phenomena* **237** (2008), pages 14–17.

[10] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[11] J. V. Grabiner, Is mathematical truth time-dependent?, *New Directions in the Philosophy of Mathematics: An Anthology*, (T. Tymoczko, ed.), Birkhäuser, 1986, pages 201–213.

[12] J. Harrison, Proof style, *Types for Proofs and Programs: International Workshop TYPES'96*, (E. Giménez and C. Paulin-Mohring, eds.), volume 1512 of *Lecture Notes in Computer Science*, Springer-Verlag, Aussois, France, 1996, pages 154–172.

[13] ______, Floating-point verification using theorem proving, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006*, (M. Bernardo and A. Cimatti, eds.), volume 3965 of *Lecture Notes in Computer Science*, Springer-Verlag, Bertinoro, Italy, 2006, pages 211–242.

[14] J. Harrison and L. Théry, A sceptic's approach to combining HOL and Maple, *Journal of Automated Reasoning* **21** (1998), pages 279–294.

[15] I. Lakatos, *Proofs and Refutations: The Logic of Mathematical Discovery*, (John Worrall and Elie Zahar, eds.), Cambridge University Press, 1976. Derived from Lakatos's Cambridge Ph.D. thesis; an earlier version was published in the *British Journal for the Philosophy of Science* vol. 14.

[16] C. W. H. Lam, How reliable is a computer-based proof?, *The Mathematical Intelligencer* **12** (1990), pages 8–12.

[17] M. Lecat, *Erreurs de Mathématiciens des origines à nos jours*, Ancne Libraire Castaigne et Libraire Ém Desbarax, Brussels, 1935.

[18] J. E. Littlewood, *Littlewood's Miscellany*, (Bella Bollobas, ed.), Cambridge University Press, 1986.

[19] S. Mac Lane, *Mathematics: Form and Function*, Springer-Verlag, 1986.

[20] D. MacKenzie, *Mechanizing Proof: Computing, Risk and Trust*, MIT Press, 2001.

[21] W. Marciszewski and R. Murawski, *Mechanization of Reasoning in a Historical Perspective*, volume 43 of *Poznań Studies in the Philosophy of the Sciences and the Humanities*, Rodopi, Amsterdam, 1995.

[22] J. McCarthy, Computer programs for checking mathematical proofs, *Proceedings of the Fifth Symposium in Pure Mathematics of the American Mathematical Society*, pages 219–227, American Mathematical Society, 1961.

[23] W. McCune, Solution of the Robbins problem, *Journal of Automated Reasoning* **19** (1997), pages 263–276.

[24] G. Pólya, *Induction and Analogy in Mathematics*, Princeton University Press, 1954.

[25] V. R. Pratt, Anatomy of the Pentium bug, *Proceedings of the 5th International Joint Conference on the theory and practice of software development (TAPSOFT'95)*, (P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, eds.), volume 915 of *Lecture Notes in Computer Science*, Springer-Verlag, Aarhus, Denmark, 1995, pages 97–107.

[26] P. Taylor, *Practical Foundations of Mathmematics*, volume 59 of *Cambridge Studies in Advanced Mathematics*, Cambridge University Press, 1999.

[27] A. N. Whitehead, *An Introduction to Mathematics*, Williams and Norgate, 1919.

[28] F. Wiedijk, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006.