The Mechanization of Mathematics

Jeremy Avigad

Communicated by Daniel Velleman

Note: The opinions expressed here are not necessarily those of Notices.

ABSTRACT. In computer science, *formal methods* are used to specify, develop, and verify hardware and software systems. Such methods hold great promise for mathematical discovery and verification of mathematics as well.

Introduction

In 1998 Thomas Hales announced a proof of the Kepler conjecture, which states that no nonoverlapping arrangement of equal-sized spheres in space can attain a density greater than that achieved by the naive packing obtained by arranging them in nested hexagonal layers. The result relied on extensive computation to enumerate certain combinatorial configurations known as "tame graphs" and to establish hundreds of nonlinear inequalities.

He submitted the result to the *Annals of Mathematics*, which assigned a team of referees to review it. Hales found the process unsatisfying: it was more than four years before the referees began their work in earnest, and they cautioned that they did not have the resources to review the body of code and vouch for its correctness. In response, he launched an effort to develop a formal proof in which every calculation, and every inference, would be fully checked by a computer. To name the project, Hales searched for a word containing the initial letters of the words "formal," "proof," and "Kepler," and settled on "Flyspeck," which means "to scrutinize, or examine carefully." The project was completed in August of 2014.[1]

In May of 2016, three computer scientists, Marijn Heule, Oliver Kullmann, and Victor Marek, announced a solution to an open problem posed by Ronald Graham. Graham had asked whether it is possible to color the positive integers red and blue in such a way that there are no monochromatic Pythagorean triples, that is, no monochromatic triple a, b, c satisfying $a^2 + b^2 = c^2$. Heule, Kullmann, and Marek determined that it is possible to color the integers from 1 to 7,824 in such a way (see Figure 1), but that there is no coloring of the integers from 1 to 7,825 with this property. They obtained this result by designing, for each n, a propositional formula that describes a coloring of $1, \ldots, n$ with no monochromatic triple. They then used a propositional satisfiability solver, together with heuristics tailored to the particular problem, to search for satisfying assignments for specific values of n.

For $n = 7{,}824$, the search was successful, yielding an explicit coloring of the corresponding range of integers. For the negative result, however, it is riskier to take the software's failure to find a coloring as an ironclad proof that there isn't one. Instead, Heule, Kullmann, and Marek developed an efficient format to encode a proof that the search was indeed exhaustive, providing a certificate that could be checked by independent means. The resulting

Jeremy Avigad is professor of philosophy and mathematical sciences at Carnegie Mellon University. His email address is avigad@cmu.edu.

For permission to reprint this article, please contact: reprint-permission@ams.org.

DOI: http://dx.doi.org/10.1090/noti1688

[1] *Hales provided an engaging account of the refereeing process and the motivation behind the Flyspeck project in a talk presented to the Isaac Newton Institute in the summer of 2017,* www.newton.ac.uk/seminar/20170710100011001.

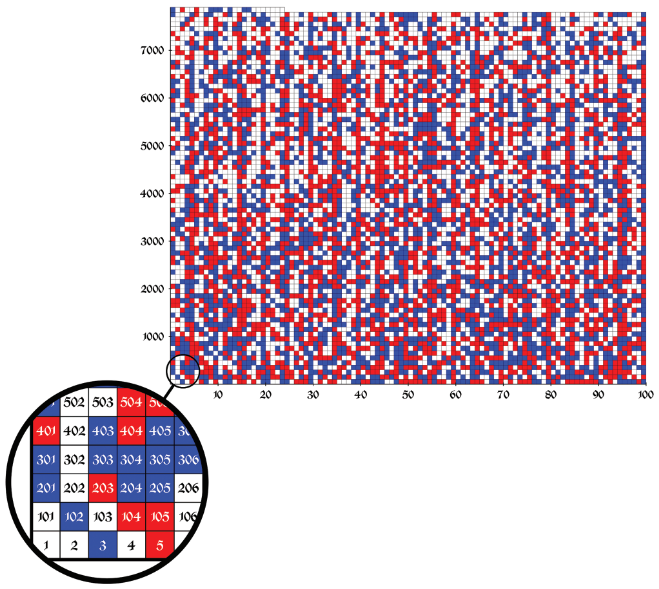


Figure 1. A family of colorings of the integers from 1 to 7,824 with no monochromatic Pythagorean triple. White squares can be colored either red or blue.

proof is 200 terabytes long, leading to popular reports in the international press of the longest proof ever found. They managed to produce a 68-gigabyte certificate with enough information for users to reproduce the proof on their own, and made it publicly available.

The use of computers in mathematics is by no means new. Numerical methods are routinely used to predict the weather, model the economy, and track climate change, as well as to make decisions and optimize outcomes in industry. Computer algebra systems like Mathematica, Maple, and Sage are widely used in applied mathematics and engineering.

By now we have even gotten used to the fact that computers can contribute to results in pure mathematics. The 1976 proof of the four color theorem by Kenneth Appel and Wolfgang Haken used the computer to check that each of a list of 1,936 maps had a required property, and to date there is no proof that can be checked by hand. In 2002, Warwick Tucker used careful calculation to show that the Lorenz attractor exists, that is, that Lorenz's original equations do indeed give rise to chaotic behavior in a precise sense. In doing so, he settled the fourteenth problem on a list of open problems prepared by Stephen Smale at the turn of the twenty-first century. In 2005 Manjul Bhargava and Jonathan Hanke used sophisticated computations to prove a conjecture by John Conway, now called the 290 theorem, which asserts that any positive definite quadratic form with integral coefficients that represents all positive integers up to 290 in fact represents all the positive integers. In 2013 Marc Lackenby and Rob Meyerhoff relied on computer assistance (as well as Perelman's proof of the geometrization conjecture) to provide a sharp bound on exceptional slopes in Thurston's Dehn surgery theorem. Other examples can be found under the Wikipedia entry for "computer-assisted proof," and in a survey by Hales [4].

But the uses of computation in the Flyspeck project and the solution to the Pythagorean triples problem have a different and less familiar character. Hales' 1998 result was a computer-assisted proof in the conventional sense, but the Flyspeck project was dedicated to *verification*, using the computer to check not only the calculations but also the pen-and-paper components of the proof, including all the background theories, down to constructions of the integers and real numbers. In the work on the Pythagorean triples problem, the computer was used to carry out a heuristic search rather than a directed computation. Moreover, in the negative case, the result of the computation was a formal proof that could be used to certify the correctness of the result.

What these two examples have in common is that they are mathematical instances of what computer scientists refer to as *formal methods*: computational methods that rely on formal logic to make mathematical assertions, specify and search for objects of interest, and verify results. In particular, both Flyspeck and the Pythagorean triples result rely crucially on formal representations of mathematical assertions and formal notions of mathematical inference and proof.

The thesis I will put forth in this article is that these two results are not isolated curiosities, but, rather, early signs of a fundamental expansion of our capacities for discovering, verifying, and communicating mathematical knowledge. The goal of this article is to provide some historical context, survey the incipient technologies, and assess their long-term prospects.

Both Flyspeck and the Pythagorean triples result rely crucially on… formal notions of mathematical inference and proof.

The Origins of Mechanized Reasoning

Computer scientists, especially those working in automated reasoning and related fields, find a patron saint in Ramon Llull, a thirteenth-century Franciscan monk from Mallorca. Llull is best known for his *Ars generalis ultima* ("ultimate general art"), a work that presents logical and visual aids designed to support reasoning that could win Muslims over to the Christian faith. For example, Llull listed sixteen of God's attributes—goodness, greatness, wisdom, perfection, eternity, and so on—and assigned a letter to each. He then designed three concentric paper circles, each of which had the corresponding letters inscribed around its border. By rotating the circles, one could form all combinations of the three letters, and thereby appreciate the multiplicity of God's attributes (see Figure 2). Other devices supported reasoning about the faculties and acts of the soul, the virtues and the vices, and so on.

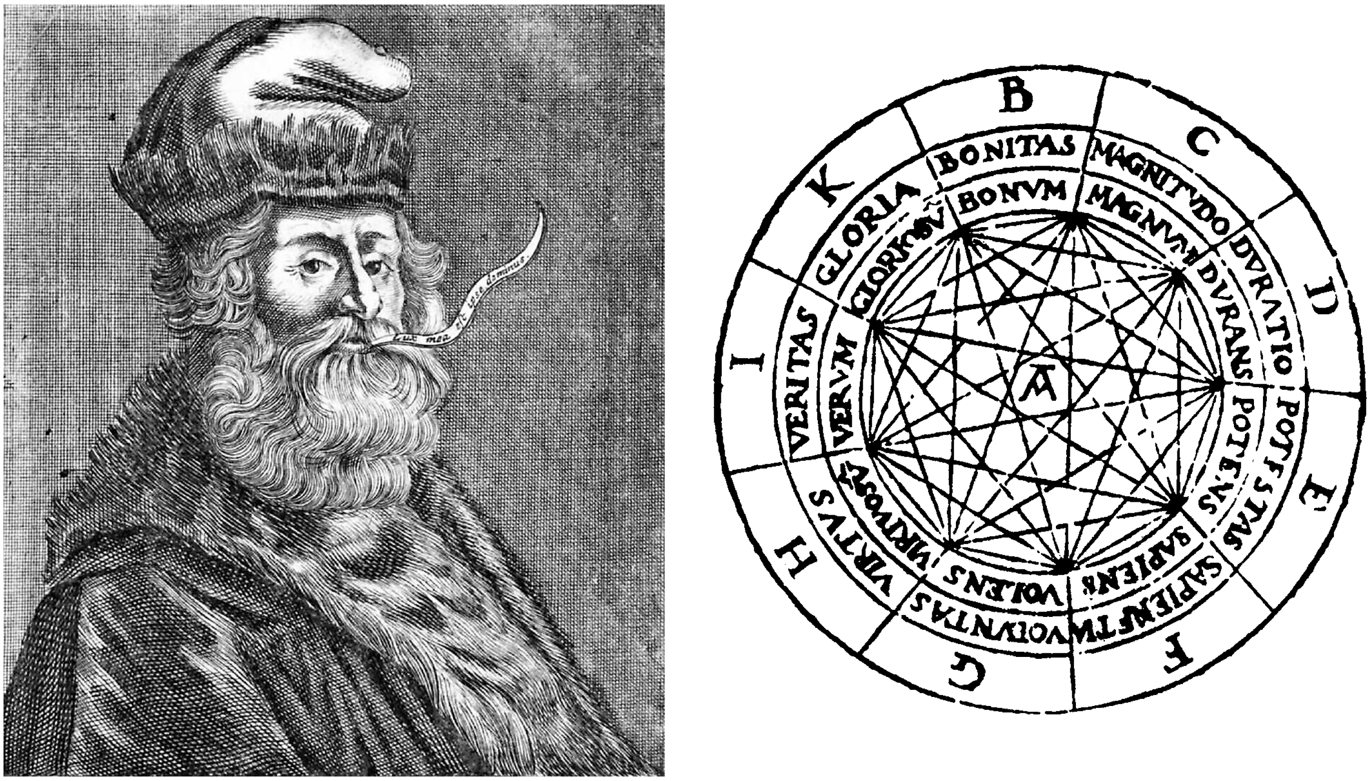


Figure 2. A thirteenth-century Franciscan monk, Ramon Llull, designed logical and visual aids to reason about the multipicity of God's attributes.

Although this work sounds quirky today, it is based on three fundamental assumptions that are now so ingrained in our thought that it is hard to appreciate their significance:

- We can represent concepts, assertions, or objects of thought with symbolic tokens.
- Compound concepts (or assertions or thoughts) can be obtained by forming combinations of more basic ones.
- Mechanical devices, even as simple as a series of concentric wheels, can be helpful in constructing and reasoning about such combinations.

Llull was influenced by an early Muslim thinker, al-Ghazali, and the first two assumptions can be found even earlier in the work of Aristotle. For example, the theory of the syllogism in *Prior Analytics* offers general arguments in which letters stand for arbitrary predicates, and Aristotle's other writings address the question of how predicates can combine to characterize or define a subject. But Llull's use of mechanical devices and procedures to support reasoning was new, and, in the eyes of many, this makes him the founder of mechanized reasoning.

Llull's ideas were an inspiration to Gottfried Leibniz.

Almost 400 years later, Llull's ideas were an inspiration to Gottfried Leibniz, who, in his doctoral dissertation, dubbed the method *ars combinatoria* ("the art of combinations"). In 1666 he wrote a treatise, *Dissertatio de arte combinatoria*, which contained a mixture of logic and modern combinatorics. The unifying theme once again was a method for combining concepts and reasoning about these combinations. In this treatise, Leibniz famously proposed the development of a *characteristica universalis*, a symbolic language that could express any rational thought, and a *calculus ratiocinator*, a mechanical method for assessing its truth.

Although Leibniz made some initial progress towards this goal, his languages and calculi covered a very restricted fragment of logical inference. It is essentially the fragment we now call *propositional logic*, rediscovered by George Boole in the middle of the nineteenth century. But soon after Boole, others began to make good on Leibniz's promise of a universal language of thought, or, at least, languages that were sufficient to represent more complex assertions. Peirce, Schröder, Frege, Peano, and others expanded logical symbolism to include quantifiers and relations. In 1879 Gottlob Frege published his landmark work, *Begriffsschrift* ("concept writing"), which presented an expressive logical language together with axioms and rules of inference. In the introduction, he situated the project clearly in the Leibnizian tradition while carefully restricting its scope to scientific language and reasoning.

In the early twentieth century, the work of David Hilbert and his students and collaborators, Ernst Zermelo's axiomatization of set theory, and Bertrand Russell and Alfred North Whitehead's *Principia Mathematica* all furthered the project of using symbolic systems to provide a foundation for mathematical reasoning. The project was so successful that, in 1931, Kurt Gödel could motivate his incompleteness theorems with the following assessment:

> The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules. The most comprehensive formal systems that have been set up hitherto are the system of *Principia mathematica* (PM) on the one hand and the Zermelo-Fraenkel axiom system of set theory (further developed by J. von Neumann) on the other. These two systems are so comprehensive that in them all methods of proof used today in mathematics are formalized, that is, reduced to a few axioms and rules of inference. [3]

This brief historical overview will help situate the work I intend to present here. To properly bridge the gap from the beginning of the twentieth century to the present, I would have to survey not only the history of logic, foundations of mathematics, and computer science but also the history of automated reasoning and interactive theorem proving. Nothing I can do in the scope of this article would do these subjects justice, so I will now set them aside and jump abruptly to the present day.

Formal Methods in Computer Science

The phrase "formal methods" is used to describe a body of methods in computer science for specifying, developing, and verifying complex hardware and software systems. The word "formal" indicates the use of formal languages to write assertions, define objects, and specify constraints. It also indicates the use of formal semantics, that is, accounts of the meaning of a syntactic expression, which can be used to specify the desired behavior of a system or the properties of an object sought. For example, an algorithm may be expected to return a tuple of numbers satisfying a given constraint, C, expressed in some specified language, whereby the logical account spells out what it *means* for an object to satisfy the symbolically expressed constraint. Finally, the word "formal" suggests the use of formal rules of inference, which can be used to verify claims or guide a search.

Put briefly, formal methods are used in computer science to say things, find things, and check things. Using an approach known as *model checking*, an engineer describes a piece of hardware or software and specifies a property that it should satisfy. A tool like a satisfiability solver (SAT solver) or satisfiability-modulo-theories solver (SMT solver) then searches for a counterexample trace, that is, an execution path that violates the specification. The search is designed to be exhaustive so that failure to find such a trace guarantees that the specification holds. In a complementary approach known as *interactive theorem proving*, the engineer seeks to construct, with the help of the computer, a fully detailed formal proof that the artifact meets its specification.

It should not be surprising that such technologies bear on mathematical activity as well. Proving the correctness of a piece of hardware or software is an instance of proving a theorem, in this case, the theorem that states that the hardware or software, described in mathematical terms, meets its specification. Searching for bugs in hardware or software is simply an instance of searching for a mathematical object that satisfies given constraints. Moreover, claims about the behavior of hardware and software are made with respect to a body of mathematical background. For example, verifying software often depends on integer or floating point arithmetic and on properties of basic combinatorial structures. Verifying a hardware control system may invoke properties of dynamical systems, differential equations, and stochastic processes.

Of course, there is a difference in character between proving ordinary mathematical theorems and proving hardware and software correct. Verification problems in computer science are generally difficult because of the volume of detail, but they typically do not have the conceptual depth one finds in mathematical proofs. But although the focus here is on mathematics, you should keep in mind that there is no sharp line between mathematical and computational uses of formal methods, and many of the systems and tools I will describe can be used for both purposes.

Verified Proof

Interactive theorem proving involves the use of computational proof assistants to construct formal proofs of mathematical claims using the axioms and rules of a formal foundation that is implemented by the system. The user of such an assistant generally has a proof in mind and works interactively with the system to transform it into a formal derivation. Proofs are presented to the system using a specialized proof language, much like a programming language. The computational assistant processes the input, complains about the parts it cannot understand, keeps track of goals and proof obligations, and responds to queries, say, about definitions and theorems in the background libraries. Most importantly, every inference is checked for correctness using a small, trusted body of code, known as the *kernel* or *trusted computing base*. Some systems even retain, in memory, a complete description of the resulting axiomatic derivation, a complex piece of data that can be exported and verified by an independent reference checker.

The choice of axiomatic foundation varies. Some systems are based on set theory, in which every object denotes a set. Predicates are then used to pick out which sets represent objects like integers, real numbers, functions, triangles, and structures. Most systems, however, implement frameworks in which every object is assigned a *type* that indicates its intended use. For example, an object of type `int` is an integer, and an object of type `int → int` is a function from integers to integers. Such an approach often permits more convenient forms of input, since a system can use knowledge of data types to work out the meaning of a given expression. It also makes it possible for a system to catch straightforward errors, such as when a user applies a function to an object of the wrong type. The complexity of the typing system can vary, however. Some versions of type theory have a natural computational interpretation, so that the definition of a function like the factorial function on the nonnegative integers comes with a means of evaluating it.

Many core theorems of mathematics have been formalized in such systems, such as the prime number theorem, the four color theorem, the Jordan curve theorem, Gödel's first and second incompleteness theorems, Dirichlet's theorem on primes in an arithmetic progression, the Cartan fixed-point theorems, and the central limit theorem. Verifying a big name theorem is always satisfying, but a more important measure of progress lies in the mathematical libraries that support them. To date, a substantial body of definitions and theorems from undergraduate mathematics has been formalized, and there are good libraries for elementary number theory, real and complex analysis, point-set topology, measure-theoretic probability, abstract algebra, Galois theory, and so on. In November of 2008 the *Notices* devoted a special issue to the topic of interactive theorem proving, which provides an overview of the state of the field at the time (see also [1]). As a result, here I will discuss only a few landmarks that have been achieved since then.

In 2012 Georges Gonthier and thirteen co-authors announced the culmination of a six-year project that resulted in the verification of the Feit–Thompson odd order theorem. Feit and Thompson's journal publication in 1963 ran 255 pages, a length that is not shocking by today's standards but was practically unheard of at the time. The formalization was carried out in Coq, a theorem prover based on a constructive type theory using a proof language designed by Gonthier known as SSReflect. The formalization included substantial libraries for finite group theory, linear algebra, and representation theory. All told, the proof comprised roughly 150,000 lines of formal proof, including 4,000 definitions and 13,000 lemmas and theorems.

Another major landmark is the completion of the formal verification of the Kepler conjecture, described in the introduction. Most of the proof was carried out in a theorem prover known as HOL light, though one component, the enumeration of tame graphs, was carried out in Isabelle.

Yet another interesting development in the last few years stems from the realization, due to Steve Awodey and Michael Warren and, independently, Vladimir Voevodsky, that dependent type theory, the logical framework used by a number of interactive theorem provers, has a novel topological interpretation. In this interpretation, data types correspond to topological spaces or, more precisely, abstract representations of topological spaces up to homotopy. Expressions that would ordinarily be understood as functions between data types are interpreted instead as continuous maps. An expression of the form $x = y$ is interpreted as saying that there is a path between x and y, and the rules for reasoning about equality in dependent type theory correspond to a common pattern of reasoning in homotopy theory in which paths are contracted down to a base point. This opens up possibilities for using interactive theorem provers to reason about subtle topological constructions. Moreover, Voevodsky showed that one can consistently add an axiom that states, roughly, that isomorphic structures are equal, which is to say, the entire language of dependent type theory respects homotopic equivalence. The field has come to be known as *homotopy type theory*, a play on the homotopical intepretation of type theory and the theory of homotopy types.

At this stage, it may seem premature to predict that formally verified proof will become common practice. Even the most striking successes in formally verified mathematics so far have done little to alter the status quo. Hales' result was published in the *Annals of Mathematics* and widely celebrated long before the formal verification was complete, and even though the verification of the Feit–Thompson theorem turned up minor misstatements and gaps in the presentations they followed, the correctness of the theorem was not in doubt, and the repairs were routine.

But the mathematical literature is filled with errors, ranging from typographical errors, missing hypotheses, and overlooked cases to mistakes that invalidate a substantial result. In a talk delivered in 2014,[2] Vladimir Voevodsky surveyed a number of substantial errors in the literature in homotopy theory and higher category theory, including a counterexample, discovered by Carlos Simpson in 1998, to the main result of a paper he himself had published with Michal Kapronov in 1989. Voevodsky ultimately turned to formal verification because he felt that it was necessary for the level of rigor and precision the subject requires.

The situation will only get worse as proofs get longer and more complex. In a 2008 opinion piece in the *Notices*, "Desperately seeking mathematical truth" [5], Melvyn Nathanson lamented the difficulties in certifying mathematical results: "We mathematicians like to talk about the 'reliability' of our literature, but it is, in fact, unreliable." His essay was not meant to be an advertisement for formal verification, but it can easily be read that way.

The mathematical literature is filled with errors.

Checking the details of a mathematical proof is far less enjoyable than exploring new concepts and ideas, but it is important nonetheless. Rigor is essential to mathematics, and even minor errors are a nuisance to those trying to read, reconstruct, and use mathematical results. Even expository gaps are frustrating, and it would be nice if we could interactively query proofs for more detail, spelling out any inferences that are not obvious to us at first. It seems inevitable that, in the long run, formal methods will deliver such functionality.

[2]`www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2014_IAS.pdf`.

Verified Computation

When Hales submitted his proof of the Kepler conjecture to the *Annals*, a sticking point was that the mathematically trained referees were not equipped to vouch for the correctness of the code. Hales and his collaborators countered this concern by verifying these computations as well as the conventional mathematical arguments. This was not the first example of a formally verified proof that involved substantial computation: Gonthier's verification of the four color theorem in Coq was of a similar nature, relying on a simplified computational approach by Robertson, Sanders, Seymour, and Thomas.

This brings us to the subtle question as to what, exactly, it means to verify a computation. Researchers working in formal verification are very sensitive to the question as to what components of a system have to be trusted to ensure the correctness of a result. Ordinary pen-and-paper proofs are checked with respect to the axioms and rules of a foundational deductive system. In that case, the trust lies with the kernel, typically a small, carefully written body of code, as well as the soundness of the axiomatic system itself, the hardware that runs the kernel, and so on. To verify the nonlinear inequalities in the Flyspeck project, Hales and a student of his, Alexey Solovyev, reworked the algorithms so that they produce proofs as they go. Whenever a calculation depended on a fact like $12 \times 7 = 84$, the algorithm would produce a formal proof, which was then checked by the kernel. In other words, every computational claim was subjected to the same standard as a pen-and-paper proof. Checking the nonlinear inequalities involved verifying floating point calculations, and the full process required roughly 5,000 processor hours on the Microsoft Azure cloud.

Another approach to verifying computation involves describing a function in the formal foundational language of a theorem prover, proving that the description meets the desired specification, and then using an automated procedure to extract a program in a conventional programming language to compute its values. The target of the extraction procedure is often a functional programming language like ML or Haskell. This approach requires a higher degree of trust, since it requires that the extraction process preserve the semantics of the formal expression. Of course, one also has to trust the target programming language and its compiler or interpreter. Even so, the verification process imposes a much higher standard of correctness than unverified code. When writing ordinary mathematical code, it is easy to make mistakes like omitting corner cases or misjudging the properties that are maintained by an iterative loop. In the approach just described, every relevant property has to be specified, and every line of code has to be shown to meet the specifications. In the Flyspeck project, the combinatorial enumeration of tame graphs was verified in this way by Tobias Nipkow and Gertrud Bauer.

There is also a middle ground in which functions are defined algorithmically within the formal system and then executed using an evaluator that is designed for that purpose. There is then a tradeoff between the complexity of the evaluator and the reliability of the result. The verification of the four color theorem used such a strategy to evaluate the computational component of the proof.

One notable effort along these lines, by Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote, and Enrico Tassi, yielded a verification of Apéry's celebrated 1973 proof of the irrationality of $\zeta(3)$. The starting point for the project was a Maple worksheet, designed by Bruno Salvy, that carried out the relevant symbolic computation. The group's strategy was to extract algebraic identities from the Maple computations and then construct formal axiomatic proofs of these identities in Coq. A fair amount of work was needed to isolate and manage side conditions that were ignored by Maple, such as showing that a symbolic expression in the denominator of a fraction is nonzero under the ambient hypotheses.

Yet another interesting project was associated with Tucker's solution to Smale's 14th problem. To demonstrate the existence of the Lorenz attractor, Tucker enclosed a Poincaré section of the flow defined by the Lorenz equations with small rectangles and showed that each rectangle (together with a cone enclosing the direction in which the attractor is expanding) is mapped by the flow inside another such rectangle (and cone). Tucker, a leading figure in the art of validated computation, relied on careful numeric computation for most of the region, coupled with a detailed analysis of the dynamics around the origin. Quite recently, Fabian Immler was able to verify the numeric computations in Isabelle. To do so, he not only formalized enough of the theory of dynamical systems to express all the relevant claims, but also defined the data structures and representations needed to carry out the computation efficiently and derived enough of their properties to show that the computation meets its specification.

Once again, on the basis of such examples, it may seem bold to predict that formally verified computation will become commonplace in mathematics. The need, however, is pressing. The increasing use of computation to establish mathematical results raises serious concerns as to their correctness, and it is interesting to see how mathematicians struggle to address this. In their 2003 paper, "New upper bounds on sphere packings. I," Cohn and Elkies provide a brief description of a search algorithm:

> To find a function g [with properties that guarantee an upper bound] … , we consider a linear combination of g_1, g_3, … , g_{4m+3}, and require it to have a root at 0 and m double roots at z_1, … , z_m. . . . We then choose the locations of z_1, … , z_m to minimize the value r of the last sign change of g. To make this choice, we do a computer search. Specifically, we make an initial guess for the locations of z_1, … , z_m, and then see whether we can perturb them to decrease r. We repeat the perturbations until reaching a local optimum.

After presenting the bounds that constitute the main result of the paper, they write:

> These bounds were calculated using a computer. However, the mathematics behind the calculations is rigorous. In particular, we use exact rational arithmetic, and apply Sturm's theorem to count real roots and make sure we do not miss any sign changes.

The passage goes on to explain how they used approximations to real-valued calculations by rational calculations without compromising correctness of the results. In their 2013 paper "The maximal number of exceptional Dehn surgeries," Lackenby and Meyerhoff turn to the topic of computation:

> We now discuss computational issues and responses arising from our parameter space analysis. The computer code was written in C++.

They then proceed to sketch the algorithms they used to carry out the calculations described in the paper, as well as the methods for interval arithmetic, and some of the optimizations they used. They also discuss the use of Snap, a program for studying arithmetic invariants of hyperbolic 3-manifolds, which incorporates exact arithmetic based on algebraic numbers. In their preprint "Universal quadratic forms and the 290 theorem" Bhargava and Hanke are forthright in worrying about the reliability of their computations:

> As with any large computation, the possibility of error is a real issue. This is especially true when using a computer, whose operation can only be viewed intermittently and whose accuracy depends on the reliability of many layers of code beneath the view of all but the most proficient computer scientist. We have taken many steps to ensure the accuracy of our computations, the most important of which are described below.

These steps include checks for correctness, careful management of roundoff errors, and, perhaps most importantly, making the source code available on a web page maintained by the authors.

The paper by Cohn and Elkies appeared in the *Annals of Mathematics*, the one by Lackenby and Meyerhoff appeared in *Inventiones Mathematicae*, and the paper by Bhargava and Hanke will appear in *Inventiones* as well. This makes it clear that substantial uses of computation have begun to infiltrate the upper echelons of pure mathematics, and the trend is likely to continue. In the passages above, the authors are doing everything they can to address concerns about the reliability of the computations, but the mathematical community does not yet have clear standards for evaluating such results. Are referees expected to read the code and certify the behavior of each subroutine? Are they expected to run the code and, perhaps, subject it to empirical testing? Can they trust the reliability of the software libraries and packages that are invoked? Should authors be required to comment their code sufficiently well for a computer-savvy referee to review it?

Whatever means we develop to address these questions have to scale. Perhaps the bodies of code associated with the examples above are manageable, but what will happen when results rely on code that is even more complicated, and, say, ten times as long? With results like the four color theorem and Hales' theorem, we are gradually getting past the vain hope that every interesting mathematical theorem will have a humanly surveyable proof. But it seems equally futile to hope that every computational proof will make use of code that can easily be understood, and so the usual difficulties associated with understanding complicated proofs will be paired with similar difficulties in understanding complicated programs.

Formal Search

Formal verification does not have a visceral appeal to most mathematicians: the work can be painstakingly difficult, and the outcome is typically just the confirmation of a result that we had good reason to believe from the start. In that respect, the Pythagorean triple theorem of Heule, Kullmann, and Marek fares much better. Here the outcome of the effort was a new theorem of mathematics, a natural Ramsey-like result, and a very pretty one at that. The result relied on paradigmatic search techniques from the formal methods community, and it seems worthwhile to explore the extent to which such methods can be put to good mathematical work.

To date, such applications of formal methods to mathematics are few and far between. In 1996 William McCune proved the Robbins conjecture, settling a question that had been open since the 1930s as to whether a certain system of equations provided an equivalent axiomatization of Boolean algebras. The result was featured in an article by Gina Kolata in the *New York Times*. But the subject matter was squarely in the field of mathematical logic, and so it is not surprising that an automated theorem prover (in this case, one designed specifically for equational reasoning) could be used for such purposes.

Systems like McCune's can also be used to explore consequences of other first-order axioms. For example, McCune himself showed that the single equation $(w((x^{-1}w)^{-1}z))((yz)^{-1}y)=x$ axiomatizes groups in a language with a binary multiplication and a unary inverse, and Kenneth Kunen later showed that this is the shortest such axiom. Kunen went on to use interactive theorem provers to contribute notable results to the theory of nonassociative structures such as loops and quasigroups. (More examples of this sort are discussed in [2].)

Since the beginning of this century, propositional satisfiability solvers have been the killer app for formal methods, permitting algorithmic solutions to problems that were previously out of reach. On the heels of the Pythagorean triples problem, Heule has recently established that the Schur number $S(5)$ is equal to 160; in other words, there is a five-coloring of the integers from 1 to 160 with no monochromatic triple a, b, c with $a + b = c$, but no such coloring of the integers from 1 to 161.

A SAT solver had a role to play in work on the Erdös discrepancy problem. Consider a sequence $(x_i)_{i>0}$, where each x_i is $\pm$ 1, and consider sums of this sequence along multiples of a fixed positive integer, such as $x_1 + x_2 + x_3 + \ldots$ and $x_2 + x_4 + x_6 + \ldots$ and $x_3 + x_6 + x_9 + \ldots$. In the 1930s, Erdös asked whether it is possible to keep the absolute value of such sums—representing the discrepancy between the number of +1's and −1's along the sequence—uniformly bounded. In other words, he asked whether there are a sequence (x_i) and a value C such that for every n and d, $\left|\sum_{i=1}^{n} x_{id}\right| \leq C$, and he conjectured that no such pair exists. In 2010 Tim Gowers launched the collaborative Polymath5 project on his blog to work on the problem. In 2014 Boris Konev and Alexei Lisitsa used a SAT solver to provide a partial result, namely, that there is no sequence satisfying the conclusion with $C = 2$. Specifically, they showed that there is a finite sequence $x_1, \ldots, x_{1,160}$ with discrepancy at most 2, but no such sequence of length 1,161. The following year, Terence Tao proved the full conjecture, with a conventional proof. This was a much more striking achievement, but we still have Konev and Lisitsa, and a SAT solver, to thank for exact bounds in the case $C = 2$. SAT solvers have been applied to other combinatorial problems as well.

The line between discovery and verification is not sharp. Anyone writing a search procedure does so with the intention that the results it produces are reliable, but, as with any piece of software, as the code becomes more complex, it becomes increasingly necessary to have mechanisms to ensure that the results are correct. This is especially true of powerful search tools, which rely on complicated tricks and heuristics to improve performance at the risk of compromising soundness. It is important that the solution to the Pythagorean triples problem produced a formal proof that could be verified independently, and, in fact, that proof has been checked by three proof checkers that themselves have been formally verified, one in Isabelle, one in Coq, and one in a theorem prover named ACL2. This provides a high degree of confidence in the correctness of the result.

Today, the use of formal methods in discovery is even less advanced than the use of formal methods in verification. The results described above depend, for the most part, on finding consequences of first-order axioms for algebraic structures, searching for finite objects satisfying combinatorial constraints, or ruling out the existence of such objects by exhaustive enumeration. It is not surprising that computers can be used to exhaust a large number of finite cases, but few mathematical problems are presented to us in that form. And spinning out consequences of algebraic axioms is a far cry from discovering consequences of rich mathematical assumptions involving heterogeneous structures and mappings between them.

But just as pure mathematicians have discovered uses for computation in number theory, algebraic topology, differential geometry, and discrete geometry, one would expect to find similarly diverse applications for formal search methods. The problem may simply be that researchers in these fields do not yet have a sense of what formal search methods can do, whereas the computer scientists who develop them do not have the expertise needed to identify the mathematical domains of application. If that is the case, it is only a matter of getting the communities to work more closely together. Combinatorics is a natural place to start, because the core concepts are easily accessible and familiar to computer scientists. But it will take real mathematical effort to understand how problems in other domains can be reduced to the task of finding finite pieces of data or ruling out the existence of such data by considering sufficiently many cases.

Indeed, for all we know, there may be lots of lovely theorems of mathematics that can *only* be proved that way. For the last two thousand years, we have been looking for proofs of a certain kind, because those are the proofs that we can survey and understand. In that respect, we may be like the drunkard looking for his keys under a streetlamp even though he lost them a block away, because that is where the light is. We should be open to the possibility that new technologies can open new mathematical vistas and afford new types of mathematical understanding. The prospect of ceding a substantial role in mathematical reasoning to the computer may be disconcerting, but it should also be exhilarating, and we should look forward to seeing where the technology takes us.

Digital Infrastructure

Contemporary digital technologies for storage, search, and communication of information provide another market for formal methods in mathematics. Mathematicians now routinely download papers, search the web for mathematical results, post questions on Math Overflow, typeset papers using LaTeX, and exchange mathematical content via email. Digital representations of mathematical knowledge are therefore central to the mathematical process. It stands to reason that mathematics can benefit from having better representations and better tools to manage them.

TeX and LaTeX have transformed mathematical dissemination and communication by providing precise means for specifying the appearance of mathematical expressions. MathML, building on XML, goes a step further, providing markup to specify the *meaning* of mathematical expressions as well. But MathML stops short of providing a foundational specification language, which is clearly desirable: imagine being able to find the statement of a theorem online, and then being able to look up the meaning of each defined term, all the way down to the primitives of an axiomatic system if necessary. That would provide clarity and uniformity, and help ensure that the results we find mean what we think they mean. The availability of such formal specifications would also support verification: we could have a shared public record of which results have been mechanically verified and how, and we could use theorems from a public repository to verify our own local results. Automated reasoning tools could make use of such background knowledge, and could, in turn, be used to support a more robust search. Contemporary *sledgehammer* tools for interactive theorem provers rely on heuristics to extract relevant theorems from a database and then use them to carry out a given inference. With such technology, one could ask whether a given statement is equivalent to, or an easy consequence of, something in a shared repository of known facts.

Figure 3. In the summer of 2017, the Isaac Newton Institute hosted a six-week workshop, *Big Proof*, dedicated to the technologies described here.

For all these purposes, formal specifications are essential. As a first step towards obtaining them, Hales has recently launched a Formal Abstracts Project, which is designed to encourage mathematicians to write formal abstracts of their papers. To process and check the definitions, he has chosen an interactive theorem prover called Lean, an open source project led by Leonardo de Moura at Microsoft Research (and to which I am a contributor). In the coming years, the Formal Abstracts project plans to seed the repository with core definitions from all branches of mathematics, and develop guidelines, tools, and infrastructure to support widespread use.

Conclusions

In the summer of 2017, the Isaac Newton Institute hosted a six-week workshop, *Big Proof*, dedicated to the technologies described here (see Figure 3).[3] As part of a panel discussion, Timothy Gowers gave a frank assessment of the new technology and the potential interest to mathematicians. He observed that the phrase "interactive proof assistant" is rather appealing until one learns that such assistants actually make proving a theorem a lot more difficult. The fact that a substantial body of undergraduate mathematics has been formalized is generally unexciting to the working mathematician, and existing tools currently offer little to improve our mathematical lives.

Gowers did enumerate three technologies that he felt would have widespread appeal. The first is a bona fide proof assistant that could work out small lemmas and results, at the level of a capable graduate student. The second is genuine search technology that can tell us whether a given fact is currently known, either because we would like to use it in a proof, or because we think we have a proof and are wondering whether it is worthwhile to work out the details. The third is a real proof checker, that is, something we can call when we think we have proved something and want confirmation that we have not made a mistake.

We are not there yet, but such technology seems to be within reach. There are no apparent conceptual hurdles that need to be overcome, though getting to that point will require a good deal of careful thought, clever engineering, experimentation, and hard work. And even before tools like these are ready for everyday use, we can hope to find pockets of mathematics where the methods provide a clear advantage: proofs that rely on nontrivial calculations, subtle arguments for which a proof assistant can provide significant validation, and problems that are more easily amenable to search techniques. Verification is not an all-or-nothing affair. Short of a fully formalized axiomatic proof, formalizing a particularly knotty or subtle lemma or verifying a key computation can lend confidence to the correctness of a result. Even just formalizing definitions and the statements of key theorems, as proposed by the Formal Abstracts project, adds helpful clarity and precision. Formal methods can also be used in education: if we teach students how to write formal proofs and informal

[3] *Talks delivered at the program are available online at* `www.newton.ac.uk/event/bpr`.

proofs at the same time, the two perspectives reinforce one another.[4]

The mathematics community needs to put some skin in the game, however. Proving theorems is not like verifying software, and computer scientists do not earn promotions or secure funding by making mathematicians happy. We need to buy into the technology if we want to reap the benefits.

To that end, institutional inertia needs to be overcome. Senior mathematicians generally do not have time to invest in developing a new technology, and it is hard enough to learn how to use the new tools, let alone contribute to their improvement. The younger generation of mathematicians has prodigious energy and computer savvy, but younger researchers would be ill-advised to invest time and effort in formal methods if it will only set back their careers. To allow them to explore the new methods, we need to give them credit for publications in journals and conferences in computer science, and recognize that the mathematical benefits will come only gradually. Ultimately, if we want to see useful technologies for mathematics, we need to hire mathematicians to develop them.

The history of mathematics is a history of doing whatever it takes to extend our cognitive reach, and designing concepts and methods that augment our capacities to understand. The computer is nothing more than a tool in that respect, but it is one that fundamentally expands the range of structures we can discover and the kinds of truths we can reliably come to know. This is as exciting a time as any in the history of mathematics, and even though we can only speculate as to what the future will bring, it should be clear that the technologies before us are well worth exploring.

References

[1] Jeremy Avigad and John Harrison, Formally verified mathematics, *Commun. ACM* **57**(4):66–75, 2014.

[2] Michael J. Beeson, The mechanization of mathematics. In *Alan Turing: Life and Legacy of a Great Thinker*, pages 77–134. Springer, Berlin, 2004. MR2172456

[3] Kurt Gödel, Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsh. Math. Phys.*, **38**(1):173–198, 1931. Reprinted with English translation in *Kurt Gödel: Collected Works*, volume 1, Feferman et al, eds., Oxford University Press, New York, 1986, pages 144–195. MR1549910

[4] Thomas C. Hales, Mathematics in the age of the Turing machine. In *Turing's Legacy: Developments from Turing's Ideas in Logic*, volume 42 of Lecture Notes in Logic, pages 253–298. Assoc. Symbol. Logic, La Jolla, CA, 2014. MR3497663

[5] Melvyn B. Nathanson, Desperately seeking mathematical proof, *Math. Intelligencer*, **31**(2):8–10, 2009. MR2505014

ACKNOWLEDGMENT. I am grateful to Jasmin Blanchette and Robert Y. Lewis for corrections, suggestions, and improvements.

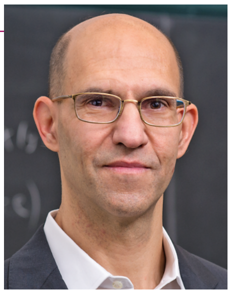
[4]*See the freely available textbook,* Logic and Proof, *by Robert Y. Lewis, Floris van Doorn, and me:* `leanprover.github.io/logic_and_proof/`.

Image Credits

Figure 1 courtesy of Marijn J. Heule.
Figure 2 images are in the public domain.
Figure 3 courtesy of Isaac Newton Institute.
Author photo courtesy of Carnegie Mellon University.

ABOUT THE AUTHOR

Jeremy Avigad's research interests include mathematical logic, formal verification, and the history and philosophy of mathematics.


Jeremy Avigad

The following is excerpted from *The Hope Circuit: A Psychologist's Journey from Helplessness to Optimism* by Martin E. P. Seligman. Copyright © 2018. Available from PublicAffairs, an imprint of Hachette Book Group, Inc.

A Full Fellowship?

I found that I could make deep contact only with the most serious students. Robin Forman was a mathematical whiz and in a band that did The Doors almost as well as The Doors themselves. His band played at my "Master Blasters," the master's open house I held periodically with loud music in my attempt to appear less geeky to my students. Robin bore an uncanny resemblance to my college roommate Wilfrid Schmid, Princeton valedictorian of 1964, now gone off to parts unknown."Harvard mathematics is my first choice, and I just got accepted, but with $5,000 minus tuition," Robin said, exuding disappointment, when he came to me in April for advice.

"That is an amazing coincidence," I replied. "My roommate, Wilfrid, also first in our class, applied to Harvard almost twenty years ago and was only given $5,000 minus tuition. He asked my advice, and I told him to phone the chairman of math at Harvard and tell him confidently, 'Perhaps, you don't know who I am.' Wilfrid did this and was promptly given a full fellowship at Harvard. Phone the chairman of math at Harvard and tell him, 'Perhaps you don't know who I am,'" I suggested. "I phoned the chairman of math at Harvard and said exactly those words," Robin reported back to me the next day. "Wilfrid Schmid is the chairman, and he gave me a full fellowship."