Proving Theorems with Computers

Kevin Buzzard

Superhuman Mathematics

How is breakthrough mathematics achieved? Here is one example, from algebraic number theory.

In his 1987 paper "Deforming Galois representations," Barry Mazur observes that the *geometric* concept of a smoothly varying complex family of representations of a group has an *arithmetic* analogue. He sets up deformation theory in the non-geometric setting of mod p and p-adic Galois representations, and makes some interesting observations about the relationship between arithmetic deformation rings and Galois cohomology. By 1990, Mazur and Tilouine have raised a profound question about whether a certain universal deformation ring coming out of this theory is isomorphic to one of Hida's Hecke algebras. In 1993 Wiles uses new techniques in commutative algebra to reduce a variant of this question to a numerical criterion, and a year later, aided by Taylor, he has pushed the strategy through. The semistable Shimura–Taniyama conjecture (at that time often called the semistable Shimura–Taniyama–Weil conjecture) follows, and hence, by earlier work of Ribet, Fermat's Last Theorem.

This is but one of very many examples where cross-fertilization has occurred in mathematics. The breadth of Mazur's mathematical knowledge (he was initially a topologist) played a key role here. In a 2014 article [Maz14] for the *Math. Intelligencer*, Mazur writes: "Reasoning by analogy is the keystone: it is present in much (perhaps all) daily mathematical thought, and is also often the inspiration behind some of the major long-range projects in mathematics."

Kevin Buzzard is a professor of mathematics at Imperial College London. His email address is k.buzzard@imperial.ac.uk.

DOI: https://doi.org/10.1090/noti2177

One might hence ask the following question: if one human had an understanding of all of modern pure mathematics simultaneously, how much further would they immediately be able to see? How many insights are there which are needed to unblock one field, but which have already been made in another?

This question can of course be dismissed as a thought experiment. Perhaps it is the kind of thing which philosophers might muse over, but it is the year 2020 and pure mathematics is much too big for one human to comprehend.

However, it is the year 2020 and hence we have computer proof systems which *are* now in theory capable of understanding all of modern pure mathematics. So why is our community not teaching it to them? This is entirely within our grasp, and we have absolutely no idea what will happen when we do.

1. Teaching Mathematics to Humans

Let us look at the way pure mathematics is learnt by humans, from undergraduate to PhD level.

1.1. The basics. Three key concepts in pure mathematics are the *definition*, the *theorem statement*, and the *proof*. We will talk more about this trichotomy later on, but for now let us focus on the concept of proof. A course introducing the formal notion of proof might cover concepts such as sets, functions, and binary relations, and basic theorems about these objects will be carefully proved. For example, there might be a proof that distinct equivalence classes for an equivalence relation are disjoint. There are many ways that one can attempt to teach this to undergraduates. Recently I have become fond of a method where I bring a set of around 100 plastic shapes coloured red, yellow, green, and blue into class. Two shapes are defined to be equivalent if they have the same colour, and it is not hard to convince the students that this is an equivalence relation,

and that the red shapes form an equivalence class, the blue shapes form another, and so on. The fact that distinct equivalence classes are disjoint is now obvious. However this is an example, not a proof. The formal proof, which I then go on to show the students, is a series of elementary steps, each of which follows from the rules of logic or the axioms of an equivalence relation.

In proofs such as these, we are operating very close to the "machine" which drives formal mathematics—the machine which tells us that if P is true, and if P implies Q, then Q is true, and other such logical rules. This axiom-based attitude continues to play an important role in subsequent classes such as first courses in group theory, linear algebra, and real analysis. The real numbers might be presented as a complete totally ordered archimedean field, that is, a structure satisfying a list of axioms. Using only these axioms, and the axioms of the foundational system we're working in (often set theory), we can build a basic theory of real analysis from first principles. The theories of sequences, limits, infinite sums, continuous functions $\mathbb{R} \to \mathbb{R}$, differentiation, integration, and so on can all be carefully built from these axioms. Similarly, a lecturer presents the axioms of a group in a first course on group theory, and from these axioms we can build the theory of subgroups, normal subgroups, group homomorphisms, kernels, images, and quotient groups, and prove the first isomorphism theorem for groups. At this stage in the development of mathematics, every proof can be chased right down to the axioms of the system we are considering, and students are expected to learn from such courses that mathematics *can* be done in this way. Much (but, as we are about to see, not all) of undergraduate pure mathematics is of this form.

1.2. **Developing intuition.** After a while it becomes inconvenient to do mathematics in a purely axiomatic fashion. For example, proving that if we remove a finite set of points from $\mathbb{R}^2$ then the resulting topological space is still path connected could of course *in theory* be done from the axioms, but in practice, rather than attempting to write down the function defining a path between two arbitrary points in the space, one would just draw a picture. The same is also true when proving basic results about contour integrals in complex analysis—there are several "proofs by picture" in a typical development of the theory. The concept of a simple closed curve in the plane having an inside and an outside will often be taken as read, although very few students will have seen a proof of the Jordan curve theorem at this point in their mathematical education. Over time, students learning mathematics begin to understand our unwritten rules of "what is allowed in practice." One is reminded of the apocryphal story of a student asking their professor whether the fact just presented to the class as "obvious" was indeed obvious, and the professor going into deep thought to emerge 20 minutes later with the reply "yes." By this point in the development of a student's education, lecturers are expecting the students to "learn to fly." Arguments in lectures may take place high above the axioms, with technical details being dismissed as obvious or easy to verify, and left to the reader (perhaps with some hints). This is the beginning of what Terry Tao [Tao09] has called the *post-rigorous* stage of mathematics. To borrow a phrase from computer science, students begin to learn the intuitive "front end" of mathematics.

1.3. **PhD research.** Those students who convince us that they can steer their mathematical arguments correctly are rewarded by being given PhD places. The prize for this "levelling up" is that they are allowed access to the mathematical literature, and from now on they can assume any result they like, as long as it is published in a reasonably prestigious journal and their advisor believes it. A typical PhD thesis in pure mathematics will contain new proofs of results in a given theory. In my personal case, this theory was the theory of p-adic Galois representations attached to modular forms. By the time they graduate, a PhD student will typically know many theorem statements concerning the objects they chose to study, and may well have contributed to this list of theorem statements themselves. Again to borrow a phrase from computer science, the student knows the *interface* to each object in their area of expertise. The student is allowed to assume any results in the interface, and might well know how to prove some of them—but possibly not all of them. For example, when I was a PhD student, I had not read the details of the proof of the theorem of Deligne which attached a p-adic Galois representation to a modular form, a key result from the interface to the theory of modular forms upon which my entire PhD work was based. In fact, at that time there was only really a sketch proof of the result in the literature. This was not however a problem, because the proof of Deligne's theorem was "known to the experts." Students are expected to get their own intuition of their area, and after a while should have a feeling about what is accessible given known results, and what requires genuinely new ideas.

In the rest of this article, I would like to discuss the idea that computers can be taught mathematics in much the same sort of way. I leave it up to the reader to decide whether they would like to be involved, but what I do believe is that these computer systems are now here, that they can eat mathematics, and that they will ultimately change the way we do both teaching and research; furthermore, the sooner these systems are noticed by mathematicians, the sooner this will happen.

2. A Brief Introduction to Interactive Theorem Provers (ITPs)

Before we talk about computer proof systems, let us consider a computer program which many of us are familiar with: LaTeX.

The computer program LaTeX is a mathematical typesetting system. Thirty years ago, only a small number of (typically young) mathematicians knew how to write LaTeX files, but now most of us do; this program is now the standard typesetting system for mathematicians. If one runs the LaTeX program on a LaTeX file, one of two things can happen. The file might contain errors, for example perhaps we accidentally used a command in normal text mode which is only valid in maths mode. In this case the LaTeX editor we are using will typically flag these errors and ask that we fix them. But when all the errors are gone, the LaTeX program will compile the LaTeX file, and the output will be a (hopefully) beautifully typeset document. This document is typically a pdf file nowadays, which can be read on a screen, printed out, or of course sent to another computer on the internet.

The computer program Lean is an interactive theorem prover (ITP). A small number of (typically young) mathematicians know how to write Lean files. A Lean file can contain definitions, theorem statements, and proofs—we shall see examples later. Just as a LaTeX file is likely to contain some parts written in "maths mode," a Lean file is likely to contain some parts written in "tactic mode."

If one runs the Lean program on a Lean file, one of two things can happen. The file might contain errors, for example perhaps not all the hypotheses of a theorem were checked (or were true!) when it was applied. In this case, the Lean editor we are using will typically flag these errors and ask that we fix them. When all the errors are gone, Lean will compile the Lean file, and the output will be... nothing at all. When this happens, Lean believes that all the definitions in your file make sense, and it believes that all the proofs in the file are correct.[1]

Note that Lean is not (just) a programming language like Python or C++ or Haskell. To give an example of the difference: in Python you could write a program which printed out the first 1000 prime numbers, or the first 1000 digits of π. In Lean you could write a proof that there are infinitely many prime numbers, or a proof that π was transcendental.

[1] *It is not strictly speaking true that the output is nothing at all—the actual output is a computer file, unreadable by humans, where each proof is represented as a complicated graph. The proofs in this form (terms in a type theory) can be independently verified by typecheckers written in other languages and running on other operating systems and other chipsets, to minimise the possibility that bugs in Lean cause incorrect proofs to be accepted as valid.*

Lean was written by Leonardo de Moura at Microsoft Research, and is free and open source software which runs on any modern operating system. It is one of many ITP systems. Others include Coq, Isabelle/HOL, Mizar, Metamath, HOL Light, Agda, Arend, and there are at least 20 more; many of these are also free and open source, and some are over 50 years old. All of these systems use slightly different logical foundations, and it is hard to automatically move a mathematical proof from one system to another, for the same reasons that it is hard to automatically translate a computer program from one language to another. However the differences in these systems will not concern us here. It suffices to say that essentially all of the examples in this article can in theory be written in essentially all of the ITP systems. We will focus on the Lean theorem prover in the examples below, but this is only because it is the system which I know best.

3. Teaching Mathematics to Computers

For the rest of this article, I would like to discuss the possibility of teaching a computer pure mathematics, following the same path as the way we teach it to humans. Let us start with the basics. Earlier on, we mentioned three fundamental mathematical concepts—the theorem statement, the proof, and the definition. We will now see about how Lean understands these concepts.

3.1. Propositions.

First let us talk about general true/false statements, a fundamental concept in mathematics. As well as theorems, mathematicians are interested in conjectures, which are true/false statements which might be believed to be true, but which are not proven. Here are some examples of true/false statements:

- $2 + 2 = 4$;
- $2 + 2 = 5$;
- Fermat's Last Theorem;
- The Riemann Hypothesis.

Two of these statements are true, one is false, and the truth value of the Riemann Hypothesis is currently unknown. Mathematicians do not really have a good word for a general true/false statement. The Riemann Hypothesis is called a conjecture, but it seems ridiculous to call $2+2 = 4$ or $2+2 = 5$ a conjecture. Note however that most mathematicians use the words Theorem, Lemma, Proposition, and Corollary to express ideas which are formally the same, whereas logicians use the word *Proposition* to mean an arbitrary true/false statement. From now on we will use the word Proposition in this way, meaning a mathematical statement which has a truth value, rather than one which is definitely true. For example, $2 + 2 = 5$ is a false Proposition. We will capitalise Proposition to remind us of this slightly non-standard usage.

Here is an example of some valid Lean code which defines a Proposition:

```
variables (X Y Z : Type)
  (f : X → Y) (g : Y → Z)

open function

definition P :=
  injective f ∧ injective g →
  injective (g ∘ f)
```

This code defines a Proposition called `P`, stating that the composite of two injective functions is injective. Note that there is no proof here—we are just observing that the theorem *statement* can be formalised in Lean. The variables X, Y, and Z were initialised to be not sets but "types"; however, in this context the two ideas coincide, the only difference being a linguistic one where we speak about terms `x` of a given type `X` and write `x : X` rather than speaking about elements x of a given set X and writing $x \in X$. Note finally that the arrow $\to$ is used both as notation for a function, and for the concept $\implies$ of implication.

Here is an example which shows that 2 + 2 = 5 is also a Proposition in Lean:

```
definition Q :=
2 + 2 = 5
```

Just to reiterate: a Proposition is any true/false statement.

3.2. **Proofs.** Creating a mathematical proof of a Proposition is like solving a puzzle, or a level of a computer game. Let us look further at this analogy. In a sudoku puzzle, one is presented with a partially filled-in grid of numbers and asked to fill in the rest of them, subject to some axioms. My first experience with sudoku was seeing levels of this game in newspapers, and solving them with a pen. Solving a sudoku level in this way can be quite inconvenient—any error may create quite a mess and might be difficult to recover from. It is far easier to solve sudoku levels on a phone app or a web browser, where errors can be erased more efficiently, and experimental lines can be explored and then painlessly accepted or rejected.

Many of us will have seen undergraduate work, written in pen, which is also quite a mess. Lean offers a structured environment where proofs can be created. As an example, let us consider the proof of the Proposition that if $f : X \to Y$ and $g : Y \to Z$ are injective functions, then so is $g \circ f$. Let us first run through the mathematical proof. By definition, our task is to prove that if $a, b \in X$ and $g(f(a)) = g(f(b))$, then $a = b$. By injectivity of f, we know that to prove $a = b$ it suffices to prove that $f(a) = f(b)$. Finally, the assertion $f(a) = f(b)$ follows from injectivity of g and our assumption.

We will go through a Lean proof of this Proposition below. As one creates the proof using a Lean editor, one can see Lean's "tactic state," representing the things Lean knows at any point during the proof. For example, just after one has applied injectivity of f, Lean's tactic state is the following:

```
X Y Z : Type,
f : X → Y,
g : Y → Z,
f_inj : injective f,
g_inj : injective g,
a b : X,
hgf : (g ∘ f) a = (g ∘ f) b
⊢ f a = f b
```

Let us go through this tactic state. The current hypotheses are listed above the "turnstile" $\vdash$, and the current goal is stated after it. The first three lines say that X, Y, and Z are types, and f and g are functions. The hypotheses `f_inj` and `g_inj` are the assertions that f and g are injective. The next line says that a and b are terms of type X, which is the type-theoretic way to say that they are elements of the set X. Our final hypothesis `hgf` is the assertion that $g(f(a)) = g(f(b))$, and our goal is to prove $f(a) = f(b)$.

As one solves the level, i.e., builds the proof, in Lean, the tactic state changes interactively, until it eventually becomes `Proof complete`. Just like theorem statements, Lean's tactic state can often be easily understood by mathematicians without any specialist knowledge of ITPs, because the notation used is close to standard mathematical notation.

Here is a complete Lean file containing the proof, written in Lean's tactic mode, with comments (written in grey, preceded by `--`, and ignored by Lean). If you are reading this article in digital format, I invite you to interact with this proof by clicking on this link, which will open up a Lean editor within a web browser. This is most definitely *not* the best way to interact with Lean—it will be slow (you might have to wait for up to ten seconds for Lean to initialise and process the file on a fast PC, and far longer on a mobile device; you also need to have some cookies enabled). But when "Lean is busy…" finally turns to "Lean is ready!" you can click around and see Lean's tactic state at any given point in the proof. If instead you install Lean on your own computer, the same process will be lightning fast.

```
-- need access to many useful tactics
import tactic

-- need injective functions
open function
```

```
-- Let X, Y, Z be types and
-- let f : X → Y and g : Y → Z be
-- functions between these types
variables (X Y Z : Type)
  (f : X → Y) (g : Y → Z)

-- Theorem: if f and g are
-- injective, then so is g ∘ f.
theorem injective_comp :
  injective f ∧ injective g →
  injective (g ∘ f) :=
begin
  -- assume f and g are injective.
  rintro ⟨f_inj, g_inj⟩,
  -- We want to prove g ∘ f is
  -- injective. So say a,b ∈ X and
  -- assume g(f(a))=g(f(b)).
  intros a b hgf,
  -- We want to prove that a = b. By
  -- injectivity of f, it suffices to
  -- prove that f(a)=f(b).
  apply f_inj,
  -- By injectivity of g, it suffices
  -- to prove g(f(a))=g(f(b)).
  apply g_inj,
  -- But this is an assumption.
  assumption,
end
```

This tactic mode proof looks mildly intimidating but basically comprehensible, in the same way that `\sum_{n=0}^{100}n^2` looked mildly intimidating but basically comprehensible before we learnt about LaTeX's maths mode. The tactics used in the proof such as `intros` and `apply` perform basic logical moves on the tactic state, and are not hard to pick up. The last line of the proof, the `assumption` tactic, solves the goal $g(f(a)) = g(f(b))$ by noting that it is equal to one of our assumptions, namely `hgf`.

Anyone concerned about how such a triviality might take five lines to prove might be interested in seeing the same proof written in Lean's "term mode":

```
theorem injective_comp' :
  injective f ∧ injective g → injective
    (g ∘ f) :=
λ ⟨f_inj, g_inj⟩ _ _ hgf, f_inj $ g_inj
    hgf
```

This complete proof—shorter than the corresponding LaTeX proof—is far harder for a beginner to understand, and also indicates something about what is going on under the hood, namely that a proof in Lean is actually a function. Indeed, Lean is a functional programming language. We will not go any further into this issue here.

Below is another tactic mode Lean proof, this time of the fact that if a and b are real numbers, then $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$. Before we embark upon it, let us consider what goes into a proof from first principles, assuming that the real numbers are a field. We first write the left-hand side as $((a+b)(a+b))(a+b)$ or $(a+b)((a+b)(a+b))$ depending on what definition we are using for x^3. Then we apply left and right distributivity several times to expand out the brackets. If one does this whilst carefully keeping track of all brackets involved, one will discover that one now needs to apply associativity and commutativity of addition and multiplication 20 times or more in order to turn the left-hand side into the right-hand side—operations which a mathematician applies intuitively and without comment. The full proof, using only the axioms of a ring, seems to be at least 30 lines long, although the exact number of axiom applications needed depends on other foundational questions such as whether 3 is defined to mean $(1+1)+1$ or $1+(1+1)$. Here is a complete proof of this result in Lean's tactic mode:

```
import tactic -- tactics
import data.real.basic -- the real
   numbers

example (a b : ℝ) :
   (a+b)^3=a^3+3*a^2*b+3*a*b^2+b^3 :=
begin
  ring,
end
```

The `ring` tactic is a high-level tactic, concealing tedious low-level work and enabling mathematicians to operate using their usual interface, well above the axioms of a ring. This tactic was implemented in Lean by Mario Carneiro, a computer scientist, following the efficient Coq implementation by Grégoire and Mahboubi in [GM05] of a "classical" decision problem algorithm for commutative rings. It has been indispensable for the subsequent development of ring theory in Lean.

A general tactic mode proof will contain a mixture of low-level and high-level tactics, corresponding to whether the corresponding human proof is operating at axiom level or well above the axioms.

Codewars (`www.codewars.com`) is a website containing programming challenges in many programming languages, including Lean. The Lean Codewars levels contain many mathematical puzzles, ranging from easy questions (proving that the sum of two odd numbers is even, for example), to far harder ones involving finding all integer solutions to the subtle Diophantine equations $x^2 - 37y^2 = 3$

and $y^2 = x^3 + 11$. Solving these harder problems involves using a range of low-level and high-level tactics. Of course one can also invoke theorems from Lean's extensive mathematics library, where various number-theoretic facts such as quadratic reciprocity are already proved. We will say more about Lean's mathematics library below.

3.3. **Definitions.** As well as Propositions and proofs, systems like Lean can understand new mathematical definitions. Many of the definitions in Lean's maths library are definitions of types or of terms. For example, the type $\mathbb{R}$ is defined to be the type of equivalence classes of Cauchy sequences of rationals, and the term π is defined to be twice the smallest positive zero of the cosine function. Groups, rings, fields, manifolds, schemes, perfectoid spaces, and many many other mathematical structures are all defined to be types in Lean, and their definitions are all readable to mathematicians. For example here is a definition of a group in Lean:

```
class group (G : Type)
 extends has_mul G, has_one G, has_inv G
   :=
(mul_assoc :
  ∀ (a b c : G), (a * b) * c = a * (b *
   c))
(one_mul : ∀ (a : G), 1 * a = a)
(mul_left_inv : ∀ (a : G), a⁻¹ * a = 1)
```

The "structural" part of a group (the multiplication, identity, and inverse) is packed into the second line, and the axioms follow afterwards. Given this definition, one can now start to prove other basic results. For example here is something which is proved very early on in the development of the interface for groups—the proof that the left identity coming from the axioms is also a right identity:

```
theorem mul_one (a : G) : a * 1 = a :=
begin
  -- exercise!
end
```

This exercise now becomes a puzzle. It is my experience that certain undergraduates enjoy solving puzzles like this in Lean, developing the basic theory of groups by completing levels of a computer game.

In 2012 Peter Scholze introduced the concept of a perfectoid algebra and a perfectoid space into mathematics. Slightly later on, Fontaine introduced the general notion of a perfectoid ring. Here is the definition of a perfectoid ring in Lean (here p is a prime number):

```
structure perfectoid_ring (R : Type)
   [Huber_ring R] extends Tate_ring R :
   Prop :=
(complete  : is_complete_hausdorff R)
(uniform   : is_uniform R)
(ramified  : ∃ ϖ : pseudo_uniformizer R,
  ϖ^p | p in R°)
(Frobenius : surjective (Frob R°/p))
```

A perfectoid ring is a complete Hausdorff Tate ring satisfying some technical hypotheses. Experts in the area will certainly be able to read and understand the gist of the code above. Perfectoid rings and perfectoid spaces were formalised in Lean by Johan Commelin, Patrick Massot, and myself; see [BCM20].

3.4. **Lean's mathematics library.** The above snippet from `perfectoid_ring` code would not compile in core Lean alone; some imports would be needed from Lean's maths library and Lean's perfectoid space library. Lean's maths library is a rapidly growing library containing a lot of undergraduate-level algebra, analysis, number theory, geometry, and topology. At the time of writing (April 2020) it contains theorems from number theory such as quadratic reciprocity, theorems from algebra such as the Hilbert basis theorem, theorems from analysis such as the inverse function theorem, open mapping theorem, and Arzelà–Ascoli theorem, definitions such as manifolds and topological spaces, smooth functions, and so on. Undergraduates at my university can (and do) solve problem sheets and past exam questions in Lean; I am convinced that it can play a role in making undergraduate teaching better (see forthcoming work of Iannone and Thoma, discussing my interventions so far).

Note however that much work remains to be done in Lean's maths library. A notable omission in complex analysis is Cauchy's integral formula (Lean is behind other systems in this regard), a notable omission in number theory is the basic theory of factorisation of ideals into prime ideals in algebraic number fields, and a notable omission in algebra is undergraduate representation theory. It is only a matter of time before these holes are filled, however, such is the pace of development right now. Enough commutative algebra was developed for undergraduates at my university to define schemes and to prove that an affine scheme is a scheme (that is, that the structure presheaf on an affine scheme is a sheaf). We have also formalised various tags in the Stacks Project [Sta18], an encyclopedic reference for modern algebraic geometry. Homological algebra is on the horizon, although we have made the design decision to set up everything within the context of abelian categories and are currently developing tactics to help with diagram chasing. By "we" I mean the collection of mathematicians and computer scientists who are collaborating to build Lean's mathematics library, an open source library of mathematics written in Lean which was started in 2017

and now contains over a quarter of a million lines of code. There are plenty of ways to get started learning the theory, and plenty of projects to work on. Within a few years I believe that the library will cover all of undergraduate pure mathematics. Another of my visions for the library is that it becomes a 21st century version of Bourbaki. Indeed much of Bourbaki's Topologie Génerale is now formalised in Lean, thanks to the efforts of (mathematician) Patrick Massot, building on earlier work of (computer scientist) Johannes Hölzl. Much of this was needed to formalise the definition of a perfectoid space. We are only just beginning to create an interface for the theory of local fields, however, so proving Scholze's tilting correspondence is still several years away. Later on, we will ask whether *proving* profound results like this is even the right thing to be doing.

3.5. **Learning to fly.** We have seen that ITPs, if taught by humans, are capable of picking up the basics of mathematics, and are (completely unsurprisingly) capable of checking proofs which stick close to the axioms of mathematics, such as most basic results in the first year of a mathematics degree. We have also seen an example of a higher-powered tactic, `ring`, which solves a problem for which working axiomatically would be a chore. How much further away from the axioms can we move?

The Yoneda lemma is a lemma in category theory where the idea is, in some sense, in the theorem statement, and the proof is to just chase the diagrams. It will come as no surprise to hear that Lean can find the proof by itself; a generic "follow your nose" tactic has been written by Scott Morrison at the Australian National University. Morrison, a mathematician, is an expert in designing high-powered tactics in Lean which can discover proofs of statements such as this. Morrison is currently concentrating on category theory, although the tactics he is developing are slowly being taken up by developers in other areas of mathematics in Lean. In particular, Lean already has some kind of primitive intuition for mathematics, although this intuition currently works better in some areas than others.

Gabriel Ebner, a computer scientist, is implementing a second approach, where versions of Lean goals are passed to an external system which specialises in solving first-order logic problems, and the external system passes back data which Lean then attempts to turn into a rigorous proof. Such techniques (calling external solvers which have been designed to solve certain types of logic problem) are referred to as "hammers" and they are already in active use in several other ITPs—Lean is still playing catch-up in this area. Indeed Sledgehammer [PB], developed by a team led by Larry Paulson, was an extremely successful tool for the Isabelle/HOL proof system. Note however that almost all experiments by computer scientists are restricted to the databases of formalised theorems which are currently available, and because available databases *still do not even cover all of undergraduate mathematics*, one imagines that there is still much room for improvement.

3.6. **Research-level mathematics.** Mathematics is vast, and growing quickly, and unless there is some kind of complete cultural change (which seems unlikely), one cannot imagine humans formalising their proofs in an ITP rather than typing them up into LaTeX any time soon (it would also make papers around four times longer; four seems to be the current "de Bruijn factor" representing the ratio between the length of a formal proof and the length of the corresponding LaTeX proof). Similarly it would take an extraordinary breakthrough in machine learning before computers can start to read human-written papers.

There have been some spectacular one-off achievements however. The two most obvious examples are [GAAea13] (a formalisation of the proof of the Feit–Thompson odd order theorem in Coq) and [HAB+17] (a formalisation of the Hales–Ferguson proof of the Kepler conjecture). These two formalisations came about for two rather different reasons. The Feit–Thompson work, led by Gonthier, was a demonstration that ITPs were capable of understanding a proof which is of Fields medal standard. The Kepler conjecture work, led by Hales, was an attempt to justify the correctness of the Hales–Ferguson proof, after the *Annals of Mathematics* chose to publish the paper proof whilst the referees claimed that they were only "99% certain" of its correctness. To give another recent example, the 2017 Ellenberg–Gijswijt proof [EG17] of the cap set conjecture, also published in the *Annals*, was formalised [DHL19] in Lean two years later by Dahmen, Hölzl, and Lewis. These examples show that it is now feasible for modern mathematics (or at least some of it) to be formalised in "real time." However the problem remains that there is no currently feasible method to turn the corpus of modern mathematics into a formally verified state.

3.7. **Formal abstracts.** But let us go back to the human PhD student, who, when they start their research, does not need to know all of the proofs of all of the theorems that they will be using. The important thing is that they know the *statements*. And teaching modern theorem statements to a computer is well within our grasp.

Tom Hales has proposed, in his *Formal Abstracts* project [Hal20], that the main theorem statements and definitions of mathematical papers be formalised in an ITP. *Proofs are not required*. Hales has chosen Lean for the system he wants to use, but there is no reason why other analogous projects cannot use other systems. One output of such a project would be a complex graph linking important mathematical concepts, and which grows over time. Hales imagines exploration tools enabling humans

to analyse this graph, like a Google Earth for mathematics. Search for mathematical theorems would suddenly become much easier—and the machine learning experts would finally have something to get their teeth into. Math Reviews and Zentralblatt contain human-written reviews of modern mathematical papers, and writing a formal abstract would in many cases be *easier* than writing a review, as only the theorem statements would be required, and once sufficiently many definitions are in the system, formalising theorem statements becomes easy. As more young mathematicians learn how to write mathematics in this kind of software, the possibility of making such a database seems to be becoming ever more real. Such a project seems to be a feasible way of digitising modern mathematics, and can be integrated into the hammers mentioned previously in order to make more powerful proof search. If PhD students are encouraged to formalise the statements of the results they are claiming to prove, and the statements of some of the theorems they use, then such a database could begin to grow very quickly indeed.

The Formal Abstracts project offers a real possibility of making an entirely new object—a digitisation of what the modern mathematician believes, and a chance for machine learning experts to see what they can make of it in a format which they can easily understand. If we, the mathematical community, build this database, then they, the computer scientists, will come. In fact, they are waiting.

Final Thoughts

I have spoken a lot about Lean, but there are many other systems available; Coq and Isabelle/HOL are also serious systems with a lot of mathematics in, and there are others too. The debate about which system is "best" is a complex one, involving technicalities about dependent types, strong normalisation, and subject reduction, and is beyond the scope of this article.

All of the systems could be used for educational purposes at the undergraduate level; I use Lean with my 1st year students and some of them love it. I run a weekly club, the Xena project, where I teach undergraduates (and they teach me) how to use Lean to do undergraduate and MSc-level mathematics in Lean. I would be extremely interested in making one of these systems more user-friendly whilst keeping it flexible enough to do a lot of undergraduate-level mathematics. An interesting related question is how to teach this topic to undergraduates—teaching material for undergraduate mathematicians is currently being developed by a team consisting of both mathematicians and computer scientists. The CoCalc website [SI20] enables teams of people to work on Lean code in a collaborative real-time environment and I have used this environment for training undergraduates to use Lean.

Whether or not they are used for teaching, it seems to me inevitable that these systems will one day change the way we do research. Writing proofs in these systems forces a human to clarify their thinking, and will perhaps lead them to discover better abstractions. Long proofs, too long for any single human to comprehend every last detail of, are becoming more commonplace. Can computers help to check these? Looking further ahead, I believe that one day our subject will experience a true culture shock, where a computer announces a proof of a conjecture which humans are interested in, and if the argument is sufficiently deep, then the computer might not be able to explain their argument using a language that humans can understand. But *well* before that happens, computers will become tools which we can use to search for and verify lemmas in all areas of pure mathematics, and the sooner an area is taught to one of these systems, the sooner computers will be able to help. The Lean Prover community website at `https://leanprover-community.github.io/` and the Lean chatroom at `https://leanprover.zulipchat.com/` are two places to start, if people have any questions about what some of us believe will be the future of mathematics.

I thank the members of the Lean Prover community, and the anonymous referee, for their feedback on an earlier version of this article.

References

[BCM20] Kevin Buzzard, Johan Commelin, and Patrick Massot, *Formalising perfectoid spaces*, Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020, pp. 299–312.

[DHL19] Sander R. Dahmen, Johannes Hölzl, and Robert Y. Lewis, *Formalizing the solution to the cap set problem*, 10th International Conference on Interactive Theorem Proving, LIPIcs. Leibniz Int. Proc. Inform., vol. 141, Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2019, Art. No. 15, 19. MR4008934

[EG17] Jordan S. Ellenberg and Dion Gijswijt, *On large subsets of $\mathbb{F}_q^n$ with no three-term arithmetic progression*, Ann. of Math. (2) **185** (2017), no. 1, 339–343, DOI 10.4007/annals.2017.185.1.8. MR3583358

[GAAea13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, and et al., *A machine-checked proof of the odd order theorem*, Interactive Theorem Proving, Lecture Notes in Comput. Sci., vol. 7998, Springer, Heidelberg, 2013, pp. 163–179, DOI 10.1007/978-3-642-39634-2_14. MR3111271

[GM05] Benjamin Grégoire and Assia Mahboubi, *Proving equalities in a commutative ring done right in Coq*, Theorem Proving in Higher Order Logics, Lecture Notes in Comput. Sci., vol. 3603, Springer, Berlin, 2005, pp. 98–113, DOI 10.1007/11541868_7. MR2197007

[HAB+17] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang

Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller, *A formal proof of the Kepler conjecture*, Forum Math. Pi **5** (2017), e2, 29, DOI 10.1017/fmp.2017.1. MR3659768

[Hal20] Tom Hales, *Formal Abstracts Project*, 2020.

[Maz14] Barry Mazur, *Is it plausible?*, Math. Intelligencer **36** (2014), no. 1, 24–33, DOI 10.1007/s00283-013-9398-0. MR3166990

[PB] Lawrence C. Paulson and Jasmin Christian Blanchette, *Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers*, The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011, pp. 1–11.

[SI20] SageMath Inc., *Cocalc collaborative computation online*, 2020. `https://cocalc.com/`.

[Sta18] The Stacks Project Authors, *Stacks Project*, 2018.

[Tao09] Terence Tao, *There's more to mathematics than rigour and proofs*, 2009.

Kevin Buzzard

Credits

Author photo is courtesy of Kezia Buzzard.