Automated reasoning for the working mathematician

Jeremy Avigad

July 17, 2019

I have been invited to give a talk in September to two colocated conferences, *Frontiers of Combining Systems* (FroCoS) and *Tableaux*. The focus of the first is "research on the development of techniques and methods for the combination and integration of formal systems, their modularization and analysis," and the focus of the second is "research on all aspects—theoretical foundations, implementation techniques, systems development and applications—of the mechanization of tableaux-based reasoning and related methods." I am flattered by the invitation, but also anxious as to what I can possibly say that might be of interest to researchers in those areas. I gather that a substantial fraction of the audience will consist of people developing automated reasoning tools, and I would like to say something about the sorts of tools that might be useful for those of us interested in formalizing mathematics.

With that in mind, I proposed the title above and the following abstract:

> The mathematical literature is filled with minor errors and imprecision, and interactive proof assistants offer hope of making mathematics more reliable and exact. Given the gap between an informal proof and a formal derivation, one would expect automated reasoning tools to play a key role in formally verified mathematics. But this expectation has not been borne out in practice. Despite technological advances, automated reasoning is far from central to the field, and many of the most impressive accomplishments to date have used surprisingly little automation. The use of automated reasoning tools in mathematical discovery has been even more limited. In this talk, I will do my best to make sense of this state of affairs and offer guidance towards developing useful mathematical tools.

The problem is that I really don't know how to make good on this promise, or even fill an hour-long talk without embarrassing myself. Here are some things I am planning to do:

- Reflect on my own experiences with automated reasoning.
- Ask as many people as I can who have formalized nontrivial mathematics to reflect on their experiences, report on what automation they have found helpful (if any), and speculate on what might make their lives better.
- Work through some examples to come up with concrete examples of places where better automation might make a difference.

This document is a start on the first and a preliminary report on the second. I am hoping it will prompt discussion and more input.

The tension between automated reasoning and interactive theorem proving has long been understood. Automated reasoning offers push-button solutions, but of limited scope: most interesting problem domains are undecidable, search is intractable, and so on. Automated reasoning does well on large, homogeneous problems that can be formulated using restrictive means, but typically fails on more open-ended reasoning problems. In contrast, there are no restrictions on the reach of interactive theorem proving, in principle: anything that can be stated and proved in mathematical terms can be proved formally. It is only a matter of how much user interaction is required, and, currently, even simple proofs require inordinate amounts of work.

I think most people working in one of these two areas agree that some kind of synthesis is needed, with high expressivity and the potential for user interaction when appropriate, but with as much tedium relegated to the machine as possible. But we are not there yet, and despite some good efforts, the ATP and ITP communities are largely segregated.

In these notes, I will focus on ATP and ITP for mathematics, that is, support for mathematical reasoning and proof. The boundaries between that and hardware and software verification are not sharp, and many of the concerns are common to both. But there are also significant differences between the two domains, which therefore offer different challenges. It is possible that the kinds of problems that arise with respect to hardware and software verification are more amenable to automation. In any case, I will stick to verification of mathematics, because that is what I know best.

I will also focus on automated support for mathematical verification, rather than mathematical discovery. The latter also holds great promise, but the uses of automated reasoning tools to date have been few and far between, and the most useful thing I can do to address the issue is point you to Marijn Heule's list of publications and encourage you to talk to him. Once again, I will stick to automated support for verification only because that is what I know best.

1 My personal story

I started playing around with Isabelle around 2002. I was soon able to formalized the law of quadratic reciprocity, together with Adam Kramer, an undergraduate at Carnegie Mellon at the time, and David Gray, an MS student. Encouraged by this first success, I decided to formalize the prime number theorem. Another undergraduate student, Paul Raff, joined in, and the proof was completed in September of 2004.[1] The code has not been maintained, but the original project page is still online.[2]

Many of the challenges in formalizing the PNT stemmed from the fact that Isabelle was still young and there were gaps in the libraries. (In fact, we managed to prove the law of quadratic reciprocity with an incorrect definition of primality in the library. We were later able to show that, with that definition, there were no integer primes. Fortunately our proof survived the easy fix.) I chose to formalize a version of the Selberg-Erdős elementary proof because the library did not even have a theory of integration, let alone complex analysis. The formalization required filling in basic

[1] Adam went on to earn a degree in social psychology, joined Facebook (https://www.apa.org/gradpsych/2011/01/kramer), and ran a controversial experiment there (https://www.businessinsider.com/adam-kramer-facebook-mood-manipulation-2014-6). David Gray went on to do a PhD in ethics, and is a member of the faculty of Carnegie Mellon Qatar (https://www.cmu.edu/dietrich/philosophy/people/faculty/david-gray.html). Paul Raff earned his PhD in Mathematics from Rutgers, worked at Amazon for a while, and now works for Microsoft (https://www.linkedin.com/in/ptotheraff).

[2] http://www.andrew.cmu.edu/user/avigad/isabelle/NumberTheory/index.html

facts about real and integer arithmetic, deriving rules for finite cardinalities, sums, and products, and so on.

Despite these gaps in the fundamentals, the automation was remarkably mature. Isabelle had, and still has, a very good term rewriter (`simp`), variants of tableau provers and automated reasoners (`auto`, `force`, `clarify`), and a good procedure for real and integer linear arithmetic (`arith`). My memory now is that I used these a lot, and that memory is corroborated by the proof scripts. For example, the last file in the PNT formalization, `PrimeNumberTheorem.thy`, is about 4,000 lines long, corresponding to about five consecutive textbook pages (cf. the link in the previous footnote). My editor tells me that `simp` is called 390 times, `auto`, 51 times, `force`, 277 times, `clarify`, 69 times, and `arith`, 246 times.

The proofs weren't pretty, and I did not make the effort to update the scripts to the next Isabelle release. Over the new few years, however, I did clean up many of the fundamental results and add them to Isabelle's core library, and I am proud of the fact that many of these contributions survive to this day.

After the prime number theorem, I eased up on formalization for a while, but came back to it in full force in 2009. I had a sabbatical from Carnegie Mellon during the 2009–2010 academic year, and spent it with Georges Gonthier and his *Mathematical Components* project in France. It was a great sabbatical: it gave me the opportunity to learn French, Coq, the SSReflect proof language, and finite group theory, all at the same time.

A few months before the sabbatical began, I visited Georges at Inria, and discovered that we had widely divergent views on the right way to go about formalization. At the time, I was moving towards a declarative style of writing proofs using Isabelle's *Isar* proof language, which yields proof scripts that are verbose but more readable. While SSReflect has declarative elements for structuring long proofs, the language is designed rather to be an very efficient tactic language. The goal is not to make proofs readable, but to allow users to carry out fundamental operations with a few well-chosen keystrokes.

The *Mathematical Components* library was built with very little automation beyond a simple `done` tactic that did a few obvious things to try to close a goal. Georges had a deep mistrust of black-box automation, which he worried was nondeterministic and wouldn't scale. I had always felt that the most robust proof scripts would look like this:

```
have A, by auto,
have B, by auto,
have C, by auto,
...
```

Avoiding mention of specific rules and theorems would allow refactoring libraries, changing theorem names, and so on, without breaking scripts. As long as `auto` remained smart enough to fill the gaps, everything would be o.k. But this is precisely what Georges worried about, namely, the reliance on heuristic black box procedures whose behavior is unspecified and constantly changing. From his point of view, the most robust proof scripts would be fully detailed and explicit.

Georges would probably object to the claim that *Mathematical Components* does not rely on automation, since it *does* rely on computation in the underlying logical framework. In systems like Martin-Löf type theory and the Calculus of Constructions, theorems like $2 + 2 = 4$ and $x + 0 = x$ follow immediately from the reflexivity of equality (in the latter case, assuming addition is defined by recursion on the second argument). This is so because the kernel proof checker can (and must) reduce both sides of the equation until they match. Similarly, the kernel will unfold definitions

and reduce as necessary to match a theorem to a goal. The *Mathematical Components* library aggressively takes advantage of these features. Georges felt that *that* type of automation is o.k., because it is deterministic, and the behavior is (at least ideally) fully specified by the logical foundation.

That never sat well with me. I have reconciled myself to the fact that sometimes relying on definitional reduction is needed for type checking, but I don't like using it for theorem proving. It somehow seems *wrong* to write a proof that only works on the assumption that boolean "or" is defined by recursion on one argument or another, or that concept A unfolds to a definition in terms of concept B, so that you can apply a theorem about B to a goal that mentions only A. Doing these things breaks abstraction and modularity. In mathematics, it doesn't matter whether we define the reals in terms of Dedekind cuts or Cauchy sequences. Once we have that the reals form a complete ordered field, our proofs should respect that interface, and we would not expect a theorem about manifolds to unfold the definition of a real numbers. So where my preferred style of proving aims to respect abstraction barriers, Georges' approach makes violating them an art unto itself.

Georges challenged my assumptions in other ways as well. Since working on the PNT, I had felt that the goal of interactive theorem proving was to figure out how to enable users to write mathematics as much as possible the way it appears in textbooks and papers. After all, mathematical language has evolved the way it has for good reasons, and I felt that our goal should be to capture all the benefits of mathematical language and style in formal terms. For Georges, however, the main challenge was always to figure out how to rework the mathematics to make it amenable to formalization. This often involved using novel representations, such as his use of hypermaps in his proof of the four-color theorem, which was of independent mathematical interest.

Maybe the best way to describe the difference between our ways of thinking is that I felt that the key to interactive theorem proving is to make the computer science look more like the mathematics, whereas Georges felt that the strategy that is called for is to make the mathematics look more like computer science. Since then, I have gravitated somewhat closer to his point of view. Like Georges, I recognize that we mathematicians have a lot to learn from computer scientists about finding good representations, and there are wisdom to be had in understanding how to choose the right ones. But at the same time, the need to focus on the low-level details of the representations often distracts from the real mathematics. In an ideal world, we should be able to read and appreciate a formal proof without being excessively conscious of the theorem prover and its formal library. These are artifacts of formalization, not essential features of the mathematics. The computational implementation is the medium, not the message.

Anyhow, I came back from France feeling as though I had done enough formalization to last a lifetime, and ready to move on to other things. But a year later, a bright undergraduate student, Luke Serafin, took my freshman seminar on the history and philosophy of mathematics in the fall and then expressed an interest in formalization in the spring. We settled on formalizing the central limit theorem in Isabelle, which became an on-again, off-again project for the next couple of years.

The most notable thing about the project was how un-notable and routine it was. We mapped out the chapters in Billingsley's textbook and simply worked through them page by page. The project tested the breadth of Isabelle's libraries and brought together a number of components—measure theory, topological notions, everyday calculus, and the theory of characteristic functions, essentially a form of Fourier analysis. As always, we had to add little bits and pieces that were missing, for example, extending the Lebesgue integral to functions from the reals to the complex numbers. Johannes Hölzl, who had developed a good deal of the analysis libraries, was instrumental in helping out, and eventually joined the project. He later generalized integration to the Bochner

integral, which handles functions to any Banach space, including both the reals and the complex numbers.

All this was about ten years after working on the PNT, but I that found `simp`, `auto`, and `arith` were still mainstays. I also called Isabelle's *Sledgehammer* occasionally, but it was not a big help. Every once in a while, when faced with a goal that required chaining one or two facts from the library, Sledgehammer would solve the goal, and save ten minutes or so of looking through browser pages. But most of the time, it just failed. At the time, Tobias Nipkow described his experience in a similar way, as did Johannes a few years later. But I will say more about this in the next section.

While working on the CLT, I was impressed by Johannes' style of writing structured proofs, essentially the `have A, by auto, have B, ...` format I described above. You can see it at play, for example, in his formalization of the Bochner integral.[3] I learned his method by looking over his shoulder on a visit to Munich. He would start by writing nicely structured Isar sketches, and then fill the gaps by hacking with tactics. In particular, he would let `auto` make partial progress, and then help it along when it got stuck. When he finally succeeded in filling all the gaps, he would turn the proof into a one-liner by calling `auto` or `simp` with enough hints to enable them to finish off the goal.

The other thing that pulled me back to interactive theorem proving at the time was the rising interest in Homotopy Type Theory. Carnegie Mellon was (and still is) a center of activity, driven in large part by the enthusiasm of Steve Awodey, a colleague of mine. I formalized properties of homotopy limits in Coq with Chris Kapulkin and Peter Lumsdaine, started the Lean 2 HoTT library during a visit to Microsoft in 2014, followed along while Floris van Doorn and others did impressive things, and contributed a bit to a construction of spectral sequences. But none of that used any automation to speak of. (If I remember correctly, Peter and Andrej Bauer wrote some Ltac tactics to automate some path constructions, but they were slow and not very useful in practice.)

When Leondardo de Moura decided to develop a new theorem prover in 2013, I was getting tired of interactive theorem proving, and looking forward to learning more about automation. But Leo convinced me that even if one cares about automation, one should build it on top of a secure, expressive foundation— not just to ensure that the automation was reliable, but to have any sense at all of what the results *mean*. The aim of the Lean project was

> ... to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs.

The Lean project web page still proclaims this goal.

To date, most of Leo's efforts (and those who have worked with him on the prover itself) have gone into the infrastructure and the automation used in the elaboration process, rather than automation for theorem proving. Here the word *elaboration* refers to the process of taking user input and inferring and then inserting all the information that is left implicit. There isn't a sharp boundary between filling in this information and proving theorems, since the latter also involves filling in bits of information that users would prefer to leave implicit. But we usually think of elaboration as being closer to parsing than automated reasoning. For example, Lean relies on mechanisms for type class inference, basically prolog-like search, that are similar to ones used by programming languages like Haskell. On the automated reasoning side, Lean does have a built-in term simplifier that is similar to Isabelle's `simp`. Moreover, because its logical foundation has a computational

[3] https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/HOL-Analysis/Bochner_Integration.html

interpretation, Lean can be used as a programming language. One of Leo's best decisions was to make it possible to use that programming language as a *metaprogramming* language, allowing users to write automation for Lean in Lean itself.

Lean is a really wonderful system to use. The syntax is clean, and the language is expressive and powerful. As recently as three years ago, I used to complain that one could count the number of mathematicians using interactive theorem provers on the fingers of one hand. (Mathematical logicians don't count; I meant mathematicians using interactive theorem provers to do mathematics, not mathematicians interested in the theory behind them.) But within the last few years a lively community of mathematicians have become avid Lean users, and Lean's external library, `mathlib`, has been growing rapidly. What is most encouraging, and even touching, is to see how much they are enjoying themselves. Lean's Zulip channel sees dozens (and sometimes hundreds) of messages every day. People love using the system, experimenting with it, talking about it, and even complaining about it. When Kevin Buzzard, a number theorist, gives a talk about Lean and says "it changed my life," nobody doubts his sincerity.

It is interesting that Lean generates this enthusiasm without substantial automation. Gabriel Ebner has written a resolution theorem prover in Lean's metaprogramming language, showing that it can be done, but it is currently too slow to be useful. A few years ago I implemented a tableaux theorem prover, `finish`, on the model of Isabelle's `auto`, and users occasionally try it out. But, from the metaprogramming language, I had to rely on some internal procedures Leo implemented experimentally to carry out term instantiation, and they were not flexible enough for my purposes. Lacking the will and expertise to hack Leo's code or write my own, I set that aside as well. Seul Baek is experimenting with various ways of implementing automated procedures in Lean, but his methods are still in prototypical form.

It turns out, however, that *small scale* automation has made a much bigger difference. When Kevin complained about how hard it was to do numeric and algebraic calculations in Lean, Mario Carneiro implemented the `norm_num` and `ring` tactics, and these are used regularly. Scott Morrison has written a `tidy` tactic that tries to close a goal by trying a battery of straightforward moves. Rob Lewis has implemented a procedure for linear arithmetic. I find a small `convert` tactic to be useful: it applies a given theorem to the goal, detects the mismatches, and leaves them as equational goals. Whenever a user suggests a useful tactic on Zulip, thanks to the metaprogramming language it is usually not long before someone in the community implements it.

But most of the time when I sit down to use Lean, I make use of very little automation at all. Occasionally I call the simplifier, but even in situations where `simp` is helpful, I tend to favor doing an explicit sequence of rewrites by hand. Lean has a theorem-naming convention that makes it easy to guess theorem names, or at least a prefix thereof. The VS Code editor (as well as Emacs) supports tab compleition, which makes it easy to locate theorem by typing a prefix and then choosing from a list of matches. This is *extremely* useful, and I use it all the time. I find it ironic that, more than fifteen years after I got started in this business, the automation that I find most useful in day-to-day theorem proving is tab completion.

2 Reports from others

Before trying to figure out what to make all of this, let me share some anecdotes and things I have heard from others. I hope the people I mention here will speak up if I am misremembering our conversations or in any way mischaracterizing their views.

Larry Paulson's Sledgehammer tool for Isabelle is a remarkable achievement. Larry once told me that it made it easy for him to port theories from the HOL Light library without knowing much about the mathematics at all. The proof scripts bear him out; for example, consider his port of the Cauchy Integral theorem.[4] The file contains 155 invocations of `metis`, the internal resolution theorem prover that is used to reconstruct proofs from data returned by Sledgehammer.

Tobias Nipkow did an evaluation of Sledgehammer in 2010[5] but a lot has changed since then. Jasmin Blanchette has contributed a tremendous amount to the effort. His 2016 survey [6] contains a wealth of data and examples, and he has presented additional compelling examples of Sledgehammer's effectiveness in talks.

But it is sometimes hard to interpret the quantitative data given in evaluations of Sledgehammer, which often test the tool on existing libraries, essentially determining the extent to which the tool can replace proofs (or parts of proofs) carried out by hand. Anecdotal evaluations, like Larry's and ones that Jasmin have given in talks, seem more to the point: we want to know how useful the tool is in day-to-day formalization. And there assessments are mixed. I have already relayed my own impression that it was not terribly useful in formalizing the central limit theorem. I once asked Johannes about his experiences, and they were similar. And, more recently, I talked about this briefly with Manuel Eberl at a meeting in Edinburgh. Manuel had recently reported on a very elegant and impressive formalization of analytic number theory, including the prime number theorem, Dirichlet's theorem on primes in arithmetic progression, properties of the ζ function, and much more—in fact, the lion's share of Apostol's textbook on analytic number theory. Manuel told me that he did not use Sledgehammer very much, and perusal of his proof scripts bear this out.[7] They are lovely examples of structured proofs, and look similar to those of Johannes: they are eminently readable, with `auto` and `simp` providing most of the justification. In two files whose links are given in the previous footnote, I found only one instance of `metis`, but even that does not seem to come from a call to Sledgehammer, since it uses only a few local hypotheses from the same file.

There is a small but growing literature on the use of machine learning techniques in automated theorem proving, with notable contributions by Josef Urban, Cezary Kaliszyk, Christian Szegedy and his group at Google Mountain View, and others.[8] This literature often offers benchmark measures of success based on the ability to automatically reinstate proofs removed from a hand-curated library. Here, too, it is hard to guess what this means for the working mathematician, i.e. how it translates to everyday use. It would be helpful to hear of experiences with machine-learning tools in practice.

I can offer some speculation as to the mismatch between Sledgehammer benchmarks and the perceived utility to everyday formalization. When I think of what it takes to formalize a theorem, the hard part is often getting the high-level details right, so that filling in the low-level details is

[4] https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/HOL-Analysis/Cauchy_Integral_Theorem.html

[5] http://www21.in.tum.de/~nipkow/pubs/ijcar10.pdf

[6] https://people.mpi-inf.mpg.de/~jblanche/

[7] Here are two examples, the first a development of properties of Dirichlet characters, and the second a proof of Dirichlet's theorem:

https://www.isa-afp.org/browser_info/current/AFP/Dirichlet_L/Dirichlet_Characters.html
https://www.isa-afp.org/browser_info/current/AFP/Dirichlet_L/Dirichlet_Theorem.html

[8] I apologize for the lazy scholarship, but I can't do better than the literature review here: https://arxiv.org/abs/1904.03241.

possible at all. We usually start by writing down some definitions and proving some basic facts. When we define X in terms of Y, we usually end up having to browse through the library for Y, to figure out what is there and figure out the library designer's intended idioms. Often, we have to add a few more facts to the library for Y. Sometimes we adjust the definition of X slightly to make the proofs go through more smoothly. Sometimes we decide to jettison the original definition in terms of Y and replace it by a definition on terms of Z, because Z's infrastructure is better suited to our purposes. We often start with long, hackish proofs, and then tinker and polish the library until all the concepts are in place and everything fits together nicely.

That is what the sledgehammer benchmarks see: the sanitized versions of the proofs where everything goes through smoothly. It's not surprising, then, that sledgehammers do well. After all, we already did all the work to set it up; then it is just a matter of knocking the lemmas down, like a row of dominoes. In fact, that is also the point at which we don't really need a sledgehammer. By the time we get to that stage, all the local definitions are fresh in our minds, as well as the necessary facts from the background library. (Another consideration is that, when building a library, one often proves some hard facts, and then provides a few easy variations and combinations that might be helpful to users. It is hard to gauge what portion of the successes are the really hard parts, and what portion are the easy variations.) What we really need are reports from people in the field, of the following form: "I tried to prove theorem X, I called Sledgehammer on these obvious goals, and these are the ones it got." In that sense, the examples in Jasmin's talks are more compelling than any benchmark. What we really want are dozens of examples of that sort, and not just the successes, but also the half-successes and the crash-and-burn failures, with realistic data as to how these are distributed. One modest success every three or four attempts may make it worthwhile to make calling a tool part of the regular workflow. One every twenty or so means that we will probably do better to focus on other approaches.

I am very struck by the fact that many of the best formalizers I know use almost no automation at all. In fact, I would say that this is true of *most* of the best formalizers I know. A number of years ago, I asked John Harrison about his use of automation in the HOL Light library, and I asked him about it again after a talk he gave at the *Big Proof* meeting in Edinburgh a few weeks ago. Both times he told me that he uses very little automation. He described this as a personal preference, namely, the desire to have "fine control" over his proofs. It is not for the lack of availability: John himself has implemented a first-order tableau prover for HOL Light, a term rewriter, decision procedures for real and algebraically closed fields, and special purpose procedures for reasoning about rings, Hilbert spaces, and so on. He is the author of the *Handbook of Practical Logic and Automated Reasoning*, which I highly recommend as an introduction to automated reasoning, especially the kind that bears on interactive theorem proving. John has also formalized a tremendous amount of interesting, elegant, and important mathematics. So it is especially notable that he doesn't make strong use of automated reasoning tools in his own formalizations.

Mario Carneiro's work on *Metamath* is another striking example. Metamath is the assembly language of interactive theorem proving, and Mario has got me to appreciate what a remarkable system it is. Its library includes 71 of the 100 theorems on Freek Wiedijk's list,[9] second only to HOL light and Isabelle. The library, which includes those 71 theorems and thousands more, lives in a single ASCII file, which can be checked for correctness in about three seconds on an unexceptionable laptop. The logic is so simple that writing a checker is an easy programming exercise, and there are dozens available. And despite the simplicity of the logic and the bare-metal interface, very little

[9] http://www.cs.ru.nl/F.Wiedijk/100/index.html

automation is used to write the proofs. Mario doesn't seem to have missed it at all when formalizing the prime number theorem, Dirichlet's theorem, and many of the other entries on Freek's list.

Even Johannes, the master of the "`have A, by auto, have B, by auto`" style of proofs, seemed eager to switch from Isabelle to Lean. Johannes has made substantial contributions to Lean's libraries, and seems to revel in marrying nice algebraic generalizations with computer-sciency abstractions. He has built the set theory, topology, and analysis libraries on top of lattices, filters, functors, and monads. This is where the expressivity of dependent type theory shows itself, and a lot of his work depends on the ability to treat algebraic structures as first-class objects. It seems that this expressivity was more important to him than having automation.

Similar considerations motivated Sébastien Gouezel, a mathematician who studies dynamical systems, to switch from Isabelle to Lean. On his web page, he writes:

> Out of curiosity, I have given a try to several proof assistants, i.e., computer programs on which one can formalize and check mathematical proofs, from the most basic statements (definition of real numbers, say) to the most advanced ones (hopefully including current research in a near or distant future). The first one I have managed to use efficiently is Isabelle/HOL. In addition to several facts that have been added to the main library (for instance conditional expectations), I have developed the following theories.
>
> However, I have been stuck somewhat by the limitations of the underlying logic in Isabelle (lack of dependent types, making it hard for instance to define the p-adic numbers as this should be a type depending on an integer parameter p, and essentially impossible to define the Gromov-Hausdorff distance between compact metric spaces without redefining everything on metric spaces from scratch, and avoiding typeclasses). These limitations are also what makes Isabelle/HOL simple enough to provide much better automation than in any other proof assistant, but still I decided to turn to a more recent system, Lean, which is less mature, has less libraries, and less automation, but where the underlying logic (essentially the same as in Coq) is stronger (and, as far as I can see, strong enough to speak in a comfortable way about all mathematical objects I am interested in).

You can see examples of Sébastien's contributions to the Lean library—in particular, his theory of the Gromov-Hausdorff distance—online.[10]

I could go on. Floris van Doorn, a recent PhD student of mine, has done lots of clever and impressive work in Homotopy Type Theory, where automation simply doesn't play a role. He and Jesse Han, a very impressive PhD student of Tom Hales, recently formalized Cohen's proof of the independence of the continuum hypothesis. The paper describes a useful bit of small-scale automation in Secton 3.2.[11] But I have already noted that most proofs in Lean are carried out without much automation, and the proofs in their repository[12] are no exception.

3 Reflection

Where do we stand? When talking about the formalization of the prime number theorem at a European *Types* meeting in 2005, I predicted that in thirty years, formalized mathematics would

[10] https://github.com/leanprover-community/mathlib/blob/master/src/topology/metric_space/gromov_hausdorff.lean

[11] https://arxiv.org/pdf/1904.10570.pdf

[12] https://github.com/flypitch/flypitch

be commonplace.[13] Despite the time and effort require to formalize even trivial arguments for the PNT, I did not see any conceptual hurdles, and argued that better libraries, better automation, and better infrastructure (for example, database management and search) would inevitably put formalization within reach of the average mathematician.

But I did think that automation would be an integral part of the story. What I had in mind, in particular, was domain-general automation to fill in those steps that, in an ordinary pen-and-paper proof, are deemed entirely obvious and in no need of any sort of justification. And now, almost halfway to the thirty-year benchmark, I have to admit that there hasn't been much improvement in that respect.

One might reasonably conclude that Georges was right, and I was wrong: what we need is not better black-box search procedures, but better languages, formally represented concepts, and libraries that encode our mathematical expertise in ways that scale to contemporary mathematical practice. In deference to my earlier claims, one could, perhaps, concede that automation might be useful to novices, who have yet to learn a system's libraries and idioms. But, one might argue, formalization inevitably requires expertise and familiarity with the libraries, and once we know a system well enough to do what we want to do, automation doesn't help.

Another concession to my claims is to admit that *domain-specific* could be useful, automation that carries out tasks that are well-defined and deterministic but tedious. Mild uses of a term rewriter to carry out obvious simplifications could fall into this category, as well as numeric calculations, ring calculations, and so on. Isabelle has procedures for establishing continuity and measurability of functions by chaining through the obvious rule applications. Kevin Buzzard has made a good case that we crucially need "transfer" procedures to mediate between representations of mathematical objects that mathematicians often identify without even realizing it.

But even though I am clearly on the defensive here, I am not ready to give up my claim that we need better domain-general automation. I don't think the current situation is tenable, or that interactive theorem proving will ever have a broad mathematical audience without better automated support. The problem is that interactive theorem proving requires us to focus on representational details that are incidental to the mathematics, and distract from it, even though they are sometimes interesting in their own right. The mathematicians using Lean right now are showing a remarkable tolerance towards learning the syntax of type theory, mastering the ins and outs of type class inference, appreciating the importance of finding the right encodings, and refactoring theories when a chosen encoding turns out to be less than ideal. But in doing so, they are warming to the interests of logicians and computer scientists. That is not a bad thing; but it is hard to make the case that this is making them better mathematicians, or that the mathematical community as a whole would do better to cultivate the same interests.[14]

I am not simply trying to reinforce traditional methodological distinctions between mathematics and computer science. I do think it is important for those two subjects to interact, and I am absolutely certain that applications of computer science to mathematics and vice versa will, in the

[13] My reseasoning was as follows. I was born in 1968, and I could remember, roughly thirty years earlier, my father bringing home the latest video game console system as a present for my brother and me. It was a version of Pong, which we immediately hooked up to our black-and-white television through the antenna connector. By that measure, thirty years seemed like an infinite amount of time, and so my prediction felt safe. After my talk, someone in the audience shrewdly pointed out that there is a much bigger market for video games than formalized mathematics. I guess it is a good thing that I am not trying to make a living as a technology visionary.

[14] Libraries of formal mathematics are like libraries of code in some ways, but different in others. Shaving a few lines off a block of computer code in a tight loop can result in substantial performance gains, whereas the corresponding task may not make a mathematical theorem easier to use or maintain. So we should be mindful of the extent to which a computer science sensibility is appropriate to the mathematics.

long run, fundamentally change our understanding of what it means to do mathematics. But I also recognize that mathematicians and computer scientists face distinct challenges. The miracle of mathematics is the way it has been able to develop conceptual innovations through the centuries that extend our cognitive reach. Mathematical exploration pushes the boundaries of what we can think about with clarity and precision. The challenge, then, is to develop abstractions that can help us solve hard problems and think better. Finding representations of those abstractions that make them amenable to formalization is important, but it is not the end goal. Formality is the medium, not the message.

Interactive theorem proving *can* help us do mathematics better, but as soon as it becomes an end in and of itself, it ceases to be mathematics. And, right now, the focus on formal syntax, theorem names, and library organization makes interactive theorem proving more of an end in and of itself. Formalization has its charms: it can be exhilarating to find just the right definitions and lemmas, and see all the pieces come together just right to find a clean and efficient proof of a hard theorem. But this is not the same as doing mathematics, and spending more time formalizing means spending less time on the mathematics. Given the time and effort that formalization requires, we can forgive the vast majority of the mathematical community for being reluctant to join the cause.

4 Recommendations

In my abstract, I promised to "offer guidance towards developing useful mathematical tools." The best strategy I can think of to provide substantive data is to experiment by formalizing some straightforward theorems in Isabelle. I will choose some examples that are roughly at the level of undergraduate homework assignments, write human-level structured proofs in Isar, and then see what it takes to fill in the gaps, relying both on internal automation and Sledgehammer as much as possible. The fact that I have been away from Isabelle for a number of years now means that the names of theorems and the details of the library are not at my fingertips, nor do I remember all the power-user tricks and idioms. So I will be working as a user familiar with the tool but not immersed in it.

I am singling out Isabelle here its internal automation and its connection to external theorem provers are so good. I am having a hard time thinking of any other system for which such a test even makes sense; perhaps PVS is one.[15]

I am open to the fact that, having set the terms, I may come to the conclusion that Isabelle's automation now really *is* capable of filling in obvious inferences in a satisfactory way. The fact that power users like Johannes and Manuel don't use Sledgehammer very much is not in and of itself a problem, if more casual users can get by equally well with Sledgehammer's support. In that case, perhaps all that is needed to bring interactive theorem proving to the mathematical mainstream is to combine Isabelle's automation with the ability of Coq or Lean to handle algebraic and structural reasoning. That would be a nice conclusion—it provides a straightforward recommendation and a clear target.

(I know that some people reading this will object to the implicit presupposition that dependent type theory is the best way to support algebraic, structural reasoning. So let me ward off those complaints by saying that I am *not* presupposing that. For all its drawbacks, dependent type theory

[15] The *Naproche* system (https://korpora-exp.zim.uni-duisburg-essen.de/naproche/) allows one to write down small sets of axioms and hypotheses, write proofs in a controlled natural language, and ship each inference in a proof to back-end automated reasoners. It does not sit on top of a large mathematical library, but it can be used to experiment with curated lists of background facts.

has the advantages of letting users be more concise in their input by providing regimented means to infer a lot of information that we often leave implicit, and providing means of detecting and reporting low-level errors, like sending a function the wrong number or wrong kinds of arguments. These features are absolutely necessary, and a think newcomers often underestimate how hard it is to get a system to provide the functionality we need. But there are a number of set-based systems on offer now, including Mizar (and a Mizar implementation in Isabelle by Cezary Kaliszyk, Karol Pak, and Josef Urban), implementations of set theory in Isabelle by Larry Paulson and Bohua Zhan, and Metamath. I'd like to avoid a flame war here, and I won't speculate as to what will work best in the long run. I would love to see mathematicians happily using a system based on set theory.)

In the meanwhile, off the top of my head, I can make some general structural observations. It seems to that the key to success is to have strong interaction between the community of users and the community of tool developers. Benchmark repositories like TPTP and SMTlib are a mixed blessing. It is undeniable that these have spurred a lot of good work, and offer clear measures to evaluate progress. But they can also lead to disconnect between developers of automated tools and their users among the ITP community. An overly slavish adherence to benchmarks is a kind of institutionalized version of "dumpster diving," going through piles of stuff that others have thrown out in the hopes of finding something valuable or useful, without performing a genuinely useful social function. Even with the best intentions, the way that benchmark data is generated may lead to systematic bias away from the real goals. And there is always a danger of overfitting, i.e. developing tools that are well-tuned to benchmark problems but do not transfer to other domains.

Interactive theorem proving, since it involves a complex interaction between users and computers, is a messy, heuristic practice. It is often hard to quantify the pain points. Ideally, we want ATP developers sharing offices with ITP users, looking over their shoulders, seeing how they muck around, and listening to them moan and complain. We in the ITP community can't lay all the blame at the feet of ATP developers, since it is up to us to provide better benchmarks and data. Once we succeed in proving a theorem—once we know how to do it—we tend to clean up our proofs and only show the world what, in hindsight, we should have done from the start. We need better reports of what we would *like* to do, and detailed accounts of what goes wrong when we try to do it.

In fact, I believe that Isabelle's successes stem from the fact that Larry Paulson and Tobias Nipkow have led system development in tandem with the development of the libraries. Some of the many students and postdocs who have worked with them over the years have focused on decision procedures, tactics, and automation, while others have focused on formalization, many have done both. Which means that there was constant give-and-take, so that formalizers knew what to ask for, and developers knew what was wanted. During my Isabelle years, I often wrote Larry and Tobias with questions, problems, and suggestions. At least a couple of times I sent them files with examples of inferences that I wished were more automatic, and I remember them coming back with annotations from Larry, and Tobias implementing suggestions the next time he tinkered with the simplifier. Larry once told me that he started his *Metitarski* project in response to complaints I had about the difficulty of proving straightforward real-valued inequalities when I was proving the prime number theorem. (Rob Lewis and I also developed a prototype prover, *Polya*, with those types of problems in mind.)

Those of us in the Lean community are looking forward to Lean 4, currently under development, which is beginning to look more like a programming language and API than a theorem prover. The idea is to give end-users the flexibility to design the system they want, freeing the core developers the burden of trying to anticipate and cater to the full range of user needs (and whims). Lean's

library, `mathlib`, is currently developed by an open community of Lean users, with discussion on Zulip and Github. The development is somewhat chaotic, and only time will tell whether the effort will scale or collapse under its own weight. But, at least so far, the process has been fun and fruitful. So maybe what we need now is to extend the collaboration to the automated reasoning community. The result would be a Wild West frontier version of the Isabelle model. I am not sure it will work, but it is worth a try.

5 Conclusions

At this point in time, I think two things are clear. First, interactive theorem proving has not made inroads to everyday mathematics. To use a phrase I picked up from Bob Solovay, the technology is not yet "ready for prime time." Second, automated reasoning is not yet at a stage where it can fill in routine steps in a mathematical proof, at the level of granularity one finds in a mathematics textbook.

What is not yet clear is the extent to which these two facts are linked. It may be the case that interactive theorem *never* becomes part of everyday mathematics. I think that is unlikely, though. Rigor is essential to mathematics, rigorous mathematics is formalizable, and formalization is a natural extension of all the means for ensuring rigor that have been developed throughout the history of the subject. One way or another, it will happen.

A second possibility is that interactive theorem proving does one day become commonplace, but without the need for domain-general automation. Perhaps it suffices to have better libraries, better languages, better infrastructure, and a few domain-specific tools here and there. Mathematics is inherently modular. It may be enough to master a fixed palette of definitional and inferential idioms, as well as the right glue to compose concepts and theorems. As I said above, though, I don't see it happening. We already have pretty good languages and formal frameworks. What makes formalization tedious is the need to spell out things that are mathematically trivial, and I cannot see that becoming a central part of the mathematical workflow.

What I think is most likely is that interactive theorem proving will eventually make it to the mainstream, and that good old fashioned AI—resolution theorem proving, tableaux theorem proving, SAT solving, SMT solving, and combination methods—will play an essential role. But I will freely admit that the data doesn't yet back this up, and the results of the last two decades have been disappointing. So those who, like me, harbor these hopes need to own up to reality. We need to think long and hard about what is going wrong, and how to get the effort back on track.