Automated Reasoning for the Working Mathematician

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

September 2019

Formal methods in computer science

Formal methods are used in computer science for

- specifying,
- developing, and
- verifying

complex hardware and software systems.

They rely on:

- *formal languages* to make assertions and express constraints,
- *formal semantics* to specify intended meaning, and
- *formal rules of inference* to verify claims and carry out search.

The methods are gaining traction in industry.

Formal methods in mathematics

Formal methods hold promise for mathematics as well, but they have not yet caught on.

Four domains of application:

- verified proof
- verified computation
- formal search and discovery
- digital infrastructure

I will focus on verified proof.

Verified proof

Interactive Theorem Proving provides one method of verifying mathematical theorems.

Working with a proof assistant, users construct a formal axiomatic proof.

In many systems, this proof can be extracted and verified independently.

Verified proof

Some systems with substantial mathematical libraries:

- Mizar (set theory)
- HOL (simple type theory)
- Isabelle (simple type theory)
- Coq (constructive dependent type theory)
- HOL Light (simple type theory)
- ACL2 (primitive recursive arithmetic)
- PVS (classical dependent type theory)
- Agda (constructive dependent type theory)
- Metamath (set theory)
- Lean (dependent type theory)

Verified proof

Some theorems formalized to date:

- 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 central limit theorem

There are good libraries for elementary number theory, real and complex analysis, point-set topology, measure-theoretic probability, abstract algebra, Galois theory, ...

The verification of the odd order theorem and the Kepler conjecture (Hales' theorem) are major accomplishments.

ATP for ITP

The questions I would like to explore today:

- What role can or should automated reasoning play in interactive theorem proving?
- Where do we stand?
- What are the main challenges?

To prepare, I decided to:

- reflect on my own experiences with automated reasoning
- gather information from friends and colleagues
- do some experiments with Isabelle and Sledgehammer
- draw some conclusions

I set up a repository: `https://github.com/avigad/arwm`

Outline of talk

Table of contents:

- reminiscence (all about me)
- hearsay (what my friends told me)
- anecdotes (what happened when I tried it out)
- speculation (what I think might work)

Reminiscence

My timeline:

- 2002: learned to use Isabelle, proved quadratic reciprocity
- 2003–2004: proved the prime number theorem (with Kevin Donnelly, David Gray, Paul Raff)
- 2005–2009: contributed the Isabelle library
- 2009–2010: worked on the odd order theorem with Gonthier and the Mathematical Components team (Coq / SSReflect)
- 2011–2012: formalized the central limit theorem with Luke Serafin and Johannes Hölzl
- 2012–2018: did some work in homotopy type theory in Coq and then in Lean
- 2013–present: worked on Lean's libraries, documentation, automation

Reminiscence

In 2002, Isabelle's library was not very extensive, and there were lots of gaps.

But the automation was surprisingly mature:

- a conditional term rewriter (`simp`)
- a procedure for linear (real and integer) arithmetic (`arith`)
- a tableau prover (`blast`)
- a general reasoner (`auto`)

I used them a lot.

Reminiscence

The last file in the proof of the prime number theorem, PNT.thy, has about 4,000 lines.

Usage:

- `simp`: 390 times
- `auto`: 51 times
- `force`: 277 times
- `clarify`: 69 times
- `arith`: 246 times

(The proofs here are horribly hackish. John Harrison, Larry Paulson, Mario Carneiro, and Manuel Eberl have much nicer proofs now.)

Reminiscence

After the PNT, I took a break. Over the next few years, I contributed various things to the Isabelle library.

I was firmly convinced that a little more domain-general automation would be the key to making ITP accessible to mathematicians.

In 2009–2010, I had an opportunity to spend a sabbatical year with the Mathematical Components group in France, thanks to Georges Gonthier.

A tale of two (or three) cities

Isabelle	Coq / SSReflect
Cambridge / Munich	Paris
simple type theory	dependent type theory
classical	constructive
declarative proofs	tactic proofs
lots of automation	almost no automation
no computation in the kernel	computation in the kernel
adapt CS to the mathematics	adapt mathematics to the CS
foundation in the background	foundation in the foreground

A tale of two (or three) cities

There are nice examples of structured Isabelle proof scripts, using automation, throughout the Isabelle library and AFP (Archive of Formal Proofs).

For example, take a look at the final proof of the prime number theorem by Eberl and Paulson or Hölzl's construction of the Bochner integral.

The next slide shows a proof in SSReflect.

For more examples, see the proof of the Lasalle invariance principle by Cyril Cohen and Damien Rouhling.

A tale of two (or three) cities

```
Theorem Burnside_normal_complement :
  'N_G(S) \subset 'C(S) -> 'O_p^'(G) ><| S = G.
Proof.
move=> cSN; set K := 'O_p^'(G); have [sSG pS _] := and3P sylS.
have [p'K]:  p^'.-group K /\ K <| G by rewrite pcore_pgroup
    pcore_normal.
case/andP=> sKG nKG; have{nKG} nKS := subset_trans sSG nKG.
have{pS p'K} tiKS: K :&: S = 1 by rewrite setIC coprime_TIg
    ?(pnat_coprime pS).
suffices{tiKS nKS} hallK: p^'.-Hall(G) K.
  rewrite sdprodE //= -/K; apply/eqP; rewrite eqEcard ?mul_subG //=.
  by rewrite TI_cardMg //= (card_Hall sylS) (card_Hall hallK) mulnC
    partnC.
pose G' := G^`(1); have nsG'G : G' <| G by rewrite der_normal.
suffices{K sKG} p'G': p^'.-group G'.
  have nsG'K: G' <| K by rewrite (normalS _ sKG) ?pcore_max.
  rewrite -(pquotient_pHall p'G') -?pquotient_pcore //= -/G'.
  by rewrite nilpotent_pcore_Hall ?abelian_nil ?der_abelian.
suffices{nsG'G} tiSG': S :&: G' = 1.
  have sylG'S : p.-Sylow(G') (G' :&: S) by rewrite (pSylow_normalI _
    sylS).
  rewrite /pgroup -[#|_|](partnC p) ?cardG_gt0 // -{sylG'S}(card_Hall
    sylG'S).
  by rewrite /= setIC tiSG' cards1 mul1n pnat_part.
apply/trivgP; rewrite /= focal_subgroup_gen ?(p_Sylow sylS) // gen_subG.
```

A tale of two (or three) cities

Georges felt that automated reasoning is unprincipled, brittle, undependable, unpredictable, not robust, not scalable.

In principle, the logical framework of DTT specifies the computational behavior.

In fact, elaboration and type checking in DTT relies on heuristics that are unpredictable, sometimes brittle, and rarely documented.

Isabelle's library ($\sim$900K loc) and the AFP ($\sim$2.3M loc) show that libraries based on automation are maintainable.

Reminiscence

I returned from France in 2010, and decided that I had done enough formalization.

I was more interested in learning about automated reasoning.

Then Luke Serafin, and undergraduate student, talked me into working on a formalization of the central limit theorem.

Chris Kapulkin talked me into working on homotopy type theory.

Leonardo de Moura talked me into working on the libraries for a new theorem prover, Lean.

Reminiscence

From the project page:

> *Lean aims 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.*

From one perspective, the goal was to build an interactive theorem prover with good automation.

From another perspective, the goal was to build an automated reasoning tool that was expressive, interactive, proof-producing, and built on top of a powerful library.

Leo convinced me that even if we care mainly about automated reasoning, it should be built on top of a formal proof system. (Otherwise, we are "not even wrong.")

Reminiscence

Leo was thorough in learning about ITP. He, Cody Roux, Grant Passmore, and I shared hundreds of e-mails talking about Isabelle, Coq, HOL, Agda, PVS, ACL2, ...

Leo wrote Lean 0.1 and then Lean 2 with Soonho Kong. I worked on the library and documentation.

The community is currently using Lean 3.

Leo and Sebastian Ullrich are working on Lean 4.

Reminiscence

In the summer of 2017, while transitioning from Lean 2 to Lean 3, the library was moved to a separate repository, dubbed *mathlib*. Mario Carneiro took over, and rewrote a lot.

And then something interesting happened: a number of mathematicians got involved. These include:

- Reid Barton (algebraic topology)
- Kevin Buzzard (algebraic number theory)
- Johan Commelin (algebraic geometry and algebraic number theory)
- Sébastien Gouëzel (dynamical systems and ergodic theory)
- Patrick Massot (differential topology and geometry)
- Scott Morrison (higher category theory and topological quantum field theories)
- Neil Strickland (stable homotopy theory)

Reminiscence

Other developments:

- Hundreds of messages are posted every day on the Lean chat forum on Zulip.
- Buzzard has been training lots of very talented undergraduate students, like Chris Hughes and Kenny Lau.
- Lean's library is growing quickly.
- Buzzard, Commelin, and Massot formalized the concept of a *perfectoid space*.
- Jesse Han and Floris van Doorn formalized a proof that CH is not provable from the axioms of set theory.
- Sander Dahmen, Johannes Hölzl, and Robert Lewis formalized the Ellenberg Gijswijt cap set theorem.

Reminiscence

There is a lot to like about Lean:

- It is based on an expressive dependent type theory.
- The system is very well designed.
- The editor environment and syntax is nice.
- The documentation is friendly.
- It supports both computational and classical reasoning.
- It uses type class inference to manage algebraic reasoning.
- It has a powerful *metaprogramming language*.

But the automation is limited, especially compared to Isabelle.

Reminiscence

Lean has a pretty good simplifier, `simp`, modeled after Isabelle's but not as powerful.

Metaprogramming has made it possible to implement more tactics in Lean:

- Me: `finish` (a poor man's `auto`)
- Mario Carneiro: `ring`, `abel`, `norm_num`
- Rob Lewis: `linarith`
- Seul Baek: `omega`
- Scott Morrison: `tidy`, `library_search`

Reminiscence

I still think automation is crucial.

Focusing on representations and names of theorems and rules is a distraction from the mathematics.

Programming languages and proof languages are about implementation, doing very specific things.

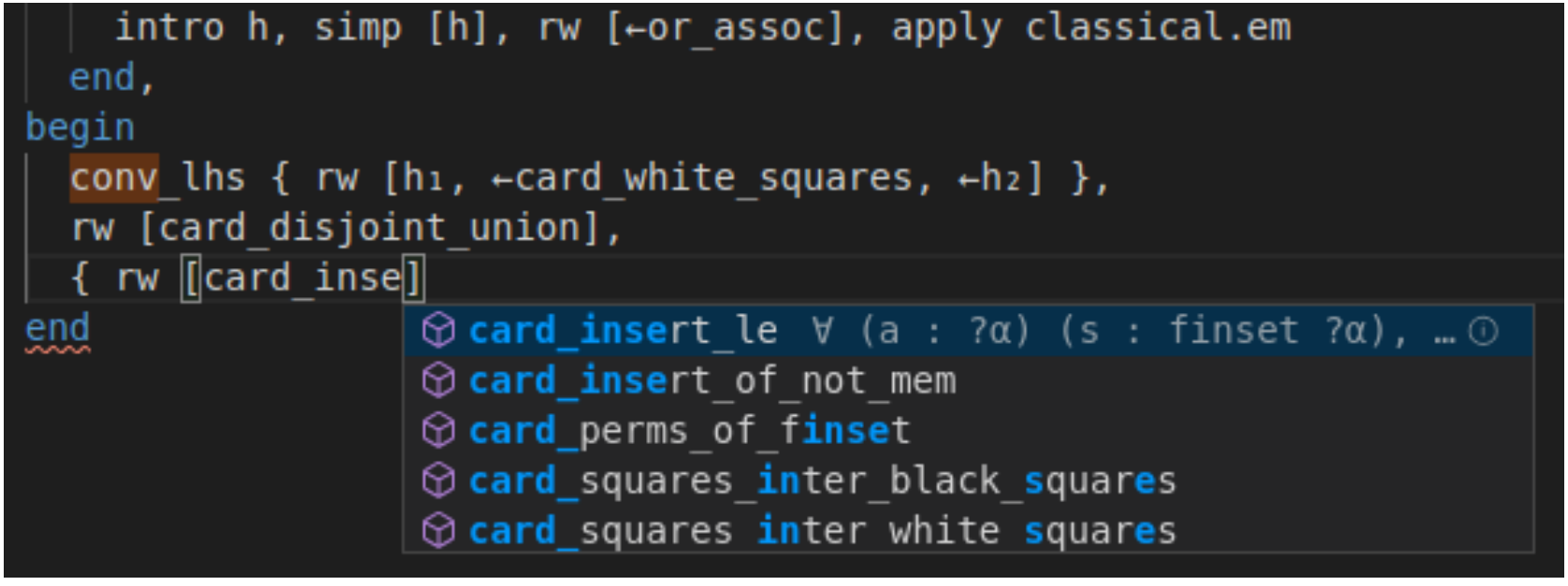
Mathematics is about abstraction, doing very general things.

Reminiscence

But these days, I don't use much automation.

I even tend to use explicit rewrites over `simp`.

Lean has good naming conventions and tab completion.



I use that a lot.

Recap

Table of contents:

Hearsay

Let's distinguish between two clusters of automated methods.

Small scale:

- used to perform small or domain-specific tasks.
- deterministic, or at least predictable
- examples: linear arithmetic, tactics for ring calculations, a `contradiction` tactic, a `continuity` tactic

Large scale:

- domain general
- often involves open-ended search using large parts of the library.
- examples: Sledgehammers, tableaux, resolution, SMT

A term rewriter like `simp` sits between the two.

Hearsay

Almost everyone likes deterministic, domain-specific stuff, like Lean's `norm_num` and `ring`. (The latter was modeled after the Coq tactic by Benjamin Gregoire and Assia Mahboubi.)

In Isabelle, Manuel Eberl uses:

- algebraic transformations (rearranging, cancelling terms)
- Gröbner basis methods (e.g. for systems of equalities in a ring)
- linear arithmetic and (occasionally) Presburger arithmetic
- occasionally Isabelle's connection to Z3 (via the `smt` method) ("but that's very specialized")
- his own limit automation tactic ("although it still has a long way to go")

Hearsay

Almost everyone likes term rewriting also.

Isabelle's simplifier is great (thanks to Tobias Nipkow). It does conditional rewriting, but also calls `auto` and `arith` to dispel side conditions.

It can be extended with special-purpose simplification procedures.

It does restricted higher-order matching and handles congruences. It will simplify under binders.

It is often used to prove facts, by simplifying them to `True`.

For example, it is good at proving side conditions like `finite S`.

Hearsay

For domain-general tools, Isabelle is the exemplar. (Also PVS, but I have not used it.)

Paulson's `auto` is excellent. It does a tableau-like search applying user-tagged `intro` and `elim` rules, but also calls `simp`, which in turn will call `arith` and `auto`.

Hearsay

Paulson's Sledgehammer, developed further by Blanchette and others, is a powerful tool. It calls resolution provers (E, Vampire, Spass) and SMT solvers (Z3, CVC4) that are shipped with the Isabelle distribution, and it reconstructs proofs.

For an excellent overview of the technology, see:

- Blanchette, Kaliszyk, Paulson, and Urban, "Hammering towards QED"
- Blanchette, Böhme, Paulson, "Extending Sledgehammer with SMT Solvers"

Isabelle's automation

As far as I can tell, everyone who uses Isabelle to formalize serious mathematics uses `auto`, `simp`, and `arith` all the time.

Usage of Sledgehammer varies.

Isabelle's automation

Paulson has ported a huge amount from HOL Light, including libraries for complex analysis and homology theory.

This involved turning tactic scripts into structured Isar proofs.

He reports that Sledgehammer was indispensible, and that he still doesn't understand the mathematics.

He gave a nice talk about this.

Isabelle's automation

The surveys cited above also provide examples of Sledgehammer in use.

Angeliki Koutsoukou-Argyraki has written a nice account of her experiences as a mathematician beginning to use Isabelle.

She reports that Sledgehammer is extremely helpful.

Yong Kiam Tan, who has worked on the CakeML project, learned Isabelle this summer and started proving theorems about dynamical systems.

He thinks Sledgehammer is great.

Isabelle's automation

But others who have formalized lots of mathematics in Isabelle rarely use Sledgehammer. These include:

- Johannes Hölzl: formalized lots of analysis and measure theory, including the central limit theorem with Luke Serafin and me.
- Fabian Immler: formalized properties of dynamical systems and manifolds, verified Tucker's computations establishing the existence of the Lorenz attractor.
- Manuel Eberl: formalized the lion's share of Apostol's number theory textbook, including the prime number theorem and Dirichlet's theorem.

Hearsay

Outside of Isabelle, some of the best formalizers I know use almost no automation.

I have already discussed Georges Gonthier and the Mathematical Components library.

John Harrison has formalized an incredible amount of mathematics in HOL Light. He tells me that he uses very little general-purpose automation.

This is surprising, because John is the author of *The Handbook of Practical Logic and Automated Reasoning*.

Hearsay

Before working on Lean, Mario Carneiro worked on *Metamath*, essentially an assembly-language like proof language for set theory (and more).

He formalized 37 of the theorems on Freek Wiedijk's list, including the prime number theorem and Dirichlet's theorem.

The logical framework is very simple, and proofs are very explicit. The entire (huge) library, including those 37 theorems, checks in a few seconds.

Mario used essentially no automation, and did not miss it.

Hearsay

Homotopy type theory uses essentially no automation (beyond elaboration and kernel reduction).

And, of course, there are all the impressive things happening in Lean, without much automation.

Hearsay

I wonder why so many mathematicians have gravitated towards Lean, despite the fact that Isabelle is more mature, with a huge library and great automation.

The main difference is that dependent type theory is better for algebraic reasoning.

Johannes Hölzl liked Lean because he could use clean algebraic abstractions to formalize analysis.

Sébastian Gouëzel, who (in his day job) studies dynamical systems and ergodic theory, switched from Isabelle to Lean.

Dependent type theory

Algebraic reasoning is important to mathematics.

Dependent type theory is one approach to handling it, but it is both a blessing and a curse.

Type theory is good for catching errors and allowing more convenient means of expression (overloading, omitting implicit information).

DTT serves two essential purposes:

- Parameterizing structures: $\mathbb{Z}/n\mathbb{Z}$ and $M_{n,n}(R)$ as rings, $\mathbb{R}^n$, C^n, $L^p(X, \mathcal{B}, \mu)$, $Aut_F(E)$, etc.
- Making structures first-class objects: we can calculate with structures, put structures on spaces of structures, consider categories of structures, etc.

Isabelle vs. Lean

From Sébastien's web page:

"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..."

Isabelle vs. Lean

"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)."

Dependent type theory

The curse: DTT requires definitional reduction.

Consider mul $(\mathbb{Z}, \text{plus}, \text{times}, 1, 0) = \text{times}$.

- LHS has type carrier $(\ldots) \to$ carrier $(\ldots) \to$ carrier $(\ldots)$.
- RHS has type $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$.

The type checker has to recognize these two as being the same.

This sameness also plays a role in elaboration, matching, and unification.

Dependent type theory

Another curse: structure needs to be inferred everywhere.

For example, $\mathbb{R}$ is an ordered field, and hence a field, and hence a ring, so the additive part is an abelian group, hence a monoid, hence a semigroup, and so $+$ is associative.

Lots of structures and operations on them can come up during proof search. Reasoners need to keep track of their properties.

This curse has to do with algebraic reasoning, not DTT *per se*.

Recap

Table of contents:

Anecdotes

To gather data, I proved some simple theorems in Isabelle, relying on automation as much as possible.

I relied chiefly on:

- `auto`, `simp`, and `linarith`
- Sledgehammer

I proved:

- the mutilated chessboard problem.
- the fact that there are infinitely many primes congruent to three modulo four.
- the intermediate value theorem.

I took notes as I went.

The mutilated chessboard problem

We are given an 8×8 chessboard with two opposite corners removed, and a set of dominoes that cover two adjacent squares.

Theorem. It is impossible to cover the mutilated chessboard with dominoes.

Proof.

- Since the chessboard is symmetric, there are just as many white squares as black squares.
- The mutilated chessboard has more white squares than black squares.
- Since every domino covers one white square and one black square, and set of disjoint dominoes covers the same number of white and black squares.

The mutilated chessboard problem

I started with a template sketch of an ideal proof.

I then resolved to use *only* the automated tools, and add intermediate statements as needed.

The result is here, and here are my notes.

I had previously formalized the theorem by hand in Lean.

Having Sledgehammer was not a big advantage.

The mutilated chessboard problem

Lessons:

1. Sledgehammer has trouble with bound variables, for example, with set abstractions and summations.
2. A difficulty with Sledgehammer is that you get no useful feedback when it fails.
3. Sledgehammer and `auto`/`simp`/`arith` have complementary strengths.
   - `auto`, `simp`, and `arith` do straightforward things. They are not clever.
   - Slegehammer is clever, but it doesn't know how to do straightforward things.
4. My approach was too rigid. Some steps are best done manually.

Being more flexible led to a nicer proof here.

Primes modulo four

Lemma. If the product of two numbers is congruent to 3 mod 4, then one of them is congruent to 3 mod 4.

Lemma. If n is congruent to 3 modulo 4, it has a prime factor congruent to 3 mod 4.

Theorem. There are infinitely many primes congruent to 3 modulo 4.

Proof. Let S be the set of such primes. If S is finite, consider $4 \cdot (\prod_{n \in S} n) - 1$. By the lemma, it has a prime divisor congruent to 3 mod 4.

Primes modulo four

Here Sledgehammer was great.

I had to set up a proof by induction manually, but otherwise Sledgehammer wrote most of the proof.

In other words, I wrote a declarative sketch, and Sledgehammer filled in most of the details, even some things I did not expect it to get.

The result is here and the notes are here.

The intermediate value theorem

Theorem. Suppose $f(x)$ is continuous on $[a, b]$, with $f(a) < 0$ and $f(b) > 0$. Then for some $x \in [a, b]$, $f(x) = 0$.

Proof. Let $y = \sup\{x \in [a, b] \mid f(x) < 0\}$.

If $f(y) < 0$, then $y < b$, and by continuity one can find a point y' to the right of y satisfying $f(y) < 0$, contradicting the fact that y is the sup.

If $f(y) > 0$, then $y > a$, and there is a neighborhood around y where all elements $x \in [a, b]$ satisfy $f(x) > 0$. But since y is the least upper bound, there will be points y' arbitrarily close to y such that $f(y') < 0$, again a contradiction.

The intermediate value theorem

The result is here and the notes are here.

Sledgehammer wasn't too helpful, but `auto` and friends were.

For example, I needed to show sup $\{x \in [a, b] \mid f(x) < 0\} \in [a, b]$.

Sledgehammer could prove this using

- $a \in \{x \in [a, b] \mid f(x) < 0\}$
- $\mathit{bdd\text{-}above}([a, b])$
- $\mathit{bdd\text{-}above}(\{x \in [a, b] \mid f(x) < 0\})$

and `auto` could prove these trivially.

But the way I figured out what hypotheses were needed was by proving it manually.

The intermediate value theorem

Another gripe: from the fact that f is continuous on $[a, b]$ and $\varepsilon > 0$, I wanted to conclude

$$\exists \delta > 0 \, \forall x \in [a, b] \, (|x - y| < \delta \to |f(x) - f(y)| < \varepsilon).$$

This is essentially the definition, but in a good library continuity will be defined for arbitrary topological spaces, with instances that unpack to this.

I had to unwrap the chain of abstractions manually to get to an elementary statement.

Recap

Table of contents:

Speculation

Most people like automation for straightforward, mostly deterministic procedures.

Opinions are still mixed regarding general-purpose search.

Three kinds of concerns:

- Technical
- Principled
- Pragmatic

Speculation

Technical concerns: using automation

- slows down formalization
- slows down compilation
- makes proofs fragile
- makes proofs hard to fix and maintain
- yields large proofs that take up too much space

Mitigating factors:

- Isabelle provides a strong evidence that it works.
- Technical problems can be overcome with good engineering.
- Compilation time, robustness, and so on are also problems in systems that don't use automation.

Speculation

Sometimes the desire to write proofs explicitly reflects a programmer's aesthetic: we want control over our code.

Frank Pfenning (talking about software verification):

> *. . . sometimes I am worried that too much automation is a bad thing, because it can mask the flaws in what one is doing. There are many examples in the study of logics, but let me use programming instead. Start with your favorite terrible programming language (Python happens to be a convenient target, or C++). Now, sure, you can reduce the number of bugs with automatic program analysis, the smarter the better. But you would nevertheless still be better off in a statically typed functional language with a decent module system. You prop up something that would be better left to wither and die.*

Speculation

But mathematics is not programming.

- Good programming languages do not necessarily make good proof languages.
- Implementation details of a formal proof have very little mathematical content.

Speculation

The third objection to the use of general automation is that it just doesn't work: ultimately, to do what we need to do, we need to learn to do things by hand.

I don't think the objection is valid, but it's consistent with current evidence.

If automation made it a lot easier to prove theorems, we'd get over the other concerns.

I'll close with some conclusions and suggestions.

Conclusions

Be wary of benchmarks.

Benchmarks are important, but they are imperfect.

- They lead to overfitting.
- They may not be representative of problems we care about.
- They are generally extracted from existing libraries, and it is often hard to tell whether provers are getting the easy stuff or the stuff where we really want help.
- The real work often goes into setting up the library in the first place.

Isabelle's automation is so good because formalization and tool development went hand in hand.

The real measure of success is whether users adopt the tools.

Conclusions

Algebraic and structural reasoning is important.

Whether we manage it with dependent type theory or other methods, it's a *sine qua non*.

All the automation in the world won't help if the system is not good for the kinds of mathematics we want to do.

Conclusions

It would be great if we could combine strengths of Sledgehammer and other tools.

Remember, `auto`, `simp`, `arith` are really good at the straightfoward steps, but not very clever.

Sledgehammer is very clever, but has no common sense.

Can we somehow get the best of both worlds?

Conclusions

Second-order reasoning is unavoidable.

Even straightforward reasoning patterns often need to deal with binders (like lambdas and set abstractions).

Conclusions

Arithmetic is important.

Lots of straightforward reasoning patterns require simple inequalities or basic arithmetic.

Conclusions

Combination methods are important.

The last few slides were all about heterogeneous forms of reasoning.

Conclusions

Users need feedback.

When a tool fails, we need to know what went wrong, and how to fix it.

It would be nice if a tool could tell us that it would be helpful to know $n > 0$ or that S is finite or that $[a, b]$ is bounded above, or that it doesn't know a good value to choose for ε.

Resolution calculi try to eliminate elements from a clause in a fixed order to avoid redundancy.

Maybe it would be better to be more aggressive about discovering near misses.

Conclusions

It would help to give users and library designers more control.

Isabelle users are happy to annotate theorems for `auto` and `simp`. This makes it possible to tune their behaviors.

Bohua Zhan used similar annotations for his `auto2` prover.

Lean users really like writing little tactics in the metaprogramming language.

If there are convenient ways for users to convey expertise and tune automation, they will take advantage of it.

Conclusions

Certificates are nice.

In the ITP community, we want to reconstruct proofs and check them ourselves.

Please provide us with as much information as you can!

Conclusions

I recognize that

- nothing I have said is really new, and
- the problems are not easy to solve.

I am here to show enthusiasm and ask you to keep ITP in mind.

I am hopeful that we are near a tipping point, and that slightly better automation will make formal methods attractive to a wide mathematical audience.

I, for one, am very grateful for the work you do.