Automated Reasoning: A Survey

John Harrison

University of Cambridge

(visiting TU München)

- What is automated reasoning?
- Theoretical and practical limits
- Successes of the AI and logic approaches
- Development of formal logic
- History of automated reasoning
- Applications
- Interactive systems
- Reflection and LCF

What is automated reasoning?

We interpret 'automated' broadly and 'reasoning' narrowly:

- We are interested in reasoning in logic and mathematics, rather than in general reasoning.
- On the other hand, we consider both fully automatic and interactive systems.

The field is also called *automated theorem proving* or *mechanized theorem proving*.

Decidable systems

There are well-known fields of logic and mathematics where validity is decidable, e.g:

- Propositional logic, e.g. $\neg(p \vee q) \Rightarrow \neg p \wedge \neg q$.
- AE fragment of first order logic, e.g. $\forall x.\, \exists y.\, P[x] \Rightarrow P[y]$.
- Linear arithmetic over $\mathbb{N}$, e.g. $x < y \Rightarrow 2x + 1 < 2y$.
- Nonlinear arithmetic over $\mathbb{R}$, e.g. $\exists x.\, x^2 - 3x + 1 = 0$.

However, this only covers small fragments of mathematics.

Theoretical limits

Full automation has strong theoretical limits, by virtue of the following (related) theorems:

- Tarski's theorem on the undefinability of truth
- Gödel's first incompleteness theorem.
- Church's theorem.

A naive proof procedure

However, there are still ways of searching for proofs that can in principle prove most of the facts of present-day mathematics (e.g. everything in Bourbaki). A crude way is follows.

1. Express the mathematical axioms ϕ and the desired theorem ψ in first order logic.
2. Dual-Skolemize the formula $\phi \Rightarrow \psi$ into the form $\exists x_1, \ldots, x_n.\, P[x_1, \ldots, x_n]$
3. Search for substitution instances such that $P[t_1^1, \ldots, t_n^1] \vee \ldots \vee P[t_1^k, \ldots, t_n^k]$ is a tautology.

Practical Limits

Even if a theory is decidable in principle, the time or space usage of the decision procedure may make it ineffective in practice.

Anyway with general methods like the above, we have the problem of searching with no upper bound on the time taken.

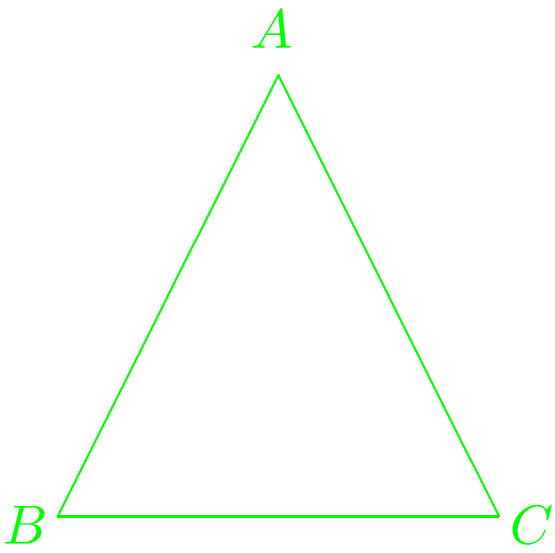
The key is to cut down *search space*. There are two main approaches:

- Look at and copy human behaviour (the AI approach)
- Use more refined search methods backed up by metatheorems (the logic approach).

There was (is?) still a controversy over whether the human-oriented 'AI' approach or the 'logic' approach is better.

A theorem in geometry

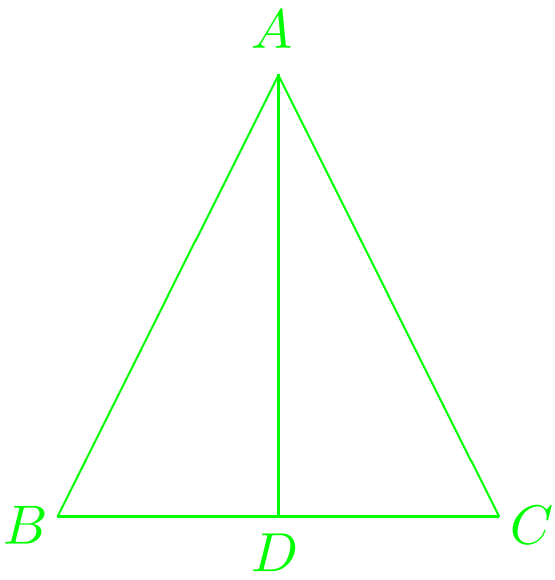
One of the early successes in automated theorem proving (on the AI side) was the proof of the following theorem:



If the sides AB and AC are equal (i.e. the triangle is isoseles), then the angles ABC and ACB are equal.

The usual proof

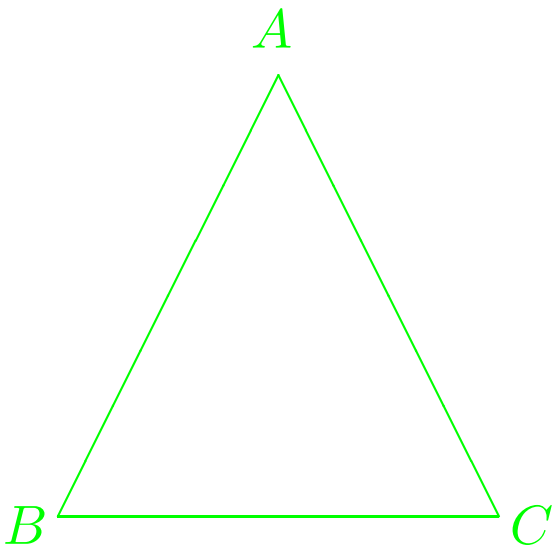
The usual proof proceeds by dropping a perpendicular down from the point A to the side BC, meeting it at a point D:



and then using the fact that the triangles ABD and ACD are congruent.

The computer's proof

The computer found an ingenious proof which had been missed by most writers on geometry (though it had already been used by Pappus).



Simply, the triangles ABC and ACB are congruent. Q.E.D.

The Robbins Conjecture (1)

A very recent success in automated reasoning, this time on the logic side, was the proof by McCune's program EQP of the Robbins Conjecture.

Huntington (1933) presented the following axioms for a Boolean algebra:

$$\begin{aligned} x + y &= y + x \\ (x + y) + z &= x + (y + z) \\ n(n(x) + y) + n(n(x) + n(y)) &= x \end{aligned}$$

Shortly thereafter, Herbert Robbins conjectured that the Huntington equation can be replaced by a simpler one:

$$n(n(x + y) + n(x + n(y))) = x$$

The Robbins Conjecture (2)

This conjecture went unproved for more than 50 years, despite being studied by many mathematicians, even including Tarski.

It because a popular target for researchers in automated reasoning.

In May 1996, it was claimed that a proof had been found automatically using the REVEAL prover. However this was traced to a bug in REVEAL.

Then, in October 1996, a correct proof was found by McCune's program EQP.

The successful search took about 8 days on an RS/6000 processor and used about 30 megabytes of memory.

Origins of mechanization

The idea of mechanizing reasoning in a manner similar to arithmetic calculation is an old one, going back at least to Hobbes.

> Reason [. . .] is nothing but Reckoning. For as Arithmeticians teach to adde and subtract in *numbers* [...] The Logicians teach the same in consequences of words [...] And as in Arithmetique, unpractised men must, and Professors themselves may often erre, and cast up false; so also in any other subject of Reasoning the ablest, most attentive, and most practised men, may deceive themselves, and inferre false conclusions.

Leibniz envisaged a *calculus ratiocinator* and a *characteristica universalis.*

Development of formal logic

We can highlight several important phases in the development of formal logic.

- The Socratic method
- Aristotle's syllogisms
- Leibniz's attempts at a *characteristica*
- Boole's algebra of logic
- Frege's *Begriffsschrift*
- Peano's *Formulaire*
- Russell and Whitehead's *Principia Mathematica*.
- Hilbert's programme
- Metamathematical studies (Gödel, Tarski, Church, Turing, ...)

Early computer experiments

The earliest uses of computers in theorem proving were in the late 50s and early 60s. Among the pioneers were:

- Newell and Simon (AI)
- Gelentner's geometry machine (AI)
- Gilmore (logical)
- Wang (logical)
- Davis and Putnam (logical)
- Prawitz (logical)

The logic approach soon began to dominate, but still had strong limitations.

Prawitz's method used a much more intelligent way of searching for ground instances, based on a simple form of unification. This was later generalized by Robinson.

More recent methods

The two most efficient general first order theorem proving methods were invented in the 60s.

- **Resolution**, invented by Alan Robinson, is a bottom-up, local, proof method based on a single, very simple, inference rule:

$$\frac{p \vee q \quad \neg p}{q}$$

- **Model elimination**, invented by Donald Loveland, is a top-down, global, proof method which in many versions is quite similar to Prolog.

These are still the big two methods today, represented by `Otter` (from Chicago) and `SETHEO` (from Munich), probably the most powerful general first order provers at present.

Higher Order Logic

Most attention has been devoted to automatic proofs in either (i) pure first order logic, or (ii) particular mathematical theories.

However, higher order logic is a promising alternative. This line has mainly been pursued by Andrews and his collaborators and led to *TPS*.

TPS uses a version of the 'connection' or 'matings' method, with higher-order unification à la Huet replacing first order unification. It can prove automatically:

- Cantor's theorem: there is no mapping from a set onto its powerset.
- If some f^n has a unique fixed point then f has a fixpoint

The Boyer-Moore Prover

Boyer and Moore's NQTHM is unusual in that it doesn't work in pure logic. Instead it uses a very simple system of 'primitive recursive arithmetic' (Skolem, Goodstein).

It has the remarkable ability to do proofs by induction automatically.

These properties make it much more useful in many real situations than provers for pure logic.

It has been used for many impressive applications, mainly in verification, which we consider later.

It is fully automatic. Nevertheless, the user still has to guide it in some way by selecting a sequence of lemmas. And there is not much control over what it does.

A new system ACL2 supersedes NQTHM in most respects.

Formalized Mathematics

One application of theorem provers is to check large bodies of existing mathematics, making them completely formal.

Peano started such a project with his *Formulaire* but did not really formalize *proofs*.

Bourbaki seems to believe in formalization 'in principle', but not in practice.

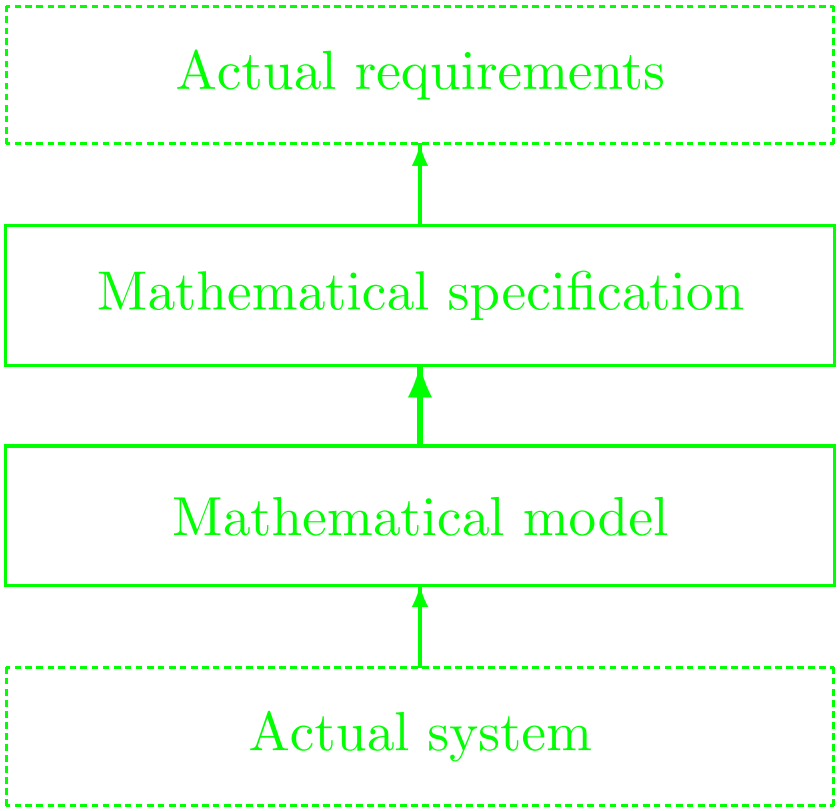
However with the help of the computer we can actually achieve formalization.

The most impressive example is the Mizar project.

There is a recent proposal for a QED Project to extend this formalization much further.

Verification

The idea of verification is to make sure computer systems (hardware, software) work correctly by formal verification of the design.



It is only the central link that is mathematically precise. The others are still informal — all we can do is try to keep them small.

Interactive theorem proving

Automated systems may be capable of impressive feats, but they are not usually much use either in mathematics or verification.

The current trend is to combine automation with human control and guidance. This idea goes back to the SAM (semi-automated mathematics) project. Other pioneering proof checkers appeared in the 70s:

- AUTOMATH (de Bruijn)
- Mizar (Trybulec et al.)
- Stanford LCF (Milner)

However, these tended to be tedious to use. What was needed was a better mix of automation with the manual controllability. Nowadays, PVS is perhaps the state of the art.

Obviousness

The ideal is to leave the subtle parts of the proofs to humans, and have the computer fill in the obvious parts. But what humans and computers find obvious are not the same. For example computers find:

$$(\forall x\ y\ z.\ P(x,y) \wedge P(y,z) \Rightarrow P(x,z)) \wedge$$
$$(\forall x\ y\ z.\ Q(x,y) \wedge Q(y,z) \Rightarrow Q(x,z)) \wedge$$
$$(\forall x\ y.\ Q(x,y) \Rightarrow Q(y,x)) \wedge$$
$$(\forall x\ y.\ P(x,y) \vee Q(x,y))$$
$$\Rightarrow (\forall x\ y.\ P(x,y)) \vee (\forall x\ y.\ Q(x,y))$$

very obvious, but most people need to think about it. Conversely, most people find McCarthy's 'mutilated checkerboard' obvious (when shown the trick) but computers have trouble. Computers are really oriented towards 'logical' obviousness.

Sound extensibility

The ideal interactive theorem prover should be extensible with new inference capabilities as the need arises.

Moreover, especially if it is to be used in verification, the system itself and any extensions should be reliable (at least logically consistent...)

There are two main approaches to this problem:

- Reflection
- LCF

Crudely speaking, the difference is between *verifying code* and *making code self-checking*.

As such it represents a general choice in designing correct software (see papers by Blum on results checking).

Reflection

Reflection in theorem proving is vaguely related to logical reflection, which involves adding rules like:

$$\frac{\vdash Pr(\ulcorner \phi \urcorner)}{\vdash \phi}$$

The idea is:

- Take a new inference rule, implemented by a piece of code C.
- Verify in the existing system that C is correct.
- Add C to the implementation of the theorem prover.

Not exactly a logical rule or principle in the traditional sense. As yet, there are few non-trivial examples.

Edinburgh LCF

The LCF alternative started with Edinburgh LCF (Milner et al.)

A core of primitive inferences is provided, each simply an ML function, returning a `thm`.

Users can write custom inference rules in ML, decomposing to these primitive inferences.

Although proofs are not explicitly constructed, theorems are members of an abstract type and can only be created by applying the primitive inferences.

Thus, derived inference rules are correct by construction, in the sense that all 'theorems' produced really are theorems.

There are many LCF descendants including HOL (Gordon, Melham et al.), Coq (Huet et al.) and Nuprl (Constable et al.).

Fully-expansive decision procedures

How can we code sophisticated derived rules that decompose to primitives? At first sight this might seem hopelessly inefficient.

- Represent inference steps as object-level theorems
- Separate search from inference

Many useful decision procedures can be coded in this manner, without unacceptable slowness.

For example, HOL has linear arithmetic, tautology checking and model elimination.

Other things, like explicit arithmetic with very large numbers, or the BDD-based fixpoint calculations in model checking, seem more challenging.

Isabelle

Isabelle (Paulson, Nipkow et al.) belongs to the LCF family but uses slightly different principles.

Inference rules are theorems in a metalogic, which itself is formalized in the traditional LCF way. This means:

- The system can be made generic, i.e. to work for a family of object logics, including HOL and ZF.
- Inference rules are represented more 'declaratively' rather than as black boxes.

on the other hand:

- The primitive rules are less efficient; they perform a higher-order unification step each time.
- The system is more complex, since it has both a metalogic and (several) object logics.

Conclusions

- Purely automatic theorem proving is an interesting research field with connections both to pure logic and to AI.
- Automatic systems can still make new contributions to mathematics in the right areas.
- Although applications mainly use interactive systems, automated subsystems are vital, and these can draw on the body of existing work.
- The LCF methodology seems a promising line of research for obtaining powerful but reliable syetems.
- Applications have their own interest. Formalizing mathematics is often fascinating, while systems verification is a challenge and potentially very valuable in practice.

Postscript

A theorem prover in 6 lines of Prolog (Beckert and Possega):

```
prove((E,F),A,B,C,D) :- !,prove(E,[F|A],B,C,D).

prove((E;F),A,B,C,D) :- !,prove(E,A,B,C,D),
                          prove(F,A,B,C,D).

prove(all(I,J),A,B,C,D) :- !,
   \+length(C,D),copy_term((I,J,C),(G,F,C)),
   append(A,[all(I,J)],E),prove(F,E,B,[G|C],D).

prove(A,_,[C|D],_,_) :-
    ((A= -(B);-(A)=B) -> (unify(B,C);
                          prove(A,[],D,_,_))).

prove(A,[E|F],B,C,D) :- prove(E,F,[A|B],C,D).
```