Automated Reasoning

Jeremy Youngquist

March 11, 2016

Abstract

This paper serves as an introduction to the field of automated reasoning and is divided into two main parts. The first part surveys the various techniques for proving theorems via a program. Various concerns are addressed starting with the general approach that programs use to prove a theorem before discussing the level of interaction with the user that the program has. I then move on to which deduction calculi are used in automated reasoning and how the language is represented in the computer. Lastly, the very practical issue of axiom selection is discussed as the choice for axioms vastly effects both the efficiency of the program as well as the length of the proof. The second part of this paper addresses the philosophical issues that arise when you begin to prove theorems using machines as opposed to humans. I consider issues relating to the level of understanding that a formal proof conveys as opposed to that of a human-produced proof and conclude by addressing the issues of program correctness and how errors can be minimized.

1 Introduction

Within the last century or so, the foundations of mathematics has been completely formalized in the language of set theory, combined with a deduction calculus that allows you to combine the axioms to prove new results. Mathematics has been reduced to a very small finite set of rules as the starting point; ZFC only has ten axioms and the natural deduction calculus has just eight rules of inference and one axiom. With such a small set of starting points and current computational power, it seems reasonable to wonder if you could write a computer program that simply applies the rules of the calculus to the axioms and have it prove new results, letting the computer run wild and contribute new theorems, complete with proofs.

As it turns out, the answer is that, yes, you can completely automate mathematics! At least in theory, that is, in practice this is much more easily said than done. Much work has been done in this field in the past few decades and the automated theorem provers are getting better than ever, but there have been strikingly few novel contributions that they have made. This move to a purely computational approach was anticipated by Leibniz, who in the 17^{th} century desired a universal language in which you could express any question and then simply calculate the answer. The range of this language would cover not only mathematics, but would even venture into philosophy[2]. Godel showed that this is in principle not possible, but the approach of automated reasoning resembles Leibniz's desire.

1.1 Famous Results of Automated Theorem Provers

Automated theorem provers have thus far not been powerful enough to make very many contributions to mathematics, most of their results stay within the range of undergraduate level mathematics. This is to be expected though; the field is relatively new and is largely a niche area that does not have nearly as many researchers as some of the more popular areas of mathematics. That being said, though, there are several important results that automatic theorem provers have given us.

One of the first important results that got the field off of it's feet was the proof of the four-color theorem. It states that given any separation of a plane into contiguous regions, no more than four colors are needed to fill in the map such that no two adjacent regions have the same color. This was a long-standing problem in combinatorics that was eventually proven by Appel and Hakken in 1976 using an automated theorem prover to exhaustively check every possible configuration[2]. It was then verified in 2004 by Gonthier using the system *Coq*, making it one of the most meticulously checked proofs in history[4].

Other famous theorems have been proven, such as the Kepler conjecture, which will be talked about later, and the Robbins Conjecture which states that all Robbins algebras are Boolean. The approach of automated theorem provers has also been used extensively to verify already proved results, from Godel's first incompleteness theorem to the prime number theorem, the Jordan Curve theorem, the fundamental theorems of algebra and calculus, and many more important theorems as a way to verify their correctness.

2 Techniques for Proving Theorems

Now that we know computers can be used to automate areas of mathematics, how would someone go about actually doing such a thing? In this section I will overview some of the main concerns and decisions that must be made when implementing an automated theorem prover. First I discuss the different types of theorem provers: Interactive vs Automatic and the advantages and disadvantages to each. We then run our attention to the method that the program will use to prove the theorem, either a direct proof or a refutation proof. Then I explain the different types of deduction calculi that are used in these provers and how they are represented in the program. Finally I end the section by considering the problem of selecting the right axioms when proving a theorem so that you have enough axioms to be able to prove it, but not too many which cause the program is bogged down with a bloated search space.

2.1 Interactive vs. Automatic Theorem Provers

There are two main classes of theorem provers: Automated Theorem Provers (ATP's) and Interactive Theorem Provers (ITP's). The main difference between them is the level of interaction that the user has, as the names might suggest. ATP's are exactly what you might expect from a program that is supposed to prove theorems: you simply give it a theorem and some axioms in the language of the program and it runs until it arrives at a proof, or it runs for the rest of eternity.

ITP's on the other hand, allow for more interaction with the user, although in some cases the lines seem to blur between ATP's and ITP's. The most clear example of an ITP would have to be the Boyer-Moore $NQTHM$ theorem prover[2]. It was originally an ATP, but it ended up shifting into an ITP where users prove facts incrementally and can even give the program hints to help it along the way.

The line between an ATP and an ITP can certainly be unclear at times and sometimes theorem provers combine them into one program. This is employed by the authors in [1]. A user inputs a fact that needs a proof into the ITP front-end, the ITP runs an axiom selection program, and feeds the theorem along with the predicted axioms to an ATP.

2.2 Direct vs. Indirect Approaches

Once we have decided upon using an ATP or an ITP, the next logical question is what method will our program use to prove a theorem from the given facts? There are two main approaches to this. The first is the more obvious approach of simply proving the theorem directly from the axioms. The program takes in the axioms, applies the logical rules to them, and eventually outputs a proof of the theorem, showing that $\Gamma \vdash \phi$ directly. The other popular choice is to indirectly prove the theorem by converting ϕ, the theorem in question, into $\neg\phi$ and showing that $\Gamma \cup \{\neg\phi\} \vdash \perp$ which by the rules of the logic system entail that $\Gamma \vdash \phi$ completing the proof.

Both of these approaches rely on the metatheoretical properties of the deduction calculi, namely soundness and completeness. Both of these results are clearly needed for the direct approach, and can be manipulated to give similar results for the indirect, or refutation, method. For the indirect calculus, soundness becomes $\Gamma \cup \{\neg\phi\} \vdash \perp$ implies that $\Gamma \models \phi$ and completeness becomes $\Gamma \models \phi$ implies $\Gamma \cup \{\neg\phi\} \vdash \perp$, both of which are unsurprising[3].

2.3 Choice of Calculi

After you decide on the type of theorem prover and the approach you want it to take, you need to decide what type of deduction calculi you will implement in the code. I will talk about two calculi, starting with natural deduction, since it is the calculi we developed and used in class, and then mentioning sequent calculus since it more popular in automated reasoning.

Proofs that use natural deduction can be thought of as trees with the assumptions being leaves and the conclusion being the root. We typically read the proof in a forwards direction, starting at the leaves and reading towards the root, seeing all the logical steps that are made along the way, but when the proof is being constructed, it is generally more effective for the program to work backwards. The program starts with the root and applies the rules of inference in a backwards fashion to arrive at the leaves[3].

One of the problems with implementing natural deduction is that you need to have a way to prevent the program from making endless chains of conjunctions and other endless logical inferences that, although logically valid, do not make any progress toward the conclusion. The way around these infinite 'loops' is to exploit the fact that natural deduction has the subnormal property, i.e. that every formula can be restricted to being a subformula of $\Gamma \cup \Delta \cup \{a\}$ where Δ is the set of assumptions made in the $\neg$-Elimination rule. This can be used to reduce the search space and control the backward application of the elimination rules[3].

The attraction to natural deduction is that it closely resembles how an average mathematician reasons and, unsurprisingly, the proofs generated are more natural to read. This makes theorem provers that use natural deduction very attractive for educational purposes, since it is easier to guide a student through a proof if they can read it more easily, but since it is more difficult to program, it is hardly ever used in academia.

Now we turn our attention to sequent calculus, which is more awkward for a human to use, but much easier to use in a program. Starting with some definitions, a sequent is an expression of the form $\Gamma \to \Delta$ where Γ and Δ are sets of formulas which may or may not be empty. Γ is the antecedent and Δ is the succedent. If $\mathcal{A}$ is an interpretation, then we say that $\mathcal{A} \models \Gamma \to \Delta$ if and only if either $\mathcal{A} \not\models \gamma$ for some $\gamma \in \Gamma$ or $\mathcal{A} \models \delta$ for some $\delta \in \Delta$[3]. Thus, to define an axiom, we write a sequent $\Gamma \to \Delta$ where $\Gamma \cap \Delta \neq \emptyset$ since no interpretation can make every formula in Γ true while at the same time making every formula in Δ false.

Proofs written in sequent calculus can be viewed in the same way as proofs in natural deduction: as trees with assumptions as leaves and the conclusion as the root. There are 14 rules of inference and 2 axioms in sequent calculus and they have several properties that make it particularly useful in automated reasoning. First, it is refutation complete, which is good because then we can perform proofs by contradiction as mentioned earlier. Second, sub formulas are simpler than their parent formulas, so when you work backwards, a refutation gets simpler and simpler the further you go. The third property is that the order in which you apply the non-quantifier rules does not matter when you work backwards, which means that you can narrow your search space since you only have to worry about combinations and not permutations of rules. A calculus with this property is called confluent[3]. These properties cause sequent calculus to lend itself quite well to automated reasoning.

2.4 Language Representation

One of the final things to decide upon when constructing your theorem prover is how to represent the calculus to the program. There are three main methods: first-order logic, typed λ-calculus, and clausal logic. I will discuss clausal logic since it is the most widely used representation[3]. Terms and literals are defined as we did in class, and a clause is a possibly empty finite disjunction of literals where each literal appears at most once. If there are no variables it is called a ground clause. Clausal logic has no quantifiers and the goal is to express everything in conjunctive normal form.

To express a formula that is already in first-order logic in clausal form, you perform two steps. First, you convert it into prenex normal form, where the expression has a string of quantifiers at the front and is followed by a quantifier free expression. For example, letting Θ be an arbitrary quantifier, $\Theta(x) \vee \Theta(y)$ is not in prenex form, but $\Theta(x)\Theta(y)(\phi(x, y))$ is. Then the second step converts it into conjunctive normal form using standard logical rules[3]. We then drop the quantifiers from the beginning of the expression.

The problem that arises is that this last step does not preserve equivalence when existential quantifiers are dropped. The solution is to introduce Skolem functions which emulate existential formulas. As an example, consider the formula $\forall x \exists y \phi(x, y)$. This formula has an existential quantifier in front which cannot be dropped. Define a function f such that $f(x) = y$. Then we can rewrite the original expression as $\forall x \phi(x, f(x))$ which does not have an existential quantifier. The process of using Skolem functions does not necessarily preserve equivalence, but does preserve satisfiability, which is all that is needed for automated reasoning[3].

2.5 Axiom Selection

One of the last hurdles encountered is to choose which axioms and theorems to allow it to start with. In principle, you could give it merely the ten axioms of Zermelo-Fraenkel set theory and it should be able to prove any theorem that ZFC can prove, but then it ends up having very little direction and having to prove many trivial things along the way. You might think then that it would be best to give it as many already proved theorems as possible to start with, but this is also impractical since that increases the search space. Unless you are Kurt Godel and are proving incompleteness, you generally 'forget' about areas of math such as number theory when you are concerned about another field such as logic.

A relatively recent approach to this problem is to use machine learning to find a minimal set of axioms which are sufficient to prove the theorem in question. Every proof can be viewed as a tree of dependencies, with each node being a fact and each edge being a dependency. Given the dependency tree of a theorem allows you to trace back through the edges to the given facts that were used to prove it. More precisely then, the axiom selection problem is, given a proof obligation c, predict the parents of c in the dependency tree[1].

To do this, the theorem prover is given a set of proofs from a pre-existing proof library that allows it to see the dependency tree, the facts which appear before it, and the representation of the theorems in the language of the theorem prover. The program then ”learns” the proofs that it was given and ranks each fact based on how likely it is to be used in future proofs. It does this by considering how often a fact is used, since the more used a fact is, the more likely it is to be used in future proofs, taking into account the context of the fact, since more recently proved theorems are more likely to be used in the next proof, comparing sub formulas of the theorem in question to axiom possibilities, since the more similar the structure of the facts the more likely there will be a proof of one from the other, and finally the types that appear in the facts[1].

Once the program ”learns” the proofs from the library it was given and ranks them, it then makes a prediction of a minimal set of axioms that are sufficient for proving the theorem that is asked about. Note that this approach does not guarantee the optimal set of axioms, but simply a prediction of what axioms are needed. After it makes a prediction, it then passes on the choice of axioms and the theorem in question to the actual theorem prover, which will hopefully return a correct proof of the theorem.

3 Philosophical Considerations

Now that we know that we *can* use automated reasoning to prove results in mathematics, we now turn our attention to the question of whether or not we *should* do this. Automated reasoning is not without it’s critics, to a large degree because we as humans are uncomfortable with leaving areas of our lives in the hand of machines. We like to have control over our own lives and having machines do our thinking for us goes against who we are. Nonetheless, this is the direction that mathematics is heading, at least in part, and it needs to be discussed.

3.1 Understanding vs. Rigor

One of the main concerns about applying automated reasoning to mathematics is that the proofs generated by a computer are achieved very mechanically and do not skip over any steps, no matter how trivial, which results in incredibly long and tedious proofs that are difficult to read and follow, albeit very rigorous. When we use a program to generate a proof we often sacrifice the understanding that a proof usually gives for extra rigor.

One of the best examples of this trade-off is in the proof of the Kepler Conjecture. The Kepler Conjecture states that in three-dimensional Euclidean space, there is no arrangement of equally sized spheres that has a greater average density than cubic close packing. The only known proof is by Thomas Hales and was produced via automated reasoning. It is 300 pages long plus around forty thousand lines of code. It was submitted to and published by the $Annals of Mathematics$, but not until after a four year review process by

12 reviewers who ended up saying that they were only "99% certain" of it's correctness[2][4][3]. This shows that, at least in this case, automated proofs may not add to our understanding of mathematics.

One of the most important aspects of mathematics is the understanding that it provides, often times the method by which a theorem is proven is of more value than the actual result of the proof, and if automated reasoning displaces this part of mathematics, then it is not of much benefit to us. However, there are some mathematicians, typically those working in the field of formal reasoning, who would argue that checking every last detail of a proof is one of the most interesting parts of mathematics and that we lose much of the depth of math when we skim over the details, labeling them as "trivial." They also respond to critics of automated reasoning by pointing out that formal proof scripts are not meant to replace normal mathematical approaches, but simply to supplement them. They provide a way to formalize mathematics in order to verify that what we believe we know is, in fact, true[2]

3.2 Program Correctness

One of the main criticisms of automated reasoning is that the correctness of the proof is only as good as the program that produces it. A typical computer program has about one bug per 100 lines of code and even the mission-critical code that was used for the space shuttles still reportedly had around one bug in 10,000 lines[4]. The objection that there may be bugs in the theorem prover is not an objection to be taken lightly. Fortunately, there are ways to circumvent this.

Clearly the axiomatic systems that are implemented carry with them the soundness that they have in their pure mathematical usage: any program based off of ZFC will be as consistent as ZFC, so the main issue is with errors in the code itself. The first and easiest way to stave off errors is to make the code small and make the source code publicly available. The kernel of HOL Light, for example, has only 500 lines of code and is written in a user-friendly language, so it very feasible for it to be checked in it's entirety by anyone familiar with automated reasoning. By making it open-source, anyone with a computer and internet access can download it and look at the code to check for errors and as a result it has been scrutinized by many logicians[4].

Recently a newer approach to verifying a the correctness of a program is to translate one system's proof into the language of another system and verify it's correctness in the second system. It is computationally easy to verify the correctness of a proof without having to reprove the theorem. In this approach, if there is a bug in one system, the other system will catch the error and show that the first proof was incorrect as long as the secondary system does not have the same error in the same location. If the probability of the individual failure rate of the systems is p, then the probability of failure across n systems is p^n and goes to zero quickly[4].

Often, proponents of automated reasoning observe that in "informal" everyday mathematical reasoning done by human mathematicians, error creeps in and mistakes end up getting published. We as humans make mistakes with faulty reasoning, incorrect assumptions, imprecise definitions, faulty intuitions and many other errors. With automated reasoning, we cannot get away with inferential gaps, make assumptions, or have unclear definitions, everything is clear, rigorous, and there are no gaps in reasoning made, so correctness should not be an issue once we know that the code is error-free[2][4]. In theory, it should be even better than a human mathematician.

4 Conclusion

Automated reasoning has a long way to go before all of it's critics are satisfied and we have powerful enough systems to prove more important results, but the field is growing and getting more and more developed each year. Within a few years it is predicted to be at the point where it is most useful for verifying long and complicated calculations, but we should not expect automating mathematics to be attractive to mathematicians until the time required to use theorem provers is similar to the time needed to verify things by hand[2]. Currently theorem provers can be difficult to work with and many hours can be spent struggling with the program to get it to behave correctly or to fill in gaps in the program's library, so it is currently not at a level where the average mathematician can use it effectively in their day to day work.

Automated theorem provers need better libraries of background mathematics, better ways to search through what has already been proven, more user-friendly interfaces, better ways to share knowledge between proof assistants, and much more. All of these issues are not insurmountable though, and the future looks bright for automated reasoning. It is expected that within just a few decades we will see many more important results proved and that formally verified proofs will become more commonplace[2].

References

[1] Jasmin Christian Blanchette and Daniel Kuhlwein, "A Survey of Axiom Selection as a Machine Learning Problem", http://www21.in.tum.de/ blanchet/axiom_sel.pdf

[2] Jeremy Avigad and John Harrison, "Formally Verified Mathematics", *Communications of the ACM*, Volume 57 Number 4, April 2014, http://cacm.acm.org/magazines/2014/4/173219-formally-verified-mathematics/abstract

[3] Portoraro, Frederic "Automated Reasoning", *The Stanford Encyclopedia of Philosophy* (Winter 2014 Edition), Edward N. Zalta (ed.), http://plato.stanford.edu/archives/win2014/entries/reasoning-automated

[4] Thomas C. Hales, "Formal Proof", *Notices of the AMS*, Volume 55 Number 11, December 2008, http://www.ams.org/notices/200811/tx081101370p.pdf