ATP Review

Alex Havrilla

August 2018

1 Introduction

ATP - Automated Theorem Proving, uses computer systems to mechanically prove theorems from an assumed set of axioms. Since the 1950s, ATP has grown into a multi-faceted field with systems including human interaction, purely automated systems, and systems residing in a gray area in between. The field grew a desire to computably verify and formalize mathematics, propagated by papers such as the *QED* Manifesto that. Many systems now exist that are building formalized mathematical libraries and developing theorem provers. This paper seeks to give an overview of several such theorem provers and the similarities between them. The systems I will be reviewing include *Automath*, *The Mizar System*, *Lean*, *TLA+*, *HOL*, *Metamath*, and *Holophrasm*.

2 Background

The first "general automated theorem prover" came in the 1950s as the Logic Theory Machine designed by Newell and Simon. This system was implemented via a set of axioms and rules of inference, and then generated theorems by creating "chains" of these axioms. However this was quickly discovered to be inefficient for proving larger, more complex theorems. Searching for the correct steps in the proof is an np complete problem, so use of *tactics* developed to incorporate some level of human intuition and strategy into the automation.

This led to the development of different proof approaches beginning with resolution theorem provers and extending other methods. Quickly meaningful ways of classifying these systems emerged. For example, often times the study of automated theorem provers includes interactive theorem provers - systems used to write proofs in conjunction with human input. What's more there are many systems that exist between this gap, including elements of both automated and interactive provers.

Another distinction arises when discussing proof verification. Some systems developed for the verification of incredibly complex proofs. The best example of this is Tom Hales proof of the Kepler Conjecture, which contained massive arrangements of inequalities that could only be verified via computational means. Many proof verifiers accomplish this by starting with premises $\{\phi_1, ..., \phi_n, \neg\psi\}$

and seek to find contradiction, thus showing $\phi_1 \wedge ... \wedge \phi_n \implies \psi$. Yet while this is helpful for verification, these kinds of methods are not helpful for building human mathematical intuition and knowledge.

Thus, many constructive proof systems, both automated and interactive, use tactics to aide in the construction of a *proof tree*. The proof tree is dually useful, as firstly it can be thought as a map of varying techniques the human/computer system have used to attempt to complete the proof. Secondly, trees are an exceedingly well understood data structure, and there are many techniques in computer science that can be applied to make tree construction and search efficient.

Regardless of what system is being used, some kind of *formal system* is necessary for syntactically expressing the mathematics in a computer operable manner. Many such formal systems exist and range widely in their size, power, and human-readability. The creation of these formal systems allows for a rigorous representation of mathematical arguments to be constructed in a chosen logic with a set of axioms while simultaneously being interpretable by a computer system. This leads to an even higher level of rigor than standard mathematics is subjected, which is the primary motivating reason for computably formalizing mathematics. This higher level of rigor is also used by certain formal systems, such as TLA+, to prove properties and inavariants of computer architecture concurrency.

Several formal systems are implemented in strongly typed functional programming languages. This takes advantage of type theory, which gives every *well-typed* expression a type. For example, the *value* 5 has type *int*. We can define abstract datatypes to represent axioms and theorems, and use library functions as inference rules to generate new theorems. Thus the truth value of a proposition is then preserved through a functional inference rule enforced by the programs type checker. This reduces a constructive proof of a new proposition to finding the correct composition of functions in the language.

Some systems extend this version of type theory further, by incorporating both dependent type theory and constructing an inductive theory of types, where even a simple type such as *int* is given type *Type*. *int* is considered a value of a parent type *Type*. This *Type* is then again in turn considered a value of type *Type 1* and so forth. This is done to allow for construction of functions which take in types as arguments. These are used in constructing mathematical foundations of the system. This construction is facilitated by the Curry-Howard isomorphism theorem, which acts as a bridge between these logics and the languages they are implemented in.

Each formal system also varies in deciding which axioms are used to construct its foundations of mathematics. Some are formed strictly around the ZFC axioms without the axiom of choice. Others are extended with the axiom of choice, and yet others are constructed with the Tarski-Grothendieck axioms, allowing for higher constructions of higher order infinities allowed by conventional ZFC. Of courese the choice of axiomatic structure affects the type of mathematics constructed from the system. Another added challenge these systems face is the addition of extra "convenience" axioms that allow for the

proving of higher level theorems without the necessity of being proven from the ground up. As more of the foundations are constructed, this allows for the removal of these "convenience" axioms.

All but one of the systems are hybrid automated theorem provers. These hybrid systems generally consist of a formal specification language to write proofs, a formalized mathematical library, and a proof assistant. The best known example of the goals of these libraries is the formalization of the "100 Most Important Proofs In Mathematics" selected in 1999 by Paul and Jack Abad. While their selection is somewhat arbitrary, the consideration of such theorems makes for nice goals to work towards. While the size of a library is likely its most relevant measure, progress towards these proofs can be interesting also.

With all this in mind, we can begin discussion of the specifics.

3 Systems

3.1 Automath

Automath is an automated proof system created by de Bruijin in 1967. Although not prevalent today, it influenced many contemporary ATPs. The most notable of these are the Mizar system and Coq.

The Automath system allowed for the expression of complete mathematical theories in a formal language. Additionally it included an automated proof checker. In the most literal sense Automath is not then an "automated theorem prover" as it does not allow for the automatic generation of theorems. However, a common theme as I mentioned is that the field of Automated Theorem Proving contains many systems that are not fully automated.

3.2 Mizar

The Mizar System was created by Andrzej Trybulec in 1973. It includes a formal language, proof assistant, and library of curated mathematics. The proof checker is written in Free Pascal. Its basic axioms are those in Tarski-Grothendieck set theory. Currently the Mizar Library contains the largest collection of "formalized mathematics" in the world. In total it has around 9400 definitions and 49000 theorems formalized. The library is continuously growing as mathematicians submit theorems, both old and new, for formal curation. This process is fairly intensive and on average it takes about a week's worth of work to formalize a single textbook page.

A Mizar article is considered the atomic unit of the library. It consists of two sections: An *environment declaration* and the *text-body*. The *environment declaration* allows the user to specify which parts of the Mizar library the proof will be assuming, which allows for a more thorough process of construction and formalization. The *text-body* is the bulk of the proof.

Despite the Mizar Language being written in ASCII with an attempt to be similar to conventional proof style, it can still be complex. The formal language is modeled to be as close to conventional mathematical proof style as possible while still allowing for formalizability. This is achieved by representing large sets of recoginized mathematical symbols corresponding to the same "meaning" given a certain environment with a *constructor*. This *constructor* can really just be thought of as an ID number. Additionally the author of a Mizar article can extend the symbol lexicon of the Mizar system by using the *vocabularies* feature, which allows for the specification of certain symbols for *tokenization*.

The process of *tokenization* occurs as a Mizar article is being checked by the formal proof checker. The processing of a Mizar article occurs in three stages: First the *accomodator* processes the *environment declaration* and creates an enivronment for the verifier. Then the *verifier* processes the text proper of the Mizar article. The verier checks for the correctness of the proof given the information in the environment and prepares a report on any errors found. Finally, if the proof is valid, the *exporter* extracts usable "knowledge" to be added to the Mizar library and prepares an article for that purpose. This process was developed for the efficient and semi-automated production of formalized "knowledge" in the Mizar library.

Additionally, several tools are available to a Mizar author in the process of writing or revising an article. The MML Query allows an author to detect overlap with knowledge already in the database, and the proof advisor provides a method for finding potentially relevant theorems to the proof in question. The *exporter* plays a particularly crucial role in the process of efficiently storing mathematical knowledge, as it extracts unique proofs or theorems from new articles to be stored in the library.

Mizar is unique in the sense that it relies little on type theoretical foundations to implement mathematical verification and construction. Thus as Mizar's creator Trybulec said "While we in Mizar have a library of mathematics, those in type theory have libraries of systems". This is an interesting point, as it will soon be shown that a great number of other systems depend heavily on type-theoretical foundations.

Necessarily logic and the foundations of mathematics constitute the greatest majority of the articles. However, the Mizar library is slanted heavily towards proofs in analysis and algebra, with very little information on computer science literature. The Mizar project has proven 69 of the 100 "most important proofs".

3.3 Lean

Lean is both a theorem prover and a programming language. The system advertises itself as a "bridge between interactive and automated theorem proving". A lite javascript version can be accessed via the browser. An open source download gives access to the full system implemented in both Emacs and Visual Studio.

The language itself is structured on the Calculus of Constructions with inductive types. With this Lean extends simple versions of type theory by giving the typeclass *Type* to type names. This creates an inductively typed structure

where *Type* itself is of type *Type 1*, *Type 1* is of *Type 2*, and so forth. This allows for conveniently defined functions that take types as arguments, which has clear analogues in higher-order logic.

Additionally Lean utilizes dependent type theory extensively to construct proposition and theorem typeclasses. This dependently typed system leads to the creation of *holes* inside of typed expressions, which can be inferred to the correct type via a process of *Elaboration* made possible by the type information conveyed in a specific input value of some type. This is how Lean implements the theorem typeclass, with certain inference rules acting as polymorphic-dependently typed functions.

There are additional structural features such as sections and namespaces used to organize theorems and definitions, as well as libraries of propositions. Using the structures, Lean allows for theorem proving in a *term-style* and a *tactic-style*. A *term-style* proof is slower and fully interactive but more intuitive and readable. Conversely the *tactic-style* uses more complex tactics provided by Lean to attempt complex proofs.

3.4 TLA+

TLA+ is a formal specification system designed by Leslie Lamport in 1999. Unlike most of the other systems discussed here, the primary purpose of this system is to detect design flaws in concurrent computer systems. It includes a model checker which checks the correctness of algorthims for invariants such as "safety" and "liveliness". This was extended to mathematical proof checking.

TLA+ inclues a mechanical proof checker called TLC. This system runs the specification written in TLA+ by executing a finite number of steps to detect runtime errors exhaustively. This can detect simple errors in order to streamline the rigorous proof verification process. In contrast, TLAPS is the TLA+ proof checker. It can be used to check specifications written in TLA+ rigorously in the specification language. It is used after TLC has checked for simple errors. The mechanics of the proof checker involve a three step process. First the SMT solver is given 5 seconds to complete the proof. If this fails the Zenon solver is given 10 seconds. Finally if this fails the Isabelle theorem prover is given 30 seconds to complete the proof. It is given another 60 seconds if necessary. The system can be customized with an array of tactics to allow for longer solver timeout lengths.

TLA+ stands for temporal logic of action. Hence its logical founations are in temporal logic. This type of logic is particularly useful for describing the action of concurrent systems. The proof system uses the ZFC axioms to construct its mathematical foundations.

TLA+ uses a number of handcrafted tactics to check to rigorously prove correctness of a specification. To construct a proof a tactic is applied and then a number of *proof obligations* are generated. Then tactics are continually applied to the proof obligations until the proof is reduced to known theorems or axioms. The proof is then semantically correct if each proof obligation is correct.

TLA+ stresses the importance of writing abstract specifications that are not tailor made to the specific architecture of the mechanical theorem prover being used. In reality a TLA+ proof is checkable if each obligation is checkable, which simply means there is backend that accepts it. A proof writer in TLA+ should not be overly concerned with the specifics of the backend.

The syntax of TLA+ is conveniently very similar to Latex. There exist many programs that can convert between the two.

3.5 HOL

The HOL ATP refers to a collection of four similar systems genrally implemented in SML. They define an abstract data type representing theorems in an opaque library and then use library functions to represent valid rules of inference to construct new theorems.

It can be implemented as both an interactive proof checker and an automatic proof checker. The four systems include HOL4, HOL Light, ProofPower, and HOL Zero. HOL4 is implemented in ML on top of common Lisp. HOL Light started as a "minimalist version" of HOL4 implemented in OCaml but has quickly become extremely popular. The other two systems are similar to HOL4 but provide more specific tools for formalization.

The implementation of these systems was inspired by the LCF system constructed by Robin Milner. A type structure and library is used to construct an interface preserving soundness of the models being constructed in pursuit of proof. The HOL4 implementation develops structure based on a model M, a *type structure* Ω, a set of constants Σ_Ω called a *signature*, and *terms*. HOL4 designates a *standard form* of models and *type structure* to ensure soundness. Every standard *type structure* must contain an atomic bool type α representing a boolean value structure.

HOL also implements a system of terms to structure proof. It contains syntactic categories of types and terms whose elements are intended to denote sets and elements. There are four kinds of type terms: type variables which represent arbitrary sets, atomic types which denote fixed sets, compound types which are formed via operators *op* of arity n to construct new types, and function types. There are also four primitive terms: Constants, Variables, Function applications, and λ abstractions.

A HOL theory is similar to conventional mathematical theory, except it also tracks what has been proven from the axioms instead of what could be proven.

In order to formally define a theory, we introduce *sequents*. A *sequent* is a pair (Γ, t) with Γ a set of terms or assumptions and t the goal. HOL then formally defines a *theory* as a 4-tuple $< Struc_\tau, Sig_\tau, Axioms_\tau, Theorems_\tau >$. $Struc_\tau$ is the type strucutre of the theory, Sig_τ is a signature over $Struc_\tau$, $Axioms_\tau$ is the set of sequents over Sig_τ, and $Theorems_\tau$ are a set of sequents over Sig_τ s.t. every members follows from the axioms. HOL develops an initial theory called *INIT* that is sound and then preserves soundness using standard models described above.

HOL users can extend theories. A theory τ is an extension of t if $Struc_t \subseteq Struc_\tau$, $Sig_t \subseteq Sig_\tau$, $Axioms_t \subseteq Axioms_\tau$, $Theorems_t \subseteq Theorems_\tau$. Extensions are constructed usually in one of two methods. These are extension by constant specification or extension by type specification. Using these tools, users can construct new theories in the HOL language. Theorems are then constructed inside HOL theories via chains of sequents.

The other languages are structured similarly to HOL4, with a similar or simplified foundation. The specific type of logic used is predicate calculus with type theory, as implied by the name HOL(Higher Order Logic). By default the system uses the axioms of ZFC minus the Axiom of Relacement plus the Axiom of Choice. The systems have had great success, with the HOL Light proof library containing 86 of 100 of the "100 most important mathematical theorems".

3.6 Coq

Coq is an interactive theorem prover inspired by Automath and LCF. Originally conceived of as a language used to computably formalize properties of programming languages, it grew to be useful in formalization of mathematical arguments. It consists of a formal language, library, proof checker, and proof assistant.

Similar to Lean, Coq uses a Calculus of Inductive Constructions, which is implemented using a type-theoretic model. The specification langauge is named Gallina. Like many other type-dependent systems, Gallina takes advantage of the Curry-Howard isomporphism to implement mathematical proof in terms of types. This means that all logical judgements become typing judgements checked at compile time by a type-checker.

The proof assistant come in two forms. The interactive version is usable through some IDE: Emacs for example. When using the interactive version the user can querry the system for available definitions and proofs and use prespecificed tactics to attempt to complete a proof. The checker may also be run via a file that is typechecked at compile time. This version acts more as a verifier than as a proof assistant.

Gallina includes a language called *The Vernacular* which is used to specify commands. The system assumes that all terms are well typed when executing commands. The range of commands includes assumptions, definitions, and annotated types.

The Coq library system is large, having completed 74 of the 100 "most important theorems". When run, the system automatically loads the *Initial Library* which contains notions of elementary logics and data-types. This can be extended by the user defined libraries.

3.7 Metamath

Metamath is a formalized language with a proof checker and formalized mathematical library. It is based on the axioms of ZFC set theory. Additionally the

decision can be made to add Tarski's axiom if desired.

The formal language is constructed with two types of variables: set variables and *wff* - well-formed formula. Set variables represent any mathematical set theoretic object. *wff*s represent propositions about set variables. This forms the basic syntactic structure of the Metamath language.

Metamath then uses the Curry-Howard isomorphism to construct a theory of classes that is an eliminable and conservative extension of set theory. Over the development of the Metamath system, the developers have been able to reduce the number of additional higher level axioms by building the mathematical foundations via ZFC. However they still retain some "convenience" axioms.

As opposed to Mizar, Metamath is structured for less professional mathematical users. It includes features such as "Music of Proofs" which generates a simple melody and harmony for each proof based on its proof tree's size and depth. This extends itself to the formal language, which is designed with as much simplicity as possible.

Mathematical proofs in the language consist of entirely symbol substitutions in the language of set variables and wffs in the case of a theorem. On the one hand this makes the proofs easy to follow, as they are entirely substitution. On this other hand this does not mirror how modern mathematics is constructed, which makes it difficult for professional use. Additionally, it makes fully extended proof trees enormous for complex proofs. For context, the proof that $2 + 2 = 4$ has around 25,000 total substitutions. Even with this complexity, the Metamath library has over 20,000 proofs. Additionally, the system has proven 67 of the 100 "greatest theorems".

3.8 Holophrasm

Holophrasm is a completely automated theorem prover designed in 2017 using AI machine learning techniques to efficiently prove test theorems.

It is structured on the Metamath system, and uses the Metamath langauge to construct a completely automated proof search. The algorithm used searches the space of partial proof trees of a metamath theorem with an "algorithm from the tree-based multi-armed bandit problem"[7].

By structuring a *goal* assertion in the Metamath system, the holophrasm system develops the partial proof tree to find a full proof tree. A full proof tree is defined by a partitioning of red and blue nodes. Red nodes correspond to expressions or *goals* to be proven. Blue nodes contain a pair of expressions (T, ϕ) where T is a theorem being applied to the parent red node and ϕ is the instantiation of the application. This blue node then gives rise to more blue nodes which are expressions of the hypotheses of the theorem T which remain to be proven.

The partial proof tree is similar in structure to the proof tree, but is necessarily a supertree of any full proof tree. In this structure red expression nodes may have multiple blue node children representing the applications of multiple potential theorems. Then once a full proof is found the partial proof tree is pruned to give the resulting full proof tree.

In order to construct the partial proof tree the system begins with the original *goal* assertion and iterates downward. At each red node the *evaluative* neural network calculates an initial value, and then a total value corresponding to the certain characteristics of the red nodes children. Then the algorithm decides whether to apply and new theorem and create an new child blue node, or traverse down the highest valued child node. The value of a blue node is then computed by the *evaluative* network by taking the minimum value of the nodes' childrens' values and adding a factor v_b calculated by the *relevance* and *generative* neural networks. This continues until a proof is found, and the partial proof tree is primmed of all the unproven *goal* expression branches. Several other techniques are implemented to improve algorithmic efficiency, but this is the main structure of the process.

Holophrasm was tested using 2720 proofs from the Metamath library. It found proofs for 388 theorems, which is a 14.3% of the test set. Unsurprisingly, Holophrasm fared comparatively better on the foundational proofs of the database, proving 45.1% of them. This is likely due to the combinatorial explosion each successive proof step experiences. What is more, when the system did discover proofs, it did so in an average of 17 passes, which is relatively fast.

4 Conclusion

Automated theorem proving includes a wide array of various systems which are not entirely automatic in nature. Each system differs in several characteristics including its rigor of formalization, foundations axioms and logic, and level of automation.

Several of the systems ultimate goal is to construct formalized libraries of mathematics rigorously machine checked by custom verification systems. This most notably includes the Mizar System, Coq, HOL, and Metamath. Each uses a similar but unique formal language to store articles. Type theory is heavily used via the Curry-Howard isomporphism to store mathematical language programatically and implement proof verification. The exception to this being the Mizar System, which does not use a type theoretic foundation.

Yet other systems focus on the specification and testing of concurrent systems: TLA+ and Coq again. These systems too have proof assistants and proof checkers, but TLA+ also has a mechanical tester that can test a specification in a brute force style. This allows for more streamlined testing processes that then allow for selective use of proof checkers to fully and rigorously formalize an invariant or theorem. These systems again use type-theoretic notions to implement formal systems.

Lean stands in between this gap, acting simultaneously as a programming language and a theorem prover.

Holophrasm stands alone as the only true "automated theorem prover". Studying its implementation gives insight into the proof tree methods of other systems like Metamath and other languages that have the option of proving theorems constructively. What Holophrasm does uniquely is implement machine

learning networks in an attempt to develop "machine learned tactics", to some degree of success.

All the systems together have formalized 93 of the 100 "most important proofs", which is an impressive feat.

5 Sources

HOL Logic Guide
HOL Development Guide
https://dal.academic.ru/dic.nsf/enwiki/4477840
http://www.wolframscience.com/nks/notes-12-9–automated-theorem-proving/
http://www.cs.ru.nl/ herman/talk-2016-China.pdf
https://leanprover.github.io/about
https://leanprover.github.io/theorem$_p roving_i n_l ean/theorem_p roving_i n_l ean.pdf$
$http://www.cs.tau.ac.il/\ msagiv/courses/ATP/lecture-1.pdf$
$https://arxiv.org/pdf/1608.02644.pdf$
$http://mizar.org/$
$https://link.springer.com/content/pdf/10.1007\%2Fs10817-017-9440-6.pdf$
$http://us.metamath.org/$
$http://www.cs.ru.nl/\ freek/100/$
$https://coq.inria.fr/$
$https://hol-theorem-prover.org/about$
$https://lamport.azurewebsites.net/tla/book-02-08-08.pdf$
$http://tla.msr-inria.inria.fr/tlaps/content/Home.html$
$https://pdfs.semanticscholar.org/presentation/16cd/fdc0153da40b4f662853fdad3b38563508e3.pdf$