Consult the Scholar

The Role of Artificial Intelligence in Theorem Proving

CHARLES AVERILL, University of Texas at Dallas, USA

Theorem proving is one of the oldest targets of artificial intelligence research, and the question of "how much of the work can be automated?" remains open. This survey examines the history and current state of AI-assisted theorem proving, covering rule-based approaches, classical machine learning, deep learning, and large language models. We organize the landscape around two paradigms: rule-based methods, which encode domain-specific reasoning strategies as explicit procedures, and machine learning methods, which train models to predict reasoning steps from data. For each paradigm we survey representative systems, describe their core techniques, and situate them in the broader context of the field's development. We discuss open problems and argue that the most productive near-term role for AI in theorem proving is to absorb the mechanical and repetitive parts of proof work, rather than to replace human mathematical reasoning.

ACM Reference Format:
Charles Averill. 2026. Consult the Scholar: The Role of Artificial Intelligence in Theorem Proving. 1, 1 (May 2026), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

For thousands of years, the advancements of the human race have been fueled by our efforts in mathematics. Agriculture, architecture, warfare, and exploration have evolved dramatically as mathematicians built powerful abstractions to efficiently reason about the world around us. This growth was enabled by tedious, careful, and error-prone calculations performed manually. To offset this overhead, we constructed elegant methods to solve problems more easily: early architects suspended chains to approximate catenary curves for building arches [9]; merchants adopted the abacus to accelerate arithmetic; Babbage designed the first mechanical calculator [60] to eliminate errors in nautical tables; and engineers built the vacuum tube systems and transistor machines that became the first modern computers. Each generation of tools was motivated by the desire to offload calculation so that human minds could focus on harder problems.

Roughly one century ago, this paradigm was uprooted as a group of mathematicians founded the field of computer science [19, 24, 73, 78] in an effort to better understand the capabilities of formal logic. This work was a direct response to Hilbert's program [92], which challenged the mathematical community to find a complete and consistent set of axioms from which all mathematical truth could be derived. Church, Turing, and their contemporaries showed that no such procedure could exist in full generality. In doing so, they developed precise definitions of computation, forming the theoretical foundations upon which all modern computers rest.

Although the machines built on these fundamental theories are primarily used for communication, commerce, and war, they were initially designed to perform automatic reasoning in mathematics. The earliest pioneers of *artificial intelligence* considered machine intelligence to be synonymous with the *automated reasoning* capabilities for mathematical and logical problems. Over the following

Author's Contact Information: Charles Averill, charles@utdallas.edu, University of Texas at Dallas, Richardson, Texas, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2026/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

decades, however, practical pressures pushed formal methods and artificial intelligence in different directions. Computer-aided mathematics became a specialized discipline focused on proof assistants and verification, while AI research focused on perception, heuristic search, and statistical learning. The two fields developed largely independent methodologies, communities, and vocabularies.

As the world grows increasingly dependent upon digital infrastructure, we are once again incentivized to pursue automated reasoning capabilities in the name of security and stability. Subtle bugs in mission-critical software, communication protocols, and formal specifications represent systemic risks at a scale that manual review and spot checks cannot address. The task of having a machine discover or verify mathematical proofs offers a path toward reliable software and systems. This renewed urgency has drawn the AI and formal methods researchers back together, and today we see the two fields converging once more as large language models, reinforcement learning, and neural search are brought to bear on automated proof generation.

A number of fundamental challenges remain open: the field has converged heavily on a few sub-problems, which receive varying amounts of interest over time; the probability-distribution-based generation of modern language models makes them poorly suited for the open-ended conjecture generation that theory exploration requires; and evaluation methodology is unsettled, as existing datasets and metrics may not adequately measure generalization to novel mathematical reasoning. Addressing these challenges has motivated a diverse landscape of approaches, broadly organized around two paradigms. *Rule-based* methods encode domain-specific reasoning strategies as explicit procedures, and *machine learning* methods train models to predict reasoning steps.

The remainder of this survey is organized as follows. Section **??** provides background on theorem provers, machine learning, and the formal foundations underlying both. Section 3 surveys the history of AI in theorem proving. Section 4 covers machine learning approaches, from classical premise selection to large language models. Section 5 covers rule-based approaches, including inductive automation and theory exploration. Section 6 presents related surveys and complementary work. Section 7 discusses open problems and promising directions.

1.1 Scope

We consider research in the context of automating the construction of formal proofs via *Artificial Intelligence* (AI) in *Interactive Theorem Provers* (ITPs) or *proof assistants* and *Automatic Theorem Provers* (ATPs). To avoid the philosophical problem of defining "AI," we restrict the scope of this survey to techniques that enhance the capabilities of ITPs and ATPs while residing in at least one of the following categories:

- *Machine Learning*: learning some function over input training data that generalizes to unseen inputs, such as k-Nearest Neighbors [30] or neural networks [58]
- *Knowledge Management*: synthesizing and exploring sets of knowledge to call upon for reasoning, such as approaches targeting premise selection and loop invariant synthesis
- *Heuristic*: encoding specific reasoning strategies as a procedure to be used by an ATP or ITP user, such as rippling [15]

These categories restrict AI proof automation efforts to those the authors find the most relevant, successful, interesting, and applicable to theorem prover users. We exclude some approaches, such as tactic languages and domain-specific tools, that are explored in more depth in other surveys as described in §6. Furthermore, this survey focuses on generic techniques that could reasonably be applied to any domain, rather than approaches that target one specific application of formal methods.

99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147

2 Background

2.1 Theorem Provers

Theorem provers are a category of software designed to check and/or generate proofs for formal mathematical statements. *Automated* Theorem Provers (ATPs) aim to automatically generate and check proofs for a given theorem, while *Interactive* Theorem Provers (ITPs) enable a user to manually describe proofs to be checked. Both have been utilized to prove complex mathematical statements.

2.1.1 Automated Theorem Proving. Automated theorem proving was among the earliest ambitions of computer science, and for a time was nearly synonymous with artificial intelligence. The Logic Theorist [63], developed in 1956, was the first program to automatically prove mathematical theorems correct, successfully discovering proofs for 38 of the 52 theorems in Principia Mathematica [86]. Early researchers were highly optimistic about the application of computers for automated reasoning. They hoped that, given a sufficiently rich encoding of mathematical knowledge, computers could autonomously discover and verify new theorems solely through symbolic reasoning.

This optimism proved difficult to sustain as problems grew in size and complexity. ATPs made rapid advances through the 1960s via heuristics and new search algorithms, but it became increasingly clear that proof search is sensitive to exponential explosions in computational complexity, and that purely automatic approaches would soon plateau when faced with any theorem requiring deep insight or background knowledge. By the 1970s, the practical demands of software verification (in which theorems are numerous, highly-specific, and complexly inter-dependent) made it apparent that human guidance was indispensable for non-trivial works.

Despite these challenges, automated theorem proving persists still, and has demonstrated great applications with relatively small human guidance. Rather than serving as end-to-end proof discovery engines, they remain as powerful sub-components that are called upon to discharge smaller problems that humans identify as necessary to solve larger problems. Systems such as ACL2 [53], which automates inductive reasoning over recursive programs, have been used to reason about complex objects such as microprocessors [14]. SMT solvers such as Z3 [26] are now routinely embedded as decision procedure backends in larger verification frameworks and ITPs. The most widely used ATPs today, (e.g., E [74], SPASS, Vampire [68]) are commonly invoked in *hammer* systems [8] that translate proof goals into first-order logic statements to automatically dispatch.

2.1.2 Interactive Theorem Proving. Interactive theorem proving emerged as a distinct field in the 1960s, motivated by the growing recognition that full automation was infeasible for large proofs. The core idea is that both the human and machine contribute what they do best: the human supplies mathematical intuition, high-level strategy, and forward reasoning steps such as key lemmas, while the machine can solve tedious, repetitive tasks and check proofs with high assurance.

Automath [61] was arguably the first interactive theorem prover, and set several patterns that would be inherited by its decendents. It was the first proof system to utilize the Curry-Howard Isomorphism [76], which describes a relationship between type systems and propositional logic, as the basis for a proof checker. The idea that a proof is a well-typed term in a suitable type theory is foundational, and became the backbone of modern ITPs such as Rocq [42]. Indeed, Rocq is a direct descendent of Automath; the development of the Calculus of Constructions [22] (the initial underlying type theory of Rocq) was influenced by it.

The CoC incorporated another core aspect of many modern ITPs: *dependent types*. Knowing that propositions, specifications, and relationships can be encoded as types, dependent typing parametrizes these by *program values*. This enables the formal statement of phrases such as "for given numbers $n,\ m$, $n + m = m + n$" (the equality is parametrized by the inputs n, m). Dependent

types substantially increase the expressiveness of a proof language, allowing specifications and proofs to be written in a single unified framework.

As the limitations of pure automation became clear through the 1970s and 1980s, ITP attracted increasing research attention, producing families of influential systems built either on Rocq's approach (encoding theorems as types) or another interpretation of the Curry-Howard Isomorphism in which theorems are more loosely associated with classes of proofs: LCF [36]. LCF introduced the key idea of implementing a proof assistant in a typed meta-language such that only the primitive inference rules can construct theorems, guaranteeing soundness by construction regardless of the complexity of structures built atop this kernel. LCF's type theory underlies HOL [35], Isabelle [65], and their descendants, which use classical higher-order logic (rather than CoC-style constructivist type theory) and have been applied to landmark verification efforts such as seL4 [54].

By the 1990s and 2000s, ITP had become the dominant paradigm for serious formalization work. ATP came to serve primarily as an automation component in larger interactive works.

2.2 Machine Learning

Machine Learning (ML) is a field of research with the aim of building computer systems that can approximate generic functions from specific training samples. In other words, ML models learn relationships between the conditions of a system and measurements of that system such that they can be used to predict the behavior of the system in unseen conditions. Canonical ML tasks include *classifying* an input into one of multiple categories, *regression* to predict continuous numerical values, *clustering* common inputs together, and *anomaly detection*.

Learning to perform these tasks entails iteratively modifying a model (some data structure encoding a function approximation), also referred to as *training*. Training is either *supervised*, where input samples are paired with labeled outputs, or *unsupervised*, where only inputs are provided. *Inference* is the process of supplying an input to a trained model to receive a prediction. In some systems, the input is referred to as a *prompt*.

Models can vary greatly in complexity, intended purpose, their ability to generalize from few training samples, etc. We present some common model types that appear often in AI theorem proving efforts.

2.2.1 k-Nearest Neighbors (k-NN). k-NN [30] is a fundamental ML algorithm used for classification and regression by grouping the k most similar points in a dataset. Its application is natural for classification, as grouping an input with its most similar training samples is an intuitive heuristic for classifying samples. For regression, an input's most similar training samples are averaged to predict the output, with the intuition that nearby inputs are likely to have nearby outputs.

2.2.2 Bayesian Network. Bayesian networks are directed acyclic graphs in which nodes represent statistical variables and edges represent probabilistic dependencies. Inference on these graphs occurs via Bayes' Theorem [6]. Bayesian networks can be used for classification by modeling the joint distribution of input features and class labels, allowing computation of the probability of each class given observations. For regression, they can model continuous variables and their dependencies, enabling prediction of a target variable as a conditional expectation given observations.

2.2.3 Neural Network (NN). Neural networks [58] are a family of ML models loosely inspired by biological neural structures. A network is organized into layers of interconnected nodes called neurons, where each neuron computes a weighted sum of its inputs and passes the result through a nonlinear activation function. Layers are stacked sequentially: an input layer receives raw features, one or more hidden layers learn intermediate representations, and an output layer produces a prediction. Training adjusts the weights of every neuron via *backpropagation*, which propagates

prediction error backward through the network and applies *gradient descent* to minimize a loss function over training data. The *depth* of a network is the number of hidden layers within, and correlates with its capacity to learn hierarchical features. NN architectures with many layers are referred to as deep neural networks.

2.2.4 Recurrent Neural Network (RNN). RNNs [71] address a challenge faced by standard NNs: handling variable-length input sequences. They accomplish this by introducing a hidden state that is updated at each position of an input sequence, effectively giving the network a form of memory over prior inputs. At each step, the network receives the current input together with the previous hidden state, producing both an output and an updated state to be carried forward. This recurrence enables RNNs to capture dependencies across sequences of arbitrary length, making them well-suited for language modeling and proof script processing.

A practical limitation of vanilla RNNs is that gradients shrink exponentially when backpropagated through many time steps, making it difficult to learn long-range dependencies. This is referred to as the *vanishing gradient problem*. Gated architectures such as the Long Short-Term Memory (LSTM) [44] and Gated Recurrent Unit (GRU) [18] mitigate this by introducing learned gates that control what information is retained or discarded as the hidden state evolves.

2.2.5 Transformer. The Transformer [83] is a neural network architecture designed to model sequences without recurrence, relying entirely on a mechanism called *self-attention*. Self-attention allows every element of an input sequence to directly interact with every other element simultaneously, rather than processing the sequence step by step as RNNs do. This enables highly parallel training on modern hardware and sidesteps the vanishing gradient problems that plague recurrent approaches. Transformers are organized into stacked blocks of attention and feed-forward layers, and can be configured to encode inputs, generate outputs, or both.

Transformers form the basis of large language models (LLMs), which have demonstrated strong generalization to downstream tasks via fine-tuning or in-context prompting, and have become the dominant architecture in recent AI theorem proving work. Notably, they require substantially larger datasets than other ML models in order to effectively learn the patterns of the input space.

2.3 Machine Learning Strategies

The previously described models must be trained via some strategy that determines what signal is used to update the model's behavior. We describe the four strategies most relevant to AI theorem proving.

2.3.1 Supervised Learning. In supervised learning, a model is trained on a dataset of input-output pairs, where each output is a ground-truth *label* provided by an external source (e.g., an expert human). The model is optimized to minimize the discrepancy between its predictions and the provided labels, typically via gradient descent on a loss function. In the theorem proving setting, supervised learning commonly takes the form of training on human-written proof corpora, where the inputs are proof states and the outputs are the strategies a human chose to apply.

2.3.2 Unsupervised Learning. Unsupervised learning does not require labeled outputs; instead, a model is trained to find structure in unlabeled input data alone. Common unsupervised objectives include clustering inputs into groups, learning compact latent representations, and modeling the distribution of inputs so that likely new samples can be generated.

2.3.3 Imitation Learning. Imitation learning trains a model to replicate the behavior of an expert by treating recorded expert demonstrations as supervised training data. In the context of ITP, the expert is typically a human prover, and demonstrations are existing proof scripts; the model learns

to predict the next proof strategy by imitating the sequence of choices the human made. A key limitation of imitation learning is *distribution shift*: at inference time the model may encounter proof states that never appeared during training, and errors can compound as the model navigates increasingly unfamiliar territory without a mechanism for recovery.

2.3.4 Reinforcement Learning. Reinforcement learning (RL) trains an agent to maximize a cumulative *reward* signal obtained by interacting with an environment. Rather than imitating fixed demonstrations, the agent explores possible actions, receives feedback on their outcomes, and updates its policy to favor actions that lead to higher reward. RL sidesteps the distribution shift problem of imitation learning by training directly on the states the agent actually visits, but requires a large number of interactions to learn effectively.

2.4 Program Analysis

Program analysis is a category of techniques for formally reasoning about the behavior and composition of computer programs. Formal program analysis usually seeks to prove some correctness or safety property for all of a program's input-output pairs. These methods contrast with standard unit testing, which usually can only prove these properties for a small subset of the (potentially infinite) input-output space.

Formal software verification frequently involves many redundant or near-trivial steps in between stating and proving deep properties of a program's sub-components. Thus, it is a highly sought-after target in proof automation, and thus in the surveyed AI proof automation approaches.

Unfortunately, formal program analysis involves distinct and uniquely challenging reasoning steps that are used less frequently in standard mathematics proofs. This is largely due to the use of *repetition* or *iteration* in programming, where mathematicians largely prefer to quickly abstract away from these concepts as they appear. Because of the ever-presence of iteration in computer programs, formal software verification does not reduce nicely to symbolic manipulation, as much of mathematics does.

The surveyed automation techniques primarily target two avenues for performing formal software verification: *invariant generation* and *dependent types*.

2.4.1 Invariant-Driven Verification. Hoare logic [43] provides a formal system for reasoning about the partial correctness of imperative programs. The central construct is the *Hoare triple* $\{P\}\ C\ \{Q\}$, which asserts that if precondition P holds before command C executes and C terminates, then postcondition Q holds afterward. Hoare logic includes inference rules that describe how triples compose: for example, the sequence rule states that $\{P\}\ C_1;\ C_2\ \{R\}$ follows from $\{P\}\ C_1\ \{Q\}$ and $\{Q\}\ C_2\ \{R\}$. This transforms the question "does this program do the right thing?" into a collection of proof obligations over a program's sub-components, which can then be discharged by a theorem prover.

The central challenge of applying Hoare logic to programs with loops is that a loop may execute an arbitrary number of times. The standard solution is a *loop invariant*: a predicate I that holds before the loop begins, is preserved by every iteration of the loop body, and is strong enough to imply the desired postcondition when the loop exits. Given a loop `while` B do C, the Hoare rule for loops requires finding an I such that $\{I \wedge B\}\ C\ \{I\}$; the invariant then discharges the entire loop with the triple $\{I\}$ `while` B do C $\{I \wedge \neg B\}$.

Consider a simple program that computes the sum of the first n natural numbers by accumulating into a variable s as i counts from 0 to n. The invariant $s = \frac{i(i-1)}{2} \wedge 0 \leq i \leq n$ captures the relationship between s and i at every iteration. It holds before the loop ($i = 0, s = 0$), is preserved by each

```
add(z, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

Fig. 1. Peano addition implemented in Prolog

increment of i and corresponding update to s, and when the loop exits with $i = n$ it implies $s = \frac{n(n-1)}{2}$, the desired postcondition.

Finding such invariants by hand for production-level software is tedious and is frequently the most labor-intensive step in a formal software verification proof. It is for this reason that invariant generation is a primary target of the proof automation techniques surveyed.

2.4.2 Dependent Types. The type systems of most programming languages assign types to expressions based solely on their syntactic structure: a function `sort` might have type `List` → `List` regardless of the content or length of its argument. *Dependent types* remove this restriction by allowing types to depend on *values*, making it possible to natively express values with types such as "a list of exactly n elements" or "a number n such that $n > 0$".

These types can encode not only primitive properties of data structures (such as list length) but can in fact encode arbitrary mathematical properties of program values. For example, consider a function `filter` that filters out list elements that do not satisfy some predicate f. In a language with dependent types, one may define the `filter` such that its return type is

$$\{l : \texttt{list} \mid \forall i,\ i < |l| \implies f(l[i]) \text{ holds}\}.$$

That is, the function itself returns (alongside the return value) a proof that all of the return value's elements satisfy f.

Dependently-typed programming languages such as Rocq [42], Lean [27], and Agda [10] utilize dependent types to allow users to state arbitrary theorems encoded as dependent types. These theorems can contain a function's input and output values and the properties that describe them. Users state their desired correctness and security properties as dependently-typed theorems and construct proofs that show that the properties hold.

2.5 Logic Programming

Logic programming is a paradigm in which a program is expressed not as a sequence of instructions but as a collection of logical *facts* and *rules* that together define a knowledge base. Computation proceeds by posing a *query* to this knowledge base and asking whether there exists an assignment of values to the query's variables that is consistent with the known facts and rules. Logic programming forms the basis for modern proof search techniques, and logic programs can reasonably be interpreted as involving proof search in their execution.

2.5.1 Knowledge Bases and Queries. Consider representing the natural numbers in Peano style, where `z` denotes zero and `s(N)` denotes the successor of `N`. Addition can then be defined entirely as a set of facts and rules in a logical programming language such as Prolog [82] as seen in Figure 1.

In Figure 1, the first clause is a fact asserting that adding zero to any `Y` yields `Y`. The second is a rule asserting that adding `s(X)` to `Y` yields `s(Z)`, provided that adding `X` to `Y` yields `Z`. A query such as `?- add(s(s(z)), s(z), R)` asks the system to find a value of `R` satisfying the relation, which it will bind to `s(s(s(z)))`.

2.5.2 Proof Search as Traversal. Answering a query requires the system to find a sequence of rule applications that derives it from the known facts. Prolog accomplishes this via depth-first search with *backtracking* over the *proof tree* induced by the rules. Backtracking is the process of re-tracing

the path already taken on the search to change some earlier decision if it is found that the current path is un-satisfiable, or if the user has requested multiple answers to a problem.

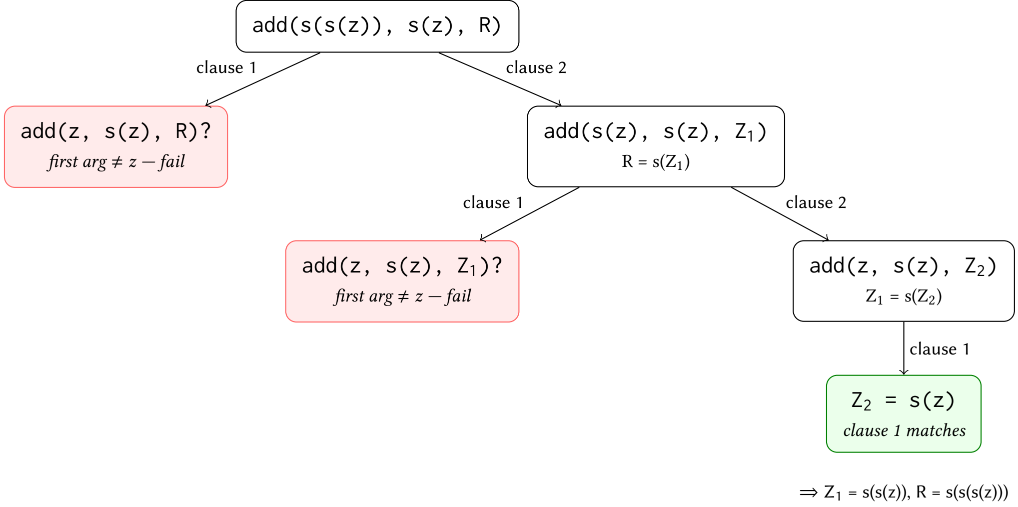

Fig. 2. Proof search tree for `add(s(s(z)), s(z), R)`

Figure 2 shows an example derivation for the query `add(s(s(z)), s(z), R)`. To answer the query, Prolog cannot apply the first clause since the first argument is not `z`, so it applies the second, unifying `X=s(z)` and `Y=s(z)`, and reducing the query to `add(s(z), s(z),` Z_1`)`. The same rule applies again, reducing further to `add(z, s(z),` Z_2`)`, at which point the first clause matches immediately, binding Z_2`=s(z)`. Unwinding, Z_1`=s(s(z))` and therefore `R=s(s(s(z)))`, the correct encoding of 3.

The structure of this search is essentially the same as tactic-based proof search in an ITP: the current query is a goal, rule applications are tactics, and the system backtracks when a chosen rule leads to a dead end. The primary distinction is that logic programming targets decidable or semi-decidable queries over explicitly enumerated facts, while ITP proof search operates over richer logics with user-defined tactics and heuristics to manage a much larger search space.

2.6 Proof Synthesis

Proof synthesis is the automatic construction of proof objects to be checked by a theorem prover, and is the end goal of proof automation. We identify four high-level tasks that are necessary for proof synthesis:

- *Tactic Prediction*: choosing which proof strategy to take next given the current proof context
- *Premise Selection*: choosing which axioms, facts, or previously-proved theories are relevant to the task at hand
- *Theory Exploration*: choosing when it is appropriate to state and prove useful lemmas, and choosing which lemmas to prove, to make progress on the current proof context
- *Autoformalization*: converting natural-language theorem statements into formal definitions

While each of these categories are relevant for ITP, tactic prediction is notably not applicable for ATPs, in which there is no notion of tactics or proof scripts.

3 Overview

The history of AI in theorem proving is a story of repeatedly revised expectations. In the earliest decades of the field, artificial intelligence was not yet clearly outlined as a discipline, and heavily overlapped with automated reasoning. Early research into automated reasoning systems was largely concerned with symbolic reasoning, problem solving, and the mechanization of logic. Systems for automated theorem proving therefore were seen as direct embodiments of intelligent, autonomous behavior.

The Logic Theorist [64] is arguably the first automated theorem prover, famed for having proved 38 of the 52 theorems in Principia Mathematica in 1956. Given its successes, it seemed briefly possible that full automation of mathematical reasoning was within reach. This optimism was sustained through the early 1960s as new search algorithms such as resolution refutation [70] augmented the capabilities of automated reasoning systems in large increments. This optimism did not survive contact with large-scale problems in mathematics and program analysis. Proof search turned out to be acutely sensitive to exponential blowup, and purely automatic approaches plateaued quickly when faced with any theorem requiring genuine mathematical insight or extensive background knowledge. By the 1970s it was clear that despite an abundance of impressive discoveries in automated theorem proving, purely-automated systems would be unable to reach the lofty goals envisioned decades earlier.

The response to this plateau was not to abandon automation but to reconceive its role. Two complementary research directions emerged. One branch, *interactive theorem proving*, relied on humans to dictate the flow and low-level details of proofs, while enabling custom automation as seen fit. Automath [61], developed by de Bruijn in the late 1960s, was the first to utilize the Curry-Howard isomorphism as the formalism behind proof encoding as typed terms, establishing the foundations that modern proof assistants such as Rocq, Lean, and Agda inherit. LCF [36], developed at Edinburgh and Stanford in the 1970s, guaranteed proof soundness by construction by implementing a proof assistant in a typed meta-language so that only primitive inference rules can construct theorems. This architecture became the backbone of HOL [35] and Isabelle [65], and remains standard today. Boyer and Moore's Nqthm [11] pioneered another branch, accepting that full generality was unnecessary for practical targets, such as software and hardware verification, and relied upon human interaction (e.g., stating intermediate results to prove in the service of larger ones) and deeply-automated systems for the specific class of theorems that target required. Nqthm's waterfall model and its successor ACL2 [53] demonstrated impressive competence on complex proof engineering problems, including the verification of commercial microprocessors [14] and the correctness of the RSA encryption algorithm.

These two branches developed largely in parallel throughout the late 20$^{\text{th}}$ century, each accumulating landmark results that clarified what the technology could and could not do. Gonthier's 2005 machine-checked proof of the Four Color Theorem in Coq [32] was a huge turning point for the field: it validated a result that had been controversial since Appel and Haken's 1976 computer-assisted proof [1], widely criticized because the computer component could not itself be checked. The Flyspeck project, completed in 2014, carried this further by producing a fully formal proof of the Kepler conjecture [38], solving a decades-old problem. In 2009, Klein et al. proved the functional correctness of the 8,700 lines of C code making up the seL4 microkernel [54], establishing that ITP was viable for industrial-scale software. The Mizar Mathematical Library [2], one of the largest repositories of formalized mathematics, further demonstrated the scalability of interactive proof development.

Meanwhile, the question of how to reduce the human burden in ITP was generating its own research thread. The hammer paradigm [8], in which ITP goals are automatically translated into

first-order logic and dispatched to off-the-shelf ATPs, was a practical answer. *Premise selection*, the challenge of choosing which lemmas from large libraries to include in the ATP call, became a recognized subproblem, and the first ML-based approaches to it arose from the Automated Reasoning in Large Theories movement. MaLARea [81] showed that a Bayes network trained on syntactic symbol co-occurrence could substantially outperform naive premise selection on large mathematical libraries, solving 165 of 252 problems on the MPTP challenge set [79] compared to 104 for the best purely symbolic baselines. This was the entry point of machine learning into theorem proving proper, not as a proof engine but as an intelligent filter on the combinatorial explosion of potentially relevant facts.

With the re-emergence of deep learning in the 2010s, the problem space was again revolutionized. DeepMath [47] in 2016 was the first to apply neural networks to premise selection, demonstrating that CNNs operating directly on formal syntax could learn relevance judgments competitive with hand-engineered features. As research effort pivoted towards focusing on interactive theorem proving, the task to solve accordingly shifted from premise selection to tactic prediction. GamePad [45], HOList [3], miniF2F [94], and CoqGym [89] contributed learning environments consisting of large corpora of human-generated proofs for models to train on. The insight driving deep learning-based systems was that the sequential structure of a proof is amenable to the same sequence modeling techniques that had succeeded in natural language processing, and that the formal proof state provides a precise training signal unavailable in informal mathematics.

The emergence of large language models produced the most recent and dramatic shift. GPT-f [66] showed that a Transformer pre-trained on mathematical text could be fine-tuned for tactic prediction, with domain-specific pre-training on arXiv papers providing a measurable advantage over general internet corpora. The key observation was that LLMs arrive at formal proof tasks already equipped with broad mathematical intuition acquired from textbooks and informal proofs, knowledge that earlier neural provers had to learn from scratch from formal libraries alone. This pre-trained knowledge proved transferable and complementary to symbolic methods. Draft, Sketch, and Prove [49] exploited it to generate informal proof outlines that guide ATP search, achieving roughly 40% on miniF2F, a 20–30 percentage point improvement over symbolic baselines alone. LeanDojo [90] then identified a methodological problem that had inflated much of the reported progress. Naive train/test splits allow theorems in the test set to share premises with nearly identical training examples, and correcting for this retroactively reduced success rates substantially across prior work.

The arc traced by AI theorem proving over the past seventy years is a gradual one. Full automation, once imagined as imminent, gave way to the more durable insight that humans and machines contribute complementary strengths: machines excel at exhaustive search, bookkeeping, and the discharge of routine sub-problems, while humans supply the mathematical intuition and strategic judgment that no system has yet replicated. The landmark results of the 21st century validated interactive proof development as a serious engineering discipline rather than an academic curiosity. The subsequent introduction of machine learning, and later large language models, has reopened questions about how much of the remaining human burden can be absorbed by automation, while simultaneously exposing new methodological challenges around evaluation and generalization. The field now stands at a productive, if unsettled, juncture: symbolic and learned methods are beginning to reinforce one another, and the boundary of what machines can handle continues to shift, though how far and how fast remains an open question.

4 Machine Learning Approaches

The application of ML to theorem proving has undergone several distinct generational shifts, broadly tracking the wider trajectory of the ML field. Early approaches treated proof automation as

a classical ML problem, engineering input features by hand and applying relatively simple models to premise selection and search heuristics. The advent of deep learning removed the need for manual feature engineering, allowing models to learn representations of formal mathematical syntax directly from data. Most recently, the rise of large language models has shifted the dominant paradigm again, enabling systems that leverage general mathematical knowledge acquired through pre-training on large corpora rather than knowledge confined to a single proof library.

Each generation has improved on the last in raw capability, but has also introduced new methodological challenges. In particular, the evaluation of ML-based theorem provers is complicated by the structured nature of proof libraries: theorems are rarely independent, and naive train/test splits may produce inflated success rates. We survey approaches across all three generations, primarily exploring approaches for premise selection, tactic prediction, and auto-formalization.

4.1 Classical Machine Learning

The first modern ML approaches we consider arose from the *Automated Reasoning in Large Theories* (ARLT) movement. ARLT marked a shift in ML approaches in ATP; early automation attempts focused on learning heuristics for domain-specific theories, or augmenting proof search techniques [77]. These approaches generally target premise selection and tactic prediction.

4.1.1 MaLARea. The earliest attempt to enhance ATP premise selection via ML for large theories is MaLARea [80, 81]. The initial approach [80] was purely syntactic, learning associations between the symbols that appear in premises and the relevance of those premises to a goal. MaLARea depends on the SNoW system [17], a Bayes network, trained for classification.

A later revision [81] of the system extended it with semantic guidance in addition to its existing syntactic pipeline. This guidance entails searching for *finite models*, sets of mathematical objects and their semantics, that invalidate the conjecture to prove when combined with the currently-selected premise set. If invalidated then another premise must be necessary, otherwise the current premise selection is more likely to be sufficient.

MaLARea significantly improved upon other approaches for the MPTP challenge set [79], a set of 252 mathematical problems. ATPs E [74] and SPASS [85] solved 89 and 81 problems, respectively, and 104 overall. MaLARea with semantic guidance solves 165 problems, a substantial improvement.

4.1.2 CoqHammer. CoqHammer [25] is a hammer for the Coq proof assistant that must contend with a challenge absent in earlier hammer systems: Coq's type theory is *dependently typed*, meaning that types can depend on values in ways that have no obvious equivalent in the first-order logic accepted by external ATPs. CoqHammer addresses this by implementing a translation layer that encodes dependently-typed expressions into first-order logic, enabling Coq goals to be dispatched to off-the-shelf ATPs. Premise selection is accomplished by a k-NN classifier trained on previously proved theorems.

When an ATP finds a proof of the translated goal, CoqHammer reconstructs a valid Coq proof term from the ATP's output, closing the proof inside the Coq kernel. The system reconstructs 17–40% of Coq's standard library theorems depending on its configurations, which include different ATP backends and number of premises selected.

4.1.3 SEPIA. SEPIA [37] takes a different approach to tactic prediction, departing from feature-based classifiers entirely in favor of learning directly from the structure of completed proofs. Given a corpus of proof traces, SEPIA applies the MINT [84] algorithm to infer a finite state machine (FSM) whose states represent proof contexts and whose transitions represent tactic applications observed in training data. At inference time, SEPIA performs a breadth-first search over the FSM to suggest the next tactic.

540 The key limitation of this approach is that the FSM is essentially a compressed record of proofs already seen, with no mechanism for generalizing to structurally novel goals. The system solves approximately 15% of theorems on average across theory libraries, but achieves 0% on some, indicating that performance is highly dependent on the similarity between test goals and the training corpus. Its reliance on storing and replaying proof traces also raises scalability concerns as the knowledge base grows, since the FSM must be re-inferred as new proofs are added.

4.1.4 TacticToe. TacticToe [31] targets tactic prediction for HOL Light by framing proof search as a greedy tree search, where the scoring function for each candidate tactic is provided by a k-NN classifier over a database of recorded tactic applications. At each proof step, TacticToe retrieves the k most similar proof states from its database and uses the tactics applied in those states to rank candidates for the current goal. Premise selection is accomplished by a separate k-NN query over a database of previously proved lemmas.

The authors pre-select the set of tactics the model may choose from, limiting the search space at the cost of generality. TacticToe solves 66% of theorems in HOL Light's standard library on a random train/test split.

4.1.5 Tactician. Tactician [7] provides a k-NN-based tactic suggestion interface directly inside the Coq proof assistant, constructing a semantic tactic database from proof traces accumulated during normal use. Rather than requiring a separate offline training phase, Tactician operates as a persistent background process that observes every proof the user completes and immediately incorporates it into the database, so the system's knowledge grows continuously alongside the user's own work. At each proof step, the user can invoke the `suggest` tactic to receive a ranked list of candidate tactics or `search` to let Tactician conduct an automated proof search.

The similarity metric used for retrieval is based on a representation of the proof state that captures both the goal and the local context, allowing Tactician to distinguish between superficially similar goals that require different proof strategies. The authors do not report quantitative evaluation statistics, but the online learning design makes Tactician particularly well-suited to long-running formalization projects where the relevant lemma library grows over time.

4.2 Deep Learning

The limitations of classical ML approaches in theorem proving stem in part from their reliance on hand-engineered input features, which require significant domain expertise to design and may fail to capture the full structure of formal mathematical expressions. Deep learning approaches address this by learning representations of proof states and theorem statements directly from data, enabling more expressive models at the cost of greater data and compute requirements.

4.2.1 DeepMath. DeepMath [47] was the first project to apply neural networks to premise selection. The core idea is to treat premise selection as a binary classification problem: given a conjecture and a candidate premise, predict whether the premise is likely to be useful in a proof. Both the conjecture and the premise are fed to separate embedding networks, and the resulting representations are combined to produce a relevance score.

Training data consists of theorems in the Mizar Mathematical Library [2] *known to be provable by ATPs*, with premises labeled as relevant or irrelevant based on whether they appear in known proofs. The authors evaluate LSTMs, CNNs, and GRUs as the embedding networks and find that CNNs outperform CNN-LSTM hybrids and GRUs, which in turn outperform plain LSTMs, suggesting that local patterns are more informative for premise selection than larger patterns. The best CNN-based models achieve between 30 and 70% premise selection accuracy on Mizar theorems known to be provable by ATPs.

4.2.2 GamePad. GamePad [45] is a learning environment for Coq that exposes the proof assistant's internal state to ML models, enabling training on two complementary tasks: *tactic prediction*, choosing the next tactic to apply, and *position prediction*, estimating the number of tactics remaining to close a proof. Position prediction is particularly notable as it provides a value function for proof search, allowing the system to prioritize proof states that are estimated to be close to completion. The system encodes proof term ASTs as inputs to RNNs, providing structured representations of the proof state rather than raw text.

GamePad reports 95% proof accuracy on a set of synthesized algebraic rewriting problems, and between 40 and 60% accuracy on theorems drawn from the Feit-Thompson formalization [33] using various ML architectures. The authors note, however, that these figures are likely inflated by test set poisoning, and that the tactic prediction dataset gives special treatment to specific tactics such as `reflexivity`, limiting the generality of the results.

4.2.3 HOList. HOList [3] provides an instrumented version of the HOL Light proof assistant as a reinforcement learning environment for tactic prediction, with the explicit goal of enabling the kind of self-play training loop that had proven successful in game-playing AI. The environment exposes a step function that accepts a tactic and returns the resulting proof state, allowing an RL agent to explore proof search without human intervention. The dataset contains approximately 30, 000 theorems, with splits designed to prevent goals descended from the same parent from appearing in both training and test sets. An example tactic prediction model trained within HOList achieves roughly 35% accuracy.

4.2.4 CoqGym. CoqGym [89] contributes a large-scale dataset of 71, 000 human-written Coq proofs drawn from a broad range of libraries, together with ASTactic, a model that treats tactic prediction as structured prediction over ASTs rather than sequence generation over tokens. The motivation is that flat token sequences can generate syntactically invalid tactics, whereas decoding into an AST enforces syntactic validity by construction at every step of generation. The decoder generates a tactic AST top-down, selecting a production rule at each node based on the current proof state encoded by an attention-based encoder.

ASTactic solves 12.2% of theorems in the test set, and combining it with state-of-the-art ATPs such as SMT solvers raises this figure to approximately 30%, suggesting that neural and symbolic approaches capture complementary proof strategies.

4.2.5 Learning to Reason in Large Theories without Imitation. Bansal et al. [4] identify a fundamental tension in applying imitation learning to theorem proving: a model trained to imitate human proof traces will only ever encounter the proof states that humans chose to visit, yet at inference time it must navigate from arbitrary starting states, including ones far from any training example. This distribution mismatch causes errors to compound, as each mistaken tactic leads to a proof state further from the training distribution.

Their proposed alternative is retrieval-augmented reinforcement learning, in which the agent is initialized with a set of known proofs but learns its policy by interacting directly with the proof assistant rather than by imitating demonstrations. During training, unnecessary premises are pruned from the context at each step, reducing the complexity of the state space the agent must navigate. The authors acknowledge that generalization to domains requiring novel mathematical concepts remains an open challenge.

4.2.6 Proverbot9001. Proverbot9001 [72] is a neural tactic prediction system for Coq targeting the practical verification of real-world software, with a particular focus on the CompCert verified C compiler. The system encodes the proof state using a graph neural network over the AST of the current goal and local context, and uses this encoding to score candidate tactics drawn from a

fixed vocabulary. Premise selection is accomplished by a heuristic component rather than a learned model, reflecting the authors' observation that learned premise selection tends to perform worse than simple heuristics on software verification benchmarks.

The system achieves a success rate of 28% on CompCert and between 13 and 20% on the `concat`, `float`, and `zfc` libraries, representing one of the first evaluations of a neural theorem prover on industrial-scale verified software.

4.3 Large Language Models

Deep learning approaches to theorem proving are fundamentally limited by the size and scope of available formal proof corpora, which are small relative to the datasets used to train modern ML models. The emergence of the Transformer architecture [83] and the subsequent scaling of language models on internet-scale text created a new opportunity: models that arrive at formal proof tasks already equipped with broad mathematical knowledge acquired from textbooks, papers, and informal proofs.

GPT-f [66] was the first to demonstrate that this pre-trained knowledge could be transferred to tactic prediction in an ITP, and the field has since converged on LLMs as the dominant approach. This shift also enables new proof synthesis strategies that were previously out of reach, such as generating informal proof sketches to guide formal search and repairing existing proofs in response to changes in the underlying code.

4.3.1 TacTok. TacTok [28] is a proof synthesis system for Coq that conditions tactic prediction on both the current proof state and the semantic content of the proof script accumulated so far, using a language model over tactic token sequences. Prior work such as ASTactic [89] encoded only the current goal, discarding the history of tactics that led to it. TacTok's key insight is that the sequence of prior tactics carries information about the proof strategy being pursued, and that this context should inform the choice of the next tactic even when it does not change the syntactic form of the current goal.

TacTok achieves a success rate of 12.9% and outperforms ASTactic on large Coq projects of 10, 000 or more theorems, though it does not surpass CoqHammer [25]. The authors suggest TacTok is best understood as a complement to hammer-style and neural approaches rather than a replacement, and note that it handles higher-order reasoning in cases where other tools fail.

4.3.2 GPT-f. GPT-f [66] was the first work to apply a generative Transformer language model to tactic prediction in an ITP, targeting the Metamath proof assistant [59]. Rather than encoding a proof state and predicting a single tactic, GPT-f generates entire proof steps autoregressively, conditioning each token on the full preceding proof context. This generative framing allows the model to produce novel tactic expressions rather than selecting from a fixed vocabulary, significantly expanding the space of proofs the system can find.

The authors find that pre-training on mathematical texts such as papers from arXiv yields substantially better performance than pre-training on general internet corpora, providing early evidence that domain-specific pre-training is important for formal reasoning tasks. Performance also scales consistently with model size across the range of models evaluated. GPT-f reports a 56% success rate on a Metamath dataset using a random train/test split, though the lack of contamination controls means this figure likely overestimates real-world generalization.

4.3.3 PACT. PACT [39] observes that human-written tactic scripts, the standard source of LLM training data for theorem proving, represent only a small fraction of the formal knowledge encoded in a proof library. Every proof term checked by Lean's kernel implicitly contains a wealth of auxiliary information, such as the types of intermediate expressions, the names and statements of lemmas

687 instantiated along the way, and the structure of the proof term itself. This information does not
688 always appear in the tactic script a human writes. PACT proposes extracting this information directly
689 from the kernel to construct a richer set of co-training tasks, including next-lemma prediction, type
690 prediction, and theorem naming, alongside the standard tactic prediction objective.
691 PACT instruments Lean to allow LLMs to interface directly with the kernel, bypassing the textual
692 tactic layer entirely. The reference implementation accomplishes a reported 35%+ success rate on a
693 dataset with a train/test split based on hashes of theorem names.
694
695 *4.3.4 Draft, Sketch, and Prove.* Draft, Sketch, and Prove [49] is motivated by the observation that
696 mathematicians rarely construct formal proofs from scratch; they typically reason informally first,
697 identifying the key steps of an argument in natural language before translating those steps into a
698 formal system. The system breaks down this workflow in two stages. First, an LLM is prompted
699 to produce an informal natural-language proof *draft* of the target theorem. This draft is then
700 translated into a formal *sketch*: a sequence of high-level proof steps expressed in the ITP's language,
701 with the detailed justifications for each step left as gaps to be filled. Each gap is then discharged
702 independently by an off-the-shelf ATP.
703 This division of labor exploits the complementary strengths of LLMs and ATPs: the LLM supplies
704 the high-level mathematical intuition that ATPs lack, while the ATPs provide the rigorous low-level
705 verification that LLMs cannot be trusted to perform. The approach achieves approximately 40%
706 success on the miniF2F benchmark [94], representing a 20–30 percentage point improvement over
707 the ATP baselines alone.
708
709 *4.3.5 Baldur: Whole-Proof Generation and Repair with Large Language Models.* Baldur [29] targets
710 proof synthesis for Isabelle/HOL and departs from the tactic-prediction paradigm of prior LLM-
711 based approaches by generating entire proofs in a single model call rather than one step at a time.
712 It fine-tunes an LLM on the PISA [48] dataset of 183,000 human-written Isabelle proofs, training it
713 to predict a complete proof given only the theorem statement.
714 Because whole-proof generation frequently produces proofs that are nearly correct but fail on
715 minor syntactic or logical errors, Baldur pairs the generation model with a separate repair model
716 trained on tuples of incorrect proofs, error messages, and correct proofs. When the generated proof
717 fails, the error message returned by Isabelle is fed back to the repair model, which proposes a
718 corrected version. Baldur covers 4.2% additional theorems when the repair model is applied on top
719 of a single generation attempt, and improves on the SOTA by 8.7% on PISA.
720
721 *4.3.6 LeanDojo.* LeanDojo [90] contributes both a retrieval-augmented tactic prediction model for
722 Lean and an important methodological critique of evaluation practices in the field. The retrieval
723 component instruments Lean's elaboration process to extract fine-grained annotations linking
724 each tactic in a proof script to the specific premises it uses, producing a dataset in which every
725 tactic application is labeled with its dependencies. At inference time, a retrieval model uses these
726 annotations to build an index of premises, allowing the tactic prediction model to condition its
727 output on facts retrieved from the library rather than relying solely on knowledge encoded in its
728 parameters.
729 Crucially, the authors demonstrate that the standard random train/test split used by most prior
730 work inflates reported success rates, because theorems in the test set frequently share premises
731 with nearly identical training set theorems. They propose a more rigorous splitting strategy that
732 ensures every theorem in the test set uses at least one premise that never appears in any training set
733 proof, and show experimentally that this reduces reported success rates substantially, retroactively
734 casting doubt on many results in the prior literature.
735

4.3.7 *PALM.*
Lu et al. [57] conduct a systematic analysis of the failure modes of GPT-based proof generation on CoqGym [89], finding a consistent pattern: LLMs are competent at producing correct high-level proof structure but frequently fail at low-level details such as correctly naming identifiers, selecting appropriate lemma instances, and generating type-correct expressions. This suggests that one bottleneck is not mathematical reasoning, but rather the model's imprecise knowledge of the specific formal library it is working with.

Based on this analysis they propose PALM, a write-then-repair method that decouples high-level proof planning from low-level syntactic correctness. PALM first generates an initial proof attempt with an LLM, then applies a symbolic repair pass that identifies type errors and undefined identifiers and attempts to fix them by querying the library for matching definitions. PALM achieves between 30 and 40% success on CoqGym using GPT-3, GPT-4, and Llama-3, with the repair pass contributing meaningfully across all model sizes.

4.3.8 *Sisyphus.*
Sisyphus [34] targets an important proof maintenance task: when a verified program is updated but its specification remains unchanged, the existing proof may break because the program's internal structure has changed (even though its observable behavior has not). Manually repairing such proofs is labor-intensive, as the prover must locate the invariants that no longer hold and reconstruct arguments for their updated versions. Sisyphus automates this process by tasking an LLM with proposing candidate invariant repairs guided by the diff between old and new program versions, then checking each candidate against the proof assistant.

If a candidate fails, the proof assistant's error messages are fed back to the LLM as additional context, allowing it to refine its proposal iteratively. This framing positions LLMs as repair oracles within an otherwise symbolic verification pipeline, exploiting their ability to generate plausible mathematical expressions without requiring them to produce formally correct proofs independently.

4.3.9 *Finding Inductive Loop Invariants using Large Language Models.*
Loopy [52] applies LLMs to one of the most labor-intensive steps in formal software verification: supplying the loop invariants required by Hoare-logic-based verification tools. Given a program and its pre- and postconditions, the system prompts an LLM to propose candidate invariants, then checks each candidate against the verifier. If the candidate fails, the verifier's error message is fed back to the LLM as additional context and a new candidate is requested. This generate-check-repair loop continues until a valid invariant is found or a budget is exhausted.

The key observation motivating the approach is that LLMs, having been trained on large corpora of code and mathematical text, have implicit knowledge of common invariant patterns that would take a human significant effort to reconstruct from scratch. This makes them well-suited as a first-pass oracle for invariant synthesis even without formal guarantees of correctness. Loopy solves 398 out of 469 benchmarks when combining LLM generation with SOTA invariant strengthening procedures and a repair step, compared to 430 solved by the purely symbolic baseline Ultimate [41]. Thus, Loopy falls short overall, but solves benchmarks that Ultimate cannot, demonstrating that LLM-generated invariants cover qualitatively different cases than symbolic methods alone.

5 Rule-Based Approaches

Rule-based approaches to proof automation have been prevalent for decades, and remain both influential and actively used. In contrast to machine learning approaches which derive proof guidance from statistical patterns in training data, rule-based systems encode explicit reasoning strategies as deterministic procedures that can be applied without any training phase. Thus, they are predictable, interpretable, and often sound by construction, properties that are difficult or impossible to guarantee in learned systems.

$$\frac{\forall x : \tau.(\forall y : \tau.y \prec x \rightarrow P(y)) \rightarrow P(x)}{\forall x : \tau.P(x)}$$

Fig. 3. General formula for well-founded induction. $\prec$ is some well-founded order on τ.

The approaches surveyed in this section cover two broad categories. *Inductive* automation systems encode heuristics that target the structural challenges specific to inductive proofs, exploiting the fact that inductive goals follow predictable patterns that brute-force search handles poorly. Inductive goals are prevalent in nearly all program verification tasks, so these automation approaches are highly valuable for verifying mission-critical software. *Theory exploration* systems address the upstream question of which lemmas are worth stating and proving in the first place, automating the construction of the lemma libraries that both rule-based and learning-based provers depend on.

5.1 Inductive Automation

Induction is a fundamental proof strategy that allows one to reason about arbitrarily large finite structures via a constant-complexity case analysis. Consider the application of induction to proving some property P of all natural numbers n: in order to prove that P holds for all $n : \mathbb{N}$, it suffices to show that P holds for $n = 0$ and that, assuming P holds for $n = k$[1], P holds for $n = k + 1$. This is commonly referred to as *successor induction*, as natural numbers are defined as either being 0 or the successor of another natural number $S(n')$. Successor induction is merely a special case of the more general *structural induction* as seen in Figure 3, in which a proof case is necessary for each form a variable can take, and any case with *recursive sub-structures* (such as the successor case for $\mathbb{N}$) assumes that P holds for the sub-structures.

Inductive proofs present a consistent structural challenge that differentiates them from most non-inductive proofs: because inductive hypotheses only refer to sub-structures of the problem, non-trivial effort is usually required to utilize the reasoning power they provide. Rule-based approaches to inductive proofs exploit common patterns used to overcome this challenge, rather than searching blindly for some sequence of proof steps that solves the proof. The systems described here were among the first to automate non-trivial inductive proofs, and remain both influential and in-use.

5.1.1 *Nqthm and ACL2.*

Nqthm [11] and its successor ACL2 [53] are ATPs for a quantifier-free, first-order logic of total recursive functions. Its historical significance is twofold: it was the first proof system to give first-class treatment to inductive proofs, and it gave rise to the *waterfall model* of automated theorem proving that has since become standard.

Users state definitions, theorems, and induction rules in a LISP-like language. All recursive functions must be accompanied by an ordinal measure that *strictly decreases* on every recursive call to prove termination[2].

The proof search consists of a fixed cascade of seven techniques, repeating until the goal is discharged or no technique makes progress:

1. *Simplification* applies *rewrite rules* (previously-defined lemmas in which the conclusion is an equality), unfolds function definitions, and dispatches decision procedures for propositional calculus, equality, and linear arithmetic
2. *Destructor Elimination* seeks to eliminate function calls (and therefore reduce the size of the theorem to prove) by applying rewrite rules to replace variables with expressions that lead to greater simplification

[1]This assumption is referred to as the "inductive hypothesis."
[2]This pattern persists today in theorem provers such as Rocq [67]

834 (3) *Cross-Fertilization* is a heuristic for applying equality hypotheses
835 (4) *Generalizing* replaces terms with variables to generalize the theorem being proved
836 (5) *Elimination of irrelevance* discards unnecessary hypotheses to prune unproductive proof
837 search branches
838 (6) *Induction* seeks to apply user-provided induction strategies

839 This iterative process of applying specific heuristics or simplification steps until the goal is simple
840 enough to be proven directly is known as the waterfall model.

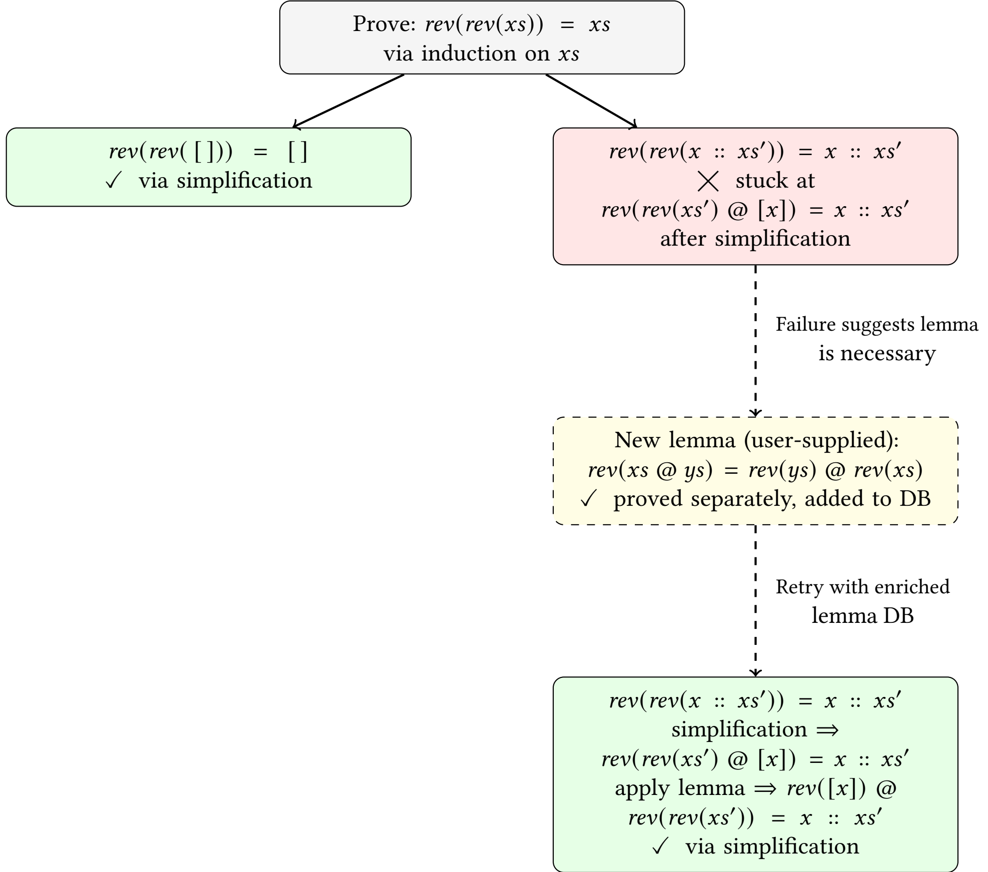

Fig. 4. Nqthm proof search for involutility of list reversal ($rev(rev(xs)) = xs$).

873 The central workflow consists of a human dispatching the theorem prover, which will either
874 succeed at proving the target theorem, or will fail. Upon failing, the human will suggest intermediate
875 lemmas to prove that will lead the theorem prover towards the proof of the main theorem.
876 Figure 4 shows an example of this workflow. The first attempt stalls in the inductive step because
877 no rule in the initial database can simplify $rev(rev(xs') @ [x])$. The user identifies the blocking
878 subgoal, dispatches the ATP to prove the distribution lemma for *rev* over append separately, and
879 adds it to the rewrite database. The second attempt then goes through.
880 This feedback loop is exceptionally powerful, as shown by the numerous applications of these
881 theorem provers to difficult problems:

883
884
885
886
887
888
889
890
891
892
893
894
895

$$
\begin{aligned}
\text{Main Theorem} \quad & \forall X\ Y\ Z, X + (Y + Z) = (X + Y) + Z \\
\text{Inductive Hypothesis} \quad & X + (Y + Z) = (X + Y) + Z \\
\text{Goal} \quad & S(X) + (Y + Z) = (S(X) + Y) + Z \\
\\
\text{Wave-front} \quad & S(X) \\
\text{Wave-hole} \quad & X \\
\text{Wave-rules} \quad & \begin{cases} S(U) + V \Rightarrow S(U + V) \\ S(U) \times V \Rightarrow U \times V + V \end{cases}
\end{aligned}
$$

Fig. 5. Components of rippling for associativity of addition

896
897
898
899
900

- Invertibility of the RSA encryption algorithm [12]
- Gödel's incompleteness theorem [75]
- Correctness of the Berkeley C string library [13]

901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931

5.1.2 Rippling. Rippling [15] is a domain-specific heuristic for manipulating inductive proof goals to automatically make progress in the proof. The technique works similarly to Nqthm and ACL2's approach of simplification via rewriting, but specifically targets bridging the gap between inductive hypotheses and conclusions, and is compatible with quantifiers. Rippling tactics have been built into multiple theorem provers [16, 87] and they fare exceptionally well on solving standard inductive proofs completely automatically.

At its core, rippling aims to apply rewriting rules such that inductive hypotheses can eventually be directly rewritten in or applied to the goal (commonly referred to as *fertilization*). Consider a proof of the associativity of addition via successor induction. In the goal of the inductive step (as in Figure 5), there exist a number of *wave-fronts* (the S functions) that prevent us from directly applying the inductive hypothesis. Using the first-presented *wave-rule*, a rewrite rule that moves the wave-front "outward" in the expression, we can convert our goal to $S(X+(Y+Z)) = S((X+Y)+Z)$, which we can trivially solve via the inductive hypothesis. This is an example of *rippling-out*, the simplest form of rippling.

Outward movement is the common case, but wave fronts may also move *inward* to instantiate universally quantified variables, *across* when a permutation of arguments is required, or under existential quantifiers when a witness must be exhibited. Conditional wave rules handle rewrites that are valid only subject to side conditions, and multiple wave rules may fire simultaneously on different subterms. Each of these cases encodes a unique reasoning strategy that enables rippling to make more intelligent progress than brute-force proof search.

5.2 Theory Exploration

Theory exploration is the automated discovery of lemmas regarding a given set of functions and datatypes. In automated theorem proving, this task is generally the only human burden beyond stating theorems to prove, making it a highly-desired target for automation. Whereas previously-described systems automate the *proof* of inductive goals given a sufficient lemma database, theory exploration addresses the question of how that database gets built in the first place. The central observation motivating the field is that mathematicians do not typically prove theorems in isolation: when introducing a new concept or function, they establish collections of basic properties relating it to existing definitions, and then use those properties as stepping stones toward deeper results.

Automating this workflow requires not just a prover but a system that can propose plausible statements worth proving before any proof is attempted.

The systems described here take different approaches to conjecture generation. Bottom-up approaches such as QuickSpec enumerate terms over a given signature and use random testing to identify likely equalities, producing a broad algebraic specification of the functions at hand. Failure-driven approaches generate conjectures in retrospect, extracting candidate lemmas from the structure of a stuck proof state. More recent work combines these perspectives with rich data structures to focus conjecture generation on lemmas that are guaranteed to make progress on a specific goal. Together these approaches address the theory exploration task, and several of them are directly integrated into ITP environments where their output can be consumed by proof automation techniques surveyed.

5.2.1 Productive Use of Failure in Inductive Proof. Ireland and Bundy present a technique [46] for automatically discovering the auxiliary lemmas that rippling requires but cannot find. Rippling stalls when no wave-rule can move the wave-front closer to a wave-hole. Upon stalling, rather than reporting failure and stopping, the system analyzes the stuck proof state to extract a candidate lemma that, if true, would allow rippling to resume. The key observation is that failure is not merely a dead end. The structure of the stuck goal contains precise information about what lemma is missing, because the wave front and wave hole are explicit annotations on the term. The theorem prover may generalizes blocking sub-terms into universally-quantified statements and submit them as new conjectures to be proved by a separate inductive proof attempt.

The technique handles two failure modes. In the first, rippling terminates with a residual goal that the fertilization step cannot close, indicating that the IH is too weak as stated. The system responds by generalizing the goal to strengthen the IH for a subsequent attempt. In the second, no wave rule applies at all, indicating that a rewrite is structurally needed that no existing wave rule can perform. The system then synthesizes a wave rule by constructing a lemma whose conclusion matches the required rewrite shape.

5.2.2 QuickSpec. QuickSpec [21] is a theory exploration system that automatically discovers equational lemmas regarding a given set of functions by random testing. Given a signature of functions and datatypes, it enumerates all valid terms up to a fixed depth and places them into equivalence classes. It then repeatedly evaluates terms on randomly generated inputs, separating any two terms that produce different outputs into distinct classes. When the classes stabilize, QuickSpec reads off one equation per class by designating a representative term and equating all others to it. The resulting conjectures have not been proved but are likely to be true, having survived falsification on hundreds of random inputs.

The key insight behind QuickSpec is that random testing is an efficient proxy for equality. Failing to find counterexamples after sufficient testing gives strong evidence that two terms are equal. QuickSpec itself only generates new conjectures; it is intended to be paired with an inductive theorem prover such as HipSpec [20] or an interactive proof assistant via a system such as Hipster [51] in order to prove and utilize the conjectures.

5.2.3 Hipster. Hipster [51] integrates theory exploration directly into Isabelle/HOL by automatically conjecturing and proving lemmas on demand within an active proof session. It translates the current Isabelle theory into Haskell and runs QuickSpec [21] to generate candidate lemmas by random testing. Once QuickSpec's generated conjectures are imported back into Isabelle, Hipster attempts to prove each one using structural induction and standard simplification. Each proved lemma is immediately available to discharge later conjectures in the same session.

5.2.4 CCLemma. CCLemma [55] is an automated prover for inductive equational goals that addresses a tension between the two dominant approaches to lemma discovery. Goal-directed approaches (those that only seek lemmas that prove the current gola) are fast but narrow-sighted, missing lemmas that relate subterms in ways the goal does not directly suggest. Theory exploration approaches (those that attempt to enumerate all provable lemmas for a set of types and functions) are more expressive but scale poorly, spending time proving lemmas that turn out to be irrelevant to the target property.

CCLemma resolves this by using e-graphs [62] and equality saturation to perform goal-directed theory exploration. E-graphs are data structures that compactly represent a large set of expressions and the equalities known to hold between them, grouping equal terms into equivalence classes. Equality saturation is the process of repeatedly applying rewrite rules to an e-graph until no new equalities can be derived, at which point the graph represents all consequences of the given rules simultaneously rather than committing to any single rewrite sequence.

Rather than enumerating candidate lemmas over the full function vocabulary, CCLemma applies all available rewrite rules simultaneously to the proof state, representing the resulting set of equal terms compactly in an e-graph. When no rewrite can merge the e-classes containing the left- and right-hand sides of the goal, CCLemma inspects the stuck e-graph to identify pairs of e-classes that random testing suggests should be equal but are not yet known to be. It generalizes the terms in these classes into a candidate lemma and attempts to prove it recursively before resuming the main proof.

The key insight is that a lemma is only worth pursuing if it can make progress on the current proof, and the e-graph makes this checkable cheaply: a lemma is useful if it would merge two currently distinct e-classes in the stuck proof state. This filters out the large majority of theory exploration candidates without requiring them to be proved first. CCLemma is evaluated on the CLAM [46] benchmark suite, as well as a new benchmark of program optimization problems requiring equivalence proofs between reference and optimized implementations. On the optimization benchmark it solves 50% more problems than CVC4 [5] (the next best tool), including properties which no prior tool could prove.

6 Related Work

Several surveys cover individual topics addressed by this one, though none evaluates the same progression of AI proof automation from rule-based approaches to modern LLM approaches. Li et al. [56] survey deep learning approaches to theorem proving comprehensively, covering premise selection, tactic prediction, and autoformalization through 2024; their scope is largely complementary to our rule-based and theory exploration sections. Zhang and Tan [93] survey conjecturing and theorem-finding systems, which covers theory exploration and automated conjecturing in more depth.

For the history and foundations of the field, Harrison, Urban, and Wiedijk [40] provide a comprehensive history of interactive theorem proving from Automath through modern systems, covering topics such as the Curry-Howard correspondence, the LCF architecture, and the development of the calculus of constructions. Ringer et al. [69] provide a thorough survey of the engineering of formally verified software covering proof assistants, proof engineering practices, and the gap between the state of the art and the demands of large-scale verification. Blanchette et al. [8] survey hammer automation tools, covering the translation, premise selection, and proof reconstruction pipeline in detail.

For theory exploration and lemma discovery specifically, Johansson [50] surveys techniques for automating the discovery of auxiliary lemmas for inductive proofs, covering both top-down failure-driven approaches and bottom-up theory exploration. Cropper and Dumančić [23] survey

inductive logic programming at its 30-year mark, covering the intersection of logic programming and machine learning that underlies several knowledge management approaches. The foundational work on automated reasoning by Wos et al. [88] provides early context for the ATP landscape that the ML approaches of §4 were built on top of.

7 Discussion

The most recent surveyed approaches have converged heavily on tactic prediction as the primary target of AI proof automation. Results in this area are promising, but lack a sturdy foundation of evaluation to stand on. LLMs fine-tuned on proof corpora are competitive with classical symbolic methods on standard benchmarks, and approaches such as "Draft, Sketch, and Prove" demonstrate that pre-trained mathematical knowledge can transfer to formal proof tasks. However, evaluation methodologies are inconsistent and appear to misinform readers of the generalizing capabilities of a model. LeanDojo demonstrated that naive train/test splits (such as those used in evaluations using CoqGym) inflate reported success rates by permitting similar theorems to span both the train and test sets. The authors propose mitigations for this effect, but do not present a fool-proof method of ensuring the sets are truly discrete. It is unclear whether existing benchmarks measure generalization to genuinely novel proofs, and careful consideration should be taken in future efforts of dataset construction.

Theory exploration is comparatively neglected despite being an arguably harder and more important problem than tactic prediction. Tactic prediction *assumes* a sufficient lemma database exists, but these lemmas often require deep insight to even pose. Modern language models are poorly suited to conjecture generation, as producing useful lemmas requires generating statements that are simultaneously true, provable, and relevant to the task at hand. Hallucinations [91] are common in current models, preventing most of these properties from being satisfied at the same time.

Full end-to-end proof generation remains out of reach for non-trivial domains. One productive near-term role for AI proof automation is to absorb the tedious and mechanical parts of proof work, rather than replacing human reasoning. However, even this conclusion is not airtight: methods for automating these tasks are prevalent [69]. Deep learning approaches are currently best suited to non-trivial, high-level *pattern matching* problems. LLMs are particularly well suited to knowledge transfer: between data types, between formalization efforts, between non-formal and formal descriptions, and so on.

8 Conclusion

The surveyed approaches apply several methodologies to a common problem among generations of humans: reducing tedious workloads so that mathematicians may focus on more difficult problems. Rule-based systems handle well-behaved, recognizable tasks reliably within their scope. Classical machine learning approaches demonstrated that the skills necessary to perform formal reasoning tasks are learnable from existing data. Deep learning approaches enhanced these skills with the capability to recognize even more complex patterns. LLMs facilitate knowledge transfer between fields of study and across the natural/formal language barrier.

Each generation has also introduced new failure modes and evaluation pitfalls. Naive benchmarking practices have overstated generalization, theory exploration remains underdeveloped relative to its importance, and full end-to-end automation of non-trivial proofs is not yet in reach.

The current state of AI proof automation reflects the broader arc of the field. Each wave of techniques has pushed the boundary of what machines can handle, and each has revealed how much remains for humans to accomplish. Whether that boundary continues to recede depends on progress in model architectures, evaluation methodology, and dataset construction.

References

[1] Kenneth Appel and Wolfgang Haken. 1977. The solution of the four-color-map problem. *Scientific American* 237, 4 (1977), 108–121.

[2] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. 2018. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *Journal of Automated Reasoning* 61, 1 (2018), 9–32.

[3] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. 2019. Holist: An environment for machine learning of higher order logic theorem proving. In *International conference on machine learning*. PMLR, 454–463.

[4] Kshitij Bansal, Christian Szegedy, Markus N Rabe, Sarah M Loos, and Viktor Toman. 2019. Learning to reason in large theories without imitation. *arXiv preprint arXiv:1905.10501* (2019).

[5] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[6] Thomas Bayes. 1958. An essay towards solving a problem in the doctrine of chances. *Biometrika* 45, 3-4 (1958), 296–315.

[7] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The Tactician: A seamless, interactive tactic learner and prover for coq. In *International Conference on Intelligent Computer Mathematics*. Springer, 271–277.

[8] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. 2016. Hammering towards QED. *Journal of Formalized Reasoning* 9, 1 (2016), 101–148.

[9] Philippe Block, Matt DeJong, and John Ochsendorf. 2006. As hangs the flexible line: Equilibrium of masonry arches. *Nexus Network Journal* 8, 2 (2006), 13–24.

[10] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[11] Robert S Boyer, Matt Kaufmann, and J Strother Moore. 1995. The Boyer-Moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications* 29, 2 (1995), 27–62.

[12] Robert S Boyer and J Strother Moore. 1984. Proof checking the RSA public key encryption algorithm. *The American Mathematical Monthly* 91, 3 (1984), 181–189.

[13] Robert S Boyer and Yuan Yu. 1996. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM (JACM)* 43, 1 (1996), 166–192.

[14] Bishop Brock, Matt Kaufmann, and J Strother Moore. 1996. ACL2 theorems about commercial microprocessors. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 275–293.

[15] Alan Bundy, Andrew Stevens, Frank Van Harmelen, Andrew Ireland, and Alan Smaill. 1993. Rippling: A heuristic for guiding inductive proofs. *Artificial intelligence* 62, 2 (1993), 185–253.

[16] Alan Bundy, Frank Van Harmelen, Christian Horn, and Alan Smaill. 1990. The oyster-clam system. In *International Conference on Automated Deduction*. Springer, 647–648.

[17] Andrew J Carlson, Chad M Cumby, Jeff L Rosen, and Dan Roth. 1999. SNoW user guide.

[18] Kyunghyun Cho, Bart Van Merriënboer, Çağlar Gulçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1724–1734.

[19] Alonzo Church. 1932. A set of postulates for the foundation of logic. *Annals of mathematics* 33, 2 (1932), 346–366.

[20] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2012. HipSpec: Automating Inductive Proofs of Program Properties.. In *ATx/WInG@ IJCAR*. 16–25.

[21] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs*. Springer, 6–21.

[22] Thierry Coquand and Gérard Huet. 1986. *The calculus of constructions*. Ph. D. Dissertation. INRIA.

[23] Andrew Cropper and Sebastijan Dumančić. 2022. Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research* 74 (2022), 765–850.

[24] Haskell Brooks Curry. 1930. Grundlagen der kombinatorischen Logik. *American journal of mathematics* 52, 4 (1930), 789–834.

[25] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61, 1 (2018), 423–453.

[26] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[27] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

[28] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

[29] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1229–1241.
[30] Evelyn Fix. 1985. *Discriminatory analysis: nonparametric discrimination, consistency properties*. Vol. 1. USAF school of Aviation Medicine.
[31] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2021. TacticToe: learning to prove with tactics. *Journal of Automated Reasoning* 65, 2 (2021), 257–286.
[32] Georges Gonthier. 2005. A computer-checked proof of the four colour theorem.
[33] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. 2013. A machine-checked proof of the odd order theorem. In *Proceedings of the 4th International Conference on Interactive Theorem Proving* (Rennes, France) *(ITP'13)*. Springer-Verlag, Berlin, Heidelberg, 163–179. doi:10.1007/978-3-642-39634-2_14
[34] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly automated proof repair for verified libraries. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 25–49.
[35] Michael JC Gordon and Tom F Melham. 1993. Introduction to HOL: A theorem proving environment for higher order logic. (1993).
[36] Michael J Gordon, Arthur J Milner, and Christopher P Wadsworth. 1979. *Edinburgh LCF: a mechanised logic of computation*. Springer.
[37] Thomas Gransden, Neil Walkinshaw, and Rajeev Raman. 2015. SEPIA: search for proofs using inferred automata. In *International Conference on Automated Deduction*. Springer, 246–255.
[38] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. 2017. A formal proof of the Kepler conjecture. In *Forum of mathematics, Pi*, Vol. 5. Cambridge University Press, e2.
[39] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. 2021. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203* (2021).
[40] John Harrison, Josef Urban, and Freek Wiedijk. 2014. History of interactive theorem proving. In *Handbook of the History of Logic*. Vol. 9. Elsevier, 135–214.
[41] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, et al. 2018. Ultimate Automizer and the Search for Perfect Interpolants: (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 447–451.
[42] Hugo Herbelin, Pierre-Marie Pédrot, coqbot, Gaëtan Gilbert, Maxime Dénès, letouzey, Emilio Jesús Gallego Arias, Matthieu Sozeau, Théo Zimmermann, Enrico Tassi, Jean-Christophe Filliatre, Pierre Roux, Guillaume Melquiond, Jason Gross, Arnaud Spiwack, barras, Pierre Boutillier, Vincent Laporte, Jim Fehrle, and MSoegtropIMC. 2026. *Rocq 9.2.0 (V9.2.0)*. doi:10.5281/zenodo.19256047
[43] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.
[44] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[45] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2018. Gamepad: A learning environment for theorem proving. *arXiv preprint arXiv:1806.00608* (2018).
[46] Andrew Ireland and Alan Bundy. 1996. Productive use of failure in inductive proof. *Journal of automated reasoning* 16, 1 (1996), 79–111.
[47] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. DeepMath–deep sequence models for premise selection. *Advances in neural information processing systems* 29 (2016).
[48] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. LISA: Language models of ISAbelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*. 378–392.
[49] Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2022. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283* (2022).
[50] Moa Johansson. 2019. Lemma discovery for induction: a survey. In *International Conference on Intelligent Computer Mathematics*. Springer, 125–139.
[51] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. 2014. Hipster: Integrating theory exploration in a proof assistant. In *International Conference on Intelligent Computer Mathematics*. Springer, 108–122.
[52] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948* (2023).

[53] Matt Kaufmann and J Strother Moore. 1996. ACL2: An industrial strength version of Nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*. IEEE, 23–34.

[54] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[55] Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. Cclemma: E-graph guided lemma discovery for inductive equational proofs. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 818–844.

[56] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939* (2024).

[57] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1509–1520.

[58] Warren S McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (1943), 115–133.

[59] Norman Megill and David A Wheeler. 2019. *Metamath: a computer language for mathematical proofs*. Lulu. com.

[60] Luigi Federico Menabrea and Ada Lovelace. 1843. *Sketch of the analytical engine invented by Charles Babbage*. Richard and John E. Taylor London.

[61] RP Nederpelt. 1970. Automath, a language for checking mathematics with a computer. *Tagung über formale Sprachen (Oberwolfach, Germany, August 30-September 5, 1970)* (1970), 27–29.

[62] Charles Gregory Nelson. 1980. *Techniques for program verification*. Stanford University.

[63] Allen Newell and Herbert Simon. 1956. The logic theory machine–A complex information processing system. *IRE Transactions on information theory* 2, 3 (1956), 61–79.

[64] Allen Newell and Herbert Simon. 1956. The logic theory machine–A complex information processing system. *IRE Transactions on information theory* 2, 3 (1956), 61–79.

[65] Lawrence C Paulson. 1994. *Isabelle: A generic theorem prover*. Springer.

[66] Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393* (2020).

[67] Rocq Prover. 2025. https://rocq-prover.org/doc/V9.2.0/refman/language/core/inductive.html#rocq:cmd.Fixpoint

[68] Alexandre Riazanov and Andrei Voronkov. 2002. The design and implementation of VAMPIRE. *AI communications* 15, 2-3 (2002), 91–110.

[69] Talia Ringer, Karl Palmskog, Ilya Sergey, Gligoric Milos, and Zachary Tatlock. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages* 5, 2-3 (2019), 102–281.

[70] John Alan Robinson. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)* 12, 1 (1965), 23–41.

[71] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning internal representations by error propagation*. Technical Report.

[72] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 1–10.

[73] Moses Schönfinkel. 1924. Über die Bausteine der mathematischen Logik. *Mathematische annalen* 92, 3 (1924), 305–316.

[74] Stephan Schulz. 2002. E–a brainiac theorem prover. *Ai Communications* 15, 2-3 (2002), 111–126.

[75] Natarajan Shankar. 1997. *Metamathematics, machines and Gödel's proof*. Number 38. Cambridge University Press.

[76] Morten Heine Sørensen and Pawel Urzyczyn. 2006. *Lectures on the Curry-Howard isomorphism*. Vol. 149. Elsevier.

[77] Christian Suttner and Wolfgang Ertel. 1990. Automatic acquisition of search guiding heuristics. In *10th International Conference on Automated Deduction*, Mark E. Stickel (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 470–484.

[78] Alan Mathison Turing et al. 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* 58, 345-363 (1936), 5.

[79] Josef Urban. 2003. MPTP 0.1 - System Description. *Electronic Notes in Theoretical Computer Science* 86, 1 (2003), 147–152. doi:10.1016/S1571-0661(04)80659-5 FTP'2003, 4th International Workshop on First-Order Theorem Proving (in connection with RDP'03, Federated Conference on Rewriting, Deduction and Programming).

[80] Josef Urban. 2007. MaLARea: a Metasystem for Automated Reasoning in Large Theories. *ISARLT* (2007).

[81] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. 2008. MaLARea SG1 - Machine Learner for Automated Reasoning with Semantic Guidance. In *Automated Reasoning*, Alessandro Armando, Peter Baumgartner, and Gilles Dowek (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 441–456.

[82] Maarten H Van Emden and Robert A Kowalski. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)* 23, 4 (1976), 733–742.

[83] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[84] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical software engineering* 21, 3 (2016), 811–853.

[85] Christoph Weidenbach. 1999. SPASS: Combining superposition, sorts and splitting. *Handbook of automated reasoning* 2 (1999), 1965–2013.

[86] Alfred North Whitehead and Bertrand Russell. 1927. *Principia mathematica*. Vol. 2. The University Press.

[87] Sean Wilson, Jacques Fleuriot, and Alan Smaill. 2010. Inductive proof automation for Coq. In *Second Coq Workshop*.

[88] Larry Wos, Ross Overbeck, Ewing Lusk, and Jim Boyle. 1983. Automated reasoning: introduction and applications. (1983).

[89] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*. PMLR, 6984–6994.

[90] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems* 36 (2023), 21573–21612.

[91] Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, Yu-Yang Liu, and Li Yuan. 2024. LLM Lies: Hallucinations are not Bugs, but Features as Adversarial Examples. arXiv:2310.01469 [cs.CL] https://arxiv.org/abs/2310.01469

[92] Richard Zach. 2003. Hilbert's program. (2003). https://plato.stanford.edu/entries/hilbert-program/

[93] Jian Zhang and Si-Cheng Tan. 2026. Automated Conjecturing and Theorem Finding: A Survey. *Journal of Computer Science and Technology* (2026), 1–21.

[94] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110* (2021).