Proof Synthesis via Symbolic and Neural Techniques

Xuyang Li

Abstract

Writing formal proofs in interactive theorem prover, for example, *Coq* and *Lean*, helps to provide rigorous correctness guarantees in diverse research fields, like software verification and mathematics. However, writing manual proofs is laborious, and there arises interest in proof synthesis. Existing proof synthesis techniques use both symbolic and neural methods to generate proofs automatically. This report tries to provide a comprehensive summary of these techniques.

1. Introduction

Research fields that need rigorous reasoning often employ formal proofs to establish correctness guarantees, particularly for critical theoretical results. Interactive theorem provers (ITP) allow researchers to write formal proofs with a special programming language, whose code is machine-checkable, to help with such purposes.

- For example, in software verification, where functional correctness properties of software are formalized into *theorems*, people have used *Coq* proof assistant [25] to implement and verify real-world OS kernels [10], system softwares [16], compilers [15, 13], network server [31], and even the entire software toolchain [1]; studies from software testing (like [30]) complementarily show the foundational benefits of verification by showing their absence of bugs, while their unverified industrial counterparts suffer.
- There is also a trend from the mathematics community to embrace ITP (like the *Lean* theorem prover [18]) to make their informal proofs formalized and certified [24]; shown in practice, ITP can help discover mistakes. Moreover, the famous four-color theorem is mechanized and verified in *Coq* proof assistant [9].

Of course, the power of ITP comes with a cost: the users must supply a proof script to guide the prover, and this necessitates additional (sometimes significant) expertise and proof engineering efforts than normal development. Like in software verification of *CompCert*, it had 100,000 Coq lines and demanded 6 person-years of effort in 2017 [12]. It is therefore very valuable to generate proofs to automate such verification processes. In addition, such automation can also help to auto-formalize the educational textbook and replenish scarce formalized documents [27].

Proof synthesis techniques discuss how to generate proofs in a systematic and automatic way. Taking the proof state below as an example, we want to figure out how to automatically apply the commutative law of additions for natural numbers:

```
1 n, m: nat
2 ============================
3 n + m = m + n
```

```
1 rewrite add_comm; reflexivity. (* what we want *)
```

In general, proof synthesis techniques fall into two categories: symbolic and learning-based (neural) ones.

- Symbolic techniques mainly try to adapt existing automated theorem provers (ATP) which support simpler logic than ITP, e.g. first-order logic, to find the proof script in ITP based on previously established theorems and current proof state [5, 2]. Although accurate and effective, these approaches suffer from high-order and inductive reasoning, limiting their usage to prove complex theorems. Some researchers try to find better symbolic automation in specific settings [23, 14].
- Neural techniques leverage the reasoning ability of language models learned from a large dataset to help predict the next proof step [7, 29]. These approaches do not guarantee the soundness of its generated proof step; nevertheless, they empirically have a decent generalizability and can better cope with high-order and inductive reasoning. Moreover, the recent rise of large language models (LLMs) provides new prospects to such problems, given their stronger reasoning capability and adaptivity; for example, in-context learning ability enables LLMs to interact with ITP and adjust its generated proof according to ITP's feedback easily.
- Some works also try to find the synergetic point of both to take advantage of both sides [17, 11, 22]; it is now clear that the combination is greater than the sum.

There are also closely relevant topics like program specification synthesis [26]; they enjoy similar favors and can be considered as a variant of proof synthesis in a specific subfield. The survey will not discuss these topics. Instead, this survey will focus on the main ideas of both symbolic and neural proof synthesis techniques and the attempts towards their synergetic combinations.

The paper is arranged as follows. After providing the introduction of the topic in Section 1, we will then move on to the concepts and tools as a background for proof synthesis in Section 2. Then in Section 3, the problem and its formalization are provided. And we discuss the high-level ideas of existing techniques in Section 4, their qualitative comparison in Section 5 and finally their numerical analysis in Section 6.

2. Background

In the section, we will introduce the basic terms of interactive theorem proving (in *Coq*), the mathematics of automatic theorem provers that are used in symbolic approaches, and the basic concepts of learning-based approaches.

2.1. Interactive Theorem Proving in Coq

Coq [25] is a popular interactive theorem prover (ITP) that helps users construct machine-checked proofs of mathematical theorems and verify software systems. We introduce key concepts through an example theorem shown in Figure 1.

Theorems: In Coq, theorems start with the keyword `Theorem` or `Lemma`, followed by a name and statement. The theorem `add_comm` in Figure 1 states that natural number addition is commutative.

Proof States: During proof construction, Coq displays the current proof state - a list of unproven goals. Each goal consists of a local context that contains hypotheses and an outstanding proof obligation. The local context appears above a double line, with the proof obligation in the following. See Appendix A for proof states of Figure 1.

Tactics: Tactics specify strategies for decomposing proof obligations into simpler subgoals. For example, the `induction n` tactic on Line 4 of Figure 1 performs induction on `n`, generating base and inductive cases as subgoals; the `apply plus_n_Sm` tactic applies another verified theorem (called `plus_n_Sm`) in global context to the current goal. Tactics can fail if applied incorrectly or with invalid arguments.

Proofs: A complete proof consists of a sequence of tactics that transform the initial theorem statement into solved subgoals. Proofs begin with `Proof` and end with `Qed`, which verifies no goals remain unsolved before adding the theorem to the global context.

```
1 Theorem add_comm :
2     forall n m : nat, n + m = m + n.
3 Proof.
4 intros n m.
5 induction n.
6 - auto.
7 - simpl. rewrite IHn. apply plus_n_Sm.
8 Qed.
```

Figure 1. A Coq theorem stating that natural number addition is commutative, and a proof of this statement.

Coq is obviously a more semantic-heavy language than natural language and other common-purpose programming language, as it contains high-order and inductive reasoning.

2.2. Automatic Theorem Proving

Symbolic proof synthesis techniques might utilize standard ATPs, such as Vampire [19], CVC5 [3], and Z3 [6]. They are mainly Satisfiability Modulo Theories (SMT) solvers, which combines boolean satisfiability (SAT) solving with reasoning over specialized theories. An SMT problem asks whether a first-order logic formula is satisfiable with respect to combinations of background theories, such as arithmetic, arrays, or bit-vectors. SMT solvers rely on heuristic optimizations based on DPLL(T) or CDCL(T) algorithms [8, 21] to efficiently determine the input's formula's satisfiability; typically, end users can make good use of them without knowing those tricks.

2.3. Machine Learning for Proof Synthesis

Classical machine learning approaches [28, 7, 20] try to train a prediction model that takes the current proof state and other relevant information as input and outputs the next tactic, in the form of abstract syntax trees (AST). They normally choose LSTM-based architectures as the model's backbone and do training on existing mechanized proofs. Researchers also discover the importance of of*premises selection*, and therefore developed a retrieval-augmented neural component to facilitate this task [29]. [11] makes a simple but efficient attempt to combine neural and symbolic techniques by learning when to call these symbolic tools.

The rise of LLMs brings new methodologies to learning-based proof synthesis. Instead of training a model from scratch, relevant approaches utilize offline or commercial LLMs to generate the proof scripts by prompting. LLMs can also help to formalize informal natural language automatically [27].

For more information on learning-based proof synthesis, this survey paper [4] is a good starting point.

3. Problem Statement

In this section, we provide formalizations of proof synthesis.

Definition 3.1. A proof state $\psi \in \Psi$ is a tuple $\langle \mathcal{H}, g, \Gamma, \mathcal{D} \rangle$, where $\mathcal{H}$ is a set of logical formulas that are current *hypotheses*, g is a logical formula representing the current proof goal, Γ is a typing environment for all free variables in $\mathcal{H}$ and g, and $\mathcal{D}$ is a set of type and term definitions that are recursively referred to by $\mathcal{H}$ and g.

If the current goal is True, we finish the current branch of proof; if there is no remaining subgoals, we reach Qed, i.e. we finish the proof. Note that the original theorem is also a (special) proof state where the hypothesis set $\mathcal{H}$ and typing context of free variable Γ are empty.

We manipulate proof states with tactics, and each of them serves a distinct purpose in proof construction (i.e., transforming the proof state). In *Coq*, the `intros` tactic moves hypotheses and variables from the goal into the proof context, while `apply` leverages existing theorems by matching their conclusions with current goals, transforming proof obligations accordingly. `simpl` performs basic computational simplification, and `destruct` allows case analysis of inductive types by breaking them into their constructors. The `rewrite` tactic handles equality substitutions, working alongside `reflexivity`" for proving straightforward equalities. For proving properties about inductive types, the `induction` tactic generates appropriate base and inductive cases, automatically providing induction hypotheses for recursive structures like natural numbers or lists, thus establishing properties that hold for all elements of an inductive type. The `exists` tactic resolves existential goals by allowing you to provide a concrete witness value, reducing the proof obligation to show that this specific value satisfies the required property. *Coq* also allows for simple automation with tactics such as `auto` or user-defined tactics.

Definition 3.2. A tactic t is defined as a state-transition function of type $T \subseteq \Psi \times \Sigma \to \Psi$. It takes in the current proof state and arguments (typed Σ) and outputs the next proof state. A proof script $\mathbb{P}$ contains a list of tactics.

From an end-to-end perspective, proof synthesis is trying to find a script $\mathbb{P}$ that can finish the proof.

Definition 3.3. A synthesis algorithm $\mathcal{S} \in \Psi \times \cdots \to \mathbb{P}$ is a prediction algorithm that takes the current proof state and other potentially useful information and outputs a proof script. If $\mathbb{P}$ finishes the proof, we say that the synthesis succeeds. Some learning-based methods perform token-by-token predictions, that is, $\mathcal{S}' \in \Psi \times \cdots \to T$.

One of the most important helpful information that the synthesizer takes is possibly relevant premises, which is a set of established theorems from the standard library or the verified code base of the user. Note that its size can be significantly large; to make the algorithm practical, a premise selection procedure is always performed in both symbolic and neural approaches to reduce the search space. For efficiency, current approaches do premise selection at the start rather than in the middle of the proof. The selection algorithm often leverages some dependence or similarity analysis for feature engineering, and use machine learning models like KNN, naive bayes, decision trees and neural network for real predictions.

Definition 3.4. A premise selection algorithm $\mathcal{R}$ takes the definition of a theorem $\psi = \langle \emptyset, g, \emptyset, \mathcal{D} \rangle$ and outputs a list of possibly relevant theorems.

For symbolic techniques that leverage ATPs, they always need to perform translations between the logic of ITP and that of ATP. Taking *CoqHammer* as an example, one needs to translate the current proof state and selected premises, whose formulas are of *Calculus of Inductive Constructions*(CIC), to First-Order Logic (FOL) formulas for the ATP; also, when the ATP finds a proof consisting of a sequence of FOL formulas, *CoqHammer* will reconstruct the proof back with ITP's tactics.

Definition 3.5. A ITP to ATP logic translation $\mathcal{T} \in \Psi \to$ *Option* F takes in the current proof state and output a FOL formula (typed F). The translation might fail.

Definition 3.6. The proof reconstruction procedure $\mathcal{T}' \in$ *List*$(F) \to$ *Option* $\mathbb{P}$ is a mapping from FOL formulas to ITP tactics that might fail.

For the LLM-based approach, a prompt is needed to enable its few-shot learning ability. Normally it describes the syntactic format of generated proof script, and a few examples. For example, in [17], the authors use a prompt similar to:

```
Help me to prove a theorem in Coq, based on provided proof state
including hypotheses and goals. Give a complete proof in only one
single code block without explanations, extra definitions, or the
initial 'Proof'. Give a complete proof in one response. Only give
the proof body, without repeating the definition and "Proof".
Avoid existing names when using intros. Write the commands separately,
without ";" and "[cmd|cmd]". Use bullets for structure, NEVER use
"{" and "}". When -, + and * are used out, proceed to double versions
like --, ++, etc. For example,
<Example 1>
<Example 2>
Definitions and lemmas related to the proof will be provided,
use them accordingly.
Solve This Proof State:
Hypotheses:
Goal:
forall n m : nat, n + m = m + n
Premises:
plus_n_Sm: forall n m : nat, S (n + m) = n + S m
plus_n_O: forall n : nat, n = n + 0
```

For evaluation, the metric *prove rate* (# of theorems proven) is used for comparisons. Normally, some techniques might have a similar or lower prove rate than others, but they are still useful if they can prove what others cannot: different approaches do not compete but complement each other.

4. Literature Review

In this section, we discuss five lines of research on proof synthesis, hoping to give a complete outlook on it.

4.1. Symbolic technique: *CoqHammer* [5]

The most classic symbolic tools are called *hammers*, and here we discuss *CoqHammer* [5]. It has three core technical components.

- First, the authors created a translation mechanism that converts Coq's dependent type theory (CIC is one of its possible implementations) into first-order logic that automated theorem provers can understand, using special predicates to encode typing information and proof terms.
- Second, they implemented premise selection using k-nearest neighbors and naive Bayes algorithms to identify relevant theorems from the available library, optimizing feature weights using TF-IDF scoring.
- Finally, they built a proof reconstruction system that converts the automated provers' output back into valid Coq proofs, using a combination of tactics including forward reasoning, congruence closure, and heuristic rewriting.

When evaluated on the Coq standard library, this pipeline successfully proved 40.8% of theorems automatically within 40 seconds on an 8-CPU system.

4.2. Neural techniques: ASTactic, Tactok and Passport [28, 7, 20]

ASTactic [28] is considered one of the first modern neural theorem provers, accompanied by a large-scale dataset CoqGym. It carefully designs the encoding, which allows adaptive tactic generations using tokens available at runtime, i.e., it is more interactive. ASTactic uses TreeLSTM as its encoder and a gated recurrent unit as its decoder; it takes a goal and a set of premises expressed as terms in *Coq* and outputs a tactic as an AST in a subset of *Coq* tactics.

Tactok [7] is a follow-up of ASTactic. The authors observe that the partial proof script also contains information that might help the subsequent proof synthesis, as tactics are not always independent.

Passport [20] switches the focus to the dataset and observe that the naming convention of global definitions, local variables, and type constructors is informative to guide proof generation. For example, local variable names `hd` and `tl` are normally the head and tail of a list. The authors carefully categorize the identifiers and encode them in training, achieving improvement.

4.3. Retrieval-Augmented nerual prover: ReProver [29]

This paper focuses on *Lean* theorem prover, providing a toolset for interactions and data extractions with *Lean*. It also provides a challenging dataset/benchmark.

The authorss emphasize a hard part of neural theorem proving (like with LLMs): *premise selection*. The authors trained a retrieval component, i.e. a premise selector that takes the current proof state as input, to argument neural theorem provers, using their toolset and data. With less resource costs, better or equivalent accuracy is achieved.

4.4. Simple combination: Thor [11]

This paper tries to incorporate language models with symbolic techniques (hammers) by simply training the model to learn which proof state can be solved by the latter. It trains the model to output a special token `<hammer>` for hammer-provable proof states.

The result of this paper show that this simple combination of language model and hammers can prove more theorems than their sums.

4.5. LLM-based technique with symbolic repair: PALM [17]

In this paper, the authors first investigate how well LLMs generate proof scripts. They discover that LLMs are good at sketching high-level proof structures but struggle with low-level details; and it's known that symbolic techniques like *CoqHammer* work better at lower level. Based on these insights, the authors develop PALM, which follows a generate-then-repair pipeline.

More detailedly, PALM works by first using an LLM (GPT-3.5, GPT-4, Llama-3-70B, and Llama-3-8B) to generate an initial proof script, then employing targeted repair mechanisms to fix common types of errors, such as wrong theorem applications, invalid references, and bullet misuse. If repairs fail, PALM uses a backtracking procedure that leverages *CoqHammer* to regenerate previous proof steps.

PALM is evaluated on **CoqGym**. The results showed that PALM significantly outperformed the existing approaches. The system also demonstrated strong generalizability across different LLMs.

4.6. LLM-based autoformalization [27]

The corpus of formal languages on the internet is quite small. For those learning-based techniques that need significant formal data, autoformalization can help by enriching the dataset via translation of informal languages. [27] tries to leverage LLM for autoformalization, and improve Thor [11] with successfully formalized data via expert iteration.

5. Qualitative Comparison

5.1. Comparison

Symbolic techniques such as *CoqHammer* [5] excel in low-level synthesis tasks, as they leverage accurate translation between different logical systems and utilize existing highly optimized ATPs to find the proofs. Such symbolic techniques are irreplaceable in the visible future. However, the significant complexity in proof language's semantics, like high-order types and inductive reasoning, which implies a large search space, poses insurmountable difficulties for scalable general-purpose symbolic proof synthesis.

The neural techniques evolve to make better use of the information in the dataset. For example, Tactok [7] extends ASTactic's [28] setting by incorporating proof script context, and Passport [20] enhances performance by leveraging naming conventions. Though neural techniques can prove unique theorems that hammers can not and are more predictably efficient, they can prove significantly lower number theorems. This might due to the probabilistic nature of neural approaches such that it might make very simple mistakes.

ReProver [29] addresses one of the key challenges in neural theorem proving: premise selection, which can be considered orthogonal to the design of model's feature engineering and architecture design. The improvement of premise selection can reduce computational resources but at the same time achieve comparable or better results. However, maintaining separate models for a single task demands greater engineering ability.

The exploration of simple hybrid approaches like Thor [11] highlights the importance of finding synergetic combinations of symbolic and neural methods. The investigation of a LLM-based approach like PALM [17] also agrees with this observation. However, the idea of Thor is way too simple; further research on relevant topics is needed.

The generate-then-repair pipeline of PALM is a useful methodology to cope with the uncertainty and fallibility of LLMs. It also incorporates the usage of *CoqHammer* for low-level proof synthesis. The mentioned repair can be considered manually designed symbolic approach that fixes the proof. And *CoqHammer* is an ultimate black box for those beyond. PALM tries to find a better synergy of hybrid symbolic and neural techniques.

Though LLM-based autoformalization [27] does not contribute directly to proof synthesis, it addresses the data scarcity problem of learning-based neural proof synthesis systems by potentially expanding its available training data.

5.2. Future work

- ATP as a gray-box: currently ATP is like a black-box oracle to generate the proof. However, when it fails, it is also possible to generate useful feedback to guide the proof synthesis system. Such feedback might contain the reason for falling (such as insufficient premises or conflicting subgoals), the usefulness of selected premises, etc. Those information can further guide the premise selection for both symbolic and neural methods, or further direct the LLM. It also suggests a better synergetic hybrid approach.
- Improved encoding: except for the naming convention, other characteristics of proof states, like the scope of existing hypothesis, might help to guide proof synthesis.
- Improved premise selection: though there have been progresses in premise selection, the ultimate solution seems undiscovered. Perhaps we still need to find better similarity or relevance analysis for amongst terms and theorems.
- Better error recovery: building upon PALM's repair mechanisms, future systems could incorporate more sophisticated error detection and recovery strategies for more detailed cases.
- Fine-tuned LLM for formal languages: current usage of LLMs only utilize its in-context learning ability to generate formal proof scripts. However, systematic fine-tuning and alignment for LLMs might help to improve its ability on proof synthesis tasks.
- Cross-system integration: research into creating proof synthesis systems that can work across different ITPs (beyond just *Coq* or *Lean*) could help standardize approaches and improve overall effectiveness.
- Domain-specific proof synthesis: when general-purpose algorithms hit the ceiling, domain-specific optimizations might shed light.
  - Like for those theorems about executable programs, sampling can provide more information and enable the so-called example-based synthesis [22, 14].
  - For those theorems about equality, special data-structures like e-graph can better summarize the relations between subterms and give hints about the proof progress [14].
  - Programs, which are the important verification objectives, can leverage systematic symbolic reasoning techniques (i.e., program analysis) to assist in verification.

In summary, more efforts are needed to utilize not only the hybrid of symbolic and neural approaches but also the characteristics of the theorem to be proved.

6. Numerical Comparison

In this section, we discuss the evaluation results of selected works. The evaluation was originally performed on different datasets (e.g. *Coq* standard library, CoqGym), but later work tried to reevaluate and compare on the same dataset. Some of the evaluations require configuration beyond the personal computer, which makes it impossible to fully reproduce their results; however, in the uploaded video, we will showcase some simple examples using these artifacts.

- The symbolic approach *CoqHammer* [5] tried to reprove the *Coq* standard library version 8.5 (9276 problems in total) on 48-core servers with 2.2GHz AMD Opteron CPUs and 320 GB RAM. Attempting synthesis with multiple provers (Vampire, E-prover, Z3), *CoqHammer* is able to achieve the prove rate 40.815% (3786 solved problems) in the validation set.

- Neural approaches gradually increase the prove rate comparing to previous methods.

  – ASTactic [28] is evaluated with 13137 testing theorems in CoqGym on machines with 16 GB RAM and two Intel Xeon Silver 4114 CPU cores. ASTactic about to prove 12.2% of the theorems alone. *CoqHammer* is also tried in the same dataset, and it achieve 24.8% prove rate. Though ASTactic underperforms *CoqHammer* in terms of prove rate, when combined with *CoqHammer* (whenever ASTactic generates a token, *CoqHammer* is invoked), 5.2% more theorems are proved than using *CoqHammer* alone.

  – TacTok [7] is also evaluated on CoqGym. It can prove 20.0% more theorems (264) that ASTactic cannot, 12.2% more theorems (115) that the combination ASTactic and *CoqHammer* cannot. No configuration detail is given.

  – Passport [20] is also evaluated on CoqGym. After considering identifiers, the prove rate improves significantly compared to ASTactic (29% more) and TacTok (14%-33% more). It does not compare with *CoqHammer*.

  Note that possibly the uncertain nature of machine learning makes the upgraded approach not a strictly superset to the former approach, i.e. the new approach is not able to prove a small portion that the previous approach can.

- The RePover retrieval-augmented neural prover RePover [29] is designed for *Lean*. With the augmented premise selector, ReProver can prove 3.6% more theorems (51.2% in total). The model training only takes five days on a single GPU.

- Thor [11] is designed for Isabelle. Taking hammers (similar to *CoqHammer* but for Isabelle) into account, Thor increases the success rate on the PISA dataset from 39% to 57%, while solving 8.2% more theorems that neither the original neural prover nor hammer can solve alone. Autoformalization [27] increases Thor's prove rate by 5.3% after two expert iterations. These two experiments are finished using a TPUVM with 8 cores, and the Isabelle process has access to up to 32 CPU cores. It takes around 4000 TPU hours to complete all the evaluation.

- The LLM-based approach PALM is also evaluated on CoqGym (though only on a subset to avoid bias). GPT3.5 is the main LLM. With the generate-then-repair pipeline and the fallback to *CoqHammer*, more than thousands of theorems are proven than using LLMs along (like for GPT3.5, prove rate increases from 3.7% to 40.4%). It also considerably outperforms previous neural approaches like Passport, and can prove 1270 theorems that none of the other approaches can prove. With more powerful LLM, the proof rate increases (for GPT-4o, 2.2% more theorems are proved than GPT-3.5); however, with small LLM like LLama-8b, PALM can still prove 3433 theorems. Ablation study is also performed to study the effectiveness of each repair mechanism, and the result shows all of them contribute to the improvement; obviously the fallback to *CoqHammer* contributes most.

We can see from these evaluations that symbolic and neural approaches complement each other. Since proof synthesis requires strict accuracy, symbolic approaches are at the heart of it to guarantee the correctness of low-level details. Neural techniques can provide helpful heuristics for high-order and inductive reasoning by providing a proof sketch, and they are also more efficient as the runtime of symbolic tools are not always predictable.

Also observed in the evaluation, LLMs seem more powerful than self-trained machine learning models due to their size. And LLMs can also work with symbolic approaches simply, though there still seems to be a space for a more systematic combination of them.

As a side component, the premise selector is useful for both symbolic and neural approaches, including LLM-based ones. And autoformalization can also help proof synthesis research, though in an indirect way, by resolving the data scarcity; moreover, it also contributes to education community by formalizing mathematic textbook.

Nevertheless, the progress in proof synthesis leveraging neural approaches is the intuitive demonstration of improvement in machine learning models' reasoning capabilities. We hope to witness and join such journey.

References

[1] A. W. Appel. Verified software toolchain. In G. Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs*, pages 135–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength smt solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

[4] L. Blaauwbroek, D. M. Cerna, T. Gauthier, J. Jakubův, C. Kaliszyk, M. Suda, and J. Urban. *Learning Guided Automated Reasoning: A Brief Survey*, pages 54–83. Springer Nature Switzerland, Cham, 2024.

[5] Ł. Czajka and C. Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Automat. Reason.*, 61(1):423–453, Feb. 2018.

[6] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] E. First, Y. Brun, and A. Guha. Tactok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.

[8] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Dpll(t): Fast decision procedures. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, pages 175–188, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] G. Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *Computer Mathematics*, pages 333–333, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[10] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, Nov. 2016. USENIX Association.

[11] A. Q. Jiang, W. Li, S. Tworkowski, K. Czechowski, T. Odrzygóźdź, P. Mił oś, Y. Wu, and M. Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 8360–8373. Curran Associates, Inc., 2022.

[12] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand. Closing the Gap – The Formally Verified Optimizing Compiler CompCert. In *SSS'17: Safety-critical Systems Symposium 2017*, Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium, pages 163–180, Bristol, United Kingdom, Feb. 2017. CreateSpace.

[13] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ml. *SIGPLAN Not.*, 49(1):179–191, Jan. 2014.

[14] C. Kurashige, R. Ji, A. Giridharan, M. Barbone, D. Noor, S. Itzhaky, R. Jhala, and N. Polikarpova. Cclemma: E-graph guided lemma discovery for inductive equational proofs. *Proc. ACM Program. Lang.*, 8(ICFP), Aug. 2024.

[15] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[16] X. Li, X. Li, W. Qiang, R. Gu, and J. Nieh. Spoq: Scaling Machine-Checkable systems verification in coq. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 851–869, Boston, MA, July 2023. USENIX Association.

[17] M. Lu, B. Delaware, and T. Zhang. Proof automation with large language models, 2024.

[18] L. d. Moura and S. Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.

[19] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, Aug. 2002.

[20] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer. Passport: Improving automated formal verification using identifiers. *ACM Trans. Program. Lang. Syst.*, 45(2), June 2023.

[21] R. Sebastiani. Lazy satisfiability modulo theories. *J. Satisf. Boolean Model. Comput.*, 3(3-4):141–224, Dec. 2007.

[22] A. Sivaraman, A. Sanchez-Stern, B. Chen, S. Lerner, and T. Millstein. Data-driven lemma synthesis for interactive proofs. *Proc. ACM Program. Lang.*, 6(OOPSLA2), Oct. 2022.

[23] Y. Sun, R. Ji, J. Fang, X. Jiang, M. Chen, and Y. Xiong. Proving functional program equivalence via directed lemma synthesis, 2024.

[24] Terence Tao. Machine assisted proof. https://terrytao.wordpress.com/wp-content/uploads/2024/03/machine-assisted-proof-notices.pdf, 2024.

[25] The Coq Development Team. The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman, 2024.

[26] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S.-C. Cheung, and C. Tian. Enchanting program specification synthesis bynbsp;large language models using static analysis andnbsp;program verification. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II*, page 302–328, Berlin, Heidelberg, 2024. Springer-Verlag.

[27] Y. Wu, A. Q. Jiang, W. Li, M. Rabe, C. Staats, M. Jamnik, and C. Szegedy. Autoformalization with large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 32353–32368. Curran Associates, Inc., 2022.

[28] K. Yang and J. Deng. Learning to prove theorems via interacting with proof assistants. *ArXiv*, abs/1905.09381, 2019.

[29] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

[30] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[31] H. Zhang, W. Honoré, N. Koh, Y. Li, Y. Li, L.-Y. Xia, L. Beringer, W. Mansky, B. Pierce, and S. Zdancewic. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

A. Appendix

```
================================
forall n m : nat, n + m = m + n
```

Figure 2. Proof state at the start.

```
n, m : nat
================================
n + m = m + n
```

Figure 3. Proof state after `intros n m`.

```
m : nat
================================
(1/2)
0 + m = m + 0
(2/2)
S n + m = m + S n
```

Figure 4. Proof state after `induction n`.

```
m : nat
================================
0 + m = m + 0
```

Figure 5. Proof state for first subgoal.

```
n, m : nat
IHn : n + m = m + n
================================
S n + m = m + S n
```

Figure 6. Proof state for second subgoal.

```
n, m : nat
IHn : n + m = m + n
================================
S (n + m) = m + S n
```

Figure 7. Proof state after `simpl`.

```
n, m : nat
IHn : n + m = m + n
================================
S (m + n) = m + S n
```

Figure 8. After rewrite `IHn`.