A Case Study on the Effectiveness of LLMs in Verification with Proof Assistants

Barış Bayazıt
baris.bayazit@mail.utoronto.ca
University of Toronto
Canada

Yao Li
liyao@pdx.edu
Portland State University
USA

Xujie Si
six@cs.toronto.edu
University of Toronto
Canada

Abstract

Large language models (LLMs) can potentially help with verification using proof assistants by automating proofs. However, it is unclear how effective LLMs are in this task. In this paper, we perform a case study based on two mature Rocq projects: the `hs-to-coq` tool and Verdi. We evaluate the effectiveness of LLMs in generating proofs by both quantitative and qualitative analysis. Our study finds that: (1) external dependencies and context in the same source file can significantly help proof generation; (2) LLMs perform great on small proofs but can also generate large proofs; (3) LLMs perform differently on different verification projects; and (4) LLMs can generate concise and smart proofs, apply classical techniques to new definitions, but can also make odd mistakes.

ACM Reference Format:
Barış Bayazıt, Yao Li, and Xujie Si. 2025. A Case Study on the Effectiveness of LLMs in Verification with Proof Assistants. In *Proceedings of International Workshop on Language Models and Programming Languages (LMPL'25).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software should be correct. But in reality, that's rarely true.

Proof assistants allow us to formally *verify* that a class of bugs is *absent* in a program via mechanized mathematical proofs. In the past two decades, various works have demonstrated that this approach is a feasible way to ensure software correctness and reliability. Some notable verified software includes the CompCert C compiler [45], the seL4 microkernel [42], the CertiKOS operating system [30], the FSCQ file system [12], *etc.* Many new tools and frameworks that support mechanized reasoning have also emerged, including program logics and frameworks for reasoning about concurrency [10, 40, 59, 79], nonterminating programs [75, 89], nondeterminism [11, 17, 60], *etc.*

However, despite all these effort, proving correctness theorems of a program in a proof assistant remains a daunting task. For example, Breitner et al. [9]'s work on verifying Haskell's `containers` library using `hs-to-coq` shows that their verification work "required 8.9 lines of proof per lines of code" [9, Section 3]. This is a significant overhead in addition to code development. Future changes to the code or the specification bring even greater chanllenges for proof maintenance and proof repair [28, 70, 71].

Large Language Models, or LLMs, on the other hand, have received great attention in their capability in performing a wide range of tasks. In particular, existing works have demonstrated LLMs' effectiveness in generating code [38] and mathematical proofs [2, 46, 83].

It is natural to ask: Can LLMs help with verification using proof assistants?

Indeed, researchers have recently started investigating this question and various new tools/frameworks for generating program correctness proofs with the help of LLMs have also emerged [25, 44, 51, 67]. However, due to the mysterious nature of LLMs [100, 102], many questions remain unanswered.

In this paper, we build on prior works and try to understand more about the effectiveness of LLMs in verification with proof assistants, by conducting a case study on two different verification projects that use Rocq Prover [81]: the `hs-to-coq` tool [9, 78] and Verdi [86, 87].

Our case study investigates the following research questions:

- **RQ1:** How do external dependencies and/or context in the same source file impact proof generation for a theorem?
- **RQ2:** How do LLMs perform on proofs of different sizes?
- **RQ3:** Is there a difference when running LLMs on different verification projects?
- **RQ4:** How is the quality of proofs generated by LLMs?

To answer these questions, we conduct a *quantitative* study for RQ1, RQ2, and RQ3, and a *qualitative* study for RQ4. Our case study shows:

- Including either external dependencies, or context in the same source file, or both can significantly improve the effectiveness of LLMs in generating proofs.

LMPL'25, October 2025, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- LLMs perform significantly better on proofs of smaller sizes. However, there is still a chance for LLMs to generate proofs consisting more than 20 tactics.
- LLMs do perform differently in the two projects we studied. For example, LLMs are less likely to generate proofs that are identical to original proofs in `hs-to-coq`. Context in the same source file also plays a more significant role in generating proofs for Verdi.
- LLMs can generate concise and smart proofs. They can also apply classical techniques such as performing case analysis on a newly-defined inductively defined proposition. On the other hand, LLMs can also generate odd, apparently failed proofs that repeat a tactic seemingly indefinitely.

In the rest of this paper, we first introduce some necessary background about Rocq Prover and program verification in Section 2. We then discuss our target codebase, namely `hs-to-coq` and Verdi, in Section 3. We describe our methodology in Section 4. We share and interpret our results (both quantitative ones and qualitative ones) in Section 5. We discuss related works in Section 6. Finally, we conclude with Section 7.

2 Background

In this section, we introduce necessary background knowledge to understand this paperq. Reader who are familiar with these concepts should feel free to skip the relevant parts.

Rocq Prover. Some commonly used proof assistants include Rocq Prover [81], Agda [1], Lean [16], F★ [80], and Isabelle [58], *etc.* In this paper, we focus on Rocq Prover. Rocq Prover is formerly known as the Coq proof assistant[1] and is one of the most commonly used proof assistants for program verification. Rocq Prover has an expressive specification language and supports full dependent types, which enables describing properties of a software system in rich details.

We show illustrate the process of theorem proving in Rocq Prover in Fig. 1. We first state a theorem in Rocq Prover using its specification language. For example, line 1 of Fig. 1a is equivalent to the mathematical proposition:

$$\forall n \in \mathbb{N}, n + 0 = n$$

where $\mathbb{N}$ is the set of all natural numbers.

We can then write a *proof script* that instructs the proof assistant to prove this theorem, as shown in lines 2–7. However, in Rocq Prover, we typically do not directly write the entire proof script. Instead, we enter an *interactive* proof mode. This step is typically marked by the **`Proof`** keyword (line 2).

When we enter the proof mode, Rocq prover will display the current context and the proof goal as shown in Fig. 1b.

[1]The name change starts in Rocq Prover version 9.0. However, we will address all versions of Rocq Prover, including versions before this name change, as Rocq Prover to avoid confusion.

```
1 Theorem add_0_r : forall n:nat, n + 0 = n.
2 Proof.
3   induction n as [ | n' IHn'].
4   - (* n = 0 *)    reflexivity.
5   - (* n = S n' *) simpl. rewrite -> IHn'.
6     reflexivity.
7 Qed.
```

(a) A Rocq theorem about natural numbers and its proof. The example comes from *Logical Foundations*, a classical textbook on Rocq Prover [66].

```
------------------------------
forall n : nat, n + 0 = n
```

(b) The context and the proof goal when we enter the proof mode (*i.e.,* right after invoking line 2).

```
------------------------------
0 + 0 = 0
```

(c) The context and the proof goal after we invoke the `induction` tactic (*i.e.,* after line 3). This is the first goal, *i.e.,* the base case.

```
n' : nat
IHn' : n' + 0 = n'
------------------------------
S n' + 0 = S n'
```

(d) The context and the proof goal after we proved the base case (*i.e.,* after line 4). This is the induction step.

```
n' : nat
IHn' : n' + 0 = n'
------------------------------
S n' = S n'
```

(e) The context and the proof goal after the `rewrite` tactic (*i.e.,* after line 5).

Figure 1. The process of proving a theorem in Rocq Prover.

The *context* is all the current hypotheses, which is empty at this point. The *proof goal* is what we need to show to finish the proof. We can manipulate the context and the proof goal using *tactics*, which are instructions to Rocq Prover about how to proceed with the proof. In this case, we decide to do an induction over n, indicated by the tactic `induction n` (line 3 in Fig. 1a). Our tactic also names some new variables via an *intro pattern* `[ | n' IHn' ]`—these names will show up later in the proof process.

After invoking the `induction` tactic, our proof goal will become two subgoals: one for the base case and one for the induction step. Rocq Prover will first ask us to prove the base case. We show the context and the goal of the base case in Fig. 1c. We can see that the context is still empty, but the goal has been changed to prove that $0 + 0 = 0$. Rocq Prover can tell that $0 + 0$ computes to 0, so proving the goal is equivalent to proving that $0 = 0$. Rocq Prover knows that = is *reflexive*, so we can discharge this goal via the `reflexivity` tactic.

Once we are done with the base case, Rocq Prover will ask us to prove the induction step. We show the context and the goal of the induction step in Fig. 1d. This time, the context contains a variable `n'` that has type `nat` and an *induction hypothesis* `IHn'` that states `n' + 0 = n'`. The names `n'` and `IHn'` come from the intro pattern in our `induction` tactic earlier. Our goal has also been changed to show that `S n' + 0 = S n'`, where `S n'` means the successor of `n'`.

In Rocq, `S n' + 0` is recursively defined as `S (n' + 0)`. We can reveal this via the `simpl` tactic. After that, we recognize that `n' + 0` equals to `n'` by the induction hypothesis `IHn'`, so we can `rewrite` using `IHn'`. The `rewrite` changes the goal to Fig. 1e, which can again be solved by the `reflexivity` tactic.

Finally, we can write a **`Qed`** in the end of the proof (line 7 in Fig. 1a). **`Qed`** is more than an end mark of a proof in Rocq Prover: it checks that the proof constructed by our proof script is indeed a correct proof of the theorem. Rocq Prover has a small trusted computing base for proof checking. This means that a proof checked by **`Qed`** is highly trustworthy.

Program Verification with Rocq Prover. We can verify properties of a program in the same way in Rocq Prover. If the program has already existed but written in another language, we need to first *embed* the syntax and/or semantics of that program in Rocq Prover [8]. Alternatively, we can also write a program directly in Rocq, prove properties about it, and then *extract* it to another language like OCaml or C. Our evaluation includes examples in both approaches (Section 3).

There have been various verification works based on Rocq Prover, including compilers [45, 96], operating systems [30], file systems [12], networked servers [43, 97], cryptographic algorithm implementations [22], *etc.* There are also various tools/frameworks supporting program verification with Rocq Prover, including Verified Software Toolchains [3], Iris [40, 79], certified abstraction layers [29, 31], mathematical components [52], and interaction trees [89, 95], *etc.* A more detailed account of Rocq Prover's ecosystem can be found in Appel [4].

3 Codebase for Evaluation

We choose two open-source Rocq Prover projects as the evaluation target: (1) the theory for Haskell's `base` library contained in the `hs-to-coq` project [9, 77, 78], and (2) the Verdi project for implementing and verifying distributed systems [86, 87].

We choose these two projects for the following reasons:

First, these two projects represent the two most typical ways to represent programs: functions and inductively defined relations. Programs in the `hs-to-coq` codebase are all purely functional programs, so they are simply defined as Rocq functions.[2] The Verdi project, on the other hand, reasons about traces and transition systems encoded by inductively defined propositions.

```haskell
take                    :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

(a) The **`take`** function defined in Haskell's `base` library.

```coq
Fixpoint take {a:Type} (n:Z) (xs:list a) :
  list a :=
  if (n <=? 0)%Z then nil
  else match xs with
       | nil => nil
       | cons y ys => cons y (take (n - 1) ys)
       end.
```

(b) The **`take`** function converted to Rocq by `hs-to-coq`. The **`Fixpoint`** keyword marks the definition of a recursive function. Haskell's types such as **`Int`** and lists `[]` are translated to Rocq types `Z` and `list`.

Figure 2. The **`take`** function defined in Haskell's `base` library and its translation in Rocq Prover produced by `hs-to-coq`.

Second, these two projects have proper sizes for an initial investigation of LLMs' effectiveness in verification. On the one hand, they are no toy projects. The theory of `base` in `hs-to-coq` contains 187 proofs and Verdi contains 579 proofs. On the other hand, these projects are not too large.

Finally, none of these projects involve advanced program logics (*e.g.,* separation logic [69], concurrent separation logic [10, 59]) or frameworks (*e.g.,* certified abstraction layers [29, 31], interaction trees [89, 95]). The absence of these advanced reasoning tools helps keep the experiments pristine.

We now talk about each project and the part we evaluate in more details.

3.1 The `hs-to-coq` Project

The `hs-to-coq` tool translates purely functional programs in Haskell to a shallow embedding in Rocq Prover. Its open-source repository contains translated code from Haskell's `base` library, `containers` library, parts of the GHC compiler, and many other examples of different sizes. We show an example of the original Haskell code and the translated Rocq code in Fig. 2. More details on how such translation works can be found in Spector-Zabusky et al. [78].

Our case study is based on the translated Rocq code, and theorems stated and proven by the `hs-to-coq` developers. Once a piece of Haskell code is translated using `hs-to-coq`,

[2] They should be called Gallina functions, to be more precise. Gallina is the specification language of Rocq Prover. However, we will not try to intentionally distinguish Gallina and Rocq in this paper.

```
Class EqLaws (t : Type) `{Eq_ t} :=
  { Eq_refl  : reflexive  _==_;
    Eq_sym   : symmetric  _==_;
    Eq_trans : transitive _==_;
    Eq_inv   : forall x y : t, x == y = ~~ (x /= y)
  }.

Class EqExact (t : Type) `{EqLaws t} :=
  { Eq_eq : forall x y : t, reflect (x = y) (x == y) }.
```

Figure 3. Laws for the **Eq** typeclass stated in Rocq Prover in `hs-to-coq`.

we can treat the translated code as regular Rocq code, so our evaluation does not rely the `hs-to-coq` tool or any Haskell code.

Our case study focuses on the theory of the `base` library. The `base` library contains a number of basic Haskell types, functions, typeclasses, and typeclass instances. The theory of `base` contains theorems for these basic types and functions, and thoerems for typeclass laws.

Typeclasses are a way to implement *overloading* (*i.e., ad-hoc polymorphisms*) in functional languages including both Haskell and Rocq Prover [34, 76, 85]. A few examples of typeclasses implemented in the `base` library includes: **Eq** for equality tests, **Ord** for total orders, `Semigroup` for concatenation, `Foldable` for "congregating" a data structure, and abstract interfaces like **Functor**, `Applicative` [53], and **Monad** [55, 84], *etc.*

Instances of these typeclasses are expected to satisfy certain laws. For example, an implementation of equality tests == in **Eq** should be reflexive, trasitive, and symmetric; the <= operator in **Ord** should be reflexive, transitive, and anti-symetric; a **Monad** should satisfy monad laws [55, 84]. The documentation of the `base` library describe these laws in details.

We show an example how `hs-to-coq`'s theory of `base` states laws for the **Eq** typeclass in Fig. 3. These laws are themselves defined as typeclasses in Rocq Prover. `Eq_refl`, `Eq_sym`, and `Eq_trans` states that == is reflexive, symmetric, and transitive, respectively. In `hs-to-coq`, `_==_` represents the equality test function and == is a notation that can be used as an infix operator. `Eq_inv` states that == and `/=` are inverse of each other. Finally, `EqExact` contains a special laws that states == always agrees with Rocq's builtin equality =. The `reflect` definition is an interesting definition that enables a classical technique in mechanized reasoning called *proof by reflection*. We will see an example that LLMs using this later in Section 5.

We choose the theory of `base` because it contains a fair amount of theorems, and the proofs in general are neither too simple nor too complicated. The longest proof script involves 43 tactics.

Other theories, such as theories for `containers`, `graph`, or the GHC compiler contain much more complicated proofs. For example, the theorem `insertBM_Desc` is about the property of the `insertBM` function of `container`'s `IntSet` data structure.[3] The handcrafted proof of this theorem is 42 lines of proof script, makes heavy use of proven lemmas, uses custom tactis, uses Ltac's `match` clause for pattern matching certain goals to solve them automatically, involves both backward reasoning and forward reasoning using `assert`. We leave investigation of these examples to future work.

The `hs-to-coq` project relies on Rocq Prover 8.10, which is an old version first released in April 2019. Unfortunately, most of its code no longer works under later versions of Rocq Prover because Rocq Prover does not support backward compatibility. For this reason, we conduct our study on Rocq Prover 8.10 as well. This should not impact the validity of this research, as the key workflow and features of Rocq Prover remain the same across these versions.

3.2 Verdi

Verdi is a framework for implementing and verifying distributed systems in Rocq Prover. Instead writing a program in a different language and embedding it in Rocq Prover, a programmer first implements their distributed systems in Rocq Prover and extract the code to OCaml. Unlike purely functional programs in Haskell's `base` library, distributed systems always contain a number of effects and interact with a network that can reorder or even drop messages. To model this, Verdi defines a special monad for implementing distributed systems and transition systems for network semantics. More details about how Verdi works can be found in Wilcox et al. [86], Woos et al. [87].

The Verdi framework has been used in various work to study the effectiveness of AI in verification. For example, it is included as part of the CoqGym benchmark [92] and has been studied by First and Brun [23], First et al. [24]. In particular, Lu et al. [51] tried applying GPT-3.5[4] to proofs in Verdi. They found that LLMs like GPT-3.5 are ineffective in finishing most of the proofs, as they collected 520 errors out of 579 theorems. They further analyzed all the errors and made the following observation [51, Section 3]:

> ...while LLMs often generate proof scripts with the right high-level structure, they often struggle with accurately addressing the sorts of low-level details that hammers excel at. For example, GPT-3.5 often knows when to use the induction tactic to decompose theorems into subgoals, but often fails to generate the right sequence of tactics to prove each subgoal...

This paper builds on these prior studies, but also investigates the effectiveness of dependencies in prompting.

[3] The data structure is a Patricia trie [56, 61].

[4] https://platform.openai.com/docs/models/gpt-3.5-turbo

The Verdi project we experiment with is the version included in CoqGym [92] and relies on Rocq Prover 8.11, to be consistent with prior studies.

4 Methodology

To evaluate how models performed under different contexts, we extracted the following information for each top-level construct using SerAPI [26] version 8.10.0+0.7.2 for `hs-to-coq`, and 8.11.0+0.11.1 for Verdi[5], along with Rocq version 8.10.2 and 8.11.0, respectively:

Name and signature: For each top-level defintion in a Rocq source file, we extracted its name (*i.e.,* the identifier bound by the construct) and its signature. For theorems, the signature consists of the entire declaration excluding the proof. For other definitions, the signature includes the entire definition.

In-file context: We define the in-file context as all lines in the file prior to the location where a theorem appears.

External dependencies, or dependencies: We define external dependencies (or dependencies for short) as any *signatures* that the original proof relies on, including definitions and theorems from other source files. If a dependency was already included in the in-file context, we exclude it from the list of dependencies to avoid repetition. Our extraction can include unnecessary dependencies. Specifically, a qualified identifiers returned by SerAPI can match identifiers defined in multiple files. In such cases, we included all matching possibilities in the dependency list.

Notations: For each dependency and file imported via Rocq's `Require Import` command, we collected all associated notation declarations. However, the definitions underlying these notations were not necessarily included as dependencies, since a notation may be used without its underlying definition being required by the proof.

Model and parameter selection. Our model selection includes both general-purpose and reasoning models with a mix of full-sized and lightweight variants:

1. GPT-4o-mini, version 2024-07-18: A smaller general-purpose model with a context length of 128,000 tokens [63].
2. GPT-4o, version 2024-11-20: A general-purpose model with a context length of 128,000 tokens [62].
3. OpenAI o4-mini, version 2025-04-16: A smaller reasoning model with a context length of 200,000 tokens [64]. The model does not support changing the default temperature through the API, but supports a reasoning effort parameter [54]. For our experiments, we have selected reasoning effort 'medium,' which is the default.
4. DeepSeek Prover V2: An open-source model based on DeepSeek V3. This model is fine tuned for theorem proving in Lean 4. The model has a context length of 163,840 tokens [90] and a parameter count of 671 billion [68]. We include this model in our case study to check if exposure to mechanized proofs in another proof assistant transfers to Rocq proofs.
5. DeepSeek R1: A large open-source reasoning model with a context length of 163,840 tokens [91], and a parameter count of 671 billion [18].

Each model was prompted with the same system message (for models supporting system prompt), and was allowed a maximum of 16,384 output tokens, configured using `max_tokens` or `max_completion_tokens` based on the model. The original context lengths for each model was preserved.

For all experiments, we set the temperature to 0.1 for models that support modifying this parameter over the API (e.g., GPT-4o). For models that do not support a custom temperature setting (e.g., o4-mini), the default value of 1.0 was used.

Prompt. We used a minimal system prompt that described (1) the information provided to the model, (2) the proof task it has to perform, and (3) the expected response format, asking the model to respond only with the proof body. The prompt also specified the current version of the Rocq available and included whether the version used `omega` in place of `lia`. We included this detail as the codebases being evaluated were relatively old, whereas the models are likely aware that `omega` is deprecated, given their more recent knowledge cutoff dates.

Variation of dependencies. We varied the prompt provided to the LLMs across four conditions: (1) full context (which we will shorten as *the informed mode* from now on), (2) without dependencies and notations, (3) without in-file context, and (4) with both removed.

When omitting the in-file context, we still include the import statements present in the file to show the model the available modules. We also extend dependencies to include the in-file dependent signatures.

Checking successful proofs. We defined a proof as successfully generated by the LLM if and only if SerAPI's `sertop` program accepted the proof when provided with (1) all lines in the file preceding the theorem (*i.e., the in-file context*), (2) the theorem's signature, (3) the LLM-generated proof body. This validation was performed using the version of SerAPI that matches the Rocq Prover version used in the corresponding codebase.

5 Evaluation Results

We now share our evaluation results and use them to answer the four research questions we proposed in Section 1.

RQ1: How do external dependencies and/or context in the same source file impact proof generation for a

[5] Our experiments were conducted on Verdi corresponding to commit `fdb4ede19d2150c254f0ebcfbed4fb9547a734b0`.

theorem? Among the four ablations we introduced in Section 4, most models achieved the highest success rate in the informed mode, as shown in Tables 1a and 1b. For both `hs-to-coq` and Verdi, success rates dropped for most models when either in-file context or dependencies were excluded, with the worst results occurring when both were excluded.

One potential consequence of including all dependencies and in-file context is the increase of input tokens. To understand this implication, we also estimated the number of tokens required in both projects. We show the statistics in Table 2.

RQ2: How do LLMs perform on proofs of different sizes? Figures 4a and 4b show the proof generation success rates in each tactic count intervals in light colors. These figures show that, with the exception of GPT-4o-mini, all LLMs have high success rates in generating proofs of small sizes. These success rates drop as the proof size increases. However, even when the proof becomes quite large, LLMs can still succeed in some cases in both projects.

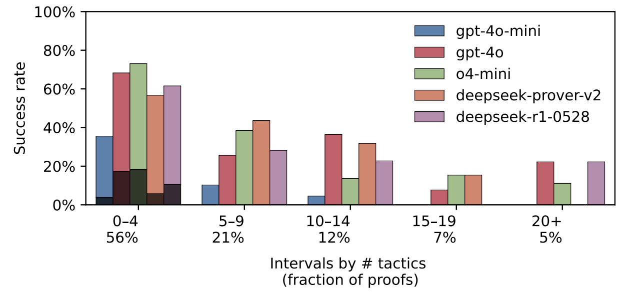
However, one question we need to address to make sure our results are valid is to check whether LLMs were generating these proofs or they have simply "memorized" all these proofs, as both projects are open-source projects available online. For this reason, we further checked if the generated proofs are identitical to original proofs. We show all the generated identical proofs, or "plagiarized" proofs, in Figs. 4a and 4b using dark colors.

The results show that LLMs indeed generate identitical proofs in both projects. In `hs-to-coq`, these are all small proofs, which have a high likelihood to be identical "by coincidence". On the other hand, some of the larger generated proofs in Verdi are identical to original proofs, suggesting that the proof might have been in these models' knowledge set.

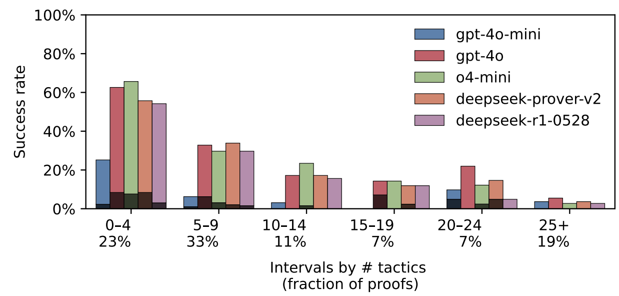
RQ3: Is there a difference when running LLMs on different verification projects? First, the impact of adding dependencies or in-file context also varies between these two projects. As seen in Table 3a, the benefits of in-file context diminished in `hs-to-coq` for proofs involving a larger number of tactics, and, in some cases, even reduced success rates for certain models. Conversely, simpler proofs with fewer tactics appeared to benefit from the in-file context.

In contrast, for Verdi, adding in-file context had a remarkably strong effect. As shown in Table 3b, external dependencies alone were mostly insufficient for handling longer proofs (*e.g.,* 20+ tactics) with a higher number of tactics in the original proof. The models were only able to perform better in the informed mode where the in-file context was provided.

Another difference between these two projects is that LLMs did not generate any proofs identical to original proofs in proofs with a larger tactic count in `hs-to-coq`.



(a) `hs-to-coq`



(b) Verdi

Figure 4. Success rates (light) vs. identically generated proofs (dark) by tactic count intervals for `hs-to-coq` and Verdi.

It is unclear why `hs-to-coq` and Verdi exhibit these difference. However, this finding suggests that studying one project may not be sufficient for improving LLMs' effectiveness in other projects.

RQ4: How is the quality of proofs generated by LLMs? In this section, we highlight some of the interesting proofs—including both successful ones and failed ones—generated by LLMs in our case study.

We compare the number of tactics in original proofs and in proofs generated by LLMs. In both projects, we find that LLMs can generate shorter proofs than original ones.

Let's start with an example in `hs-to-coq`. We show an original proof demonstrating that `unit` is a monoid that satisfies all the `Monoid` typeclass laws in Fig. 5a. The theorem statement itself is not important. The original proof works by first `split`ting the theorem into four subgoals, each representing one monoid property. The proof then unfolds a number of definitions—a style that is consistent with many other proofs in the same file. Then for each subgoal, the proof proceed by either a case analysis or an induction.

We show a proof generated by OpenAI o4-mini and DeepSeek-R1-0528 with *no* external dependency or same-file context in Fig. 5b. The proof is much simpler: it first uses the `constructor` tactic, which does the same thing as `split` in the original here. Then, LLMs "realize" that all subgoals can be solved using the same sequence of tactics: `intros []`

Table 1. Success counts and rates across different settings for `hs-to-coq` and Verdi.

(a) `hs-to-coq` (187 theorems)

Model	Informed	No in-file context	No dependencies	Neither
GPT-4o-mini	42 (22.5%)	35 (18.7%)	44 (23.5%)	31 (16.6%)
GPT-4o	92 (49.2%)	73 (39.0%)	83 (44.4%)	48 (25.7%)
o4-mini	97 (51.9%)	81 (43.3%)	81 (43.3%)	52 (27.8%)
DeepSeek Prover V2	85 (45.5%)	76 (40.6%)	71 (38.0%)	58 (31.0%)
DeepSeek R1	82 (43.9%)	74 (39.6%)	84 (44.9%)	48 (25.7%)

(b) Verdi (579 theorems)

Model	Informed	No in-file context	No dependencies	Neither
GPT-4o-mini	55 (9.5%)	27 (4.7%)	57 (9.8%)	17 (2.9%)
GPT-4o	177 (30.6%)	117 (20.2%)	170 (29.4%)	38 (6.6%)
o4-mini	172 (29.7%)	124 (21.4%)	177 (30.6%)	45 (7.8%)
DeepSeek Prover V2	164 (28.3%)	108 (18.7%)	159 (27.5%)	42 (7.3%)
DeepSeek R1	148 (25.6%)	123 (21.2%)	140 (24.2%)	40 (6.9%)

Table 2. Estimated prompt token counts for each setting, excluding the system prompt (rounded to nearest integer). The token counts were estimated using OpenAI's TikToken library [65].

Project	**Condition**	**Mean**	**Median**	**Max**
`hs-to-coq`	Informed	3379	3162	10223
	No dependencies	1766	1292	6833
	No in-file context	1916	1862	5720
	Neither	152	147	228
`Verdi`	Informed	6944	5488	25357
	No dependencies	5653	4393	19289
	No in-file context	2559	1618	20674
	Neither	174	167	445

to introduce a variable into the context *and* perform a case analysis on that variable at the same time, then `auto` for automatically discharging each goal.

An even smarter proof generated by LLMs can be found in Verdi. We show the theorem statement in Fig. 6. The theorem describes a relation between two variables `failed` and `net` when they are both in a multi-step transition relation defined by `step_ordered_dynamic_failure_star`—the exact definition of this step relation is not important. The original proof in Verdi is 24 lines of proof script, involves an induction, and Ltac's `match` statement.

However, OpenAI o4-mini is able to find a proof consists of only 4 basic tactics on the informed mode, as shown in Fig. 6. This is because the contrapositive of this proposition has already been proven as a theorem right before this theorem (called `ordered_dynamic_failed_then_initialized`). OpenAI o4-mini "recognizes" this connection between the two theorems and proves this theorem by simply `apply`ing its contrapositive.

```
Instance instance_MonoidLaws_unit :
  MonoidLaws unit.
Proof.
  split;
    unfold op_zlzlzgzg__, Semigroup__unit,
         op_zlzlzgzg____,
         Base.Semigroup__unit_op_zlzlzgzg__;
    unfold mappend, mempty, mconcat,
         Monoid__unit, mappend__,  mconcat__,
         Base.Monoid__unit_mappend,
         Base.Monoid__unit_mempty,
         Base.Monoid__unit_mconcat.
  - intro x. destruct x. auto.
  - intro x. destruct x. auto.
  - intros. auto.
  - intros x. induction x; simpl. auto. auto.
Qed.
```

(a) The original proof for showing that `unit` is a monoid that satisfies all the `Monoid` typeclass laws.

```
Proof.
  constructor; intros []; auto.
Qed.
```

(b) A proof for the same theorem generated by OpenAI o4-mini and DeepSeek-R1-0528. The two models generate the same proof for this theorem.

Figure 5. A comparison between the original proof and a proof generated by LLMs for theorem `instance_MonoidLaws_unit` in `hs-to-coq`.

Table 3. Percent gain in success rates from no in-file context (dependencies only) to informed per model and interval (with interval share in %).

(a) `hs-to-coq`

Model	0-4 (56%)	5-9 (21%)	10-14 (12%)	15-19 (7%)	20+ (5%)
GPT-4o	16.4	0.0	9.1	-7.7	11.1
GPT-4o-mini	8.7	-2.5	0.0	-7.7	0.0
o4-mini	14.4	10.3	-13.7	7.7	-11.1
DeepSeek Prover V2	0.0	18.0	9.1	0.0	0.0
DeepSeek R1	4.8	10.3	0.0	-15.4	11.1

(b) Verdi

Model	0-4 (23%)	5-9 (33%)	10-14 (11%)	15-19 (7%)	20-24 (7%)	25+ (19%)
GPT-4o-mini	7.6	4.1	3.1	0.0	9.8	3.7
GPT-4o	9.9	11.4	7.8	11.9	22.0	5.5
o4-mini	10.6	6.8	12.5	11.9	12.2	2.8
DeepSeek Prover V2	3.8	15.1	12.5	9.5	14.6	3.7
DeepSeek R1	4.6	4.7	3.1	7.1	4.9	2.8

```
Lemma ordered_dynamic_state_not_initialized_not_failed :
forall net failed tr,
  step_ordered_dynamic_failure_star
    step_ordered_dynamic_failure_init
    (failed, net) tr ->
  forall n, ~ In n (odnwNodes net) ->
  ~ In n failed.
(* The following proof is generated by OpenAI o4-mini. *)
Proof.
  intros net failed tr Hstar n Hnot Hin.
  apply Hnot.
  eapply ordered_dynamic_failed_then_initialized; eauto.
Qed.
```

Figure 6. A Rocq theorem found in Verdi (in the file `core/DynamicNetLemmas.v`) and a proof generated by OpenAI o4-mini. We omit the original proof found in Verdi because the proof script is 29 tactics long.

We should point out that the two theorems shown in Figs. 5 and 6 can also be solved using classical tools like CoqHammer [14, 15]. CoqHammer can solve the `hs-to-coq` theorem (Fig. 5) with its own tactic called `sfirstorder`. For the Verdi theorem (Fig. 6), it performs a proof search using an external automated theorem prover and also finds that the theorem can be proven with the help of its contrapositive, similar to the proof generated by LLMs. Nevertheless, it is impressive that LLMs are able to find these simple proofs given only one shot without a feedback loop.

The next theorem that LLMs come up with a simpler proof is the most surprising to us, and the theorem cannot be solved by CoqHammer. We show the theorem and its original proof in Fig. 7a. The theorem states that the pair `a * b` satisifies the `EqExact` law (Fig. 3) if both `a` and `b` satisfies this law. We show the original proof script in Fig. 7a to demonstrate the complexity of the original proof and to compare it with a proof generated by LLMs, but the reader should not try to read the proof script without Rocq Prover's interactive environment. The key structure of the proof is to perform two case analysis indicated by the two uses of the `destruct` tactic: (1) if two variables of type `a` are equal by ==, and (2) if two variables of type `b` are equal by ==.

We show a proof generated by DeepSeek-R1-0528 with the same-file context but *without* external dependencies in Fig. 7b. The proof is more concise. A main reason is that, instead of doing a case analysis on == like in the original proof, this proof peforms a case analysis on `Eq_eq`, whose type is an inductively defined proposition `reflect` (Fig. 3) that relates == and =. Intuitively, peforming a case analysis on `reflect` gives us more information than just ==. For example, the original proof uses `move /Eq_eq` to convert between == and = in various places, but such a step is unnecessary in the generated proof.

Performing a case analysis on an inductively defined proposition like `reflect` to "extract more information" is a classical technique in mechanized verification discussed in Rocq Prover textbooks [66, Chapter "Inductively Defined Propositions"]. However, `Eq_eq` is a function defined in `hs-to-coq`—to be more precise, in the same file as `EqExact_pair`—and a case analysis on `Eq_eq` was used only once in the same file before this theorem, but DeepSeek-R1-0528 is still able to generate a proof like this in one shot without a feedback loop.

```
Instance EqExact_pair {a b}
  `{EqExact a} `{EqExact b} : EqExact (a * b).
Proof.
  split; rewrite /op_zeze__ /op_zsze__
    /Eq_pair___ /op_zeze____ /op_zsze____.
  case =>[??] [??] //=. destruct (_ == _) eqn:?.
  - rewrite andb_true_l. move /Eq_eq in Heqb0.
    destruct (_ == _) eqn:?.
    + constructor. move /Eq_eq in Heqb1.
      subst. reflexivity.
    + constructor. move /Eq_eq in Heqb1.
      intro. apply Heqb1.
      inversion H5; reflexivity.
  - rewrite andb_false_l. constructor.
    move /Eq_eq in Heqb0. intro.
    inversion H5. apply Heqb0. assumption.
Qed.
```

(a) The original proof for showing that the pair `a * b` satisfies the `EqExact` law (Fig. 3) if both `a` and `b` satisfies this law.

```
Proof.
  split; unfold op_zeze__, op_zsze__,
    Eq_pair___, op_zeze____, op_zsze____
    => - [x1 y1] [x2 y2] /=.
  - case: (Eq_eq x1 x2) => [-> | NEQx];
    case: (Eq_eq y1 y2) => [-> | NEQy];
    constructor; congruence.
Qed.
```

(b) A proof for the same theorem generated by DeepSeek-R1-0528.

Figure 7. A comparison between the original proof and a proof generated by LLMs for theorem `instance_MonoidLaws_unit` in `hs-to-coq`.

On the negative side, we find that LLMs can stutter in generating some proofs. We show a theorem in `hs-to-coq` and its original proof in Fig. 8a. The proof is simple and straightforward. However, GPT 4o-mini generates a failed proof in the informed mode shown in Fig. 8b. The proof fails at the first `unfold`. However, a more odd phenomena is that, after the first `rewrite /op_zeze__`, the proof just keeps repeating the same tactic, seemingly indefinitely.

This phenomena is not common, but it has appeared in both `hs-to-coq` and Verdi, in different theorems, various LLMs, when invoking various tactics. However, we cannot deduce or conjecture what are the reasons that this happens.

Limitations. In terms of design, our experimental setup focused exclusively on single-pass proof generation, without incorporating a feedback loop. Although we recorded errors produced by `sertop` for each LLM and setting for future work, we did not analyze the model's ability to respond to

```
Lemma simpl_list_cons_eq a `{Eq_ a} (x y :a) xs ys :
  (cons x xs) == (cons y ys) = (x == y) && (xs == ys).
Proof.
    unfold op_zeze__, op_zeze____, Eq_list.
    simpl. auto.
 Qed.
```

(a) The theorem `simpl_list_cons_eq` theorem in `hs-to-coq` and its original proof.

```
(* The following proof is generated by GPT 4o-mini. *)
Proof.
  unfold Eq___list, op_zeze__, op_zeze____, eqlist.
  rewrite /eqlist.
  destruct xs, ys; simpl.
  - rewrite /op_zeze__.
    rewrite /op_zeze__.
    rewrite /op_zeze__.
    (* The proof goes on by repeating [rewrite]s. *)
```

(b) A failed proof generated by GPT 4o-mini that repeats the same tactic.

Figure 8. A comparison between the original proof and a failed proof generated by LLMs for theorem `simpl_list_cons_eq` in `hs-to-coq`.

errors. Moreover, when analyzing dependencies, we limited our scope to files within the same Rocq project. While the models were given the import statements in each setting, the external libraries were not indexed. Consequently, we assumed that LLMs would have implicit knowledge of widely used libraries such as ssreflect, or StructTact, InfSeqExt, and Cheerios (for Verdi), but did not verify this directly.

A further limitation lies in our dataset, which, while containing a substantial number of theorems, covers only two Rocq projects. Rocq projects may naturally vary in their structure and organization, which may heavily impact the results for settings with oen of dependencies or in-file contexts.

Finally, our experiments were conducted using Rocq version 8.10.2 and 8.11.0, which are both relatively old. While this choice was necessary to ensure compatibility with the codebases we studied, it may impact the relevance of results for newer versions of Rocq, if the LLMs we used were trained on more recent versions of the language.

6 Related Work

Benchmarks for proofs. CoqGym are pioneers in providing an extensive Rocq benchmark for machine learning models [92]. The benchmark containing 71K proofs from 123 real-life projects. It has been used by various studies on proof automation such as First and Brun [23], First et al. [24], Lu et al. [51]. These works are also inspiration of case studies presented in this paper. However, one issues with CoqGym is that it relies on older versions of Rocq Prover. For this reason, more recent tools like the `CoqPilot` benchmarking framework chooses to build their own datasets [44].

Outside Rocq Prover, there are many benchmarks for other proof assistants or formal-method tools, such as Dafny-Bench [50], LeanDojo [93], miniCodeProps [49], FVAPPS [20], VerifyThisBench [19], Verina [94], *etc.*

Proof automation. Proof automation has always been a goal in research on proof assistants. Most of these works rely on automated theorem provers (ATPs) like SAT/SMT solvers. For example, SMTCoq [5] uses SAT/SMT solvers to prove theorems and then reconstructs Rocq proofs from it. CoqHammer [14, 15] defines a set of automation tactics for dependent type theory, uses external ATPs to find a proof, and then constructs a proof using its own automation tactics by taking *hints* from proofs found by ATPs. In this way, CoqHammer is able to construct Rocq proofs that use intuitionistic logic with the help of ATPs that work on classical logic.

Other proof automation tools like Tactician [7] uses machine learning (but not LLMs) instead. It provides suggestions for the next tactic based on "previously written tactics". CoqGym, the benchmark for Rocq proofs, also includes a tool called ASTactic. The tool is trained on CoqGym and uses deep learning to generate proofs automatically [92]. Some more recent works in this area includes Proverbot9001 [72], Passport [73], QEDCartographer [74], *etc.*

LLMs and proof assistants. There have been a few recent works that investigate the capabilities of LLMs in generating proofs for proof assistants. We have already discussed Lu et al. [51]'s study on Verdi in Section 3.2. Qin et al. [67] studied FSCQ, a verified file system [12]. They conjectured that one reason LLMs fail to generate proofs is that LLMs struggle to find relevant lemmas when too many lemmas are given in a prompt [67, Section 4.3].

There have also been many works that leverage the power of LLMs to build proof-automation tools. For example, Baldur uses fine-tuned LLMs to generate whole proofs for Isabelle/HOL [25]. Their evaluation of Baldur on the PISA dataset [36] further shows that LLMs outperform small-model-driven search-based methods. *PALM* builds on their observation on Verdi (Section 3.2) and uses a generate-then-repair approach that combines LLMs and symbolic methods (*e.g.,* CoqHammer [14, 15]) to generate Rocq proofs [51]. Draft, Sketch, and Prove (DSP) uses LLMs to generate a sketch of a formal proof and then uses ATPs to fill in the missing details in the sketch [37]. Some other works in this area include Hu et al. [32], Kasibatla et al. [41], Lin et al. [48], Thompson et al. [82], Zhang et al. [98], *etc.*

Premise selection for proof generation. Premise selection refers to the process of selecting relevant *premises* such as definitions and lemmas [35]. This is a common process used by many proof-generation works. For example, *PALM* uses Term Frequency-Inverse Document Frequency (TF-IDF) [39] and k nearest neighbors (KNN) [21] to select relevant premises. `CoqPilot` selects premises based on "metrics such as distance from the generation target or similarity with other theorem statements" [44].

Our work takes a much simpler approach by directly including dependencies and in-file context in the prompt. Prior works like Baldur did a similar thing, but they only included in-file context [25, Section 2.3].

LLMs and math. LLMs have been studied extensively in the context of mathematics. Earlier research focuses on benchmarking LLMs with simple math reasoning tasks [6, 13, 101]. Recently, Olympiad-level math theorem proving have been successfully tackled by LLMs [2, 46, 83]. There has also been rapid progresses in auto-formalizing mathematics [47, 57, 88].

7 Conclusion

In this paper, we conduct a case study based on two real-world Rocq projects: the `hs-to-coq` project and Verdi. Our case study shows that LLMs can be effective in generating whole proofs for program correctness theorems. More specifically, we show that external dependencies and in-file context can significantly help with proof generation. We also find that LLMs perform well on small proofs. While its effectiveness degrades when the proof size increases, there is still a decent chance for it to generate whole proofs. However, our study also shows that the effectiveness characteristics of LLMs differ in different verification projects, which suggests that studying one project may not be sufficient for improving LLMs' effectiveness in other projects. Finally, we find that LLMs can generate concise and smart proof scripts, can apply classical techniques to new definitions, but can also produce meaningless stuttering proofs for unknown reasons.

We believe that LLMs for verification with proof assistants is a promising direction that deserves more attention. Program verification is suitable for tools like LLMs that are unpredicatable and can hallucinate [33, 99]. First, proofs are *not* computational. A generated *inefficient* proof have little to no impact compared with a generated inefficient program. Second, the proof checking mechanisms in proof assistants (*e.g.,* **`Qed`** of Rocq Prover) can safeguard generated proofs to make sure that they are correct.

Verification with proof assistants can be potentially much more useful in software engineering if proof automation can be significantly improved. Indeed, researchers have argued that one major reason that formal methods are rarely used in software development today is its social aspect [27]. It will greatly improve the usability of formal methods (and hence reliability of software) if LLMs can help with proof automation.

Acknowledgments

We thank all the anonymous reviewers of LMPL 2025 for their thoughtful and constructive comments to this paper and their suggestions for potential future directions for this work. We thank Yiming Lin for his feedback to a draft of this paper.

References

[1] Agda Developers. 2025. *Agda*. https://agda.readthedocs.io/

[2] AlphaProof. 2024. AI achieves silver-medal standard solving International Mathematical Olympiad problems Published. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/

[3] Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

[4] Andrew W. Appel. 2022. Coq's vibrant ecosystem for verification engineering (invited talk). In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 2–11. https://doi.org/10.1145/3497775.3503951

[5] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, 135–150. https://doi.org/10.1007/978-3-642-25379-9_12

[6] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. 2023. ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. *CoRR* abs/2302.12433 (2023). https://doi.org/10.48550/ARXIV.2302.12433 arXiv:2302.12433

[7] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The Tactician - A Seamless, Interactive Tactic Learner and Prover for Coq. In *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12236)*, Christoph Benzmüller and Bruce R. Miller (Eds.). Springer, 271–277. https://doi.org/10.1007/978-3-030-53518-6_17

[8] Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings (IFIP Transactions, Vol. A-10)*, Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute (Eds.). North-Holland, 129–156.

[9] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua M. Cohen, and Stephanie Weirich. 2021. Ready, Set, Verify! Applying `hs-to-coqm` to real-world Haskell code. *J. Funct. Program.* 31 (2021), e5. https://doi.org/10.1017/S0956796820000283

[10] Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3170)*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

[11] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 45, 1 (2023), 5:1–5:43. https://doi.org/10.1145/3579834

[12] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 18–37. https://doi.org/10.1145/2815400.2815402

[13] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *CoRR* abs/2110.14168 (2021). arXiv:2110.14168 https://arxiv.org/abs/2110.14168

[14] Lukasz Czajka, Burak Ekici, and Cezary Kaliszyk. 2018. Concrete Semantics with Coq and CoqHammer. In *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11006)*, Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef (Eds.). Springer, 53–59. https://doi.org/10.1007/978-3-319-96812-4_5

[15] Lukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reason.* 61, 1-4 (2018), 423–453. https://doi.org/10.1007/S10817-018-9458-4

[16] Leonardo de Moura and Sebastian Ullrich. 2021. 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 (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[17] Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7041)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer, 155–171. https://doi.org/10.1007/978-3-642-24690-6_12

[18] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao

Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[19] Xun Deng, Sicheng Zhong, Andreas G. Veneris, Fan Long, and Xujie Si. 2025. VerifyThisBench: Generating Code, Specifications, and Proofs All at Once. *CoRR* abs/2505.19271 (2025). https://doi.org/10.48550/ARXIV.2505.19271 arXiv:2505.19271

[20] Quinn Dougherty and Ronak Mehta. 2025. Proving the Coding Interview: A Benchmark for Formally Verified Code Generation. In *IEEE/ACM International Workshop on Large Language Models for Code, LLM4Code@ICSE 2025, Ottawa, ON, Canada, May 3, 2025*. IEEE, 72–79. https://doi.org/10.1109/LLM4CODE66737.2025.00014

[21] Sahibsingh A. Dudani. 1976. The Distance-Weighted k-Nearest-Neighbor Rule. *IEEE Trans. Syst. Man Cybern.* 6, 4 (1976), 325–327. https://doi.org/10.1109/TSMC.1976.5408784

[22] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Oper. Syst. Rev.* 54, 1 (2020), 23–30. https://doi.org/10.1145/3421473.3421477

[23] Emily First and Yuriy Brun. 2022. Diversity-Driven Automated Formal Verification. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1–13. https://doi.org/10.1145/3510003.3510138

[24] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 231:1–231:31. https://doi.org/10.1145/3428299

[25] 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, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1229–1241. https://doi.org/10.1145/3611643.3616243

[26] Emilio Jesús Gallego Arias. 2016. *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq.* Technical Report. MINES ParisTech. https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408

[27] Joseph A. Goguen and Luqi. 1995. Formal Methods and Social Context in Software Development. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/-FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 915)*, Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach (Eds.). Springer, 62–81. https://doi.org/10.1007/3-540-59293-8_187

[28] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. https://doi.org/10.1145/3591221

[29] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 595–608. https://doi.org/10.1145/2676726.2676975

[30] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[31] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 646–661. https://doi.org/10.1145/3192366.3192381

[32] Jilin Hu, Jianyu Zhang, Yongwang Zhao, and Talia Ringer. 2025. HybridProver: Augmenting Theorem Proving with LLM-Driven Proof Synthesis and Refinement. *CoRR* abs/2505.15740 (2025). https://doi.org/10.48550/ARXIV.2505.15740 arXiv:2505.15740

[33] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2 (2025), 42:1–42:55. https://doi.org/10.1145/3703155

[34] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. 1992. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.* 27, 5 (May 1992), 1–164. https://doi.org/10.1145/130697.130699

[35] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. DeepMath - Deep Sequence Models for Premise Selection. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 2235–2243. https://proceedings.neurips.cc/paper/2016/hash/f197002b9a0853eca5e046d9ca4663d5-Abstract.html

[36] 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.

[37] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=SMa9EAovKMC

[38] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *CoRR* abs/2406.00515 (2024). https://doi.org/10.48550/ARXIV.2406.00515 arXiv:2406.00515

[39] Karen Spärck Jones. 2004. A statistical interpretation of term specificity and its application in retrieval. *J. Documentation* 60, 5 (2004), 493–502. https://doi.org/10.1108/00220410410560573

[40] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[41] Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. 2024. Cobblestone: Iterative Automation for Formal Verification. *CoRR* abs/2410.19940 (2024). https://doi.org/10.48550/ARXIV.2410.19940 arXiv:2410.19940

[42] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel.. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220. https://doi.org/10.1145/1629575.1629596

[43] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

[44] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 2382–2385. https://doi.org/10.1145/3691620.3695357

[45] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[46] Zenan Li, Zhaoyu Li, Wen Tang, Xian Zhang, Yuan Yao, Xujie Si, Fan Yang, Kaiyu Yang, and Xiaoxing Ma. 2025. Proving Olympiad Inequalities by Synergizing LLMs and Symbolic Reasoning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=FiyS0ecSm0

[47] Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Xian Zhang, Fan Yang, and Xiaoxing Ma. 2024. Autoformalize Mathematical Statements by Symbolic Equivalence and Semantic Consistency. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/6034a661584af6c28fd97a6f23e56c0a-Abstract-Conference.html

[48] Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. 2024. FVEL: Interactive Formal Verification Environment with Large Language Models via Theorem Proving. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/62c6d7893b13a13c659cb815852dd00d-Abstract-Datasets_and_Benchmarks_Track.html

[49] Evan Lohn and Sean Welleck. 2024. miniCodeProps: a Minimal Benchmark for Proving Code Properties. *CoRR* abs/2406.11915 (2024). https://doi.org/10.48550/ARXIV.2406.11915 arXiv:2406.11915

[50] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2025. DafnyBench: A Benchmark for Formal Software Verification. *Trans. Mach. Learn. Res.* 2025 (2025). https://openreview.net/forum?id=yBgTVWccIx

[51] 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, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1509–1520. https://doi.org/10.1145/3691620.3695521

[52] Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.7118596

[53] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[54] Microsoft Azure AI Foundry Documentation. 2025. Azure OpenAI Reasoning Models. https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/reasoning. Accessed: 2025-07-02.

[55] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[56] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (1968), 514–534. https://doi.org/10.1145/321479.321481

[57] Logan Murphy, Kaiyu Yang, Jialiang Sun, Zhaoyu Li, Anima Anandkumar, and Xujie Si. 2024. Autoformalizing Euclidean Geometry. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=bylZbZOsGA

[58] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[59] Peter W. O'Hearn. 2007. Separation logic and concurrent resource management. In *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, Greg Morrisett and Mooly Sagiv (Eds.). ACM, 1. https://doi.org/10.1145/1296907.1296908

[60] Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. https://doi.org/10.1145/3371078

[61] Chris Okasaki and Andy Gill. 1998. Fast mergeable integer maps. In *ACM SIGPLAN Workshop on ML*. 77–86.

[62] OpenAI. 2024. GPT-4o. https://platform.openai.com/docs/models/gpt-4o. Accessed: 2025-07-08.

[63] OpenAI. 2024. GPT-4o Mini. https://platform.openai.com/docs/models/gpt-4o-mini. Accessed: 2025-07-08.

[64] OpenAI. 2025. o4-mini. https://platform.openai.com/docs/models/o4-mini. Accessed: 2025-07-08.

[65] OpenAI. 2025. tiktoken. https://github.com/openai/tiktoken. Accessed 2025-07-08.

[66] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2025. *Logical Foundations*. Electronic textbook. Version 6.7.1 https://softwarefoundations.cis.upenn.edu/lf-6.7.1/.

[67] Jianxing Qin, Alexander Du, Danfeng Zhang, Matthew Lentz, and Danyang Zhuo. 2025. Can Large Language Models Verify System Software? A Case Study Using FSCQ as a Benchmark. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems, HotOS 2025, Banff, AB, Canada, May 14-16, 2025*. ACM, 34–41. https://doi.org/10.1145/3713082.3730382

[68] Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. arXiv:2504.21801 [cs.CL] https://arxiv.org/abs/2504.21801

[69] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[70] Talia Ringer. 2021. *Proof Repair.* Ph. D. Dissertation. University of Washington, USA. https://hdl.handle.net/1773/47429

[71] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Found. Trends Program. Lang.* 5, 2-3 (2019), 102–281. https://doi.org/10.1561/2500000045

[72] Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. 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, MAPL@PLDI 2020, London, UK, June 15, 2020*, Koushik Sen and Mayur Naik (Eds.). ACM, 1–10. https://doi.org/10.1145/3394450.3397466

[73] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving Automated Formal Verification Using Identifiers. *ACM Trans. Program. Lang. Syst.* 45, 2 (2023), 12:1–12:30. https://doi.org/10.1145/3593374

[74] Alex Sanchez-Stern, Abhishek Varghese, Zhanna Kaufman, Shizhuo Dylan Zhang, Talia Ringer, and Yuriy Brun. 2025. QEDCartographer: Automating Formal Verification Using Reward-Free Reinforcement Learning. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025.* IEEE, 307–320. https://doi.org/10.1109/ICSE55347.2025.00033

[75] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434307

[76] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293.

[77] Antal Spector-Zabusky, Joachim Breitner, Yao Li, and Stephanie Weirich. 2019. Embracing a mechanized formalization gap. *CoRR* abs/1910.11724 (2019). arXiv:1910.11724 http://arxiv.org/abs/1910.11724

[78] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. https://doi.org/10.1145/3167092

[79] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

[80] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2013. Secure distributed programming with value-dependent types. *J. Funct. Program.* 23, 4 (2013), 402–451. https://doi.org/10.1017/S0956796813000142

[81] The Rocq Development Team. 2025. *The Rocq Prover.* https://doi.org/10.5281/zenodo.15149629

[82] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025.* IEEE, 347–359. https://doi.org/10.1109/ICSE55347.2025.00161

[83] Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nat.* 625, 7995 (2024), 476–482. https://doi.org/10.1038/S41586-023-06747-5

[84] Philip Wadler. 1992. Comprehending Monads. *Math. Struct. Comput. Sci.* 2, 4 (1992), 461–493. https://doi.org/10.1017/S0960129500001560

[85] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[86] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. *SIGPLAN Not.* 50, 6 (June 2015), 357–368. https://doi.org/10.1145/2813885.2737958

[87] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 154–165. https://doi.org/10.1145/2854065.2854081

[88] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/d0c6bc641a56bebee9d985b937307367-Abstract-Conference.html

[89] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[90] Xinference. 2025. deepseek-prover-v2. https://inference.readthedocs.io/en/latest/models/builtin/llm/deepseek-prover-v2.html Accessed: 2025-07-08.

[91] Xinference. 2025. deepseek-r1. https://inference.readthedocs.io/en/latest/models/builtin/llm/deepseek-r1.html Accessed: 2025-07-08.

[92] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6984–6994. http://proceedings.mlr.press/v97/yang19a.html

[93] Kaiyu Yang, Aidan M. 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. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html

[94] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. 2025. VERINA: Benchmarking Verifiable Code Generation. *CoRR* abs/2505.23135 (2025). https://doi.org/10.48550/ARXIV.2505.23135 arXiv:2505.23135

[95] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal reasoning about layered monadic interpreters. *Proc. ACM Program. Lang.* 6, ICFP (2022), 254–282. https://doi.org/10.1145/3547630

[96] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572

[97] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on*

Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193), Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPICS.ITP.2021.32

[98] Lichen Zhang, Shuai Lu, and Nan Duan. 2024. Selene: Pioneering Automated Proof in Software Verification. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 1776–1789. https://doi.org/10.18653/V1/2024.ACL-LONG.98

[99] Shizhuo Dylan Zhang, Talia Ringer, and Emily First. 2023. Getting More out of Large Language Models for Proofs. *CoRR* abs/2305.04369 (2023). https://doi.org/10.48550/ARXIV.2305.04369 arXiv:2305.04369

[100] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. 2024. Explainability for Large Language Models: A Survey. *ACM Trans. Intell. Syst. Technol.* 15, 2 (2024), 20:1–20:38. https://doi.org/10.1145/3639372

[101] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. miniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=9ZPegFuFTFv

[102] Zihan Zhou, Minfeng Zhu, and Wei Chen. 2025. A human-centric perspective on interpretability in large language models. *Vis. Informatics* 9, 1 (2025), 1. https://doi.org/10.1016/J.VISINF.2025.03.001