Proof Automation with Large Language Models

Minghai Lu
lu1074@purdue.edu
Purdue University
West Lafayette, IN, USA

Benjamin Delaware
bendy@purdue.edu
Purdue University
West Lafayette, IN, USA

Tianyi Zhang
tianyi@purdue.edu
Purdue University
West Lafayette, IN, USA

ABSTRACT

Interactive theorem provers such as Coq are powerful tools to formally guarantee the correctness of software. However, using these tools requires significant manual effort and expertise. While Large Language Models (LLMs) have shown promise in automatically generating informal proofs in natural language, they are less effective at generating formal proofs in interactive theorem provers. In this paper, we conduct a formative study to identify common mistakes made by LLMs when asked to generate formal proofs. By analyzing 520 proof generation errors made by GPT-3.5, we found that GPT-3.5 often identified the correct high-level structure of a proof, but struggled to get the lower-level details correct. Based on this insight, we propose *PALM*, a novel generate-then-repair approach that first prompts an LLM to generate an initial proof and then leverages targeted symbolic methods to iteratively repair low-level problems. We evaluate *PALM* on a large dataset that includes more than 10K theorems. Our results show that *PALM* significantly outperforms other state-of-the-art approaches, successfully proving 76.6% to 180.4% more theorems. Moreover, *PALM* proves 1270 theorems beyond the reach of existing approaches. We also demonstrate the generalizability of *PALM* across different LLMs.

KEYWORDS

Software and its engineering → *Software verification*; *Formal software verification.*

1 INTRODUCTION

Correctness is crucial to software systems. Interactive theorem provers (ITPs) such as Coq [43], Isabelle [34] and Lean [17], are powerful tools for providing semantically rich guarantees about software. In an ITP, users can state and prove formal theorems about a program; these proofs are then mechanically checked by the ITP, providing a strong, foundational guarantee about its correctness. This strategy has been successfully applied to several application domains, including compilers [31], distributed systems [46], and

OS kernels [29]. While powerful, this approach comes at a cost, as users must supply a *proof script* that helps the ITP construct the proof of the desired theorem. Constructing these proof scripts can require considerable effort. For example, it took 6 person-years to write 100,000 lines of Coq proof scripts to verify the CompCert C compiler [31].

Many proof automation techniques have been proposed to reduce the effort required by ITPs. These techniques mainly fall into two categories: symbolic methods [15, 28, 36, 44] and machine learning methods [20, 21, 39, 48]. Symbolic methods use a combination of previously established theorems and external automated theorem provers (ATPs), such as Z3 [16] and CVC5 [14], to automate the proof of a theorem. While effective, these approaches are constrained by their inability to perform higher-order and inductive reasoning, limiting their ability to prove complex theorems. Machine learning methods utilize models to predict the next proof step in a heuristic-guided search process. These methods do not have the same limitations as symbolic approaches but require a significant amount of training data [20, 21, 48].

Recently, pretrained Large Language Models (LLMs) have shown promise in generating informal natural language proofs [47], suggesting a potential to further improve existing proof automation approaches. Unfortunately, even state-of-the-art LLMs are ineffective at generating formal proofs in one shot: GPT-3.5 proves 3.7% of theorems in our evaluation, and Llama-3-70b-Instruct proves 3.6%. In order to understand why, we have conducted a formative study to analyze mistakes that GPT-3.5 made when generating formal proofs. In this study, we analyzed 579 theorems of varied complexity and identified seven categories of errors. Overall, we found that while GPT-3.5 often produced proofs with the right high-level structure, it struggled getting lower-level details of these proofs correct. Promisingly, we also observed that many of these errors can be potentially fixed using symbolic methods, including heuristic-based search and proof repair.

Guided by this formative study, we propose *PALM*, a novel generate-then-repair approach that combines LLMs and symbolic methods. Our key insight is to use LLMs to generate an initial proof that is likely to have the correct high-level structure, and then use targeted symbolic methods to iteratively repair low-level problems related to individual proof steps. *PALM* relies on four repair mechanisms that target the common types of errors identified in our formative study. If our repair mechanisms fail, *PALM* uses a backtracking procedure to regenerate previous proof steps in an attempt to fix errors in the high-level proof structure. Although *PALM* targets Coq, its underlying principles can be applied to other ITPs, such as Isabelle [34] and Lean [17].

To evaluate the effectiveness of our approach, we have conducted an extensive evaluation using the CoqGym dataset [48] with 10842 theorems. Our results suggest that *PALM* can successfully prove

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

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

40.4% of the theorems, significantly outperforming the state-of-the-art methods *Passport* [39], *Proverbot9001* [38] and *Draft, Sketch, and Prove (DSP)* [27], which only prove 14.4%, 17.1% and 22.9% theorems, respectively. Moreover, we have conducted experiments to demonstrate the effectiveness of each component in *PALM* and the generalizability of *PALM* across different LLMs.

In summary, this paper presents the following contributions:

(1) We conduct a formative study to identify the common errors made by GPT-3.5 while proving theorems in Coq.

(2) We propose *PALM*, a novel proof automation approach that combines LLMs and symbolic methods in a generate-then-repair pipeline.

(3) We evaluate *PALM* on a large dataset and demonstrate that *PALM* significantly outperforms existing methods. An artifact containing the source code of *PALM* and a replication package is publicly available [7].

2 PRELIMINARIES

2.1 Interactive Theorem Proving in Coq

The Coq proof assistant [43] is a popular tool for developing machine-checked proofs of mathematical theorems and verifying complex software systems. Coq helps users interactively construct these proofs using a set of proof *tactics*. This section first introduces the basic concepts of interactive proof development in Coq, and then illustrates the process via an example theorem shown in Figure 1.

Theorems: In Coq, the definition of a *theorem* typically starts with the keyword Theorem or Lemma, followed by its name and the theorem statement. Figure 1 shows the theorem *add_comm*, which states that natural number addition is commutative.[1] This is then followed by a *proof script*, a sequence of tactics that explain how to build a proof of the desired statement. Proof scripts are typically developed in an interactive proof mode. Processing the first line of Figure 1 causes Coq to enter proof mode. During the proof process, users can freely reuse previously proven theorems.

Proof States: In proof mode, Coq's interface displays the current *proof state*, i.e., a list of unproven *goals*. Each of these goals is a pair of a local context *lc* and an outstanding proof obligation *st*. A local context includes hypotheses and assumptions that can be used to prove *st*; these are distinct from the set of previously proven theorems, which are part of the global context. Figure 2 shows the intermediate proof states that appear during the proof of *add_comm*: each listing shows the proof states shown to the user

---

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

(a) Proof state at the start.

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

(b) Proof state after Figure 1 Line 3 (`intros n m`).

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

(c) Proof state after Figure 1 Line 4 (`induction n`).

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

(d) Proof state after Figure 1 Line 5 (the first subgoal).

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

(e) Proof state after Figure 1 Line 7 (the second subgoal).

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

(f) Proof state after Figure 1 Line 8 (`simpl`).

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

(g) Proof state after Figure 1 Line 9 (`rewrite IHn`).

Figure 2: Proof state after the execution of each tactic in the proof of addition's commutativity.

after processing each tactic in Figure 1. Following the conventions of Coq's user interface, the local context is shown above the double line, and the current proof obligation is shown below.

Tactics: Tactics specify strategies for decomposing the current proof obligation into a set of simpler subgoals, in order to eventually produce a complete proof. Conceptually, a tactic t is a state-transition function: $t \in S \times \sum \rightarrow S'$, where S is a goal, $\sum$ is a set of arguments if any, and S' is the set of resulting goals. As an example, the tactic `induction n` on Line 4 of Figure 1 tells Coq to do induction on the natural number n in the local context. Processing this tactic transforms the proof state in Figure 2b to the proof state in Figure 2c, which has two subgoals: (1) a base case in which n is 0, and (2) an inductive case in which n is an arbitrary natural number.[2] Note that Coq only displays the local context of the first goal when there are multiple goals. Importantly, a tactic can fail if, for example, it is applied to a proof state of the wrong form or it is supplied with wrong arguments. Coq reports the failure back to the user when this occurs.

Proofs: A proof of a theorem consists of a sequence of tactics that transform the initial goal, i.e., the theorem statement, into subgoals until none remain. The beginning and end of a proof are delimited

---

by the `Proof` and `Qed` commands (Lines 2 and 11 of Figure 1). The latter command prompts Coq's kernel to check that no outstanding proof obligations remain. If so, Coq exits proof mode with success and the theorem is added to the global context.

We now illustrate these concepts using the example of `add_comm` in Figure 1. At the beginning of the proof (Line 2), the proof state consists of a single goal that corresponds to the top-level theorem statement (Figure 2a). At Line 3, the tactic `intros n m` tells Coq to move the universally quantified variables `n` and `m` into the local context (Figure 2b). Then, the aforementioned `induction n` tactic performs induction on `n` (Line 4), resulting in two subgoals corresponding to the base case and inductive case (Figure 2c). We then prove the first subgoal with a *bullet* "`-`" (Line 5), which marks the beginning of the subgoal's proof and causes Coq to display only this subgoal to the user (Figure 2d). After proving this subgoal, we prove the next subgoal with the same bullet symbol (Line 7). These bullets help organize the proof by marking the beginning of each subgoal and instructing Coq to ensure one subgoal is proven before moving to the next.

We solve the base case (Figure 2d) by invoking the `auto` tactic (Line 6), which uses symbolic-based proof automation to discharge simple goals. Next we move on to the second subgoal—the inductive case. Importantly, this goal includes an inductive hypothesis in its local context. We first use the `simpl` tactic (Line 8), which simplifies the goal by evaluating the `+` operator (Figure 2f). Next, we use the `rewrite IHn` tactic (Line 9), which substitutes the left-hand side of the inductive hypothesis (`IHn`) in the goal with its right-hand side (Figure 2g). Finally, we `apply` a previously proven theorem `plus_n_Sm: forall n m : nat, 1 + (n + m) = n + (1 + m)` from the standard Coq library (Line 10). This theorem establishes that adding 1 to the sum of `n + m` is the same as adding `n` to `m + 1`. A theorem of the form $A \Longrightarrow B$ can be applied to a goal if its conclusion (`B`) matches the current proof obligation, resulting in a new goal corresponding to its hypothesis (`A`). The `apply plus_n_Sm` tactic directly solves the current goal, because the conclusion of `plus_n_Sm` matches and `plus_n_Sm` has no premises. Since no goals remain, the proof is complete, and we use the `Qed` command to finish the proof of `add_comm`.

2.2 Hammers

To facilitate proof construction, Coq is equipped with many established proof automation tactics (e.g., `auto`). These tactics either completely solve the current goal, or leave it unchanged if they fail. Among them, *hammers* [15, 28, 36] are powerful tactics that dispatch goals using external automated theorem provers (ATPs), such as Vampire [37], CVC5 [14], E [3] and Z3 [16]. Many popular ITPs have hammers, including CoqHammer [15] for Coq, Sledge-Hammer [36] for Isabelle, and HOLyHammer [28] for HOL Light.

At a high level, hammers work by first encoding the current goal into a form solvable by an ATP, typically a formula in first-order logic. This is necessary because ITPs support much richer logic, e.g., higher-order logic, than most ATPs. In order to enable the underlying ATP to use previously proven theorems, a subset of the theorems in the global context are encoded alongside the current goal, the task of selecting a relevant set of these theorems is sometimes called *premise selection* [13]. Early hammers relied

on heuristics to select premises, while modern hammers typically utilize machine learning algorithms for this purpose. After a goal and the selected premises have been encoded, hammers invoke an ATP. If successful, the proof found by the ATP is translated to a form that can be understood by an ITP. Hammers are typically invoked by applying specific tactics. CoqHammer offers a collection of tactics such as `hammer`, `hfcrush`, and `qsimpl`, each of which automatically proves goals using different strategies. While powerful, hammers only perform a subset of reasoning available to an ITP: they typically do not perform induction, for example. This limits their ability to directly prove complex theorems. Nonetheless, they are effective at accurately solving small subgoals. For instance, the `hammer` tactic is able to completely solve the goals in Figures 2d and 2e, while Coq's `auto` tactic only solves the first goal.

3 FORMATIVE STUDY

While LLMs have previously been used to generate proofs in Coq, they have not proven particularly effective at the task [22, 42, 50]. To understand the root causes of this, we have conducted a formative study to identify the common errors made by LLMs when asked to generate proof scripts. In this study, we evaluated the ability of GPT-3.5 [4] to prove 579 theorems from Verdi, a distributed system verification project [46]. Verdi has also been used in other studies [20, 21, 48]. We carefully designed our prompt based on the widely used retrieval augmented generation (RAG) method [32]. This prompt is also used by *PALM*, and is described in more detail in Sections 4.2 and 4.3. We prompted GPT-3.5-turbo-1106 API, the latest version available at the time of this study with the default decoding temperature. For each theorem, we sampled only one proof script. We ran the generated proof in Coq, and recorded the error message of the first encountered error in the proof.

We collected a total of 520 errors and conducted an in-depth manual analysis, following the grounded theory [23] and the open coding method [24]. The first author first labeled 100 errors and came up with an initial categorization. He then discussed and refined the labels and the categorization with the last author in two meetings. The first author then labeled and categorized the remaining errors based on the refined labels and categorization. Finally, all authors met to discuss and finalize the categorization, where the second author, an expert in theorem proving, offered insights that further enhanced the categorization. The whole process took approximately 52 person-hours. We categorized the 520 errors into seven types, as shown in Table 1.

Type	# of occurrences	Percentage (%)
Wrong theorem application	258	49.6
Invalid reference	79	15.2
Incorrect rewrite	61	11.7
Redundant introductions	56	10.8
Tactic misuse	44	8.5
Bullet misuse	19	3.7
Miscellaneous errors	3	0.6
Total	**520**	**100**

Table 1: The number of occurrences and percentage of each type of error.

1. Wrong theorem application (49.6%): When there is a theorem or hypothesis of the form `H: ` $A \Longrightarrow B$ and the current goal is B,

the `apply H` tactic can be used to replace the goal with H's premise (A). This tactic requires that the conclusion of H matches the goal. Attempting to apply a theorem or hypothesis whose conclusion does not match the goal will cause the tactic to fail. For example, in Figure 3, the proof state has a hypothesis `H: m = n`, which cannot be applied because it does not match the goal `n = m`. Applying H fails with the error message "*Unable to unify 'm=n' with 'n=m'.*"

```
n, m: nat
H: m = n
===========================
n = m
```

Figure 3: `apply H` causes a wrong theorem application: "Unable to unify 'm=n' with 'n=m' ".

2. Invalid reference (15.2%): LLMs can generate incorrect references, such as a hypothesis that does not exist in the local context or a theorem that cannot be found in the environment. This is a form of LLM hallucination [50].

3. Incorrect rewrite (11.7%): Given an equation `Heq`, the `rewrite Heq` tactic replaces occurrences of the left-hand side of `Heq` in the goal with the right-hand side of `Heq`. The error occurs when a theorem or hypothesis is used for rewriting but its left-hand side does not match any subterms in the goal. For instance, consider the proof state presented in Figure 5. The `rewrite H2` tactic fails with the error message "*Found no subterm matching 'b' in the current goal.*"

4. Redundant introductions (10.8%): The `intros` tactic is used to move universally quantified variables and assumptions into the local context. In some cases, LLMs produce proofs that use `intros` to introduce a term with a name that is already in the local context, or when there is nothing that can be moved to the local context.

5. Tactic misuse (8.5%): Some tactics can only be used with specific arguments. This error occurs when a tactic is given an argument that does not satisfy its requirements. For example, `destruct` and `induction` can only be applied to arguments with inductive data types such as natural numbers. Both tactics fail with the error "*Not an inductive product*" when applied to a non-inductive argument. Conversely, the tactic `unfold` cannot be applied to arguments with inductive types, and will throw an error "*Cannot turn inductive into an evaluable reference.*"

Other built-in tactics can be misused in a way specific to the tactic. For example, the `reflexivity` tactic causes an error when applied to a goal that is not an equality between equivalent terms, while the `contradiction` tactic fails when the local context does not contain a contradiction.

6. Bullet misuse (3.7%): LLMs can generate proofs which misuse bullets in two ways: (1) the proof tries to proceed to the next goal before the current one is solved, and (2) the proof uses the wrong bullet to focus on a goal. Figure 6 illustrates this misuse through two incorrect proofs. In the first proof, the second bullet symbol should be "–" instead of "+" (Line 6). This leads to an error "*Wrong bullet +: Expecting –.*" In the second proof, `simpl` fails to completely solve the first subgoal (Line 12), so trying to proceed to the second subgoal while the first is unsolved leads to an error "*Wrong bullet –: Current bullet – is not finished.*"

7. Miscellaneous errors (0.6%) Some errors do not fit into the previously defined categories. For example, LLMs can generate

special commands like `Abort`, which terminates a proof without an error before it is complete.

A key insight from this formative study is that 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. On the other hand, CoqHammer is good at addressing these subgoals using ATPs. In addition, we found that many proof generation errors are relatively straightforward to fix, e.g., through rule-based transformation, without the need of regenerating the proof from scratch. For instance, both cases of bullet misuse can be repaired by systematically inserting the correct bullet.

4 APPROACH

Guided by our formative study, we propose *PALM*, a proof automation approach that combines LLMs and symbolic methods. Figure 4 provides an overview of *PALM*. *PALM* includes three components: (1) a retrieval-augmented proof generation method, (2) a set of repair mechanisms, and (3) a backtracking procedure.

4.1 The Overall Algorithm

Algorithm 1 describes the overall generate-then-repair procedure used by *PALM*. The inputs are a theorem statement t, an environment *env*, and a language model *LM*. First, using the retrieval augmented generation (RAG) method described in Section 4.2, *PALM* retrieves relevant premises from *env* based on t (Line 3). Next, *PALM* creates a prompt using t and the selected premises (Line 4), and prompts *LM* to obtain an initial proof script (Line 5). *PALM* then executes these tactics in Coq (Lines 6-15). If an error occurs, *PALM* employs a set of repair mechanisms to fix the problem based on the error message, the tactic that throws the error, and the current proof state (Line 9). If *PALM* cannot fix an error, it invokes the backtracking procedure (Line 11) described in Algorithm 2, which attempts to fix the previous proof using CoqHammer. The proof is successful if no goals remain unsolved after all tactics have been executed (Line 16).

Algorithm 1 Framework

1: **Input**: Theorem statement t, Environment *env*, Language Model *LM*
2: **function** PROVE(t, *env*, *LM*)
3:     $PS \leftarrow$ RetrievePremises(t, *env*)
4:     $pt \leftarrow$ BuildPrompt(t, PS)
5:     $TCS \leftarrow LM$.query(pt)
6:     **for** $tc \in$ TCS **do**
7:         $error \leftarrow$ Coq.execute(tc)
8:         **if** $error$ **then**
9:             $repaired \leftarrow$ Repair($error$, tc, *current proof state*)
10:             **if** not $repaired$ **then**
11:                 $proof \leftarrow$ Backtracking(*current goal*)
12:                 **if** $proof$ is not None **then**
13:                     Coq.execute($proof$)
14:                 **else**
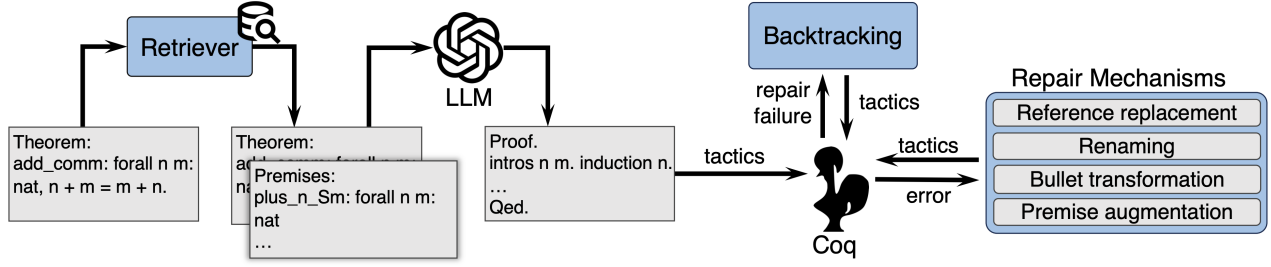15:   **return** False
16:     **return** NoUnsolvedGoals()

Figure 4: Overview of *PALM*.

```
a, b, c: nat
H1: a = b
H2: b = c
===========================
a = c
```

Figure 5: `rewrite H2` fails with: "Found no subterm matching 'b' in the current goal".

```
1  Theorem add_comm : forall n m, n + m = m + n.
2  (* wrong proof *)
3  Proof.
4    intros. induction n.
5    - auto.
6    + (* proof for the inductive case *)
7  Qed.
8
9  (* wrong proof *)
10 Proof.
11   intros. induction n.
12   - simpl.
13   - (* proof for the inductive case *)
14 Qed.
```

Figure 6: Two examples of bullet misuse.

In the rest of this section, we describe each component of *PALM* using the example shown in Figure 7. The proof script shown in the figure was generated by GPT-3.5 and contains several errors. The correct proof produced by *PALM* is shown in Figure 8.

```
1  Lemma sqr_le: forall a: Z, a <= a * a.
2  Proof.
3    intros. destruct a.
4    - reflexivity.
5    - induction p.
6      + simpl. ring.
7      + apply Z_le_dec.
8      + apply Z.le_refl.
9    - apply Z.eq_le_incl.
10 Qed.
```

Figure 7: A theorem stating $a \leq a \times a$ for any integer a (Line 1), and an erroneous proof (Lines 2 to 10) produced by GPT-3.5.

```
1  Lemma sqr_le : forall a : Z, a <= a * a.
2  Proof.
3    intros. destruct a.
4    - reflexivity.
5    - chfcrush use: Zlt_le_succ, Pos2Z.is_pos,
6        Z.le_mul_diag_r.
7    - hfcrush.
8  Qed.
```

Figure 8: The correct proof found by *PALM*.

4.2 Premise Retrieval

High-quality context is essential for LLMs to produce accurate responses. For theorem proving, we consider the previously proven theorems and definitions available in the environment as the context of constructing a proof script.
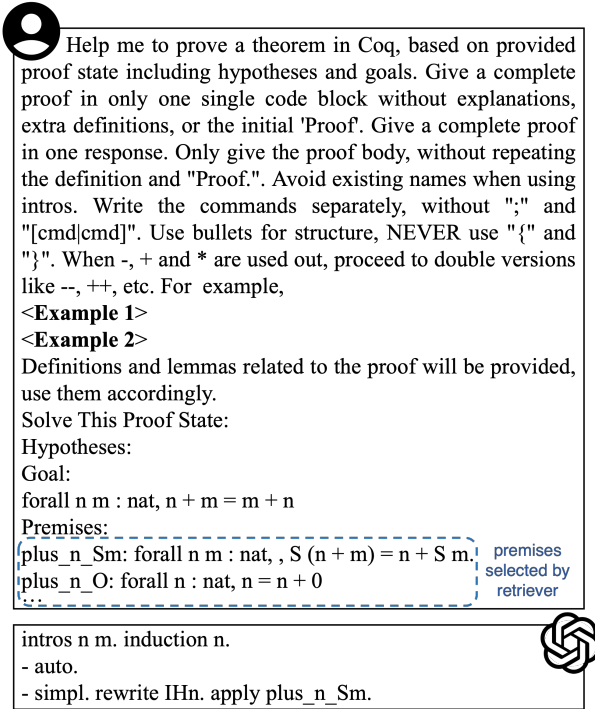
Given there are many available theorems and definitions, it is difficult to encode all of them in the proof generation prompt. Thus, we develop an information retrieval method to identify the ones relevant to the theorem to be proven. Specifically, *PALM* predicts relevant premises using Term Frequency-Inverse Document Frequency (TF-IDF) [40] and k nearest neighbors (KNN) [18]. While more advanced methods such as deep learning [13] can be more accurate, they also require significant amounts of training data and can take a longer time to make predictions [19, 30]. The premises predicted by the KNN algorithm are initially ranked by their TF-IDF scores. *PALM* then employs the BM25 algorithm [11] to rerank these premises based on their text similarity to the statement of the theorem, since we observed that BM25 tends to rank premises used in human-written proofs higher than TF-IDF.

4.3 Prompt Design

To optimize the quality of the initial proof generated by the LLMs, we carefully designed the prompt used by *PALM* following strategies for few-shot in-context learning. Our strategy for designing this prompt was inspired by recent findings that LLMs can produce instructions that are superior or equivalent to those crafted by humans [53]. We first asked GPT-4 to infer the five most effective instructions for two theorems accompanied by human-written proof scripts. Next, we constructed candidate prompts by combining these five sets of instructions with the two examples, premises and a new theorem to be proven. The inclusion of the two example theorems in this query was meant to demonstrate the correct Coq syntax and our desired proof style. The first author then manually examined 20 proofs produced by the LLM in response to these prompts and chose the prompt that yielded the highest quality proofs. Proof quality was assessed based on correctness and adherence to the instructions, e.g., using bullets for structure, etc. When multiple proofs met these criteria, the simplest (i.e. shortest) correct proof was preferred. Figure 9 illustrates the final prompt template with an example and the response of GPT-3.5.

4.4 Repair Mechanisms

The proofs produced by LLMs typically feature a good high-level structure that decomposes the proof into reasonable subgoals. However, most of these proofs are rejected by Coq due to errors, as

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

premises selected by retriever

intros n m. induction n.
- auto.
- simpl. rewrite IHn. apply plus_n_Sm.

Figure 9: An example of the prompt constructed by *PALM*, and the response of GPT-3.5.

discussed in Section 3. To address this issue, we have developed a set of repair mechanisms to handle the common types of errors.

Reference replacement When LLMs generate tactics referencing undefined theorems or hypotheses, *PALM* systematically searches in the local and global context for theorems and hypotheses with similar names, in order to find suitable replacements. Specifically, *PALM* first collects a set of candidates, including the relevant theorems selected by the retrieval method, and hypotheses in the local context. It then ranks these candidates using BM25 based on the similarity of their names to the initial undefined name. Then, *PALM* iteratively replaces the undefined reference name in the tactic with each candidate and asks Coq to re-execute the updated tactic, until the the tactic succeeds. For example, if a candidate proof uses the tactic `apply in_remove_all` but `in_remove_all` does not exist, *PALM* searches for similar reference names. It first ranks the selected theorems and hypotheses based on the similarity of their names to `in_remove_all`. Then, *PALM* iteratively replaces `in_remove_all` with the candidates and eventually finds a tactic `apply in_remove_all_preserve`, which solves the goal.

Renaming If a proof script tries to introduce a term using `intros` but the specified name already exists in the local context, *PALM* appends an apostrophe to the specified name and updates the tactic accordingly. For example, if the tactic `intros H` is used but `H` already exists in the local context as a hypothesis, *PALM* updates the tactic to `intros H'`. If there is nothing that can be moved to the local context, *PALM* simply drops the `intros` tactic from the current proof script.

Bullet transformation *PALM* handles bullet misuse in two ways, depending on the two specific scenarios described in Section

3. First, if the current goal has been solved and the next goal is focused on using the wrong bullet, Coq will indicate the expected bullet, and *PALM* will simply update the proof to use it. Second, if the proof attempts to proceed to the next goal or finish the proof while there are still unsolved goals, *PALM* will delegate the repair effort to the backtracking procedure described in the next section.

To illustrate this, consider the last subgoal produced by `destruct a`, as shown in Line 9 of Figure 7. The `apply Z.eq_le_incl` tactic fails to fully solve this subgoal, so attempting to finish the proof with Qed (Line 10) causes an error. To fix this, *PALM* starts the backtracking procedure using the goal that results from the `apply Z.eq_le_incl` tactic. Eventually, the backtracking procedure replaces the `apply Z.eq_le_incl` tactic with the `hfcrush` tactic (Line 6 in Figure 8) and completely solves the goal.

Premise augmentation LLMs can produce proof scripts that misuse a theorem, resulting in a *wrong theorem application*, *wrong rewriting*, or *tactic misuse* error. Despite this, the misused theorems are still potentially helpful: although used improperly in the proof script, they might still aid in solving the goal if used in a different manner. Based on this insight, *PALM* leverages CoqHammer to determine how to use these theorems correctly. Specifically, it employs the `qsimpl` tactic provided by CoqHammer, which accepts a list of theorems as arguments. `qsimpl` uses sophisticated heuristics to identify which theorems can be applied and simplifies the current goal accordingly. Similar tactics are available in other proof assistants. *PALM* executes `qsimpl` with a misused theorem as the argument, allowing it to automatically discover the correct usage of this theorem. For example, if a tactic `apply Zlt_le_succ` causes an error because its conclusion does not match the current goal, *PALM* will execute `qsimpl use: Zlt_le_succ` to utilize this theorem despite its initial misuse.

4.5 Backtracking

PALM leverages CoqHammer to solve goals that the initial script fails to prove due to errors that cannot be repaired. Although other proof automation techniques could be employed, we found hammers to be effective in practice. Applying a wrong tactic can result in a new goal that is more difficult, or even impossible to prove. Thus, when CoqHammer fails to solve a goal, it is clear that *PALM* needs to backtrack to an earlier point in the proof to see if it can be solved instead. Particularly, if a tactic produces multiple subgoals, all these subgoals must be proven. If *PALM* fails to prove any of them, it needs to revert to the goal before that tactic. For example, when reasoning by induction, if the base case is proven but the inductive case fails, the entire induction attempt has failed, and we need to try a different proof strategy instead of induction.

Algorithm 2 presents our backtracking procedure, which aims to prove unsolved goals using CoqHammer. The input to the procedure is an unsolved goal *g*. If *g* is successfully solved by CoqHammer, it returns the proof found by CoqHammer immediately (Lines 4-5). Otherwise, *PALM* reverts to the goal before the last applied tactic, and tries CoqHammer again. If the last command is a bullet (Line 6), it means that our algorithm will not be able to prove this subgoal using CoqHammer. When this happens, the algorithm identifies the tactic that produced this subgoal as the *root* (Line 7), and then discards *root* and all its associated subgoals (Line 8). Having the

---

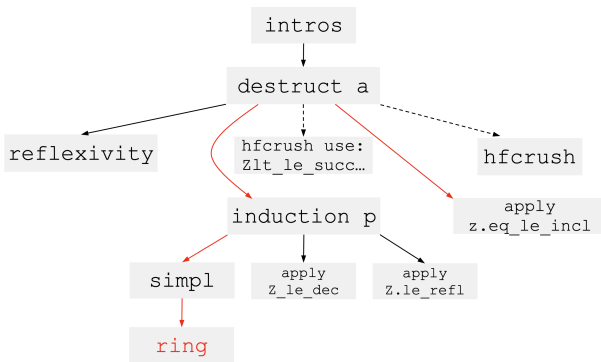**Algorithm 2** Backtracking

```
 1:  Input: Unsolved Goal g
 2:  function BACKTRACK(g)
 3:      while exist tactics do
 4:          if g is solved by CoqHammer then
 5:              return CoqHammer.getProof()
 6:          else if the last tactic is a bullet then
 7:              root ← the tactic produces this subgoal
 8:              discard root and its subgoals
 9:          else
10:              Coq.undo()
11:      return None
```

LLM structure its proof using bullets helps PALM identify which parts of the proof need to be dropped when a subgoal fails. If the last tactic is not a bullet, *PALM* simply reverts to the goal before that tactic (Line 10). This loop continues until CoqHammer succeeds (Line 5) or no tactics remain in which case the repair attempt fails (Line 3).

We demonstrate the backtracking procedure on the `induction p` tactic and its subgoals (shown in Lines 5-8 of Figure 7). Initially, all tactics up to `ring` (Line 6) are executed without errors. However, the `ring` tactic fails and cannot be repaired, so *PALM* starts backtracking. The input is the goal resulting from the execution of the last tactic, `simpl`. Algorithm 2 invokes CoqHammer, but it fails to solve the goal, so *PALM* reverts the `simpl` tactic and invokes CoqHammer again, but CoqHammer still fails. At this point, the algorithm hits a bullet ("+"), and there are no remaining tactics that can be repaired using CoqHammer. This indicates that the first subgoal produced by `induction p` cannot be proven, leading to the failure of the entire induction attempt. Accordingly, *PALM* discards the `induction p` tactic (Line 5) along with the tactics corresponding to all its subgoals (Lines 6-8). Algorithm 2 then reverts to the second subgoal produced by `destruct a`, which is successfully solved by CoqHammer. The proof found by CoqHammer is presented in Lines 5-6 of Figure 8.

Figure 10: Visualization of our backtracking repair algorithm. The red lines indicate the reverted tactics, and the dashed lines indicate the tactics found by CoqHammer during backtracking.

5 EVALUATION

Our experimental evaluation of our approach addresses four key research questions:

- **RQ1:** Is *PALM* more effective at proving theorems than other state-of-the-art proof automation approaches?
- **RQ2:** Can *PALM* generalize to other LLMs with different parameter sizes?
- **RQ3:** How much does each component of *PALM* contribute to its effectiveness?
- **RQ4:** Is *PALM* time-efficient?

We conducted experiments on a workstation with an AMD EPYC 7313 CPU, an NVIDIA A5500 GPU, and 512GB memory. The operating system was 64-bit Ubuntu 22.04 LTS.

5.1 Comparison baselines

We compare *PALM* against three state-of-the-art proof automation approaches: *Passport* [39], *Proverbot9001* [38] and *Draft, Sketch, and Prove (DSP)* [27]. Both *Passport* and *Proverbot9001* are machine learning methods. *Passport* employs a Tree-LSTM [41] to model proof states, incomplete proof scripts, and identifiers in proofs. *Proverbot9001* adopts an RNN to model manually engineered features of the proof states. *DSP* prompts LLMs to translate natural language proofs into formal proof sketches that outline high-level proof steps without low-level details. The informal proofs can be either written by humans or generated by LLMs. It then uses off-the-shelf proof automation tools such as hammers to fill in the gaps. Unlike *DSP*, *PALM* does not require informal proofs, and employs repair mechanisms and a backtracking procedure to fix proof errors. As human-written proofs were unavailable for the benchmarks used in our test set, in order to reproduce *DSP*, we used GPT-3.5 to generate informal proofs and sketches, and use CoqHammer as the underlying proof automation tool.

5.2 Benchmark construction

Following prior work [38, 39, 48], we use the test set of CoqGym [48] as the evaluation dataset, which consists of 13,137 theorems from 27 open-source Coq projects. Since the theorems from the Verdi project used in our formative study are also included in CoqGym, we exclude them to avoid biases. As we ran the baselines on CoqGym, we found that *Passport* is compatible exclusively with Coq 8.9, and relies on CoqGym's original dataset. *Proverbot9001*, which does not use CoqGym, supports only newer versions of Coq, namely Coq 8.10, 8.11, and 8.12. To ensure fairness, our evaluation is conducted on a subset of CoqGym, including 10842 theorems that are compatible with all relevant versions of Coq. We implement *PALM* for Coq 8.10, 8.11, and 8.12, since many language features and standard libraries of Coq 8.9 are outdated [2].

5.3 Results

In RQ1, we compare the performance of *PALM* using GPT-3.5 as the underlying LLM against the baselines. In RQ2, we evaluate the performance of *PALM* when using different LLMs.

5.3.1 RQ1: Effectiveness of PALM. Table 2 shows the number and percentage of theorems each approach can successfully prove. Compared with existing approaches, *Passport*, *Proverbot9001*, and *DSP*,

Approach	# of Theorems Proven
Passport	1561 (14.4%)
Proverbot9001	1849 (17.1%)
Draft, Sketch, and Prove (DSP)	2478 (22.9%)
GPT-3.5	402(3.7%)
+ *PALM*	4377 (40.4%)
GPT-4o	689(6.4%)
+ *PALM*	4614 (42.6%)
Llama-3-70b-Instruct	386(3.6%)
+ *PALM*	4155 (38.3%)
Llama-3-8b-Instruct	7(0.1%)
+ *PALM*	3433 (31.7%)

Table 2: Theorems proved by each approach.

PALM proves 180.4%, 136.7%, and 76.6% more theorems, respectively. Since *Passport* and *Proverbot9001* use less powerful LSTM and RNN models, they prove fewer theorems than *PALM* and *DSP* which leverage LLMs. *DSP* underperforms *PALM* due to two reasons. First, we use GPT-3.5 to generate informal proofs needed by *DSP*, but it may introduce errors in the generation process. Second, since *DSP* does not repair errors, any errors in the generated proof, no matter how big or small the errors are, will lead to a proof failure. Compared with *DSP*, *PALM* repairs errors in the proofs generated by the LLM, and performs backtracking to regenerate previous proof steps when hammers fail.

Figure 11 presents a Venn diagram illustrating the theorems proven by each approach. All four approaches can collectively prove 4821 (44.5%) theorems, of which only 444 cannot be proven by *PALM*. The three baselines are able to prove 3616 distinct theorems in total, and *PALM* outperforms their combination by 21.0%. Moreover, *PALM* proves 1270 theorems that none of the other approaches can prove.
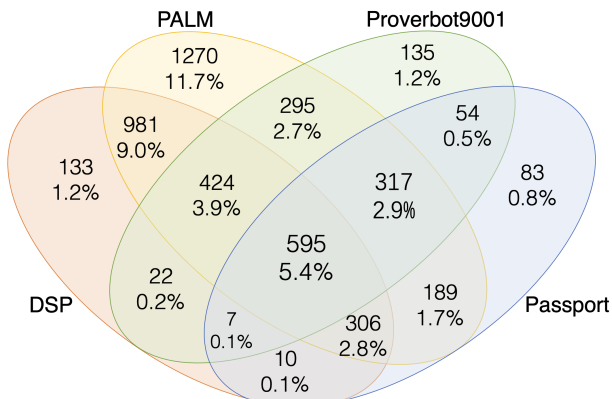

Figure 11: Breakdown of theorems proven by each combination of approaches.

We further analyze the complexity of the theorems that *PALM* proves, using the number of tactics in a proof as a proxy metric for theorem complexity. Figure 12 shows the distribution of theorems that are proven or not proven by *PALM*, categorized by the number of tactics in the ground-truth proofs. The average number of tactics in the ground-truth proofs is 5.84 and the median is 4, suggesting

PALM is more effective with simpler proofs. Moreover, *PALM* can prove 129 theorems that require 20 tactics or more, outperforming *Passport* (11), *Proverbot9001* (30) and *DSP* (48).
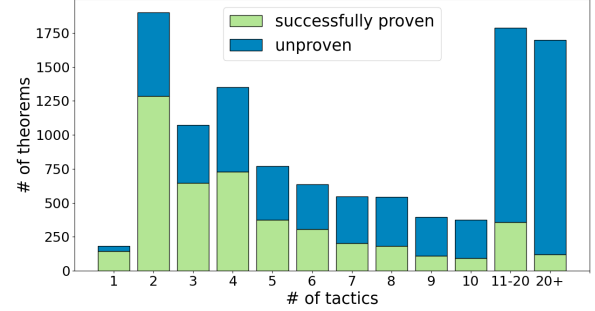
Figure 12: Distribution of theorems that are proven or not proven by *PALM*, categorized by the number of tactics in the ground-truth proofs.

> *Finding 1: PALM is more effective than Passport, Proverbot9001 and DSP on our benchmarks, proving significantly more theorems. Notably, PALM proves 1270 theorems that none of the other approaches can prove. Additionally, PALM can prove a larger number of complex theorems than other approaches.*

5.3.2 RQ2: Generalizability of PALM. To demonstrate the generalizability of *PALM* across LLMs with different parameter sizes, we further evaluate *PALM* with GPT-4o [6], Llama-3-70B-Instruct [5] and Llama-8B-Instruct [5] as the underlying LLMs.

Table 2 presents the theorems proven by each LLM individually, and by *PALM* when using them as underlying LLMs. We observe that all evaluated LLMs perform poorly when used alone, proving only 0.1%-6.4% of theorems. Augmenting these LLMs with *PALM* significantly improves the performance. With the most powerful GPT-4o model, *PALM* proves 4614 theorems, achieving a 5.5% absolute improvement compared with using the second most powerful LLM, GPT-3.5. This highlights the potential enhancements *PALM* can achieve with the latest LLMs. When using Llama-3-70B-Instruct, *PALM* proves 4155 theorems, which is comparable with the result obtained using GPT-3.5. When using the smaller Llama-8B-Instruct, *PALM* proves 3433 theorems, 21.6% fewer than when using GPT-3.5. Despite this, *PALM* still outperforms *DSP* by 38.5%, suggesting it can be effective even when using less powerful LLMs. Using all four LLMs, *PALM* successfully proves a total of 5210 theorems.

> *Finding 2: PALM generalizes to other LLMs of different parameter sizes, and performs better when using larger LLMs.*

5.3.3 RQ3: Effectiveness of each component. We have conducted an ablation study to evaluate the effectiveness of each component within *PALM*.

Effectiveness of the repair mechanisms. To study the effectiveness of each repair mechanism, we constructed four variants of *PALM*: *PALM_{ref}*, *PALM_{rename}*, *PALM_{bullet}*, and *PALM_{aug}*. These variants disable the reference replacement, renaming, bullet

Technique Variant	# of Theorems Proven
$PALM_{\_ref}$	4249 (39.2%)
$PALM_{\_rename}$	4175 (38.5%)
$PALM_{\_bullet}$	4225 (39.0%)
$PALM_{\_aug}$	4094 (37.8%)
$PALM_{\_backtrack}$	702 (6.5%)
$PALM_{\_retriever}$	4147 (38.2%)
PALM (GPT-3.5)	4377 (40.4%)

Table 3: Effectiveness of each *PALM* component.

transformation, and premise augmentation mechanisms, respectively. Table 3 presents the evaluation results of *PALM* and the variants.

Overall, *PALM* consistently proves 3.0%-6.9% more theorems than each variant, indicating the importance of each of our repair mechanisms. Furthermore, all variants continue to prove 65.2%-71.5% more theorems than *DSP*, demonstrating that *PALM* remains effective even when equipped with partial repair mechanisms.

> *Finding 3: Each repair mechanism of PALM contributes to its ability to prove theorems.*

Effectiveness of backtracking. We evaluated the effectiveness of the backtracking procedure (Algorithm 2) by constructing an additional variant, called $PALM_{\_backtrack}$. It does not perform backtracking when it fails to repair an error, and immediately terminates the proof process instead. As shown in Table 3, *PALM* significantly outperforms $PALM_{\_backtrack}$ by 523.5%, indicating that the backtracking procedure is essential to proving many theorems.

> *Finding 4: The backtracking procedure is essential to PALM's effectiveness, enabling it to prove 5× more theorems than only utilizing the repair mechanisms.*

Effectiveness of our premise retriever. To investigate the effectiveness of our premise retriever, we constructed a variant called $PALM_{\_retriever}$, which does not add any premises to the proof generation prompt.

Table 3 shows that *PALM* outperforms $PALM_{\_retriever}$ by 5.5%, which underscores that the premise retriever enables LLMs to produce higher-quality proof scripts.

> *Finding 5: The premise retriever is useful to PALM, helping it to prove 5.5% more theorems.*

5.3.4 RQ4: Efficiency of PALM. On average, *PALM* takes 32.89 seconds to successfully prove a theorem, while *Passport*, *Proverbot9001* and *DSP* require 3.1, 4.7 and 8.2 seconds, respectively. The main source of time overhead for *PALM* is its use of CoqHammer. On average, CoqHammer is invoked 1.96 times per proof, with each invocation having a timeout of 10 seconds. This additional time is justified by *PALM*'s ability to prove more complex theorems than other approaches. We further examined the time each approach takes on all theorems, regardless of whether they were successfully proven or not. On average, *PALM* takes 105.6 seconds, while *Passport*, *Proverbot9001* and *DSP* take 67.2, 31.8 and 20.6 seconds

respectively. Additionally, each successful CoqHammer invocation averages 4.3 seconds, with 79.3% of successful invocations completing in under 5 seconds. This suggests that *PALM* could prove a substantial number of theorems even with a shorter CoqHammer time limit, while a longer limit would potentially benefit more complex proofs.

> *Finding 6: On average, PALM takes longer to prove theorems than other approaches, but this overhead is acceptable given that PALM proves more complex theorems.*

5.4 Case Studies

Despite its effectiveness, *PALM* still fails to prove 59.6% theorems in our dataset. To understand the underlying reasons for these failures, we randomly sampled 100 theorems that *PALM* fails to prove and conducted a manual analysis. Table 4 outlines the 3 primary reasons for these failures.[3] To illustrate these reasons further, we now describe a typical case of failure for each.

Reason	# occurrences
Premises not retrieved	58 (58%)
Premises retrieved but not used	14 (14%)
Tactics not used	39 (39%)

Table 4: The reasons causing *PALM* to fail.

5.4.1 Missing premises. A key reason for *PALM*'s failures (58%) is the omission of necessary premises in the retrieval process. Figure 13 presents a theorem that *PALM* fails to prove because a critical premise, reduceplus_cb1, was not retrieved. Hence this premise cannot be used by the LLM, hindering the proof process.

```
Theorem reducestar_cb1 :
 forall (a : poly A0 eqA ltM) (b : list (Term A n))
   (Q : list (poly A0 eqA ltM)),
 reducestar A A0 A1 eqA invA minusA multA divA eqA_dec n
   ltM ltM_dec Q
   (s2p A A0 eqA n ltM a) b -> CombLinear (a :: Q) b.

 (* Human written proof *)
 intros a b Q H'; inversion H'; auto.
 apply reduceplus_cb1; auto.

 (* LLM generated proof *)
 intros a b Q Hred. induction Hred.
 - constructor. - apply CombLinear_1; auto.
```

Figure 13: A failure case [9] because `reduceplus_cb1` is not retrieved.

5.4.2 Premises retrieved but not used. In 14% of the failures, even when a premise is successfully retrieved and included in the prompt, it may not be used by the LLM. Figure 14 shows a case where the lemmas map_insert and map_map_exchange are included in the prompt, but they are not used by the LLM, causing *PALM*'s failure to prove the theorem. Although providing CoqHammer with unused retrieved premises during the backtracking process could solve such issues, we choose not to do so, as providing too many unrelated premises slows down CoqHammer and can lead it to timeout.

---

[3]The columns in Table 4 sum to more than 100% because a single theorem can fail for multiple reasons.

```
Lemma map_insert_map:
  forall A (f g h : A -> A) x (a : A) e,
  (forall a, f (g a) = g (h a)) ->
  map f (insert x a (map g e)) =
  insert x (f a) (map g (map h e)).

  (* Human written proof *)
  intros. rewrite map_insert. f_equal.
  eapply map_map_exchange. eauto.

  (* LLM generated proof *)
  intros. apply map_insert_eq. apply H.
```

Figure 14: A failure case [10] where the LLM does not use `map_insert` and `map_map_exchange` provided in the prompt.

5.4.3 Tactics not used. Some theorems require specific tactics to be proven, and *PALM* will fail if these tactics are not included in the proof script generated by the LLM. This accounts for 39% of the failure cases. Figure 15 shows an example where the proof of a theorem requires the use of the `induction` tactic. Since the initial proof script did not include this tactic, and both CoqHammer and our repair mechanisms do not perform induction, *PALM* cannot prove this theorem.

```
Lemma last_holder'_unlock_none : forall tr h c,
  last_holder' h tr = Some c ->
  slast_holder' h (tr ++ [(Client c, inl Unlock)])=None.

  (* Human written proof *)
  induction tr; intros; simpl in *; repeat break_match;
      intuition. congruence.

  (* LLM generated proof *)
  intros tr h c i n H1 H2 H3 H4.
  apply (last_holder'_no_out_inv tr h (Client c) n).
  apply H1.
```

Figure 15: A failure case [8] where the LLM does not perform induction.

6 DISCUSSION

6.1 Threats to Validity

Internal validity. One threat to internal validity comes from the inherent randomness of LLMs. This randomness is due to the use of temperature sampling [12, 54] as the decoding strategy, where LLMs randomly select the next token based on a probability distribution. To reduce this threat, we conduct large-scale experiments using three state-of-the-art and widely used LLMs: GPT-3.5, GPT-4o, Llama-3-70B-Instruct, and Llama-3-8B-Instruct, as the underlying LLMs for *PALM*. We evaluate their performance across a benchmark consisting of 10842 theorems from diverse domains. The consistent results observed in our experiments help reduce this threat. Another threat is that due to the limitation of computational resources and evaluation time, we only run each of our experiments once; this may introduce statistical biases into our results.

External validity. The threat to external validity is alone the generalizability of our experimental results. We implement and evaluate only on Coq, while other widely used ITPs, such as Isabelle, HOL Light, and Mizar, are not included. Nonetheless, we believe the approach and algorithm in *PALM* can be easily applied to other ITPs that use tactics for proof construction and support automation tools like CoqHammer. However, the specifics will need to be adapted

for different tactic languages. For example, Isabelle structures subgoals using the 'case' keyword instead of bullets, thus the bullet transformation needs to be modified. We plan to extend *PALM*'s implementation to other ITPs, and evaluate it across more diverse datasets in our future work.

Construct validity. One potential threat to construct validity is that we use the number of tactics in ground-truth proofs as a proxy metric for the theorem complexity. This metric may not accurately reflect the real complexity of a theorem.

6.2 Limitations and Future Work

PALM fundamentally depends on the initial proof script generated by LLMs. If the LLM generates a completely wrong initial proof, *PALM* struggles to fix it. Future improvements to *PALM* could involve leveraging LLMs to repair incorrect proofs [22] or sampling multiple initial proofs.

The initial proof script sometimes fails to use relevant tactics, such as a user-defined tactic with an ambiguous or uninformative name. As a result, *PALM* cannot effectively prove theorems that depend on custom user-defined tactics. This can be improved by adopting more powerful retrievers [49] that learn from the usage patterns of these user-defined tactics.

Finally, we did not spend significant effort optimizing the prompt used by *PALM*, since our focus was not on prompt engineering. Different combinations of instructions or using a more advanced prompting design, such as Chain-of-Thought [45] and Least-to-Most [52] prompting, may improve the performance of *PALM*. These approaches are worth exploring in future work.

7 RELATED WORK

Machine Learning for Formal Verification. There have been various machine learning-based techniques that aim to automatically generate formal proofs for different ITPs. ASTactic [48] is the first deep learning-based proof generation technique for ITPs. It leverages Tree-LSTM [41] to model proof states with all Coq terms parsed into abstract syntax trees (ASTs), and searches for a complete proof via depth-first search (DFS). Many other techniques have been proposed to enhance the performance of ASTactic. TacTok [21], for example, models not only the proof states, but also the incomplete proof scripts to provide more context information. To enlarge the search space, Diva [20] combines multiple models that are trained with different hyperparameters, such as learning rate and embedding size, and different orderings of training data. Passport [39] further extends ASTacic and TacTok by adding new encoding mechanisms for identifiers in proof scripts. These techniques are all evaluated on the CoqGym [48] dataset. Proverbot9001 [38] learns to predict the tactics and arguments using an RNN model and a set of manually engineered features. It also leverages advanced search algorithms such as A-star, and several pruning techniques.

Unlike existing machine learning methods that require significant training, *PALM* leverages LLMs and does not require any training or fine-tuning. Instead of using search strategies, *PALM* employs repair mechanisms and a backtracking procedure to address errors and solve the goals that LLMs fail to prove.

Language Models for Formal Verification. Recently, there has been considerable interest in applying LLMs to formal verification. The

most related work is Draft, Sketch, and Prove (DSP) [27]. Similar to *PALM*, DSP also synergizes LLMs and automated theorem provers. DSP uses LLMs to translate natural language proofs (i.e., informal proofs) into formal proof sketches that outline high-level steps without low-level details. Then, it uses off-the-shelf proof automation tools such as hammers to fill in the gaps. In contrast, *PALM* does not require informal proofs to guide the generation of machine-checked proofs. While DSP reports a failure once proof automation tools cannot fill in a gap, *PALM* employs a backtracking procedure to regenerate previous proof steps when hammer fails. Additionally, *PALM* adopts repair mechanisms to address common errors made by LLMs.

Minerva [33] is a LLM trained on mathematical datasets and achieves state-of-the-art performance on quantitative reasoning tasks. Baldur [22] fine-tunes Minerva to create (1) a *proof generation model* that generates whole proofs given a theorem, and (2) a *proof repair model* that repairs an incorrect proof given the error message. To train the proof generation model, it constructs a dataset by concatenating the proof steps of each theorem from the PISA dataset [25]. The PISA dataset consists of 183K theorems collected from the Isabelle standard library [35] and the Archive of Formal Proofs [1]. To train the proof repair model, it samples from the proof generation model for each theorem in PISA, and records the error messages returned from the ITP for each erroneous proof. The dataset comprises tuples of incorrect proofs, error messages, and correct proofs. Compared with Baldur, *PALM* adopts error-specific repair mechanisms to effectively address the errors. Although Baldur does not perform any search, it needs 64 samples for proof generation and 32 samples for proof repair to achieve high performance, while *PALM* only samples once from LLMs. We have not compared the performance of *PALM* with Baldur because the Minerva model used by Baldur is not open-sourced, and reproducing Baldur by fine-tuning publicly accessible LLMs such as Llama-3 would require extensive computational resources. For instance, the proof generation model of Baldur is fine-tuned on 64 Google TPU v3, with a total of 1024 GB memory. To fine-tune Llama-3-8b with the same settings, over 512 GB GPU memory (around 7 A100s) is required. Furthermore, the proof repair dataset of Baldur consists of 150K tuples of wrong proofs, error messages, and correct proofs in Isabelle. To extend Baldur for Coq, a similar dataset would need to be constructed for Coq.

Thor [26] augments the PISA dataset [25] by invoking SledgeHammer [36] in each step of the proofs in the dataset, and adding successful invocations of SledgeHammer to the dataset. Thor trains a decoder-only transformer model (700M parameters) on this enhanced dataset. This model is designed to learn when to invoke hammers during a proof, and guide a search process. Unlike Thor, which does not perform premise retrieval, *PALM* adopts a premise retriever to enhance the performance of LLMs. Instead of performing a computationally expensive proof search, *PALM* leverages LLMs to produce well-structured initial proofs and adopts repair mechanisms to fix common errors. *PALM* is not directly comparable with Thor, because Thor's model is specifically trained for Isabelle proofs rather than Coq, which cannot be reproduced on Coq with reasonable effort.

Copra [42] uses the state-of-the-art GPT-4 model to guide a depth-first search process. In each step, GPT-4 is prompted with

the proof state, previous proof steps, the incorrect steps, and the corresponding error messages to avoid recurrent errors. Copra can be further augmented by incorporating premise retrieval and generating informal proofs from informal theorem statements if they exist. However, Copra's effectiveness is highly dependent on the capability of the LLMs it uses. For example, Copra proves 26.63% theorem in the miniF2F dataset [51] using GPT-4, but only proves 9.02% using GPT-3.5. Moreover, Copra does not directly repair incorrect tactics and instead prompts the LLM with incorrect tactics and error messages. In contrast, *PALM* adopts a set of symbolic-based repair mechanisms to correct erroneous tactics effectively, and demonstrates consistent performance across LLMs.

8 CONCLUSION

Large Language Models (LLMs) have shown promise in automatically generating informal proofs in natural language, but these systems have proven to be less effective at generating formal proofs in interactive theorem provers (ITPs). This paper described a formative study that identified common errors made by GPT-3.5 when generating machine-checked proofs. Guided by these findings, we proposed *PALM*, which combines LLMs and symbolic methods to automatically prove theorems in an ITP. *PALM* adopts a premise retriever to select relevant premises such as lemmas and definitions, in order to enhance the quality of proofs generated by an LLMs. It additionally uses a set of repair mechanisms and a backtracking algorithm to correct errors in proof scripts generated by an LLM. We evaluated *PALM* on a dataset of 10842 theorems. In the evaluation, *PALM* significantly outperforms existing approaches, and demonstrates its generalizability across different LLMs. Furthermore, our ablation study suggests that all components of *PALM* are effective.

ACKNOWLEDGEMENTS

REFERENCES
[1] 2024. Archive of Formal Proofs. https://www.isa-afp.org/index.html.
[2] 2024. Changelogs of Coq. https://coq.inria.fr/doc/V8.12.0/refman/changes.html.
[3] 2024. Eprover. http://www.eprover.org.
[4] 2024. GPT-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.
[5] 2024. GPT-3.5-turbo. https://llama.meta.com/llama3.
[6] 2024. GPT-4o. https://openai.com/index/hello-gpt-4o.
[7] 2024. PALM's source code. https://github.com/lachinygair/PALM.
[8] 2024. A failure case because inductive reasoning is not used. https://github.com/uwplse/verdi/blob/b7f77848819878b1faf0e2e6a730f9bb850130be/theories/Systems/LiveLockServ.v#L1112.
[9] 2024. A failure case because key premises are not retrieved. https://github.com/coq-community/buchberger/blob/92f377ac39c0aec3e6ef77d4c2b26318990e2145/theories/Pcomb.v#L703.
[10] 2024. A failure case because the LLM does not use premises. https://github.com/coq-community/dblib/blob/master/src/Environments.v#L550.
[11] 2024. BM25, wikipedia. https://en.wikipedia.org/wiki/Okapi_BM25.
[12] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. 1985. A learning algorithm for Boltzmann machines. *Cognitive science* 9, 1 (1985), 147–169.
[13] Alexander A. Alemi, François Chollet, Niklas Een, Geoffrey Irving, Christian Szegedy, and Josef Urban. 2016. DeepMath - deep sequence models for premise selection. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) (NIPS'16). Curran Associates Inc., Red Hook, NY, USA, 2243–2251.
[14] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare

Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[15] źUkasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reason.* 61, 1–4 (jun 2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

[16] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[17] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388.

[18] Sahibsingh A Dudani. 1976. The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics* 4 (1976), 325–327.

[19] Michael Färber and Cezary Kaliszyk. 2015. Random forests for premise selection. In *International Symposium on Frontiers of Combining Systems*. Springer, 325–340.

[20] Emily First and Yuriy Brun. 2022. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering*. 749–761.

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

[22] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1229–1241.

[23] Barney Glaser and Anselm Strauss. 2017. *Discovery of grounded theory: Strategies for qualitative research*. Routledge.

[24] Beverley Hancock, Elizabeth Ockleford, Kate Windridge, et al. 2001. *An introduction to qualitative research*. Trent focus group London.

[25] 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.

[26] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems* 35 (2022), 8360–8373.

[27] Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2022. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283* (2022).

[28] Cezary Kaliszyk and Josef Urban. 2015. HOL (y) Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science* 9 (2015), 5–22.

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

[30] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. 2012. Overview and evaluation of premise selection techniques for large theory mathematics. In *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings 6*. Springer, 378–392.

[31] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE. http://xavierleroy.org/publi/erts2018_compcert.pdf

[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[33] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022. Solving quantitative reasoning problems with language models, 2022. *URL https://arxiv. org/abs/2206.14858* (2022).

[34] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg.

[35] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.

[36] Lawrence C. Paulson and Jasmin Christian Blanchette. 2012. Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive

[37] Theorem Provers. In *IWIL 2010. The 8th International Workshop on the Implementation of Logics (EPiC Series in Computing, Vol. 2)*, Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska (Eds.). EasyChair, 1–11. https://doi.org/10.29007/36dt

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

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

[39] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving automated formal verification using identifiers. *ACM Transactions on Programming Languages and Systems* 45, 2 (2023), 1–30.

[40] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[41] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[42] Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. *arXiv preprint arXiv:2310.04353* (2023).

[43] The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

[44] The Coq Development Team. 2024. Programmable proof search – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman/proofs/automatic-tactics/auto.html.

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[46] 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 (jun 2015), 357–368. https://doi.org/10.1145/2813885.2737958

[47] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in Neural Information Processing Systems* 35 (2022), 32353–32368.

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

[49] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2024. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems* 36 (2024).

[50] Shizhuo Dylan Zhang, Talia Ringer, and Emily First. 2023. Getting More out of Large Language Models for Proofs. *arXiv preprint arXiv:2305.04369* (2023).

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

[52] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).

[53] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).

[54] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.