GUIDED PROOF SEARCH USING LARGE LANGUAGE MODELS AND LEMMA EXTRACTION IN COQ

Tarun Prasad
Harvard University
m.tarunprasad@gmail.com

Nada Amin
Harvard University
namin@seas.harvard.edu

ABSTRACT

Interactive theorem provers are powerful tools for formalizing mathematics and verifying the correctness of software. However, they require significant background and effort to use due to the tedious nature of writing formal proofs and have not seen widespread adoption. Recent advances in machine learning have enabled formal proofs to be auto-generated; most existing approaches that use language models, however, have focused on generating proofs step-by-step and using them in conjunction with an expensive search algorithm to find proofs. In 2023, First et al. introduced Baldur, a system that uses language models to generate entire proofs at once for theorems in the Isabelle proof assistant. Our work studies the feasibility of a similar whole-proof generation procedure for Coq and introduces a novel approach to automated theorem proving that recursively extracts lemmas at failure points in the proof generation process, allowing the system to break complex theorems down into simpler subproblems. We evaluate these approaches on a dataset of 724 theorems from the Software Foundations textbook and show that GPT-4 can generate whole-proofs for 66.44% of the theorems. Additionally, when augmented with our lemma extraction method, GPT-4 sees a 19.54% improvement to achieve a success rate of 79.42%, thus marginally outperforming CoqHammer—a state-of-the-art automated reasoning tool—which proves 78.73% of the theorems. We also evaluate the much smaller open-source model Phind CodeLlama, which depicts a 103.23% improvement over its baseline when utilizing lemma extraction. We release our Coq playground that contains an implementation of this procedure along with the dataset and evaluation results through an open-source repository to encourage further research in this area.

1 INTRODUCTION

Formal logic studies the rules of correct reasoning and provides a symbolic framework for expressing and proving theorems. Traditionally, proofs have been written by mathematicians and computer scientists in natural language using high-level abstractions, with the emphasis being on communicating the major ideas as opposed to fleshing out every minor detail. *Formal* proofs differ from these in that the emphasis is on precision, rigor, and verifiability. As a consequence, formal proofs can be checked for correctness using a computer, usually through an *interactive theorem prover* or *proof assistant* such as Coq (The Coq Development Team, 2023), Lean (de Moura et al., 2015), Isabelle (Nipkow et al., 2002), or Metamath (Megill & Wheeler, 2019).

However, the process of finding these proofs continues to be an unsolved problem in artificial intelligence. While it is straightforward to verify a formal proof once it has been written in a formal language, *automated theorem proving*, the process of searching for or generating these proofs automatically, is computationally intractable. Nevertheless, there have been various attempts to automate the theorem proving process. One major class of automated theorem provers are based on satisfiability modulo theories (SMT) solvers, which use search algorithms in conjunction with heuristics and time-outs to attempt to find a proof in a reasonable amount of time. While these solvers are effective for many simple problems and are often used to automate the verification of small steps within a larger proof, they are not well-suited for the task of finding complex proofs from scratch due to the vast search space of possible proofs.

More recently, machine learning has been used to guide the search for formal proofs. While large language models (LLMs) have been shown to be effective at generating some simple proofs, these approaches are still in their infancy and require further research before they can be used in practice. Indeed, there tend to be frequent logical errors and hallucinations even in natural language responses to prompts, so these errors are likely to be even more common in the context of formal logic.

Automating parts of the formal proof writing process will be highly significant for not only those who currently use proof assistants in their work—mathematicians trying to formalize and verify their theorems and proofs, and engineers writing highly precise and critical code for processors, compilers, or operating systems—but also those who would benefit by their use but don't do so due to their steep learning curve and tediousness. Software developers, for example, commonly evaluate their code at runtime against a unit test suite that is often not fully comprehensive and lets bugs go unnoticed. Advancements in automated formal verification could make this unit testing process unnecessary by simply proving through static analysis that their code matches the required specifications, without them having to write the cumbersome proofs themselves.

Our contributions in this paper are two-fold.

- We study whole-proof generation using large language models for the Coq proof assistant, quantifying the fraction of theorems in our dataset provable with only a single prompt.
- We introduce a novel approach to automated theorem proving that repairs incorrectly generated proofs by recursively extracting lemmas at failure points. This method enables breaking complex theorems down into simpler subproblems that are then easier to prove using a whole-proof approach. We then evaluate this approach against existing proof automation tools and also demonstrate significant gains over single-step whole-proof generation.

1.1 Background

Interactive theorem provers, also known as proof assistants, are software tools that provide an implementation of a formal system and allow users to construct and then verify formal proofs by expressing them as computer programs. Coq (The Coq Development Team, 2023) is a theorem prover that was first released in 1989 and is used in both academia and industry for formal verification of software and hardware systems, as well as for formalizing mathematics and proving theorems.

Proofs in Coq are commonly written using a sequence of tactics, which are commands that are used to manipulate the proof state of a proof until it is complete. The proof state is simply a list of *goals* that, if proven in their entirety, will complete the proof. Each goal consists of a *conclusion* (the proposition that needs to be proven) and a *local context* containing local definitions, variables, and hypotheses that can be used in the proof. Depending on the tactic, it may be necessary to pass one or more arguments to it; these arguments may include terms containing constants, definitions from the local context, as well as declarations in the *global environment* that may include definitions and previously-proven theorems and lemmas in the file or in an imported file.

Consider the following Coq proof of the closed-form expression for the sum of the first n natural numbers, a prototypical example of a proof by induction.

```
Require Import ArithRing.

Fixpoint sum_n (n : nat) : nat :=
  match n with
  | 0 => 0
  | S n' => n + sum_n n'
  end.

Lemma sum_n_formula_helper : forall n : nat,
  2 * sum_n (S n) = 2 * (S n) + 2 * sum_n n.
Proof.
  intro n. simpl. ring.
Qed.

Theorem sum_n_formula : forall n : nat, 2 * sum_n n = n * (n + 1).
```

```
Proof.
  intro n. induction n as [|n' IH].
  - simpl. reflexivity.
  - rewrite -> sum_n_formula_helper. rewrite -> IH. ring.
Qed.
```

The proof of the main theorem `sum_n_formula` begins by introducing the natural number `n` and using the `induction` tactic to split the proof into two cases: the base case of `n = 0` (easily proven by simplifying to `0 = 0` and then using the reflexivity of equality), and the inductive step. This second case uses a helper lemma in addition to the inductive hypothesis.

2 Methods

Large language models have had much success in recent years at understanding and generating code in various programming languages (Chen et al., 2021). The formal languages used by interactive theorem provers, however, differ from these in a few different ways, not least of which is the limited availability of human-written programs in these languages due to the significantly smaller communities and the tediousness of writing formal proofs by hand. Nevertheless, LLMs have shown promise in theorem proving and there has been much research in the last few years on other ways of leveraging them for this purpose, beyond training and fine-tuning.

As discussed in section 4, recent attempts at using LLMs have used one of two approaches: either a step-by-step proof generation process that generates one proof step at a time to grow a search tree that is used in conjunction with an expensive search algorithm to find a complete proof, or a more direct approach that generates the entire proof at once. We will focus on the latter approach which was used in Baldur (First et al., 2023), a recent work that will serve as the basis for our own investigations.

The authors of Baldur focus on the Isabelle proof assistant, where proofs can be written using a declarative proof language called Isar that includes the intermediate proof states within the proof itself, hence allowing it to be human-readable. This makes it more amenable for language models to generate proofs in a manner akin to chain-of-thought as the output of the model includes the proof states at each step, allowing the transformer to consider these when generating the next step. In contrast, Coq uses a procedural proof language that is not human-readable and requires the user to step through the series of tactics in the proof one by one using the proof assistant's verifier to view the intermediate proof states and decide what tactic to use next. This makes whole-proof generation in Coq a possibly more challenging task for language models and motivates our study of their effectiveness in this setting (see appendix A for a more detailed comparison).

Given the likely difficulty of generating whole proofs in Coq, we also propose a method of breaking theorems down into smaller lemmas with the aim of improving the chances of finding a correct proof. This is motivated by our hypothesis that large language models will be able to reason about smaller, simpler theorems with fewer proof steps more accurately than complex theorems with highly involved proofs. Now how do we decide what lemmas to extract for a given theorem? Previous works adjacent to ours have used data-driven lemma synthesis or large language models to generate proof sketches as a series of assertions or "local lemmas" based on an informal proof draft. See section 4 for a discussion of these methods.

In this work, we propose simply generating a whole-proof, attempting to verify its correctness, and then extracting a lemma that would complete the goal at the point of the first error encountered during the verification. This technique is motivated by our hypothesis that LLMs are likely to make mistakes in the fine-grained details of a proof—hallucinations of identifiers or inapplicable tactics, for instance—when generating whole proofs, but are more likely to get the general proof structure right. To the best of our knowledge, this is the first work to explore this method.

To summarize, we ask the following research questions:

1. **How effective are large language models at generating whole proofs of theorems in Coq?**
2. **Can their performance on the whole-proof generation task be improved by recursively extracting lemmas at failure points during proof generation?**

2.1 Lemma Extraction Algorithm

Given the statement of a theorem and a global environment, our proposed method of proof search proceeds as follows.

1. Generate a whole-proof attempt of the theorem using a large language model. In addition to the theorem statement itself, the model is also given access to the statements of the definitions and theorems in the global environment as part of the prompt to allow it to use these in its proof. We do not include proofs of these definitions and theorems to avoid exceeding the context window size of the model.
2. Attempt to verify the proof using the Coq verifier. If the proof is correct, we are done. If an error is encountered, proceed to the next step.
3. Using an annotated version of the sentences in the proof attempt, identify the index i_{err} of the first sentence that caused the error.
4. Extract the first goal of the proof state at the sentence with index $i_{err} - 1$.
5. Construct a lemma that uses the variables and hypotheses from the extracted goal as premises and the goal's conclusion as its conclusion.
6. Recursively generate a whole-proof attempt for the lemma with the same global environment as the original theorem, recursing until either a successful proof of the lemma is found or a maximum depth is reached.
7. Replace the portion of the original theorem's proof that attempted to prove the extracted goal with an application of the lemma (i.e. using the `apply` tactic). Also modify the global environment of the theorem to include the lemma.
8. Repeat the process for the next error encountered, if any.

2.2 An Example

To illustrate the lemma extraction algorithm, consider the following example based on the induction proof in Coq discussed in section 1.1. Suppose we have previously proven the lemma `sum_n_formula_helper`, so the global environment so far includes this lemma in addition to the definition `sum_n` that recursively computes the sum of the first n natural numbers. Also suppose we provide the entirety of this global environment to a large language model that then generates the following proof attempt.

```
Theorem sum_n_formula : forall n : nat, 2 * sum_n n = n * (n + 1).
Proof.
  induction n as [| n' IHn'].
  - simpl. reflexivity.
  - ring.
Qed.
```

This proof is valid up to the base case. At the beginning of the inductive step, the proof state looks as follows.

```
1 goal
  n' : nat
  IHn' : 2 * sum_n n' = n' * (n' + 1)
  ============================
  2 * sum_n (S n') = S n' * (S n' + 1)
```

The next proof step as output by the model is the `ring` tactic, which produces an error.

```
Tactic failure: not a valid ring equation.
```

Indeed, the `ring` tactic is not applicable here as the goal's conclusion depends on the `sum_n` function and is not simply derivable from the axioms of a ring or semi-ring. The inductive hypothesis is needed to complete the proof, and the `sum_n_formula_helper` lemma is also useful. We extract the hypotheses and conclusion of the goal at the point just before the error (shown above), and use them to construct the following lemma.

```
  Lemma extracted_lemma : forall n' : nat,
    forall IHn' : 2 * sum_n n' = n' * (n' + 1),
    2 * sum_n (S n') = S n' * (S n' + 1).
```

Now we recursively generate a proof attempt for `extracted_lemma` using the same global environment as the original theorem. Suppose the model generates the following correct proof.

```
Proof.
  intro n'. intro IHn'.
  rewrite sum_n_formula_helper. rewrite IHn'.
  ring.
Qed.
```

Then, it suffices to inject this new lemma along with its proof into the proof script before the definition of `sum_n_formula`, and then replace the portion of the original proof within the inductive step with a single `apply` tactic taking the lemma as an argument. This completes the proof of `sum_n_formula` as shown below.

```
Theorem sum_n_formula : forall n : nat, 2 * sum_n n = n * (n + 1).
Proof.
  induction n as [| n' IHn'].
  - simpl. reflexivity.
  - apply (@extracted_lemma n' IHn').
Qed.
```

In this example, the model correctly identifies that the original theorem requires a proof by induction and also correctly proves the simple base case. The proof of the inductive step, however, results in an error in the form of a failed tactic. Extracting a lemma for this inductive step allows the proof state (i.e. the subgoal containing the hypotheses and conclusion) to be passed to the model as a "reminder" of what needs to be proven, allowing it to focus on the details of that subgoal while having access to the local context. This is in contrast to the original whole-proof attempt, where the model had to generate tactics with no guidance as to what remained to be proven after each step.

2.3 Implementation

Our Coq playground is available as an open-source repository[1]. The implementation is written in Python and uses Alectryon (Pit-Claudel, 2020), a Python interface for Coq, to interact with the verifier. The repository also contains the Coq source files from which the dataset used for evaluation is constructed (see section 3.2) as well as the generated proof scripts and logs. Successful proofs are given the filename `thm{thm_idx}_{thm_name}.v` while failed proofs are named `thm{thm_idx}_err_{thm_name}.v`.

Although our experiments use specific models as discussed in section 3.1, the playground is designed to be model-agnostic and can be used with any large language model, including GPT models in OpenAI's suite, pre-trained models hosted inside a model repository on HuggingFace, as well as locally trained or fine-tuned models.

More detailed instructions on how to use the playground to prove a specific theorem or run experiments are available in the README of the repository.

3 Evaluation

To evaluate our lemma extraction approach, we compare the performance of a large language model used in conjunction with lemma extraction to that of a single whole-proof approach as used in Baldur. We further report baseline results of existing automation tools in Coq, specifically the `auto`

[1] `https://github.com/mtarunpr/coq-prover`

and `intuition` tactics as well as the `hammer` tactic from the state-of-the-art CoqHammer plugin (Czajka & Kaliszyk, 2018). We use `hammer` with the CVC4 (Barrett et al., 2011) automated theorem prover and the default time limit of 20 seconds. We consider a proof to be valid if it is successfully verified by the Coq proof assistant and does not contain sentences that allow it to exit a proof without completing it, such as `Admitted` or `Abort`.

3.1 MODELS

We run experiments using two large language models: GPT-4, specifically the `gpt-4-1106-preview` model, and the open-source model `Phind-CodeLlama-34B-v2`[2]. The former is a general purpose foundational model available through the OpenAI API, while the latter is a much smaller 34B parameter model that is a fine-tuned and instruction-following variant of CodeLlama (Roziere et al., 2023), which itself is a variant of the foundational model Llama 2 (Touvron et al., 2023) that has been specifically fine-tuned to perform well on code generation tasks. At the time of writing, Phind CodeLlama is the state-of-the-art among open-source models for code generation tasks and achieves a performance of 73.8% pass@1 on the HumanEval benchmark (Chen et al., 2021). We focus our investigations on these base models and do not further fine-tune them on Coq-specific data.

It is worth noting that the pre-training data for these models may have been contaminated with some or all of the theorems in our dataset, and this should be kept in mind when interpreting results. We do not believe this to be a significant issue, however, since this work focuses on improvements in performance when using our lemma extraction approach, as opposed to merely the base performance of these models on this dataset.

3.2 DATASET

We use the theorems and exercises from the Software Foundations, Volume 1: Logical Foundations textbook (Pierce et al., 2023), sourced from a public GitHub repository[3] available under the MIT License, as our evaluation dataset. This dataset consists of 724 theorems and exercises in Coq, which we use to evaluate the whole-proof generation success rate of the base models, as well as the performance of our theorem proving approach and the automation baselines. These theorems are split across 17 source files—each corresponding to one chapter—spanning a variety of topics in logic and functional programming, including induction, lists, maps, polymorphism, and parsing.

A copy of this dataset is included in our repository under the `data/raw/software_foundations` directory. The generated proofs for the theorems are available in the `proofs_gpt4` and `proofs_phind` subdirectories.

3.3 RESULTS

Table 1: Performance of LLMs with lemma extraction compared to baseline methods

Method	Count	%	Improvement %
GPT-4 + Lemma Extraction	575	79.42%	19.54%
GPT-4 Baseline	481	66.44%	
Phind-CodeLlama + Lemma Extraction	189	26.10%	103.23%
Phind-CodeLlama Baseline	93	12.84%	
`auto`	260	35.91%	
`intuition`	331	45.72%	
`hammer`	570	78.73%	

Table 1 shows a comparison of the performance of the lemma extraction approach on the 724 theorems in the dataset using each of the two language models, measured as the percentage of theorems in the dataset successfully proven, against various baseline methods. The improvement percentage

[2] `https://huggingface.co/Phind/Phind-CodeLlama-34B-v2`

[3] `https://github.com/marshall-lee/software_foundations`

is calculated as the percentage increase in the number of theorems proven by a given language model when using lemma extraction compared to the same model without.

The results show, firstly, that whole-proof generation is a promising approach for theorem proving in Coq, with GPT-4 achieving a success rate of 66.44% even without any fine-tuning. This is significantly higher than the 35.91% and 45.72% achieved by `auto` and `intuition` respectively, but lower than the 78.73% achieved by `hammer`.

The Phind CodeLlama model, however, performs remarkably more poorly than GPT-4, with a success rate of only 12.84%. Indeed, even the automation tools built into Coq—`auto` and `intuition`—are able to prove a larger fraction of the theorems in the dataset than Phind CodeLLama, with or without lemma extraction. This is despite such proofs being solvable using a single tactic, indicating that the language model is unable to identify theorems that are simple enough to be provable using these tactics; instead, the proofs output by the model attempt to prove the theorem from scratch, without the help of automation. This is likely due to a combination of the small parameter size of the model and the limited quantity of Coq proofs in its pre-training data.

Next, we look at the performance of both models when augmented with the lemma extraction approach. GPT-4 with lemma extraction is able to prove 79.42% of the theorems in the dataset, a 19.54% improvement over its baseline performance. Notably, this improved success rate even slightly exceeds the performance of the `hammer` tactic. In the case of Phind CodeLlama, the success rate more than doubles in the presence of lemma extraction: successful proofs were found for 26.10% of the theorems in the dataset, i.e. a 103.23% improvement over the whole-proof method.

These results have several implications on possibilities for the future of automated theorem provers. Our results suggest that lemma extraction is reasonably effective for large foundational models, but particularly effective for smaller models like Phind CodeLlama. Despite the latter's poor performance on the whole-proof task, the lemma extraction approach is able to significantly improve its success rate, even though the same model is used to prove the extracted lemmas with no additional step-by-step guidance. If such techniques were to be integrated into plugins for Coq, models at the scale of GPT-4 may be far too large to run locally, but there may be possibilities for smaller open-source models like Phind CodeLlama to be used as tools to assist in proof writing. Having said that, further research on fine-tuned versions of these open-source models is needed to determine if they can outperform existing automation tools with the help of lemma extraction.

To conclude, we answer the research questions posed at the beginning of the section.

1. **How effective are large language models at generating whole proofs of theorems in Coq?**
   Large language models have shown to be reasonably effective at generating whole proofs of theorems in Coq, with GPT-4 achieving a success rate of 66.44% on the dataset and outperforming built-in automation tools like `auto` and `intuition` but not powerful external reasoning tools such as CoqHammer. Smaller models, however, seem to perform much worse without fine-tuning on Coq proofs: Phind CodeLlama, for instance, has a success rate of only 12.84% and is outperformed by even simple automation tools like `auto`.
2. **Can their performance on the whole-proof generation task be improved by recursively extracting lemmas at failure points during proof generation?**
   Yes, there is evidence to show that the performance of large language models can be significantly improved by extracting lemmas at failure points during proof generation and recursively generating whole proofs for these lemmas. GPT-4 solves 79.42% of the theorems in the dataset, thus marginally exceeding the success rate of even the CoqHammer tool. Phind CodeLlama on the other hand solves 26.10% using this approach, representing a remarkable 103.23% improvement over its baseline performance.

4 RELATED WORK

Early attempts at using neural networks for theorem proving predate the advent of LLMs. In 2019, Huang et al. introduced GamePad, an RNN-based learning environment for theorem proving in Coq that involves two components: position evaluation and tactic evaluation. CoqGym, a dataset and learning environment for theorem proving in Coq consisting of 71K human-written proofs, was published soon after (Yang & Deng, 2019) along with ASTactic, a neural theorem prover that proves theorems by interacting with the Coq proof assistant, using an encoder-decoder architecture.

These were followed by several investigations into the use of large language models for step-by-step proof search including GPT-f (Polu & Sutskever, 2020) and HyperTree Proof Search (HTPS) Lample et al. (2022), as well as approaches involving diversity-driven theorem proving that aim to integrate language models with hammers or combine the strengths of multiple methods (Jiang et al., 2022; First & Brun, 2022). Sivaraman et al. (2022) propose a method for synthesizing lemmas that may be useful in proving a given theorem, and then evaluate their approach with the help of the existing Proverbot9001 (Sanchez-Stern et al., 2020) that uses neural methods for theorem proving: this work treats lemma synthesis as a data-driven program synthesis problem, where examples generated from the current proof state serve as the data.

In a significant departure from previous attempts at using search techniques that incorporate language models as only one component of the larger search algorithm, Baldur (First et al., 2023) attempts to directly generate whole proofs at once instead of one step at a time for the Isabelle/HOL theorem prover. In this work, the authors study the feasibility of whole-proof generation along with proof repair using a transformer-based model. While Baldur is designed to work with any language model, the authors use Minerva (Lewkowycz et al., 2022) to run their experiments, which is pre-trained on a mathematics corpus. More recent attempts at whole-proof generation have involved training on synthetic data (Xin et al., 2024), using a continually growing library of verified lemmas (Wang et al., 2023), as well as iterative proof-repair using simple methods like undefined reference replacement and renaming (PALM) (Lu et al., 2024). While our work builds off existing whole-proof and proof-repair attempts such as Baldur and PALM, it differs in that it introduces the novel approach of lemma extraction as a proof-repair tool.

LeanDojo (Yang et al., 2024) is a Lean playground that contains toolkits, data, and benchmarks for theorem proving in Lean released in conjunction with ReProver, a retrieval-augmented prover that fetches relevant premises from Lean's library to assist the model in proving theorems. Draft, Sketch, and Prove (DSP) (Jiang et al., 2023) is an approach that proposes a method to formalize entire proofs of theorems using three steps: generate a draft of a proof in natural language, convert this to a formal proof sketch, and then fill in the gaps using an off-the-shelf automated theorem prover like Z3.

Besides the major works discussed above, there have been several other attempts at using machine learning for related problems: some have focused on using reinforcement learning and MCTS-based approaches for formal proving and code generation (Gauthier et al., 2021; Wu et al., 2021; Dou et al., 2024; Brandfonbrener et al., 2024), while others have focused on Olympiad geometry problems (Trinh et al., 2024) as well as natural-language approaches to math problems (Welleck et al., 2021; Lightman et al., 2024).

5 Conclusion

In this paper, we introduced, investigated, and released an implementation of a novel approach to automated theorem proving in Coq that makes use of whole-proof generation in conjunction with lemma extraction at failure points to come up with proofs of theorems. This work differs from most previous approaches described in section 4 in that it does not use expensive search-based methods to find proofs, nor does it generate tactics one step at a time. Similar to Baldur (First et al., 2023), it generates whole proofs at once, but it does so in a guided manner by "reminding" the language model of what needs to be shown at each error point through the use of lemmas, which are then proven in the same way.

There are several directions in which this work can be extended. Lemma extraction can be used in conjunction with retrieval augmented generation to retrieve relevant premises and include them in the prompt, in a manner similar to LeanDojo (Yang et al., 2024), to overcome context size limitations. Additionally, lemmas that are likely to be useful in the proof can be identified even before the proof is generated. Using the `lfind` tactic in Coq, for instance, in conjunction with whole-proof generation is one possible direction for research (see data-driven lemma synthesis discussed in section 4). Instead of explicitly extracting lemmas, one can also use proof sketches as was done in DSP (Jiang et al., 2023); in Coq, such a sketch could be created as a sequence of `assert` tactics that act as the lemmas.

Acknowledgments

We would like to acknowledge the Harvard SEAS Computing group for help with access to compute resources. We also thank the anonymous reviewers for their valuable feedback. The first author was supported in part by the Office of Undergraduate Research and Fellowships through the Harvard College Research Program (HCRP) in Fall 2023.

References

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer (eds.), *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pp. 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL https://doi.org/10.1007/978-3-642-22110-1_14.

David Brandfonbrener, Simon Henniger, Sibi Raja, Tarun Prasad, Chloe Loughridge, Federico Cassano, Sabrina Ruixin Hu, Jianang Yang, William E Byrd, Robert Zinkov, et al. Vermcts: Synthesizing multi-step programs using a verifier, a large language model, and tree search. *arXiv preprint arXiv:2402.08147*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

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

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pp. 378–388. Springer, 2015.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

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

Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 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*, pp. 1229–1241, 2023.

Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: learning to prove with tactics. *Journal of Automated Reasoning*, 65:257–286, 2021.

Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=r1xwKoR9Y7.

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

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=SMa9EAovKMC`.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in Neural Information Processing Systems*, 35:26337–26349, 2022.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 2022.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=v8L0pN6EOi`.

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

Norman Megill and David A Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. URL `http://us.metamath.org/downloads/metamath.pdf`.

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2023. URL `https://softwarefoundations.cis.upenn.edu`. Version 6.5.

Clément Pit-Claudel. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, pp. 155–174, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381765. doi: 10.1145/3426425.3426940. URL `https://pit-claudel.fr/clement/papers/alectryon-SLE20.pdf`.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

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

Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. Data-driven lemma synthesis for interactive proofs. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):505–531, 2022.

The Coq Development Team. The Coq proof assistant, July 2023. URL `https://doi.org/10.5281/zenodo.8161141`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. NaturalProofs: Mathematical theorem proving in natural language. 2021.

Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. TacticZero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342, 2021.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

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

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

A COQ VS ISABELLE

Proofs written using Isar in Isabelle include the intermediate proof states within the proof itself, for instance in the `have` and the `assume` steps, making them more human-readable albeit more verbose. Take, for instance, the following proof of a simple theorem from propositional logic written in Isabelle.

```
theorem distribute:
  "A /\ (B \/ C) ==> (A /\ B) \/ (A /\ C)"
proof -
  assume "A /\ (B \/ C)"
  then have A: "A" and BC: "B \/ C"
    by auto
  {
    assume "B"
    with A have "A /\ B" by simp
    then have "(A /\ B) \/ (A /\ C)"
      by simp
  }
  moreover
  {
    assume "C"
    with A have "A /\ C" by simp
    then have "(A /\ B) \/ (A /\ C)"
      by simp
  }
  ultimately show "(A /\ B) \/ (A /\ C)"
    using BC by blast
qed
```

The structure of the Isabelle proof is not too dissimilar from its structure in natural language: the proof begins by assuming the premise and then proceeds to name the hypotheses before splitting up into two cases. Each case then involves explicitly writing out the current proof state (such as `have "A /\ C"` and `then have "(A /\ B) \/ (A /\ C)"`) at every step.

In Coq, the structure remains similar, except that the proof state is not explicitly written out at each step; instead the interpreter maintains the state as it verifies each step of the proof. This makes it harder for a human reader as well as for a language model to keep track of things like which variable name corresponds to which hypothesis, what the current proof state is, and what remains to be shown.

```coq
Variables A B C : Prop.

Theorem distribute : A /\ (B \/ C)
  -> (A /\ B) \/ (A /\ C).
Proof.
  intro H.
  destruct H as [H1 H2].
  destruct H2 as [H2 | H3].
  - left. split.
    + exact H1.
    + exact H2.
  - right. split.
    + exact H1.
    + exact H3.
Qed.
```

B Prompts Used

We provide the prompt templates used in the experiments in this paper; these are also available in the Coq playground.

Each time a theorem or lemma needs to proven, we make a request to the model to generate a proof, without retaining context from previous requests to avoid bias and repetition of the same errors. We use a system prompt along with a user message.

System message:

~~~
You are an automated theorem prover that can prove theorems and
lemmas in Coq. Your entire response must be valid Coq code. You
should explain your reasoning (what the proof steps are
attempting to do), but only in comments inside the Coq code. The
following messages will all consist of a theorem statement
(possibly preceded by necessary definitions, imports, etc.), and
your response must be a valid Coq proof of that theorem. Your
response must be in this format: ```coq
Proof.
<proof>. Qed.
```. Remember: do not add any other text besides Coq code and do
not repeat any imports, definitions, lemmas, etc. provided in the
prompt.
~~~

User message:

```
#### CONTEXT

{global environment}

#### THEOREM TO BE PROVEN
```

```
{theorem statement}
```

We also use the language model to generate names for the extracted lemmas, which are then suffixed with an index to ensure uniqueness.

System message:

```
You are a proof helper for Coq that can come up with descriptive
names for lemmas and theorems based on the statement of the
proposition. Specifically, Replace `helper_lemma` with a better,
more descriptive, name for the following lemma(s) in Coq. Your
entire response must be valid Coq code. Your response must be in
this format: ```coq\nLemma <new_lemma_name> :
<lemma_statement>.\n```.
```

User message:

```
{lemma statement}
```

Note that the OpenAI API for GPT-4 accepts the system and user messages as separate inputs. For Phind CodeLlama, we use the following template, in accordance with the Alpaca/Vicuna instruction format as specified in the model's documentation.

```
### System Prompt
{system message}

### User Message
{user message}

### Assistant
```

C EXAMPLES OF PROOFS FOUND

This appendix contains examples of theorems that were unable to be proven by a single whole-proof sample from the language model, but were successfully proven by the model when broken down into lemmas. We present one randomly sampled example each for GPT-4 and Phind CodeLlama. For each theorem, we include the ground-truth proof from the dataset, the incorrect whole-proof attempt by the model, the error, and the final correct proof (along with lemmas extracted). We retain comments generated by the models, but omit the global environment. The complete proof scripts including the global environment can be found in the repository.

EXAMPLE 1 (GPT-4): THEOREM 676 (ADD_COMM)

The theorem in this example is a simple result about the commutativity of addition that can be proven using induction. While the model correctly identifies this, it makes errors in both the base case and the inductive step that result in four lemmas being necessary to complete the proof.

Ground-truth proof:

```
Theorem add_comm : forall n m : nat,
  n + m = m + n.
Proof.
  intros n m.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    simpl. rewrite <- plus_n_O. reflexivity.
```

```
  - (* n = S n' *)
    simpl. rewrite <- plus_n_Sm. rewrite <- IHn'. reflexivity.
Qed.
```

Incorrect whole-proof attempt:

```
Theorem add_comm : forall n m : nat,
  n + m = m + n.
Proof.
(* We prove this lemma by induction on n. *)
  intros n.
  induction n as [| n' IHn'].
  - (* The base case: 0 + 0 = 0 *)
      reflexivity.
  - (* The inductive case: S n' + 0 = S n' *)
      simpl.
  rewrite IHn'.
  reflexivity.
Qed.
```

Error in step `reflexivity.`

```
In environment
m : nat
Unable to unify "m + 0" with
"0 + m".
```

Error in step `rewrite IHn'.`

```
Found no subterm matching
"n' + ?M1094" in the current goal.
```

Correct proof after lemma extraction:

```
Lemma add_zero_comm_1 : forall m : nat, 0 + m = m + 0.
Proof.
  intros m.
  rewrite <- plus_n_O.
  reflexivity.
Qed.

Lemma add_zero_comm_0 : forall m : nat, 0 + m = m + 0.
Proof.
   (* We use induction on `n` to prove that for all natural numbers
    `n`, `n + 0 = n`. *)
  apply (@add_zero_comm_1 ).
Qed.

Lemma succ_plus_comm_2 : forall n' : nat, forall IHn' : forall m : nat,
  n' + m = m + n', forall m : nat, S (n' + m) = m + S n'.
Proof.
(* n' is fixed by forall so we can introduce the second premise (IHn')
   which is the inductive hypothesis that will be used later in the proof *)
  intros n' IHn' m.
(* Apply the previously introduced plus_n_Sm theorem which states that
   S (n + m) = n + S m *)
  rewrite <- plus_n_Sm.
(* Now apply our inductive hypothesis which will transform
   n' + m into m + n' *)
  rewrite IHn'.
```

```
(* At this point we have S m + n', which we want to show is equal to
   m + S n'.*)
  reflexivity.
Qed.

Lemma succ_plus_comm_1 : forall n' : nat, forall IHn' : forall m : nat,
  n' + m = m + n', forall m : nat, S (n' + m) = m + S n'.
Proof.
(* Let's prove the lemma by introducing the necessary variables. *)
  intros n' IHn' m.
(* Use the provided induction hypothesis (IHn'). *)
  apply (@succ_plus_comm_2 n' IHn' m).
Qed.

Theorem add_comm : forall n m : nat,
  n + m = m + n.
Proof.
(* We prove this lemma by induction on n. *)
  intros n.
  induction n as [| n' IHn'].
  - (* The base case: 0 + 0 = 0 *)
      apply (@add_zero_comm_0 ).
  - (* The inductive case: S n' + 0 = S n' *)
      simpl.
  apply (@succ_plus_comm_1 n' IHn').
Qed.
```

EXAMPLE 2 (PHIND-CODELLAMA): THEOREM 87 (FACTOR_IS_O)

This example depicts an error due to the use of `reflexivity` when the goal is not quite yet a trivial equality but requires some simplification first.

Ground-truth proof:

```
Lemma factor_is_O:
  forall n m : nat, n = 0 \/ m = 0 -> n * m = 0.
Proof.
  (* This pattern implicitly does case analysis on
     [n = 0 \/ m = 0] *)
  intros n m [Hn | Hm].
  - (* Here, [n = 0] *)
    rewrite Hn. reflexivity.
  - (* Here, [m = 0] *)
    rewrite Hm. rewrite <- mult_n_O.
    reflexivity.
Qed.
```

Incorrect whole-proof attempt:

```
Lemma factor_is_O : forall n m : nat, n = 0 \/ m = 0 -> n * m = 0.
Proof.
  intros n m H.
  destruct H as [Hn | Hm].
  - (* n = 0 *)
    rewrite Hn.
  reflexivity.
  - (* m = 0 *)
    rewrite Hm.
  reflexivity.
Qed.
```

Error in step `reflexivity.`

```
In environment
n, m : nat
Hm : m = 0
Unable to unify "0" with "n * 0".
```

Correct proof after lemma extraction:

```
Lemma mult_zero_right_neutral_1 : forall n' m : nat,
  forall Hm : m = 0, forall IHn' : n' * 0 = 0, n' * m = m.
Proof.
intros n' m Hm IHn'.
rewrite Hm.
(* We know that m = 0, so n' * m = n' * 0 by the equality m = 0. *)
  rewrite IHn'.
(* We know that n' * 0 = 0 by the induction hypothesis. *)
(* Therefore, n' * m = 0. *)
  reflexivity.
Qed.

Lemma multiplication_of_zero_0 : forall n m : nat,
  forall Hm : m = 0, n * 0 = 0.
Proof.
  intros n m Hm.
  induction n as [|n' IHn'].
  - (* n = 0 *)
    simpl.
  reflexivity.
  - (* n = S n' *)
    simpl.
  rewrite <- Hm.
  apply (@mult_zero_right_neutral_1 n' m Hm IHn').
Qed.

Lemma factor_is_O : forall n m : nat, n = 0 \/ m = 0 -> n * m = 0.
Proof.
  intros n m H.
  destruct H as [Hn | Hm].
  - (* n = 0 *)
    rewrite Hn.
  reflexivity.
  - (* m = 0 *)
    rewrite Hm.
    apply (@multiplication_of_zero_0 n m Hm).
Qed.
```