CoqPilot, a plugin for LLM-based generation of proofs

Andrei Kozyrev
JetBrains Research
Germany
Constructor University
Bremen, Germany

Gleb Solovev
JetBrains Research
Germany
Constructor University
Bremen, Germany

Nikita Khramov
JetBrains Research
Germany
Constructor University
Bremen, Germany

Anton Podkopaev
JetBrains Research
the Netherlands
Constructor University
Bremen, Germany

ABSTRACT

We present CoqPilot, a VS Code extension designed to help automate writing of Coq proofs. The plugin collects the parts of proofs marked with the admit tactic in a Coq file, *i.e.,* proof holes, and combines LLMs along with non-machine-learning methods to generate proof candidates for the holes. Then, CoqPilot checks if each proof candidate solves the given subgoal and, if successful, replaces the hole with it. The focus of CoqPilot is twofold. Firstly, we want to allow users to seamlessly combine multiple Coq generation approaches and provide a zero-setup experience for our tool. Secondly, we want to deliver a platform for LLM-based experiments on Coq proof generation. We developed a benchmarking system for Coq generation methods, available in the plugin, and conducted an experiment using it, showcasing the framework's possibilities. *Demo of* CoqPilot *is* available at: https://youtu.be/oB1Lx-So9Lo. *Code at*: https://github.com/JetBrains-Research/coqpilot

KEYWORDS

LLM, Coq, code generation

ACM Reference Format:
Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of International Conference on Automated Software Engineering (ASE '24).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Testing has always been essential for making reliable software. For some specific domains, such as aerospace engineering, banking infrastructure, or medical devices, bugs in critical systems may lead to catastrophic consequences [7, 13]. Formal software verification ensures that software operates correctly and safely by proving its correctness against the specification [15]. Under an assumption of a well-constructed specification, formal verification provides stronger guarantees than traditional testing methods, such as unit or integration testing, due to its exhaustive nature. To date, there exist a number of interactive theorem provers (ITP), such as Coq [2], Isabelle [14], or Lean [6]. They are designed to assist users with the construction of formal specifications and verification of formal proofs. For example, Coqhelps in development by providing a robust framework for defining mathematical assertions and ensuring the logical consistency of complex formal proofs. The formal verification approach has proved to be fruitful: for instance, CompCert [12], a C compiler written in Coq, was the only C compiler in which an extensive study found no bugs [21].

Coq is an interactive proof system, where proofs are constructed step-by-step using so-called *tactics*. When applied, tactics change the state of the current proof. In particular, a tactic may apply an already proven lemma, destruct the assumption to perform case analysis, apply induction reasoning, and much more. At any point of the proof, the proof state shown to the user will contain information about the current target statement and the assumptions under which it has to be proven. When the statement is empty, the proof is complete. If the proof contains an error or is not constructed correctly, Coq's system will tell that the proof is invalid and provide comprehensive information on the origin of the problem.

Writing formal proofs is an exceptionally time-consuming task and requires considerable experience from the programmer [16]. Various approaches for Coq generation are already present, both machine-learning-based and not. CoqHammer [5] translates the Coq's logic into untyped first-order logic and searches for the proof. The K-NN [3] approach, implemented as a back-end in Tactician [4], predicts tactics based on what has been used in similar cases. Other approaches are based on generative models [9, 17, 18, 20]. Recently, Large Language Models (LLMs) have gained strong code-generation capabilities [11]. Combined with tools for automatic code verification, we may be able to produce high-quality, reliable code seamlessly.

Some developed models and tools for Coq generation may require significant setup and/or lack integration into the platform for end users [9, 18, 20]. One other space of improvement for existing non-deterministic proof search processes is to use the information provided by the Coq's system. Even for a human, writing Coq code in a notepad instead of a proper Coq IDE would be harder than in a typical programming language. Interactive stepping through each tactic invocation and updated goal states provide the necessary information during the process of writing proofs. Fortunately, such information can be gathered automatically and used for proof generation.

We propose CoqPilot, a VSCode plugin designed to deliver a convenient generation of Coq code using LLMs and other methods. We studied possible external enhancements to generating Coq code with general-purpose models. The automatic checking of multiple generated proof candidates was developed to pick and present only the valid one to the user. We implemented premise selection for better LLM prompting and created an LLM-guided mechanism that attempts fixing failing proofs with the help of the Coq's error messages. To evaluate the performance of the described solutions, we

ASE '24, October 27–November 01, 2024, Sacramento, CA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

implemented a benchmarking framework for our extension. The framework allows efficiently conducting experiments on Coq generating using different models. We experimented with this framework, comparing several LLMs in Coq generation and evaluating if our contributions boost their performance.

To allow automatic proof checking, we implemented a higher level module, wrapping Coq Language Server[1] and providing abstractions such as the one to check if the given proof for the theorem is valid in a particular environment. We used the particular Coq language server implementation [8], and from now onwards, we will refer to it as Coq-LSP.

The proposed `CoqPilot`'s architecture is modular regarding the target language, requiring minimal code changes to adapt to another language. `CoqPilot` integrates popular LLMs and allows users to include tools like Tactician and CoqHammer in the proof generation pipeline. During implementation, we addressed the challenges of using commercial LLMs, including managing token limits and handling failures, by developing mechanisms for retries and clear user feedback.

In Section 2, we describe the plugin and the challenges we overcame during its development process. Section 3 discusses the created benchmarking framework. In Section 4, we describe the experiment we conducted and evaluate the features proposed in Section 2. In Section 5, we cover related work, and in Section 6, we conclude and glance at future work.

2 COQPILOT

In Coq, a goal represents a statement or proposition to be proven. One typically starts the proof with the statement you want to establish as true. Then, one applies tactics and transforms the current goal into one or more subgoals that are simpler or more manageable. Special `admit` tactic allows skipping a subgoal to permit further progress on the rest of the proof. If the proof contains admits, it is considered incomplete. Each `admit` corresponds to a self-contained goal with the hypotheses and the conclusion. Say we have a Coq file with a number of unfinished proofs containing admits. `CoqPilot` runs over the admits and tries to substitute them with correct proofs.

We designed `CoqPilot` to serve as a tool for combining different approaches to Coq generation. We implemented multiple ways to fetch completion and infrastructure around Coq-LSP to check proof candidates. We will refer to each way of fetching completion as a *service*. Currently, the available services are OpenAI API, LLMs running locally through LMStudio, JetBrains AI Platform, and completion via predefined automation tactics. Coq automation tools such as Tactician [4] and CoqHammer [5], which are triggered using special tactics, could be added to the pipeline through the predefined tactics to unite generation capabilities with `CoqPilot`.

A particular setup for the completion request is denoted as *model parameters*. For LLM-based services, model parameters include the LLM name, the temperature, the prompt, the number of choices to make (*i.e.,* the number of completions the model should generate), and other specifications. As LLMs are not deterministic, making several attempts for each model is beneficial. This result is backed up in Section 4. The setup of `CoqPilot` consists of a list of model parameters for each of the chosen services.

[1]Language Server Protocol: https://microsoft.github.io/language-server-protocol/

While implementing the described approach, we encountered several difficulties that affected the `CoqPilot`'s final architecture. Different proof holes in Coq have independent states, and we intend to generate completion for distinct holes in parallel. This requires introducing safety of concurrency to our developed proof-checking mechanisms since Coq-LSP cannot process parallel requests. Our goal was to develop an infrastructure with interchangeable components to allow users to easily add new services and prepare the ground to interchange Coq with another ITP.

Accurate error handling presents other challenges. Services such as OpenAI have various types of errors, which are supposed to be handled differently. Some may be classified as parameter validation errors and presented to the users; others may be service errors. One of the critical failures, which mainly occurs during benchmarking, is caused by exceeding token limits. Commercial LLM providers restrict their models' usage rates. Local token counters are imprecise, which makes it challenging to overcome these limitations completely. Therefore, correctly handling such errors becomes crucial for presenting them to the user in an understandable manner. To address this issue, we developed a custom error class hierarchy, differentiating between configuration errors, generation failures, and connection errors, and repacked specific service errors into these appropriate classes. The implementation reports and logs errors based on their types, supporting both user and benchmarking modes.

`CoqPilot` offers many configurable parameters. They help the user to set up both the plugin behavior and the experiments using the benchmark. We have implemented a parameter resolution framework, which correctly handles errors, allowing a programmer to write reliable resolvers for new services and parameters.

One of our contributions is enhancing the capabilities of general-purpose LLMs in generating Coq code. Given a position to perform completion, we can get the desired statement to prove and the hypotheses under which it should hold. However, this is usually not enough. Writing Coq proofs, a human often recalls other lemmas and objects in the corresponding file/project. It may be challenging for the model to deal with a theorem isolated from its context. To address this problem, we perform premise selection[2] for the theorems within the same file and use them as a few-shot prompt to an LLM. During few-shot prompting, several concrete examples of how the task needs to be solved are provided. Few-shot prompting gives the model a better understanding of the problem context and structures the format of its output. Due to token limitations and the model's context window size, we can usually only take a subset of theorems from the file. We choose optimal premises using metrics such as distance from the generation target or similarity with other theorem statements.

Also, we may extract helpful information from the Coq's system when proof candidates fail. In particular, we may get the error that occurred and use it to try to fix the failing proof. Baldur [10] used an idea of proof repairing to train a separate proof fixing model. A similar to `CoqPilot` approach with general purpose LLMs may be found in Copra [19]. When `CoqPilot`'s general pipeline does not find the proof, we launch a multi-turn communication process with

[2]Retrieval of facts from some given knowledge base that can help the model and advance the proof.

an LLM. The number of completions to fetch per turn and the depth d are predefined in settings by the user. We send the compilation error and special prompt to the LLM and ask it to fix it. If the proof is still not accepted by Coq afterward, we repeat the process, but at a maximum of $d - 1$ times.

3 COQPILOT BENCHMARK

We aimed to develop a benchmarking framework to evaluate the current effectiveness of features implemented in CoqPilot and find space for further improvements. Specifically, our research questions included (i) how well general purpose LLMs can write Coq proofs, (ii) to which extent does CoqPilot improve the LLM approach to Coq generation, and (iii) which additional value CoqPilot, using general-purpose LLMs, contributes to other Coq automation tools such as CoqHammer and Tactician?

Implementing such a framework brought up several issues that we solved. The main peculiarity of our benchmarking approach is that we need to send a large number of tokens to each model. In order to maintain reasonable performance, we aim to make requests as fast as possible. However, there is usually a limitation on the number of tokens that can be sent in a short time frame. To overcome this, we considered the requests to each model in each service to complete the given goal as a separate asynchronous *task*. We heuristically determined the necessary *waiting* time for these tasks to comply with the mentioned limitation.

The developed benchmark framework provides a number of possibilities. First, it allows the gathering of information about the internal state of CoqPilot, *e.g.,* the theorems chosen for the context and the number of used tokens. Second, thanks to the implemented interfaces and the CoqPilot's architecture, the developed framework can be conveniently scaled for experimenting with other tools. In this work, we experimented with Tactician and CoqHammer. Moreover, it is possible to generate tailored reports based on the results of the experiments that were conducted.

4 EVALUATION

To evaluate the performance of CoqPilot, we required a dataset with a large number of human-written theorems and proofs. As it was said before, CoqPilot depends on Coq-LSP, which is not version agnostic and supports Coq versions starting from 8.15. Due to this limitation, it was impossible to fully leverage the CoqGym dataset [20] for our experiments as it contains projects requiring older Coq versions. We have decided to limit ourselves to Coq 8.19 as the latest version available. We have chosen the IMM project[3] for our experiment. The project consists of a large number of proofs and supports Coq 8.19. Moreover, the IMM is of particular interest to our lab since it is developed there.

The data for the experiments was prepared as follows. We decided to consider only the proofs of at most 20 tactics, as we initially developed CoqPilot to help users generate subgoals or smaller lemmas. Theorems with proofs with such lengths amount to 83% of proofs in the IMM project. Due to the amount of computing and financial resources at our disposal, we have been unable to experiment on the entire project. Therefore, we decided to use a relatively small subset of 300 theorems. Moreover, we wanted to split the dataset into three groups based on the length of proofs measured in tactics. This was done to provide a clearer interpretation of the results. The group sizes were chosen with respect to the distribution of proof lengths in the given project so that the results of the experiments would be representative of the entire project. The final group sizes are presented in Table 1.

During the main experiment, we evaluated how many theorems from the constructed dataset could be proven using different methods. We used Coq's built-in first-order reasoning tactic with automation `firstorder auto with *` as a baseline. We have chosen GPT-4o, GPT-3.5, Anthropic Claude, and the open-sourced LLaMA-2 13B Chat as models. The average number of theorems sent to a model varied from the context window size, *e.g.,* for GPT-4o it was 52. Completion choices were equal to 12 for GPT-4o, 20 for GPT-3.5 and LLaMA and 7 for Anthropic Claude. The multi-round feature was disabled due to the exhaustive tokens consumption of this feature. Along with the models listed above, we have tested Tactician and CoqHammer with timeouts of 30, 60, and 90 seconds for the three groups, respectively. If the proof was not found during the specified timeout, we consider the theorem as unsolved. The percentages in the table cells represent the proportion of theorems in each group successfully solved using the specified method. More details are provided in the experiment report.[4]

Table 1: Benchmarking results

Reference proof length	$\leqslant 4$	5–8	9–20	Total
Group size	131	98	71	300
`firstorder auto with *`	11%	2%	1%	6%
OpenAI GPT-3.5	29%	17%	6%	20%
OpenAI GPT-4o	50%	26%	15%	34%
LLaMA-2 13B Chat	2%	0%	0%	0.5%
Anthropic Claude	21%	7%	7%	13%
All models together	57%	32%	18%	39%
Tactician	45%	23%	10%	29%
CoqHammer	23%	4%	0%	11%
All methods together	71%	45%	23%	51%

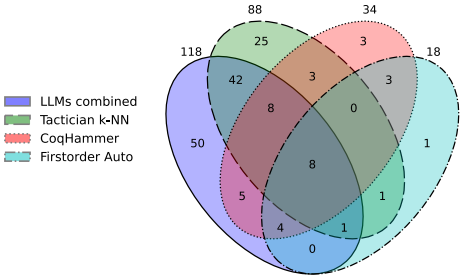
GPT-4o with CoqPilot's approach can prove 34% theorems, as seen in Table 1, with 51% of them being proved on the first attempt. GPT-3.5 can only prove 30% on the first attempt, which can be explained by the smaller context window size, resulting in fewer chosen premises. Another noticeable result is that among each group, the collectible effort of all models is stronger than any individual one. It shows that the approach of CoqPilot to using a sequence of different models altogether is promising. A combination of four models used through CoqPilot, CoqHammer, and Tactician can prove 51% theorems. The user can invoke this powerful combination from CoqPilot with a single call. Figure 1 shows methods corresponding to the sets of theorems they can prove.

Additionally, to examine how varying the number of premises sent to the model impacts the results, we compared GPT-4o with a different number of premises used in context on another 50 samples

[3]IMM: https://github.com/weakmemory/imm

[4]https://github.com/JetBrains-Research/coqpilot/tree/main/etc/docs/benchmark

Figure 1: Venn diagram of the proven theorems



from the IMM project. Results show that the model can solve 0% with 0 theorems as premises, 8% with 1 theorem, and 32% theorems with the maximum possible number of premises.[5] Another experiment with GPT-4o and 50 theorems showed that the multi-round mechanism with depth 2 and width 2 fixes 2 proofs in addition to the ones that were already generated correctly.

The results demonstrate the benefits of using `CoqPilot` instead of plain LLMs, showcase additional value to other Coq provers, and highlight the usability of using multiple generation methods at once via `CoqPilot`.

5 RELATED WORK

Many Coq generation tools require a long setup and are hardly integrated into the Coq development workflow. Proofster [1] is a web interface for Coq proof synthesis and exploration. Tactician [4] tries to generate Coq proofs after invocation by the special tactics. Copra [19] is an agent for theorem proving, which repeatedly uses a general-purpose LLM for completion. Our work serves similar purposes with a focus on a couple of factors. We aim to develop a plugin that incorporates well into a typical user's workflow and provides setup-free experience. We have built our tool around uniting many approaches and seamlessly allowing users to try all available tools for their problems. This pipeline also brings convenience in experimenting. Another focus is automatically boosting non-deterministic Coq generation tools with the Coq's proof checker. Along with that, we implemented fetching completion from common LLM providers. Tools such as Tactician can be used in `CoqPilot` as services via predefined tactics without any additional effort from the user.

6 CONCLUSION

We presented `CoqPilot`, a VSCode plugin for Coq generation that requires minimal setup. We allow users to seamlessly switch between different Coq generation methods and easily add new ones. We contributed techniques that boost the performance of general-purpose LLMs. Compared to one-shot plain GPT-4o invocation, which can solve 0% theorems from the compiled dataset, GPT-4o with `CoqPilot`' modifications gets 34%. As shown in Table 1, the joint effort of four models integrated into `CoqPilot`, achieves 39%. We contributed a highly configurable experiment framework for testing methods implemented in `CoqPilot` for Coq generation.

[5]The maximum possible number of premises is calculated as the maximal number of premises that fit into the model's context window.

ACKNOWLEDGMENTS

This paper has been greatly improved by the comments of Yaroslav Golubev, Ekaterina Verbitskaia and Marat Akhin. We thank Emilio Jesús Gallego Arias for his work on Coq-LSP.

REFERENCES

[1] Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, Talia Ringer, and Yuriy Brun. 2023. Proofster: Automated formal verification. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 26–30.
[2] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.
[3] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. Tactic learning and proving for the Coq proof assistant. *arXiv preprint arXiv:2003.09140* (2020).
[4] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The tactician: A seamless, interactive tactic learner and prover for coq. In *International Conference on Intelligent Computer Mathematics*. Springer, 271–277.
[5] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453.
[6] 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: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer, 378–388.
[7] Henrico Dolfing. 2019. *The $440 Million Software Error at Knight Capital*. Retrieved June 3, 2024 from https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html
[8] Emilio Jesús Gallego Arias et al. 2022. Visual Studio Code Extension and Language Server Protocol for Coq. https://github.com/ejgallego/coq-lsp
[9] 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.
[10] 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.
[11] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515* (2024).
[12] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.
[13] Jessica MacNeil. 2019. *Mariner 1 destroyed due to code error, July 22, 1962*. Retrieved June 3, 2024 from https://www.edn.com/mariner-1-destroyed-due-to-code-error-july-22-1962/
[14] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.
[15] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2023. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook.
[16] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281.
[17] Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. 2024. Graph2Tac: Learning Hierarchical Representations of Math Concepts in Theorem proving. *arXiv preprint arXiv:2401.02949* (2024).
[18] 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.
[19] Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. *arXiv preprint arXiv:2310.04353* (2023).
[20] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*. PMLR, 6984–6994.
[21] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009