`CoqPilot`, a plugin for LLM-based generation of proofs

Andrei Kozyrev[13], Gleb Solovev[13], Nikita Khramov[13], and Anton Podkopaev[23]
first.last@jetbrains.com

[1] JetBrains Research, Germany
[2] JetBrains Research, the Netherlands
[3] Constructor University Bremen, Germany

Abstract

We present `CoqPilot`, a VS Code extension designed to help automate the 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 provide a zero-setup experience for end-users. Secondly, we want to provide a platform for LLM-based experiments with generation of Coq proofs.

Code available at https://github.com/JetBrains-Research/coqpilot

Large Language Models (LLMs) have recently demonstrated their remarkable ability to generate code in a variety of programming languages. A natural second frontier is to use LLMs for generating code for proof assistants. There have already been works dedicated to using LLMs for theorem proof generating [5, 7, 13, 16, 14, 4]. Noticeable research by OpenAI [13] has shown how transformers could be used to successfully generate formal languages. Most of the recent works focused on one step at a time generation, followed by a proof search [7]. However, attempts to generate the complete proof using LLMs are also present [5]. Extensive research was conducted on how to improve the model itself and achieve better generation metrics.

In this particular work, we focus on maxing out the generation capabilities, regardless of the particular model. Main contributions include (i) studying possible external enhancements to the process of generating Coq code with non-fine-tuned models *and* (ii) creating an applied tool for convenient generation of Coq code using LLMs, as well as facilitating easy conduction of experiments and research based on it.

In contrast to other works and tools, we aim to create a zero-setup experience for the user. `CoqPilot` is our VS Code plugin [8], which needs just a particular API key, assigned in the settings to start running. Projects like ASTactic [16], TacTok [4], and ProverBot9001 [14] learn predictive models, yet lack an interaction interface for the end users. Proofster [2] provides a web interface for Coq code generation, yet this interface is not integrated into the code-writing process. `CoqPilot` integrates directly into the currently popular IDE choice for Coq – VSCode. Compared with Tactician [1], we provide a built-in opportunity to experiment with many general-purpose LLMs. Users can easily configure many model parameters through plugin settings and combine different approaches to boost performance. Moreover, approaches like Tactician and CoqHammer [17], called via a special tactic, are easily integrated into `CoqPilot`.

A common setting in which `CoqPilot` works is as follows: an open Coq file with a number of successfully proven theorems, and several goals, containing `admit`. In such a setting, `CoqPilot` uses already proven theorems as a few-shot prompt for the LLM and tries to retrieve completion for all the admitted goals independently.

Any admitted goal in Coq could be represented as a standalone theorem, using the hypotheses and the conclusion at the point of `admit`. We formulate the problem for the LLM as

follows: given the theorem's statement, generate the proof for it. LLMs tend to perform better on few-shot prompting[1] examples [9], compared to zero-shot. Theorems from the active file are collected, and a few are used as prompts for the chosen LLMs. The selection of theorems is based on certain metrics, such as their distance from the target theorem that needs to be solved. We often cannot use the complete list of theorems, as the LLM's context is limited.

The main strength of such formal methods like Coq is the ability to automatically perform type-checking. In tandem with the generative capabilities of LLMs, it allows us to check the correctness of the generated code. To allow automatic proof checking, we implemented a higher level module, wrapping Coq Language Server[2] and providing useful abstractions over it such as the one to check if proof is valid in a given environment. We used the particular Coq language server implementation [3] and from now onwards will refer to it as Coq-LSP. For each target theorem, we generate n potential proofs, check all of them, and in case of success, return it as an answer. In case of failure, we launch a process aiming to fix the failing proof. This process is similar to the one used in Copra [15]. Given the depth d and the number of completions requested each time m, for each incorrect proof, we start a multi-round communication process with an LLM. We send the compilation error along with the special prompt to the LLM and ask them 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.

We also aimed to create a benchmarking environment to evaluate how well different LLMs and other techniques can generate Coq code. Due to specifics of Coq, Coq-LSP is not Coq version agnostic. Hence, neither are we. Moreover, Coq-LSP supports Coq versions starting from 8.15/8.17. As a consequence, we could not use CoqGym [16] as a dataset provider, as it contains projects requiring old Coq versions. Currently maintained Coq version by us is 8.19 as the latest one available.

We have conducted an experiment with a set of theorems from the IMM project [12]. IMM, which supports the desired Coq 8.19, is of particular interest to our lab. LLMs we have picked for evaluation include GPT models [10, 11], LLaMA [6], and Anthropic Claude 2.1. Moreover, we tried firstorder reasoning tactic `firstorder auto with *` as a baseline. From each model, we attempted to sample a correct proof for the theorem up to 20 times. In this experiment, we did not try to do proof fixing. To pick the dataset, we took all proven theorems from the IMM project. Then we have chosen only theorems shorter than 20 tactics (83% of the original amount). We divided theorems by their human-written proof lengths into three groups: three to four, five to eight, and nine to twenty lines long. Then, we randomly chose 45 theorems from the dataset with group sizes proportional to the initial distribution.

Reference proof length	$\leqslant 4$	$5-8$	$9-20$	Total
Group size	20	14	11	45
OpenAI GPT-3.5	35%	7%	18%	22%
OpenAI GPT-4	55%	7%	9%	28%
LLaMA-2 13B Chat	5%	0%	0%	2%
Anthropic Claude 2.1	25%	7%	0%	13%
Firstorder	25%	7%	0%	13%
All methods	60%	21%	18%	38%

Table 1: Benchmarking results

[1] During few-shot prompting, several concrete examples of how the task is to be solved are provided. Zero-shot prompting implies the system prompt is used without examples.

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

A notable 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. The benchmarking tool and the report on experiments are published in the repository.[3]

References

[1] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The Tactician: A seamless, interactive tactic learner and prover for coq, 2020. https://arxiv.org/abs/2008.00120.

[2] Agrawal et al. Proofster: Automated formal verification, 2023.

[3] Emilio Jesús Gallego Arias et al. Visual studio code extension and language server protocol for coq, 2022.

[4] Emily First, Yuriy Brun, and Arjun Guha. TacTok: semantics-aware proof synthesis, 11 2020.

[5] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models, 2023. https://arxiv.org/abs/2303.04910.

[6] Meta GenAI. LLaMA 2: Open foundation and fine-tuned chat models, 2023. https://arxiv.org/abs/2307.09288.

[7] Albert Q. 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, 2022. https://arxiv.org/abs/2205.10893.

[8] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. CoqPilot, a visual studio code extension, designed to help automate writing of coq proofs, using large language models, 2023. https://marketplace.visualstudio.com/items?itemName=JetBrains-Research.coqpilot.

[9] Huan Ma, Changqing Zhang, Yatao Bian, Lemao Liu, Zhirui Zhang, Peilin Zhao, Shu Zhang, Huazhu Fu, Qinghua Hu, and Bingzhe Wu. Fairness-guided few-shot prompting for large language models, 2023. https://arxiv.org/abs/2303.13217.

[10] OpenAI. Language models are few-shot learners, 2020. https://arxiv.org/abs/2005.14165.

[11] OpenAI. GPT-4 technical report, 2024. https://arxiv.org/abs/2303.08774.

[12] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis et al. Intermediate memory model and compilation correctness proofs for it, 2019. https://github.com/weakmemory/imm.

[13] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. https://arxiv.org/abs/2009.03393v1.

[14] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks, 2020. https://arxiv.org/abs/1907.07794.

[15] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving, 2024. https://arxiv.org/abs/2310.04353.

[16] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants, 2019. https://arxiv.org/abs/1905.09381.

[17] Czajka Ł and Kaliszyk C. Hammer for coq: Automation for dependent type theory, 2018.

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