CoqPyt: Proof Navigation in Python in the Era of LLMs

Pedro Carrott[*]
pedro.carrott@imperial.ac.uk
Imperial College London
London, UK

Nuno Saavedra[*]
nuno.saavedra@tecnico.ulisboa.pt
INESC-ID/IST, University of Lisbon
Lisbon, Portugal

Kyle Thompson
r7thompson@ucsd.edu
University of California San Diego
San Diego, USA

Sorin Lerner
lerner@cs.ucsd.edu
University of California San Diego
San Diego, USA

João F. Ferreira
joao@joaoff.com
INESC-ID/IST, University of Lisbon
Lisbon, Portugal

Emily First
emfirst@ucsd.edu
University of California San Diego
San Diego, USA

ABSTRACT

Proof assistants enable users to develop machine-checked proofs regarding software-related properties. Unfortunately, the interactive nature of these proof assistants imposes most of the proof burden on the user, making formal verification a complex, and time-consuming endeavor. Recent automation techniques based on neural methods address this issue, but require good programmatic support for collecting data and interacting with proof assistants. This paper presents CoqPyt, a Python tool for interacting with the Coq proof assistant. CoqPyt improves on other Coq-related tools by providing novel features, such as the extraction of rich premise data. We expect our work to aid development of tools and techniques, especially LLM-based, designed for proof synthesis and repair. A video describing and demonstrating CoqPyt is available at: https://youtu.be/fk74o0rePM8.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; **Development frameworks and environments**.

KEYWORDS

Coq, Data extraction, Theorem proving, Retrieval augmentation

1 INTRODUCTION

Formal software verification is an incredibly effective method for developing high quality software, as it ensures that a software program adheres to a predefined formal specification. For example, a study compared industrial standard compilers (*e.g.*, GCC and LLVM) to CompCert, a C compiler verified using the Coq proof assistant [25], and CompCert was the only one for which no bugs were found [30]. Unfortunately, even though formal software verification provides valuable guarantees, its development is still too costly. CompCert took 100,000 lines of Coq code and 6 person-years to formally verify [13]. For this reason, it is important to find methods that decrease the cost of formal verification.

One recent approach to address this is the use of neural methods to automate proof synthesis. *Neural theorem provers*, given a partial proof and the *proof state*, use neural networks to predict the next likely *proof steps*. A proof assistant is then used to evaluate the candidate proof steps, returning either new proof states or errors. Neural theorem provers iterate on this procedure, performing *proof search* to traverse the space of possible proofs. A complementary task is automated proof repair [20], which is necessary to perform when specifications or dependencies change and result in broken proofs. To train these neural models and implement proof search, sufficient programmatic support is thus required for collecting training examples and driving the theorem prover.

In this paper, our target proof assistant is Coq due to its popularity for building verified software systems. Existing tools, such as Coq Serapy [22] and PyCoq [9], provide programmatic support for interacting with Coq in Python, and all current neural theorem provers for Coq utilize them. These tools implement Python bindings for Coq SerAPI [7], which is a library for machine-to-machine interaction with Coq. However, with the announcement that Coq SerAPI will be deprecated and replaced with Coq LSP [26], a language server for Coq, the need arises for a new tool to continue supporting existing features. Fortunately, this also creates a great opportunity to support new features that prior tools did not.

In the emerging era of large language models (LLMs), additional functionality is desirable in tools for interfacing with Coq in Python in order to promote LLM-based approaches for Coq proof synthesis and repair. Namely, the ability to collect rich contextual data is not supported by any of the existing programmatic tools for controlling Coq. LLM-based neural theorem provers for Lean and Isabelle proof assistants rely on *retrieval augmentation*, wherein they retrieve premises, such as lemmas and definitions, that are relevant to the proof goal, and condition the next tactic generation on those premises [17, 29]. For the approach to be effective, they collect fine-grained annotations of which premises are used in proofs as well as which premises are accessible from a proof. However, existing tools for Coq do not collect that type of premise data.

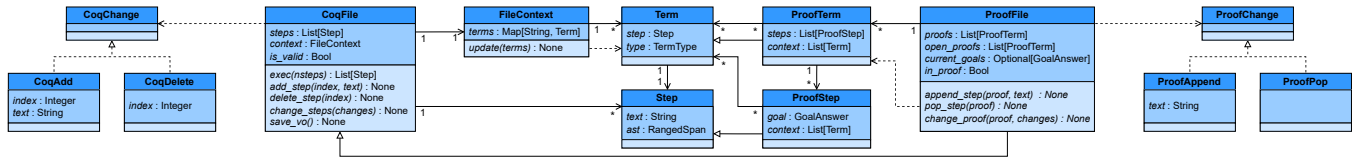[*]Both authors contributed equally to this research.

Figure 1: UML diagram for a simplified view of the domain of CoqPyt. The selected attributes and methods are all public.

We introduce CoqPyt, a tool that enables interaction with Coq in Python using Coq LSP as the backend. CoqPyt continues to support existing capabilities essential to training data collection and proof search, but goes beyond state-of-the-art tools for Coq by supporting the ability to collect rich contextual data, among other novel features. Our tool supports the newest versions of Coq (8.17 to 8.19), which will facilitate mining software repositories for new, enriched training and evaluation data. We foresee that our work will help promote future research into neural proof synthesis and repair for Coq, especially as LLM capabilities continue to evolve. CoqPyt is open-source and available at: https://github.com/sr-lab/coqpyt.

2 COQPYT

CoqPyt is a Python tool that enables developers and researchers to interact programmatically with Coq. It also allows users to extract textual and structural information from Coq files. We now describe the structure and main functionalities of CoqPyt, providing an illustrative example on how to interact with the tool.

2.1 The Coq Proof Assistant

Coq provides an interactive environment with a rich type system suitable for theorem proving. A theorem in Coq is a type definition, which can be proven by constructing a *proof term* with the stated theorem type. Since writing a proof term directly is difficult, Coq allows users to write a *proof script* consisting of a sequence of high-level *tactics* (*e.g.*, induction or reflexivity). Each tactic guides Coq in a search for the desired proof term, refining the state until no new obligations hold. After a tactic application, Coq displays the current *proof state*, which includes the goals to prove and the local context. When executed, a complete proof script generates the proof term.

2.2 Coq Language Server Protocol

To interact with Coq files, we use Coq LSP [26], a language server that provides Coq-related features. Coq LSP was developed to replace Coq SerAPI [7], a well-known tool in the Coq community that implements a protocol for interacting specifically with Coq files. Unlike Coq SerAPI, Coq LSP follows a standardized protocol for interacting with text documents known as Language Server Protocol (LSP) [16]. LSP servers, such as Coq LSP, provide useful language features, such as auto complete. Client applications, such as IDEs, can use these features by running their own LSP client.

Besides the usual LSP features, Coq LSP provides Coq-specific functionality, namely requesting the proof state at any given point of a Coq file. Coq LSP also allows users to request the abstract syntax tree (AST) of the entire document, which corresponds to a list of *steps* in the file, each with its own AST representation. A step can be a *term definition* (*e.g.*, definitions and theorems) or a

proof step (*e.g.*, mid-proof tactic application), determined by the AST contents of each step. For erroneous Coq files, Coq LSP is able to ignore erroneous steps and evaluate the remaining steps of the file, allowing any subsequent non-erroneous steps to define new terms. Coq LSP also enables project-wise imports by saving compiled .vo files, which may also benefit from this error-ignoring approach, given that valid files may still import terms from invalid files.

2.3 Interaction with Coq Files

CoqPyt implements a client for Coq LSP providing the interface shown in Figure 1. This client implementation is encapsulated within CoqFile, a class that abstracts the state of an actual Coq file, hiding the interaction with Coq LSP. For instance, the is_valid boolean flag indicates if the underlying Coq file is valid and the save_vo method allows the user to compile the file.

A CoqFile object provides an exec method which allows users to take an arbirary number of *forward* and *backward* steps through the file. The method returns a list of Steps corresponding to the steps that were executed or backtracked during the call. Each step contains its textual representation in the file and the AST contents.

Given that steps may be term definitions, a FileContext object is kept within the CoqFile, indexing by name all terms that have been defined until the last executed step. Any term defined inside a module has its name prefixed by the module path. Although notations are nameless terms, these are also stored in the context, indexed by the string pattern used to define them. For each defined term, a Term object is created containing its corresponding step, as well as its *term type*. This type depends on whether the term is a theorem, a notation, a tactic or any other definable Coq construct.

2.4 Proof Navigation

With CoqPyt, it is also possible to track the proof state throughout the file. A CoqFile can be instantiated through the ProofFile subclass, which allows users to manage the file's proof context. As steps are taken, one can enter (or leave) proof mode, which will activate (or deactivate) the in_proof boolean flag. If there are any on-going goals, these can be accessed via the current_goals attribute. Goals are represented as a GoalAnswer object, which mimics the structure of goals in the Coq LSP response [26].

In addition to allowing users to manage a file's proof context, ProofFile instances also fetch all of the file's *imported* terms. Thus, a ProofFile captures terms in a file and in its dependencies. In practice, the ProofFile context is initialized by instantiating a CoqFile for each library imported by the file and extracting the context of each library to the new ProofFile instance. Since creating a ProofFile requires instantiating multiple CoqFiles, it is more expensive to create a ProofFile than a CoqFile.

A `ProofFile` holds information about all proofs found in the file until the last executed step. Proofs with open goals are kept in the `open_proofs` attribute. Multiple open proofs occur in files with *nested* proofs, as we need to open inner proofs before closing outer proofs. Each proof is a `ProofTerm`, a `Term` with a `context` attribute, listing all terms found in the current context used to define the proof term. This contextual information is valuable for neural-based models and is not accessible from prior Coq-related tools.

A `ProofTerm` also contains an ordered list of `ProofSteps`, corresponding to all steps taken until the proof is closed (or until the last executed step if the proof is open). The `ProofStep` class enables a mapping from each proof step to the intermediate proof goals it attempts to solve or simplify. A `ProofStep` is thus constructed by augmenting the respective `Step` with the intermediate proof goals before that step is taken. Similarly to `ProofTerm`, a `ProofStep` also has a `context` attribute listing the terms used in the step.

2.5 Proof Modification

CoqPyt allows steps to be *added to* or *deleted from* the file. Allowing such modifications on Coq files yields an interface well suited for proof development. Invalid changes, such as adding inexisting tactics, may lead to explicit errors or undefined behaviour, depending on the nature of the error. For erroneous modifications, the file will revert to its original state, ignoring the requested changes.

Through the `add_step` method, it is possible to define terms, introduce theorems and apply tactics to solve proof obligations. Conversely, `delete_step` enables the removal of steps, such as added tactics which do not alter the proof goals as intended. Both methods allow editing in arbitrary parts of the file: `delete_step` requires an index i to delete the ith step in the file; `add_step` requires the index of the step which precedes the new step. To add a new step, its textual representation must also be provided.

Modifications may also be batched as transactions. A sequence of `CoqChanges` can be provided to the `change_steps` method, which will perform all modifications. A `CoqChange` can be either a `CoqAdd` or a `CoqDelete`, which serve as data wrappers for the parameters of `add_step` and `delete_step`, respectively. During the transaction, intermediate invalid states are allowed as long as the final state is valid. For example, it is possible to add a `Qed` before adding the actual proof steps, which cannot be done through successive `add_step` calls. To directly modify a proof, `ProofFile` provides the methods `append_step` and `pop_step` for single step modifications, as well as a `change_proof` method for a transaction of `ProofChanges`.

2.6 Use Cases

The features that CoqPyt offers are particularly useful for proof search and learning-based tasks, such as neural theorem proving. To get a sense of CoqPyt's interface for collecting data and conducting proof search, consider the file `test.v` in Listing 1, where we have a Coq file that defines a property about reversing a list. Listing 2 shows how we can leverage CoqPyt to collect data from the file shown in Listing 1. We can see which terms are available to the file via the `pf.context` attribute (line 3). For example, all the terms from the imported package `List.v` are available to the proofs in the file. Likewise, we show how to retrieve information from each proof step in the file via the `ProofStep` attributes (line 6).

```
1  Require Import List.
2  Lemma rev_append: forall {a} (l1 l2: list a),
3    rev (l1 ++ l2) = rev l2 ++ rev l1.
4  Proof.
5  intros a l1 l2. induction l1; intros.
6    - simpl. rewrite app_nil_r. reflexivity.
7    - simpl. rewrite IHl1.
8  Admitted.
```

Listing 1: Example Coq file `test.v` with a property of `rev`.

```
1  with ProofFile("test.v") as pf:
2    pf.exec(len(pf.steps))
3    print(pf.context)
4    for proof in pf.proofs:
5      for step in proof.steps:
6        print(step.text, step.ast, step.context, step.goals)
```

Listing 2: Proof data available with CoqPyt.

```
1  incorrect = [" reflexivity.", "\nQed."]
2  correct = [" rewrite app_assoc."] + incorrect
3  with ProofFile("test.v") as pf:
4    pf.exec(len(pf.steps))
5    unproven = pf.unproven_proofs[0]
6    for attempt in [incorrect, correct]:
7      changes = [ProofPop()]  # Admitted
8      for s in attempt:
9        changes.append(ProofAppend(s))
10     try:
11       pf.change_proof(unproven, changes)
12       print("Proof succeeded!")
13       break
14     except InvalidChangeException:
15       print("Proof attempt not valid.")
```

Listing 3: Proof attempts with CoqPyt.

Developing neural-based theorem provers requires the ability to attempt possibly erroneous proofs during proof search. For example, suppose we wanted to complete the proof `rev_append` from Listing 1. Listing 3 shows how we can use the `change_proof` method to conduct proof attempts. We first obtain the `ProofTerm` object of the proof `rev_append` (line 5). We then delete the "`Admitted.`" step from the proof of `rev_append` (line 7) and add the steps associated with our new proof attempt (line 9). When we attempt to add the erroneous steps from the proof attempt `incorrect` (line 11), an error is returned and `test.v` will remain in its original state. In turn, when the valid steps from the proof attempt `correct` are added (line 11), the changes are applied to `test.v`.

3 RELATED WORK

Most neural theorem provers for Coq use either Coq Serapy [22, 24], PyCoq [2], or a custom Python class [4, 5, 23, 28] to create training examples and perform proof search. The neural theorem prover GamePad [8] instead modifies Coq itself to record intermediate proof states. The CoqGym benchmark dataset [28], collected using a Python class, is the state-of-the-art benchmark for Coq. However, it was collected in 2019 and has only been updated to support versions 8.10 and 8.12 of Coq, while the newest version is 8.19. CoqPyt will render neural theorem provers useful for new Coq developments since it can be used for mining software repositories for newer Coq training data and for evaluation.

Table 1: Evaluation of CoqPyt's performance on CompCert. PCC stands for Pearson correlation coefficient.

Metric/Feature	Count		Execution time (s)									
	Proofs	Steps	Load file	Execute file	Add step		Delete step		Change steps (Add)		Change steps (Delete)	
					Beginning	On pointer	Beginning	On pointer	Beginning	On pointer	Beginning	On pointer
$\mu(\sigma^2)$	32.3 (48.1)	609.6 (828.1)	6.6 (7.3)	31.2 (63.8)	1.1 (1.8)	0.9 (1.5)	1.0 (1.7)	0.9 (1.6)	1.5 (2.7)	1.0 (1.7)	1.1 (1.8)	0.8 (1.5)
$\tilde{x}$	12.0	258.0	3.9	9.6	0.4	0.3	0.3	0.3	0.5	0.3	0.4	0.3
PCC (steps)	-	1.00	0.59	0.72	0.90	0.92	0.94	0.92	0.87	0.92	0.90	0.91

Table 2: Feature comparison between CoqPyt and similar tools (✓ full support, ~ partial support, ✗ no support).

Feature	CoqPyt	Coq LSP	Coq Serapy
Get proof state	✓	✓	✓
Check file validity	✓	✓	✓
Execute/Modify steps	✓/✓	~/✗	✓/~
Extract step context	✓	✗	✗
Track modules/terms/proofs	✓/✓/✓	✗/✓/✗	✓/✗/~

LeanDojo [29] and PISA [10] are learning environments for Lean and Isabelle, allowing for data extraction and proof search interaction. The neural theorem provers that use these environments are LLM-based and use retrieval augmentation [11, 17, 29]. CoqPyt would enable retrieval augmented LLM approaches in Coq.

Pretrained LLMs have been shown to have quantitative reasoning capabilities [3, 18, 27], especially when they undergo continued training on math data [1, 14]. To synthesize proofs, they can be fine-tuned on proof data [6, 10, 29], few-shot prompted [1, 12, 31], or even zero-shot prompted [24, 29]. However, the question of test leakage arises in any evaluation that uses pretrained LLMs. CoqPyt will allow for data collection after the pretraining cutoff date.

During proof development, proof engineers are constantly performing proof repair [21]. The first work in automating this task uses symbolic tools for automated proof repair in Coq [20], and has since been applied to other proof systems [15]. Baldur is the first work to use neural methods (in particular, LLMs) to repair proofs, but as part of its proof synthesis approach [6]. With the creation of a large-scale dataset of proof repair instances [19], the creation of LLM methods to automate real-world proof repairs is forthcoming.

4 EVALUATION

Feature Evaluation. Table 2 compares the features of CoqPyt to similar tools. When compared to Coq LSP, CoqPyt supports a more extensive range of features by using the data supplied by Coq LSP and constructing additional components of Coq program logic upon it. For instance, Coq LSP provides partial support for operating with steps by enabling retrieval of the proof state at any position of the file. However, it lacks implementation for navigating backward or forward from a specific step, which is provided by CoqPyt. Compared to Coq Serapy, CoqPyt offers complete support for an additional four features: (1) modifying steps – while Coq Serapy supports the addition or deletion of steps at the current point of execution, it does not offer the capability to add or delete steps at arbitrary positions within the file; (2) extract step context – Coq Serapy does not collect the terms used in each step; (3) track terms – Coq Serapy does not retain a record of the terms defined up

to the current execution point; (4) track proofs – even though Coq Serapy supports commands such as navigating to the next proof, it does not maintain a record of all the proofs already defined in a file.

Performance Evaluation. To evaluate CoqPyt's performance, we used the project CompCert [13] since it is a popular project with over 1,700 stars on GitHub and 259 Coq files with considerable complexity. We ran the experiments inside a Docker container on an Intel Xeon Silver 4210R CPU @ 2.40GHz. Table 1 summarizes our results. CoqPyt was able to run on 193 files, 58 files contained Coq-related errors, 6 files generated out-of-memory errors, and 2 files crashed for unknown reasons. The column *Count* describes metrics for the number of steps and proofs of CompCert. We ran all the features in each Coq file. For features that modify steps, we performed the modification in the beginning of the file and at the current point of execution, after executing the whole file. We performed each case 5 times for each feature. This approach is driven by the understanding that the number of required operations for these features increases as the distance to the point of execution rises. To evaluate change_steps, we performed a single addition or deletion to make it simpler to compare to the other features. The results for file execution are dependent on a cache that saves our loading of Coq libraries. As more libraries were cached, the time to execute the files decreased. By calculating the Pearson correlation coefficient, we conclude that the execution time for all features is positively correlated to the number of steps in a file. A replication package is available at: https://zenodo.org/records/10292580.

5 CONCLUSION

We developed CoqPyt, a Python tool for interacting with Coq. It offers valuable features previously unavailable in similar Coq-related tools, namely per-step context extraction and arbitrary file and proof modification. CoqPyt does not yet explore the full potential of context extraction, as it does not consider a small subset of Coq's rich environment (*e.g.*, module types and section-local constructs) nor does it fetch context recursively, *i.e.*, for each term in a step context we ignore the context of the term itself. Nonetheless, we have shown that CoqPyt provides rich features and reasonable execution times, enabling an expressive interaction style with Coq. These contributions are expected to ease the automation of machine-checked proof synthesis and repair in the emerging era of LLMs.

REFERENCES

[1] Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. 2023. Llemma: An Open Language Model For Mathematics. arXiv:2310.10631

[2] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The Tactician. In *Intelligent Computer Mathematics (CICM 2020)*, Christoph Benzmüller and Bruce Miller (Eds.). Springer, Cham, 271–277. https://doi.org/10.1007/978-3-030-53518-6_17

[3] Aakanksha Chowdhery et al. 2022. PaLM: Scaling Language Modeling with Pathways. *CoRR* abs/2204.02311 (2022). https://doi.org/10.48550/arXiv.2204.02311 arXiv:2204.02311

[4] Emily First and Yuriy Brun. 2022. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 749–761. https://doi.org/10.1145/3510003.3510138

[5] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 231 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428299

[6] 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* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1229–1241. https://doi.org/10.1145/3611643.3616243

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

[8] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *ICLR*. https://openreview.net/forum?id=r1xwKoR9Y7

[9] IBM. 2021. pycoq. https://github.com/IBM/pycoq.

[10] Albert Q. Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. LISA: Language models of ISAbelle proofs. *6th Conference on Artificial Intelligence and Theorem Proving* (Sept. 2021).

[11] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Mił oś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding Hammers to Integrate Language Models and Automated Theorem Provers. In *Advances in Neural Information Processing Systems (NeurIPS 2022, Vol. 35)*. Curran Associates, Inc., 8360–8373.

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

[13] 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*. SEE, Toulouse, France. https://inria.hal.science/hal-01238879

[14] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving Quantitative Reasoning Problems with Language Models. *CoRR* abs/2206.14858 (2022). https://doi.org/10.48550/arXiv.2206.14858

[15] Paolo Masci and Aaron Dutle. 2022. Proof Mate: An Interactive Proof Helper for PVS (Tool Paper). In *NASA Formal Methods Symposium*. Springer, 809–815.

[16] Microsoft. 2023. Language Server Protocol. https://microsoft.github.io/language-server-protocol.

[17] Maciej Mikuła, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. 2023. Magnushammer: A Transformer-based Approach to Premise Selection.

[18] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774

[19] Tom Reichel, R Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer. 2023. Proof repair infrastructure for supervised models: Building a large proof repair dataset. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2023.26

[20] Talia Ringer. 2021. *Proof Repair.* Ph. D. Dissertation. University of Washington.

[21] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL instrumentation for Coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3372885.3373823

[22] 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* (London, UK) *(MAPL 2020)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3394450.3397466

[23] 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, Article 12 (June 2023), 30 pages. https://doi.org/10.1145/3593374

[24] Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A Language-Agent Approach to Formal Theorem-Proving. arXiv:2310.04353

[25] The Coq Development Team. 2024. Coq – Release 8.19.1. https://coq.inria.fr.

[26] The Coq LSP Development Team. 2024. Coq LSP – Release 0.1.8. https://github.com/ejgallego/coq-lsp.

[27] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288

[28] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. PMLR, 6984–6994.

[29] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems (NeurIPS 2023, Vol. 36)*. Curran Associates, Inc., 21573–21612.

[30] 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* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[31] Shizhuo Dylan Zhang, Talia Ringer, and Emily First. 2023. Getting More out of Large Language Models for Proofs. *8th Conference on Artificial Intelligence and Theorem Proving* (Sept. 2023). arXiv:2305.04369