NEURAL APPROACHES TO THEOREM SEARCH & PROOF REPAIR

BY

THOMAS REICHEL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Adviser:

Professor Talia Ringer

ABSTRACT

This interdisciplinary formal methods/machine learning thesis builds neural automation for two proof-centric tasks that catalyze the reuse of existing proofs: (1) natural language theorem search, in which theorems and their corresponding proofs are retrieved from a database using natural language descriptions and (2) proof repair, in which proofs broken by external changes are mended. The theorem search model is also used as a component of the proof repair tool, allowing it to better interact with the environment. Each task is tackled holistically: we contribute datasets, fine-tuned large language models, and the end-user tools needed to make use of those models.

ACKNOWLEDGMENTS

I've shuffled the order below so that I didn't have to rank my relative gratitudes.
To Cosmo, for sharing the most refined taste in restaurants on campus.
To Hannah, for bringing me food from an event I missed.
To Kalen McGowen, for travel planning and then quite flexible travel repair.
To Tejus, Taylor, and Sahan, for helping with life stuff and keeping me sane.
To Professor Talia Ringer, for this opportunity and guidance therein.
To my family, for so much that this margin is too narrow to contain.
To Alex, for organizing skiing trips.
To Max Fan, for help in a weekend hackathon that started ProofDB.
To Anthony, for excellent explanations.
To Chris, for helpful advice.
To Wesley Henderson, Andrew Touchet, and Andrew Gardner for their tireless work [1].
To James Adduci, for letting me stay in office hours for an excessively long time.
Thanks.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Overview 1
1.2 Background 2
1.3 Reading Guide 7

CHAPTER 2 THEOREM SEARCH 9
2.1 Web Search Interface and Index 9
2.2 Local Client 17
2.3 Model and Data 21
2.4 Model Evaluation 26
2.5 Conclusion and Future Work 29

CHAPTER 3 PROOF REPAIR DATASET 31
3.1 A Proof Repair Dataset 31
3.2 Building the Proof Repair Dataset 33
3.3 Challenges 38
3.4 Conclusions 43

CHAPTER 4 PROOF REPAIR MODEL 44
4.1 Data 44
4.2 Model 48
4.3 Repair Tool 49
4.4 Evaluation 55
4.5 Conclusion and Future Work 60

CHAPTER 5 RELATED WORK 62
5.1 Datasets & Benchmark Suites 62
5.2 Theorem Search 62
5.3 Automated Proof Synthesis 63
5.4 Proof Repair 64

APPENDIX A PROOF SEARCH APPENDIX 65
A.1 Full List of Artifact Links 65
A.2 Synthetic Data Example 65
A.3 Verbatim Prompt Given to GPT3.5 to Generate Test Synthetic Searches . . 67
A.4 Binary Embedding Quantization Guarantees Proof Sketch 67

APPENDIX B PROOF REPAIR MODEL APPENDIX 71
B.1 Full List of Artifact Links 71
B.2 Related Theorems Example 71
B.3 Repair Model Prompt Example 73

REFERENCES 77

CHAPTER 1: INTRODUCTION[1]

1.1 OVERVIEW

Machine learning (ML) is coming for proofs. Recent years have seen a surge in interest in ML for proofs—one reflected by the many recent research venues [2, 3, 4], papers [5, 6, 7], tools [8, 9, 10], industrial research groups [11, 12], and funding opportunities [13, 14] centering on or prominently featuring ML for proofs. This interest is well-placed because ML for proofs draws benefits from both of its constituent fields: modern ML models are capable but untrustworthy while formal methods are complicated but convincing. In this thesis, we combine contemporary ML techniques with domain-specific adaptations to build automation for the following proof-centric tasks, focusing on the Coq [15] theorem prover:

1. **Theorem Search**: Locating specific formal proofs of relevant theorems from a (natural language) description.

2. **Proof Repair** [16]: Maintaining a working formal proof even as time changes the underlying language and dependencies of the proof.

Theorem Search helps formal methods practitioners immediately reuse that which has been previously proven by locating desirable theorems for the practitioner. At least several tens of thousands of statements have been proven in Coq, many of which are statements of general utility. However, there is no universal agreement on the notation and presentation for theorems. A search engine that exposes an interface as flexible as natural language could cut through some of the haze, helping developers utilize prior works they had no familiarity with.

Using a language model pipeline we created, to our knowledge, the first public synthetic datasets for training theorem search models that generate meaningful vector embeddings of theorems. We trained one such model using Llemma [17] as a base and found it competitive against other general purpose embedding models on theorem search tasks. The resulting model is used to create and maintain a searchable vector database of theorems. We embedded the contents of a large selection of popular libraries and designed a web interface allowing users to submit queries and discover relevant theorems from libraries they might not have even had installed. For users that need a search index over their own, custom work, we provide a Coq-integrated plugin that cooperates with our server to provide search capabilities over custom Coq theories.

[1]This chapter contains previously published work [1] that we have permission to reproduce.

Proof Repair [16] helps formal methods practitioners reuse that which has been previously proven by mending proofs which have broken due to changes in the language or environment, bringing them back to a compilable state. In contrast to proof synthesis, where a theorem is proven wholesale on request, proof repair tries to prove a theorem starting with a proof that *used* to work, and now doesn't, for some reason or another. In this setting, we also have access to at least a partial description of the changes made in the environment that could have broken the theorem. Proof Repair can take advantage of the structure of the previously working proof as well as the changes visible in the environment to essentially perform proof synthesis with *extra hints*. Need for proof repair occurs naturally as dependencies, language specifications, and requirements shift and break over time. Indeed, a proof need only be synthesized once but repaired indefinitely. Thanks to proof irrelevance, we can automatically determine if ML-generated proof repairs are correct, motivating the possibility of completely automated and correct neural proof repair to mitigate this burden.

To our knowledge, we created the first dataset of real proof repairs from Coq projects [1]. This dataset includes specialized features beyond just the text of the repair, such as the state of the proof as presented by Coq and abstract syntax trees of proof statements, which we utilized in fine-tuning the first specialized proof repair model from the pre-trained Llemma [17] model. We design a end-user repair tool, ProofDoctor, that integrates the proof repair model, the theorem search model as a means for the proof repair model to access the environment, and several domain-specific heuristics which aim to improve proof repair success. We briefly evaluate our tool by running it against a couple of real, broken projects and discussing the results.

1.2 BACKGROUND

1.2.1 Interactive Theorem Proving & Coq

Motivating Formal Methods In contrast to approaches like testing and fuzzing for determining software reliability, the field of formal methods focuses on logically proving that specifications are correct in all cases. For instance, a sorting procedure can be proven to always return a list of integers in ascending order. If such a proof is performed in an interactive theorem prover, such as Coq, then the proof can be distributed. A developer who receives this proof needs only to check that the proof compiles, that the statement that was proven matches the desired property, and have faith in the correctness of the theorem prover, supporting software, and computer hardware. If these conditions are met, then the developer can rest assured that the specified property will always hold. The list sort function,

for instance, will never return a list with any elements in non-ascending order. The strong trust that formal methods can provide is desirable in situations where software errors can lead to harm, such as software in transport and economic settings [18].

Coq The Coq [15] theorem prover is an Interactive Theorem Prover (ITP) based on the Calculus of Inductive Constructions [19]. It exploits the Curry-Howard isomorphism [20, 21] to represent propositions in its type system. In Coq, as well as many other ITPs, a theorem is a type and a proof of that theorem is a term which type-checks to the theorem's type. In practice, it is unwieldy to manually write out a proof term (e.g. "not_not" of Listing 1.1). Coq's most popular alternative to manual proof terms is LTac [22], a "tactical" metaprogramming language which allows the user to write a proof in a number of steps which build and transform a proof. Using LTac and other tactic languages, Coq gives feedback about the current state of the proof after each tactic that is crucial to determining what to do next—especially in large proofs where the bookkeeping involved becomes vast. The tactics provided by LTac vary from simplistic, such as the "exact" tactic, which requires the user to manually write out the proof term for the current goal, to highly complex automation, such as the "lia" tactic, which can solve equalities and inequalities over linear integer arithmetic completely automatically. For specialized tasks, practitioners can also define custom LTac tactics. This thesis focuses on theorem search and proof repair within the Coq language.

Complexity and Burden Though formal guarantees are desirable, obtaining them is difficult due to societal, practical, and computational issues:

1. Practitioners of formal methods are relatively few. Furthermore, formal methods as a field contains a breadth of disparate tooling and methodology, so even experienced practitioners are subject to steep learning curves when learning new approaches. As a sample of the variety available just within Coq, Listing 1.1 proves a toy theorem using 3 different proof languages contained within Coq by default.

2. Outside of decidable theories featured in SMT solvers, proving is, in general, an undecidable problem. Depending on the proof assistant, users get varying levels of support from proof automation, but frequently have to determine and transcribe the high level structure of the proof themselves.

3. Tooling and environments to support practitioners are suboptimal. This is in general true of all software, but, as we will see, formal methods lends itself to certain improvements. This thesis largely aims to improve this kind of burden.

Listing 1.1: Simple Coq example in which the booleans and boolean negation are defined. A simple proof that double negation cancels is performed using three distinct grammars for proving within Coq [15]: Gallina, LTac [22], and SSReflect [23].

```
(* Defining booleans. *)
Inductive bool := true | false.

(* Boolean negation. *)
Definition not (x:bool) : bool :=
  match x with
  | true ⇒ false
  | false ⇒ true
  end.

(* A manual proof term. *)
Definition not_not : forall x : bool, not (not x) = x :=
  fun x ⇒ match x with
  | true ⇒ eq_refl
  | false ⇒ eq_refl
  end.

(* LTac. *)
Theorem not_not' : forall x : bool, not (not x) = x.
Proof.
  intros.
  destruct x; reflexivity.
Qed.

From Coq Require Import ssreflect ssrfun ssrbool.

(* SSReflect. *)
Theorem not_not'' : forall x: bool, not (not x) = x.
Proof.
  elim ⇒ //.
Qed.
```

1.2.2 Large Language Models

Machine Learning, Contemporary Trends, and Accessibility The last several years bore witness to an extreme paradigm shift in the field of machine learning. Contemporary models used for language modeling, image comprehension, voice recognition/synthesis and other applications are now orders of magnitude larger [24], trained on orders of magnitude more data, and are generally more effective than models trained in the previous decade. As state-of-the-art language models now consist of billions (occasionally trillions [25]) of

parameters trained on trillions of tokens, parties not in possession of bleeding edge GPU clusters are not currently able to directly compete in model creation. Current top-of-the-line consumer cards with ≤24GBs of VRAM (the entire NVIDIA RTX 3000/4000 series) cannot even run inference using the common 7 billion parameter model at single-precision fidelity as 7 billion parameters at 4 bytes each requires 28 gigabytes of memory.

Thankfully, several advancements have been made which enables the participation of the broader academic community in research involving large models. Firstly, post-training model quantization has proven effective for large models, allowing the model to be stored and run in floating point and integer formats which use substantially less than single-precision floating point's 32 bits. Shockingly, 4 bit quantized models are competitive [26] despite being lossily compressed to 8x smaller than a single-precision model. Good 4-bit quantization makes it feasible to inference 33 billion parameter models on consumer hardware without storing a portion of the model outside of GPU VRAM, a workaround which dramatically reduces performance. Though 4-bit quantization makes inference accessible, it leads to problems with training: a 4-bit datatype can only take on 16 possible values, so it is not possible to make small, incremental changes to the model. Modern machine learning training techniques are still based on gradient descent and progress by making those small, incremental changes. This prompts us to use a second technique, QLORA [27], which does not attempt to apply gradient updates to 4 bit data. Instead of applying gradient updates to the weights of the model, QLORA applies gradient updates to low rank matrices that correspond to matrices in the original model and 'patch' them. Using QLORA, we can fine-tune a 7 billion parameter model within the memory of a consumer GPU, which would otherwise lead to a quick exhaustion of VRAM. Prior works such as these make the contributions in this thesis possible.

Causal Language Modelling The "Causal Language Modelling" task refers to predicting the next token (chunk of text) for a given context. By repeatedly querying a model for the next token and then adding that token to the model's context, a causal language model can write a continuous stream of text. For instance, a causal language model might complete the phrase "There are no good models," with the suffix "only practical ones."

Causal language models can come in a variety of architectures [28, 29, 30], but for the purposes of this thesis we will only concern ourselves with the highly adopted decoder-only transformer architecture [31]. Briefly, a causal language model of this kind looks up each input token in a mapping from tokens to vectors called an embedding. These vectors pass through a series of transformer blocks, which contextualize each token's embedding by querying (attending to) the embeddings of tokens before it in the sequence. Decoder-only models typically only attend to the past so that the embeddings of each token don't

change when a new token is added to the end of the sequence, accelerating causal language modelling. Finally, the last hidden layer output is multiplied against a final matrix called the "language model head", which converts the relatively small dimensional model output to a vector with one dimension per possible token. To select the next token, various sampling methods can be employed. One of the simplest is to convert the output to probabilities with a softmax activation and sample according to those probabilities.

Causal language models, when trained appropriately, can automate some tasks previously considered to be too subjective or to require too much "common sense" for a machine to handle autonomously. When the task can be textually described and the output is textual, some tasks can be automated by simply providing a language model the description of its task, examples, inputs, and asking the model to perform the task [32]. Then, the model tries to complete the output. We use language models in this thesis to perform repetitive open-ended textual reasoning, such as guessing what searches someone might use to find a certain theorem in a hypothetical search engine, that otherwise would have taken dozens or hundreds of hours to manually label for the number of samples involved. For non-textual tasks, research is ongoing into multi-modal language models which can take sound [33], images [34], and even videos [35] as input or output. In this thesis, we focus on only the text modality.

Though pre-trained causal language models are the most useful general purpose models to date, they are still not quite as reliable as a motivated human. These models are prone to *hallucinations*, which are plausible-sounding outputs that are not factually grounded. Hallucinations are present in models even after specialized training phases meant to make models more helpful (e.g. RLHF [36]). One study's investigation found that 46.4% of generated responses from several LLMs contained factual errors across multi-disciplinary testing [37]. However, in the proof repair setting of this thesis, any repair proposed by the model can be checked by Coq for correctness, making proof repair an appropriate task for current-term language models.

Embedding Models A different task benefiting from increasing ML capabilities is embedding. An embedding is a mapping of some arbitrary set of things, such as text, images, time-series, sounds, video, etc. to a vector space. We can train models which map things to vectors in a way that preserves some useful information about the things being embedded. For instance, models can be trained to embed questions and answers such that questions are near relevant answers [38]. Typically, 'nearness' is defined as increasing cosine similarity or dot product.

Since embedding models can store information that is typically hard to quantify by con-

ventional means and similarity between vectors can be calculated or estimated very quickly, embedding models are frequently run on huge collections of items to create vector databases. Then, one can run the embedding model once on a new entry (e.g. a user's question) and rapidly find similar entries (e.g. factual answers) in the database. The theorem search model we contribute follows this rough architecture.

Retrieval Augmented Generation The acronym 'RAG' was originally coined in Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks [39]. When a language model, such as Llama, answers a question without a context, it relies on the information that has been trained into it (parametric memory). Language models can also access non-parametric information through text given as context, for instance a language model can summarize the information in a news article when the text of the article is given, even if it wasn't trained on that new article's text. Unfortunately, the amount of non-parametric information a model can access is limited by model and hardware architectures. RAG attempts to put just the *most relevant* information available in the model's context by searching a vector database generated by an embedding model. In the final chapter of this thesis, we use our theorem search model to fetch the most relevant theorems for our repair model in a RAG-like way.

1.3 READING GUIDE

Chapter 1 This chapter. We reused a portion of the introduction from the previously published work predominately featured in Chapter 3 (PRISM [1]) to our overview here.

Chapter 2 We begin by discussing our work in natural language theorem search. We produce a dataset of natural language searches for Coq theorems (making some use of Chapter 3's infrastructure), and train a model that allows quick search over large volumes of proofs. We create a web client that allows anyone to make use of this service to search a central repository of common theorems as well as a local client that allows users to search custom sets of theorems in the user's current Coq environment. This chapter contains content from an extended abstract submitted to AITP which was expanded upon for this thesis:

> T. Reichel and T. Ringer, "Proofdb: A prototype natural language coq search engine," to appear in 9th Conference on Artificial Intelligence and Theorem Proving (AITP), 2024.

Chapter 3 This chapter is adapted from Proof Repair Infrastructure for Supervised Models (PRISM) [1], a previously published work in which we create crucial infrastructure for generating proof repair data, as well as general purpose tasks such as building Coq projects and parsing sentences from Coq source. The infrastructure we develop here is used throughout this thesis, especially in Chapter 4. A complete citation for PRISM follows:

> T. Reichel, R. W. Henderson, A. Touchet, A. Gardner, and T. Ringer, "Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset," in *14th International Conference on Interactive Theorem Proving (ITP 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Naumowicz and R. Thiemann, Eds., vol. 268. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.26 pp. 26:1–26:20.

Chapter 4 We elaborate upon Chapter 3 by turning the data into concrete textual prompts and model responses, training a repair model on our textual dataset, and creating a tool, ProofDoctor, that attempts repairs using the repair model. ProofDoctor includes some heuristics to improve the speed of repairs. Notably, it allows the repair model to interface with the environment using the proof search model presented in Chapter 2. We perform a brief evaluation of ProofDoctor's capabilities.

Chapter 5 Related work for the contributions in the previous chapters. This related works section contains content from both PRISM [1] and the ProofDB extended abstract [40].

CHAPTER 2: THEOREM SEARCH[2]

When attempting to locate a useful lemma, formal methods practitioners may not know a verbatim textual fragment of a desired type, but instead only an informal notion of the theorem sought-out. For instance, a practitioner working on a probablistic proof may desire a theorem which they only specify is a "concentration bound on expectation". We illustrate this search, among other examples, in Figure 2.1. To support those individuals searching for natural language notions we present ProofDB, which takes a plain-English query and retrieves relevant results from a database of formal proofs. The contributions of this chapter are

1. a novel synthetic dataset of natural language searches for Coq theorems (Section 2.3)
2. an LLM fine-tuned for theorem embedding by training on the dataset (Subsection 2.3.1) and an evaluation of that model relative to other general purpose embedding models (Section 2.4)
3. a front-end website[3] for searching a large database of theorems using the trained LLM which provides helpful metadata and links to generated documentation (Section 2.1)
4. a Coq plugin that can provide natural language search over custom theorems in a user's local proving environment, even on relatively low-resource hardware, using APIs provided through the web user interface (Section 2.2)

The interlinks between parts of this chapter are diagrammed in Figure 2.2. A list of links to artifacts, such as models and datasets, are given in Appendix A.1.

This chapter makes some use of PRISM infrastructure developed in Chapter 3 to aid in interacting with Coq source code. Then, in Chapter 4, the work in this section enables ProofDoctor to interact with the theorems present in the environment of a Coq session.

2.1 WEB SEARCH INTERFACE AND INDEX

One of our contrbutions is a web interface for proof search web. Firstly, it provides access to natural language search over a large collection of theorems (approx 90,000) from the Coq Platform [41], a comprehensive collection of common and useful Coq packages. Whenever possible, we also provide some useful metadata, such as the types of universal and existential

[2]This chapter contains previously published work [40] that we have permission to reproduce.
[3]https://proofdb.tompreichel.com

Figure 2.1: Example 'non-trivial' searches on our web UI using our search model.

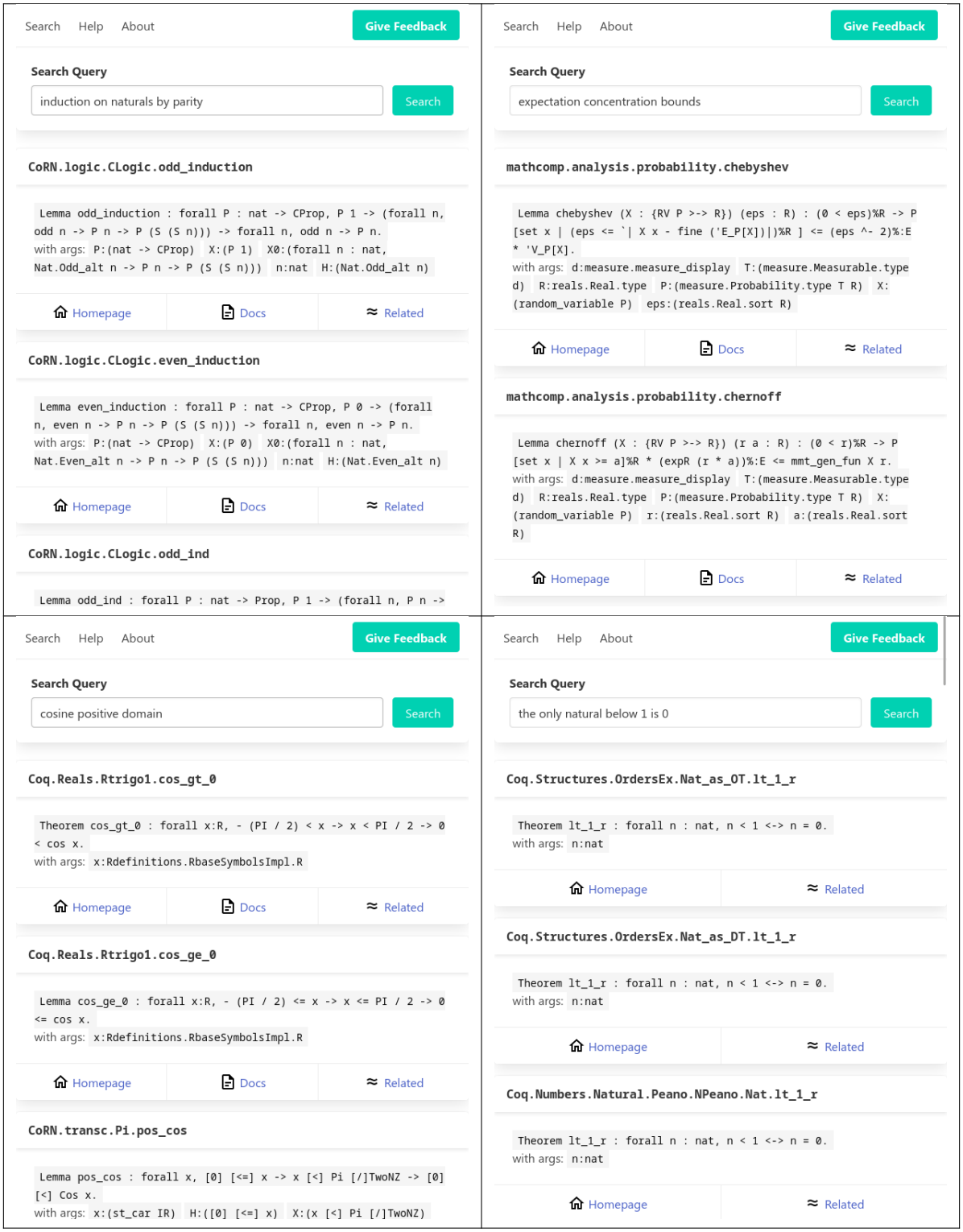
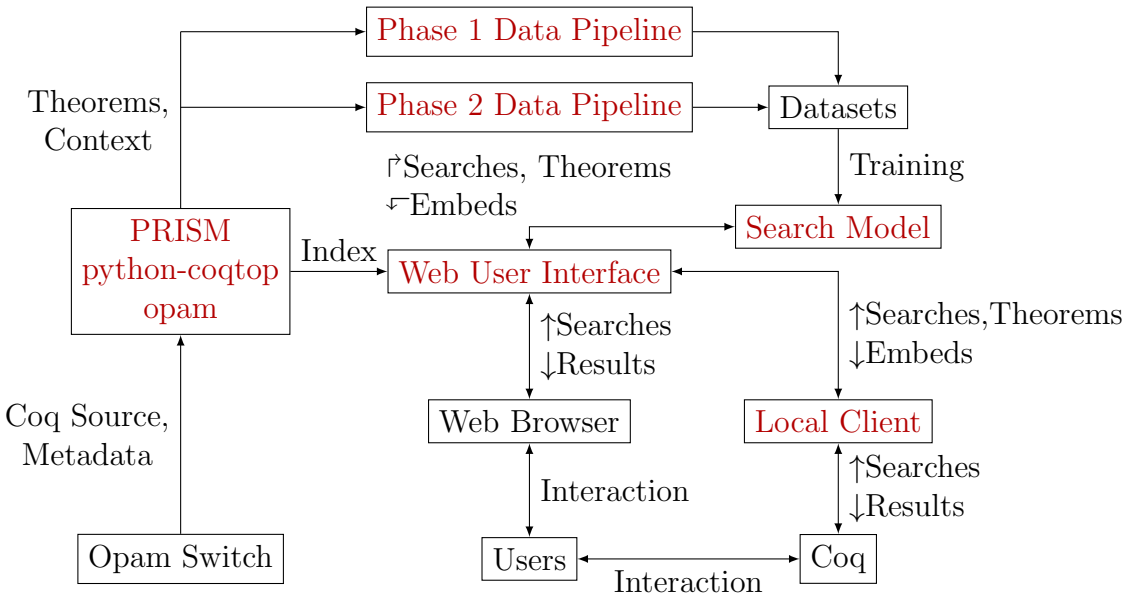


Figure 2.2: Relationships and communications between parts of this work. If you're viewing this thesis as a PDF, the components of the diagram should be mostly **clickable**.



qualifiers at the head of the theorem (users can also use metadata as a filter to e.g. find only theorems operating on natural numbers), links to the homepages of package associated with each theorem, and links to synthetic Coqdoc documentation pages. We designed automation to gather all of this information from Opam switches with Coq packages installed so that the search index can be amended and updated. In this sub-section we describe the methods by which information is gathered to create the index. Sources for these methods and the web search interface itself are available in the appendix A.1.

2.1.1 Interacting with Coq Using `python-coqtop`

In the course of extracting theorems to build the index, we must programmatically interact with the Coq interpreter. Several approaches exist with differing capabilities and trade-offs:

1. coq-serapi [42], a long-standing tool intended for use-cases like our own, which provides machine parseable output for a variety of queries. PyCoq [43] exists as a python wrapper for coq-serapi. For the majority of the duration of this thesis's work, coq-serapi had been announced to be deprecated [44] in favor of the new coq-lsp.

2. coq-lsp [45], a Language Server Protocol implementation for Coq with approximate feature parity to coq-serapi as well as new features to support IDEs and other use-cases. CoqPyt [46] exists as a python wrapper for coq-lsp. For the majority of the

duration of this thesis's work, coq-lsp was in active development.

3. coqtop.py [47], which is a simple wrapper around Coq's command-line read-eval-print-loop coqtop, using pexpect.

Due to the period in which the work was completed, we implemented a bare-bones wrapper around coqtop, Coq's command line REPL, similar to coqtop.py [47]. The main differences between `python-coqtop` and coqtop.py are as follows:

1. `python-coqtop` takes advantage of Python's asyncio library. When a Coq command is run, `python-coqtop` yields control of the thread until results are available so that other coroutines can run. This allows a single process using `python-coqtop` to manage input and output for several Coq subprocesses concurrently without multithreading or multiprocessing by attending to each process as it waits for results from the others.

2. `python-coqtop` can write multiple commands into the standard input of a Coq subprocess without waiting for commands to finish, then determine which parts of the resulting standard output and standard error pertain to each command that was run. Tasks that needs to unconditionally run several commands in a sequence but examine the output of each individually could benefit from decreased overhead.

`Python-coqtop` was used for some miscellanous research tasks as well as fundamentally in the web search database setup script which creates the index for the web search interface. `python-coqtop` is also used as the primary means of interacting with Coq in ProofDoctor (Chapter 4). Notably, the core feature of 'running commands and getting stdout and stderr for those commands' seems to work even on the oldest and newest versions of Coq available in opam, which suggests this is a relatively stable interface. We frequently use a utility from PRISM [1] of Chapter 3 to preprocess Coq documents into a list of sentences before running them with `python-coqtop`, since PRISM's heuristic sentence parser (Subsection 3.3.2) is robust to comments, some strange vernacular syntax[4], and handling bullets in proofs.

2.1.2 Opam Metadata

The opam package manager keeps track of which files are installed into an opam switch by which packages. By requesting this list for each package installed by way of invoking `'opam show --list-files'`, we obtain a mapping from Coq files to packages. By recording the

[4]For instance, it is possible to define vernacular syntax that allows the text "`*)`", which usually denotes the end of a comment block, to appear in valid source code without opening a comment block.

files we find each theorem in, we can look up the corresponding opam package from our mapping and associate each theorem with its package. This allows us to take advantage of metadata that opam stores about each package, such as a project's home-page, obtainable by invoking `'opam show -f [field] [package]'`. We display some of this information in our web-UI, such as a link to the registered homepage of the package which we indexed the theorem from. This acknowledges the package's authors and allows users to download that package or view other resources the developers make available for them.

2.1.3 Indexing Theorems

Typically, when a Coq package is installed into a switch, the source files are also installed. Indeed, for a distribution of the Coq platform [41] for Coq 8.18, released September 2023, there are 8,410 `.vo` files (compiled Coq files). 8,166 of these have a corresponding `.v` Coq source file of the same name in the same directory for a satisfactory 97% coverage rate. Due to the high availability of installed source files, we use them as part of our indexing pipeline. First, we use `coqdoc` to process each source file and remove the bodies of all the proofs, leaving just theorem statements, definitions, and other organizational structure behind. Then, we use hand-written regexes to pick out theorem statements and module boundaries.

> **Remark 2.1.** Module boundaries are of particular importance: a theorem written within a module cannot be accessed from outside that module by default, except by providing the path into the module, such as in the case of `Nat.add_comm`. If we do not properly reconstruct the path of a theorem, we cannot tell anyone how to use it!

This gives us a list of the theorems explicitly stated within a file. Combining the file path, module boundaries, and theorem names give us the full logical path of the theorems we find in the file. However, other theorems exist which are never explicitly written in a source file. There are a myriad number of other ways for a theorem to enter the environment:

1. Coq automatically defines an induction schema as a byproduct of defining any inductive type.

2. Built-in Induction/Boolean Equality/Inversion such as `Scheme` and `Derive Inversion` can automatically generate theorems.

3. Functors (parameterized modules) can be instantiated multiple times, creating an arbitrary number of theorems in a single command.

4. Coq supports low-level OCaml extensions which can make arbitrary changes to the environment. Many of these automatically generate theorems.

It is clear that we need to account for these in some way. Since we are indexing parts of built Coq libraries, our approach is to try importing each of these files and using the built-in `Search` command to list all of the theorems apparently coming from that files. We also use this as an opportunity to sanity check the theorems we found explicitly within the source code, as they should also appear in this list, and removing those which are mistakes.

> **Remark 2.2.** The reader might have noticed that we should technically find all of the theorems explicitly written in the source two ways: first by extracting their plain-text and secondly by using the built-in `Search` command. Even though we could rely solely upon the `Search` command and find all of the theorems we already found in the source, we prefer to use the plain-text that was written by the developer whenever possible over the output of `Search`, since the two outputs frequently differ. Coq's highly extensible notation system allows developers to add new syntax to the language, even syntax that conflicts with other defined syntax. Conflicting syntax is usually stored in different scopes, which are enabled and disabled at will. Since we don't know which scopes are preferable for working with theorems, we defer to however the author of the theorem actually wrote the theorem. For instance, in the default scopes, the theorem `Coq.ZArith.Zpower.Zpower_exp` is printed as follows:
>
> ```
> Zpower_exp:
>   forall x n m : Z,
>   (n >= 0)%Z ->
>   (m >= 0)%Z ->
>   (x ^ (n + m))%Z = (x ^ n * x ^ m)%Z
> ```
>
> But the plain-text version we find in the source files and reproduce in the web-UI does away with the ugly repetition of `%Z`, all without having to find and open the 'Z' scope first, since we simply take the theorem as the author wrote it.

One limitation here is that we do not currently gather *all* theorems present in a switch. A small percentage of files yield errors– possibly attributable to custom Coq plugins which

might be present, or inconsistencies between the assumptions we make about Coq's syntax and the kinds of custom vernacular notations introduced in libraries, or bugs.

2.1.4 Indexing the Types of Theorem Arguments

As stated previously, we prefer to use the plain-text of theorems as they were written by the developer whenever possible since it does away with some notational issues. However, it also introduces inaccuracies caused by developer shorthand. Developers frequently make use of Coq's implicit typing. Consider the following theorem, whose text was taken from source code:

```
Theorem add_comm p q : p + q = q + p.
```
(2.1)

Does this theorem refer to the naturals? Could it refer to rationals, complex numbers, or the reals[5]? We cannot tell from the text alone. The author of this theorem allowed Coq to infer the types of the arguments and we cannot tell which of several definitions of '+' is being used here. We don't want to lose the author's choice of notation scopes, but we also want to know the types of arguments even if the author did not write them. This motivates us to take a pass over our theorem index and gather the types of all theorem arguments.

In order to index the types of theorem arguments, we import each of the files and make use of Coq's embedded LTac tactic language. LTac ($\mathcal{L}_{tac}$) is primarily intended to ease proof development by providing automation. However, since $\mathcal{L}_{tac}$ has the capability to examine and manipulate Coq terms, we can also use it as a metaprogramming language in order to extract information about Coq terms. To determine the types of the arguments of a theorem of interest, we perform the following steps using a `coqtop` session:

1. Start a proof of `True` so that we can begin using $\mathcal{L}_{tac}$.
2. Pose the theorem of interest as `H`.
3. Cut[6] the type of H, then discard `H`. This creates a secondary goal to prove the theorem of interest. We do this because the LTac has more capabilities when it comes to manipulating the goal of a proof (that is what LTac is primarily intended to do), so we want to make the current goal the theorem of interest.
4. Prove the first, trivial, goal, leaving us with just the second goal, which is to prove the theorem of interest.

[5] It turns out that `p` and `q` are binary positives.
[6] take as assumption, but with the obligation of proving the assumption later

5. Repeatedly try to introduce hypotheses/arguments or existential variables, which eliminates them from the goal and places them into the proof context, until no further progress can be made.

6. Print the names and the types of all the things we just added to the local context. This is accomplished by attempting to match anything in the proof context, then forcing a failure. $\mathcal{L}_{tac}$'s `multimatch` is capable of backtracking, so it will proceed to the next thing in the context, print it, fail, and so on, until nothing is left to print. We parse the resultant output in Python.

The pseudo-routine above can be seen written concretely in Coq/$\mathcal{L}_{tac}$ in Listing 2.1. This routine can extract arguments and existential qualifiers from theorems if the theorem begins with them, but does not have the capability to list arguments which are bound deeper into the expression, such as in the case of `(exists x, P x) /\ (exists x, Q x)`, as we only provide a way to eliminate existential and universal quantification that occur at the beginning of a theorem and the outermost (head) symbol of the expression is the 'and' (`/\`), rather than a forall or exists.

Listing 2.1: Printing the names and types of a theorem's arguments using $\mathcal{L}_{tac}$.

```
(* 1. *) Goal True.
  (* 2. *) pose (theorem_name) as H.
  (* 3. *) let Htype := type of H in cut Htype; clear H.
  (* 4. *) 1:intros. 1:apply I.
  (* 5. *) repeat (try intros; try match goal with
                                  | ⊢ exists unnamed : ?H, _ ⇒
                                      let ident := fresh unnamed in
                                        evar (ident:H); exists ident
                                  end).
  (* 6. *) tryif (multimatch goal with
                     H:?T ⊢ _ ⇒
                       idtac H ":·:" T; fail
                   end) then idtac else idtac.
Abort.
```

2.1.5 Coqdocs

In order to ground our search results, we'd like to be able to provide a link into Coq's automatically generated documentation (Coqdocs) from our web interface. The coqdocs preserve the comments and surrounding definitions of the source, and if provided with the

necessary information, link each identifier in the source to its definition, allowing a user to investigate the components of an unfamiliar theorem. Unfortunately, the `coqdoc` utility which generates these files needs information beyond source code in order to generate links to the definitions of identifiers. By default, when Coq source is compiled, a "globalization" or "glob" file is produced as a byproduct. This glob file annotates the origins of each identifier in the source. `coqdoc` only produces links to the definitions of identifiers if it is provided with the appropriate glob files and told how the directories of the filesystem map to actual import names by use of command line flags. Glob files are not usually installed alongside source files, so the globalization information we need is lost after the packages of interest were built. Thankfully, it is possible to direct opam to rebuild all of the packages in a switch and save the build directories, which enables access to all of the glob files for the Coq packages installed in a switch. Finally, we generate a mapping from the build directories to their ultimate import names by observing that the first line of a glob file describes the intended import name for that file and using a small script to generate the mapping.

> **Remark 2.3.** A limitation of this approach is that we rely on the internal structure of glob files, which is considered a legacy format. The Coq developers provide no guarantee of stability for the format and ongoing work seeks to replace it with a "more structured and strongly-typed API" [48].

Combining all of this information allows us to issue a single `coqdoc` invocation which builds switch-wide, interlinked documentation that we link to in our web user-interface whenever possible for each search result.

2.2 LOCAL CLIENT

Some informal user feedback we got on ProofDB was that it was not helpful in ongoing research projects where the majority of theorems developers wanted to locate were not in popular libraries but elsewhere in the work-in-progress project they were editing. For these users it would be ideal if we could search the environment that exists in their Coq interpreter at the moment the search was run. Since we would like to provide the client to users with low-resource machines, running the 7 billion parameter language model on the client's computer was not considered.

> **Remark 2.4.** As consumer hardware changes, it may become possible for the client to become self-contained and not rely on an external API. Indeed, in late May

2024, Microsoft announced upcoming AI features for end-users which have **minimum** hardware requirements including a 40 trillion operation per second 'neural processing unit' [49], which indicates that near-term user hardware may not need to rely on an API for neural proof search. For now, the client relies on API access to a running instance of the web search.

Building this local client, even with an external API, poses some challenges:

1. We must have an integrated client within the user's running Coq interpreter, which is challenging in-of-itself.

2. We prefer to not have the server persistently record the theorems associated with each client to perform searches over a client's custom theorems for them. This would make our server very stateful, which is undesirable. Ideally, the client would manage its own embeddings, but transferring the embeddings can be costly: 4000 theorems embeddings naively transmitted would weigh 65 MB at 4096 single-precision dimensions per theorem.

3. In order to search the theorems present in a user's running interpreter we must first embed them. The list of theorems could number in the thousands. Embedding them on the fly could take minutes. This is a bad user experience.

We address these challenges below.

2.2.1 Integrated Client

Coq has a plugin system that allows arbitrary external OCaml plugins to be loaded at runtime and interact with the language, usually by providing new commands. In our case, we complement the existing `Search` command with a `NLSearch` command. We hook into the existing search command's logic to provide access to features that end-users are probably familiar with, particularly the set of filters the built-in search command provides, such as filtering for theorems that contain a particular term. `NLSearch` takes a single additional string argument which acts as the natural language query. Any additional arguments are passed to the existing search machinery to produce initial search results, then those results are ranked using the natural language search parameter.

The client makes an HTTP request to our API. For end users that have their own server running, it is possible to specify a custom API endpoint URL without recompiling the plugin. The client sends the server a list of theorems it needs embedded as well as the

natural language search the user is making. Then, the embeds received from the server are cached so that they do not have to be retreived again for the remainder of the session. To complete a search, the client calculates the Hamming distance between the embedding for the search query and the embeddings for the theorems, which is not prohibitively costly[7].

2.2.2 Quantizing Embeddings to Save Bandwidth and Compute

Just as our model can be quantized to less than single-precision and still function adequately, our embeddings can be quantized to lower precision to allow faster transmission and computation at the cost of some quality. We follow a method described by Xinyang Yi, Constantine Caramanis, and Eric Price in their paper on the fundamental limits of binary embeddings [50]. In this setting we assume we are working with unit vectors on the surface of the unit hypersphere. The paper suggests generating k random planes that cross through the origin and divide the unit hypersphere in two. Each vector is quantized by recording a 1 or a 0 for each plane, depending on which side of the plane the vector is on. By varying k, we can preserve variably more or less bits of information from the vectors. The paper goes on to show that the expectation of the Hamming distance between the generated binary vectors is proportional to the angular distance between the original vectors, which is the metric we use to fetch results in the non-quantized case. Finally, the authors use a concentration bound to get strong probablistic guarantees on the amount of inaccuracy introduced by quantization. Since the authors do not provide a very explicit proof of this, only an outline, we give a sketch in Appendix A.4 which concludes with a formula to help pick k.

> **Remark 2.5.** Several tech companies in the industry have announced support for binary embeddings [51, 52, 53], and all seem to use roughly the same approaches, such as retraining the embedding model to emit vectors better suited for quantization or storing the full vectors alongside the binary quantized vectors to use for re-ranking the top search results using the distances between the full-precision vectors. Usually, the embeddings are quantized by converting each dimension of the vector into a '1' if it is positive and '0' otherwise, yielding 1 bit per dimension. These approaches have some downsides, such as needing to retrain the embedding model for quantization or needing to also store the un-quantized vectors. Most have weak guarantees on accuracy relative to the original embeddings, which are

[7] Hamming distance was computed in under 3 seconds between a search embedding and 14,000 theorems embeddings obtained by importing `Arith`, `List`, and `Reals` simultaneously on Coq 8.17. We used a single CPU thread on a AMD Ryzen 7 1700

unappealing to us. Picking an appropriate number of planes following the theory in Xinyang Yi et al.'s paper [50] seems to allow us to side-step all these issues adequately.

2.2.3 Optimizing Embedding Latency

When the user makes a search, they have to wait for our server to embed the theorems in their environment. Our first line of defense against latency are embedding caches on the server and client side, which are preferred over generating or requesting new theorem embeddings. Nonetheless, small changes in the client's code, such as functor instantiation, can result in hundreds of new theorems that must be embedded quickly to be considered in a search. Ultimately, we may still have to generate embeddings quickly, no matter how well we cache. We use a couple of small optimizations on the server side in order to more rapidly embed theorems. Batching is a well known optimization which reduces the amount of times the parts of the model has to be loaded by inferencing several inputs at once. This can greatly increase the speed of inference, but increasing batch size comes with the following negatives:

1. Increased memory usage proportional to the number of simultaneous inferences performed in one batch.

2. Under typical implementations such as Huggingface's transformer implementations, the sequences in a batch must all be padded to the same length. Padding without truncating makes every sequence as memory-intensive as the largest sequence in the batch. Furthermore, although the padding is ultimately inert w.r.t. the final output of the model, it imposes computational cost comparable to additional non-padding tokens[8].

We want to perform large batch inference, but we do not want to run out of memory and we want to avoid padding the inputs. To avoid running out of memory, we vary the size of batches depending on the length of the largest sequence that would be included in the batch. When the sequences are short, we can have a larger batch, and when they are long, we take a smaller batch. To minimize padding, we sort the sequences by length and run inference on them in the sorted order. This means that the sequences that are batched together are all around the same length, reducing the amount of padding required.

[8]Presumably because PyTorch operations are typically not specialized to deal with sparsely occupied tensors and simply compute over the entire tensor, then mask out the garbage data later.

We found a prior work that describes these same alterations to batching strategies and confirms a speed-up [54]. As a quick replication, we embed a dataset of 1000 strings consisting of 15 to 100 random combinations of the words "rock", "paper", and "scissors" using a 7 billion parameter model. We embed it using three methods: batch size 1 embedding, batch size 4 embedding, and then the method we describe above, restricted to produce batches with fewer total tokens (including padding) than any batch the "batch size 4" method produces.

Figure 2.3: Table of embedding benchmark times for various batch sizes compared to the optimizations we describe.

Method	Batch Size 1	Batch Size 4	Simple Opts
Time (m:ss)	3:49	2:48	**1:53**

Evidently, these adjustments can shave some time off of embedding, even when restricted to embedding as many tokens in a single batch as the batch size 4 trial.

2.3 MODEL AND DATA

The core functionality of this project depends on a way to produce semantically rich embeddings from the text of theorems. Powerful commercial and open-source general purpose solutions exist to embed arbitrary text, but we estimate that formal methods is a sufficiently niche field that fine-tuning a model specifically for theorem search would improve performance over general purpose models. We provide (limited) benchmarks to test this hypothesis in Subsection 2.4. The main hurdles are as follows:

1. Selecting a (appropriately pre-trained) model architecture.
2. Creating dataset(s) to improve the model's performance.

We ultimately trained a large language model on two phases of synthetic data, one after the other:

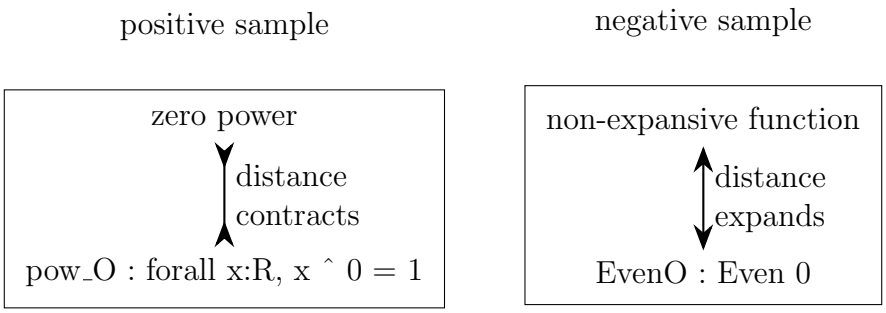
1. Simple contrastive data consisting of tuples of (THEOREM, SEARCH, LABEL) where label represents if the theorem and search are deemed to be related (positive, label is 1) or unrelated (negative, label is 0). The model learned to generate embeddings of search and theorem which were close if the label was positive and far otherwise.
2. A triplet dataset consisting of tuples of (POSITIVE THEOREM, NEGATIVE THEOREM, ANCHOR SEARCH) where we deem the search anchor to be "more related" to the

positive theorem than to the negative theorem. This teaches the model to generate embeddings such that search is closer to the positive theorem than the negative theorem.

2.3.1 Model

Current trends in machine learning [55, 56] push the state of the art with training sets and models exponentially larger [25, 57] than those conventionally used in previous decades. Training contemporary competitive models from scratch is prohibitively expensive[9], but researchers with relatively modest resources can leverage advances in resource-efficient training techniques [27] to fine-tune large models pre-trained on a wide domain of tasks to excel in a narrow domain. These trends led us to fine-tune the Llemma [17] language model, which was trained on a dataset of mathematics and formal methods texts, including Coq code, making it an appropriate base for learning insightful representations of Coq theorems. Notably, Llemma is a causal language model, not an embedding model. It was trained to predict successive tokens of text, not to produce semantically rich embeddings. In order to take advantage of Llemma's capabilities for embedding, we train its final hidden layer output to behave as an embedding by training on the datasets we describe below.

Figure 2.4: A positive and negative example from our phase 1 dataset. The objective function contracts the distance in a positive sample and expands it in a negative sample.



2.3.2 Generating Simple Contrastive Data

To train our model we need a dataset that captures notions of relevancy between theorems and natural language searches. As far as we are aware, no such prior dataset was available. To make one, we used infrastructure from Chapter 3 to extract a list of theorem definitions

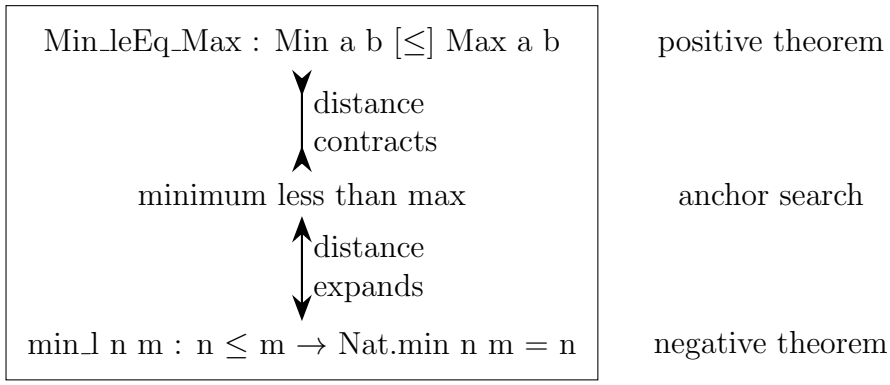
[9]Training the Llama-2 7B param model took 184,320 hours of GPU time on Nvidia A100s [58] which would cost over $200,000 if bought at Lambda Lab's current cloud prices.

from files and a window of context consisting of the source code around those theorems. Using this context, we prompted an open-weight LLM (33B parameter Code Llama [59]) to try to write an explanation of what each theorem meant intuitively, in natural language. When the context window did not contain at least a minimum threshold of comments from the developers, we discarded those theorems, since we expected the model to perform better when given a 'hint' from the developer, rather than solely interpreting Coq source. The synthetic summarization attempt along with the theorem definition was then used to prompt that same model to try to write searches a human being might try in order to retrieve a target theorem (data example in Appendix A.2). The generated searches constituted the positive samples– searches and theorems that are related. Negative examples were generated by randomly associating a theorem and search query from the pool of all theorems and search queries, which are unrelated on average. Note that we did not expect the synthetic data pipeline to produce consistently good results, just to produce results that were right frequently enough to contain more signal than noise at training time. We fine-tuned Llemma on this dataset using a loss[10] which moved the embeddings of positive samples closer together in the embedding space and negative pairs of text towards orthogonality, which gave us our first iteration 'proofdb' model. This is a kind of contrastive learning [60]. This dataset is linked in Appendix A.1 as the phase 1 dataset.

This approach has flaws. Prompting the model for "searches a user would use to find this theorem" often leads to technically right but semantically useless searches such as "coq theorem", "mathematical fact", or using the theorem's exact name in the search, which isn't what users would typically require a natural language search for. Additionally, the loss objective function has some strange consequences: the only way for the model to achieve 0 loss on a positive example is for the vector embedding of a search to be exactly equal to the accompanying theorem's embedding. Since we don't expect a search query to contain all of the information in the theorem, we don't expect the theorem embedding to be exactly equal to it! Finally, the negative examples we generate by randomly picking theorems and searches are not particularly informative to the model, since randomly picked searches and theorem are already far apart on average. We continued training our model on a smaller dataset that addresses these shortcomings in the next subsection.

[10]We used a highly non-standard loss which labelled positive theorems wih a value of 1 and negative theorems with a value of 0. Loss was calculated as the mean squared error of the cosine similarity of the pair from the label, so positive pairs moved closer together and negative pairs were pushed closer to orthogonality. We recommend using a standard contrastive loss for future experiments.

Figure 2.5: Pictoral representation of a triplet from the phase 2 dataset. The loss function contracts the distance between the anchor and the positive or expands the distance between the anchor and negative until the anchor is sufficiently closer to the positive than the negative.



2.3.3 Generating Triplet Data

We wish to resolve the following problems with the previous dataset:

1. Negative samples are easy to distinguish from positive samples because they are randomly selected. Literature suggests that negative samples should be 'close to positive' or 'hard' for good results [61, 62, 63].

2. Generated searches are frequently much more vague or specific than a search a user would actually try.

For this dataset, we move away from positive and negative samples and towards a triplet [63] format. Instead of judging that pairs of searches and theorems are absolutely positively or negatively related, we try here to synthesize searches which are relatively more related to one theorem than another and train the model to move the search's embedding closer to the positive than the negative theorem. Intuitively, triplets teach the model to correctly rank two theorems for a particular search. The structure of a single entry in our dataset is now (POSITIVE THEOREM, NEGATIVE THEOREM, ANCHOR SEARCH). An illustrated example can be seen in Figure 2.5.

We illustrate the set of searches we try to synthesize for a pair of theorems as follows. Let $P(M(t) = x)$ be the probability of a language model generating a synthetic search x for some theorem t. Let x_1 and x_2 be theorems. Consider the following set of searches:

$$\left\{ s \mid e^{16} > \frac{P(M(x_1) = s)}{P(M(x_2) = s)} > e^{9} \right\} \tag{2.2}$$

This is the set of searches that are "more probable" for some theorem x_1 than some other theorem x_2 but not excessively more probable. In practice, we generate a search just as we did

before, targeting just the positive theorem, while simultaneously measuring the probability that the current generation would have been generated if the model were prompted to write a search for the negative theorem instead.

> **Remark 2.6.** We apply a number of heuristics during search generation in an attempt to more frequently synthesize good, challenging triples with probability ratios between these thresholds. These include refusing to generate any token which would immediately increase the probability ratio by a large amount (often the exact title of the positive theorem), sampling multiple outputs, and pruning the generation of partially completed outputs that already exceed the probability ratio of a previous try in favor of another attempt.

To explain our choice of probability ratio thresholds, e^9 and e^{16}, we illustrate the relationship between the probability ratio and the quality of the data, we present some triplets with varying probability ratios in the following table.

Figure 2.6: Several examples of triplets from the dataset, roughly grouped by magnitude of probability ratio.

Theorem x_1 (positive)	Theorem x_2 (negative)	Generated Search	Pr. Ratio
lcmsr0 : (@lcmsr R) [::] = 1	lcmr0 a : lcmr a 0 = 0	least common denominator	0.92
Qpower_1 : ∀ n, 1^n == 1	pow1 : forall n:nat, 1 ^ n = 1	power of 1	1.10
Qle_refl x : x ≤ x	Qlt_irrefl x : x<x	reflexive property of Qle	64,000
Rinv_opp r : / -r = - /r	Rinv_0 : / 0 = 0	reciprocal of a negative real number	526,000
Zabs_div_rht : ∀ a : Z, $\frac{a}{\text{Z.abs } a}$ = Z.sgn a	Zquot_0_r a : $\frac{a}{0} = 0$	Z.sgn a = $\frac{a}{\text{Z.abs } a}$	1.66e15
div_le_0 : ∀ p x, 0 ≤ x → 0 ≤ $\frac{x}{2^p}$	dvdnn m : m %| m	0 ≤ $\frac{x}{2^p}$	1.99e15

We make the following observations:

1. When the probability ratio is close to 1, as in the first group of rows of the table, the model would have given the generated search for either theorem A or B at the same rate. In other words, the generated search shouldn't distinguish the two theorems. It's possible that the positive and negative theorems are indistinguishable to the search generating model or the search that was generated was very vague. Either way, this is not an informative triple.

2. When the probability ratio is very high, as in the last group of rows of the table, the model has generated a search which it would only say for the positive theorem and almost never for the negative theorem. The search must be much more specific to the positive theorem than the negative theorem, which is a case of an "easy negative".

We aim between these two extremes. By excluding the low probability ratio case, we filter out vague searches. By excluding the high probability ratio case, we filter out "easy negatives" and unrealistically specific searches. We posit that the resulting dataset is more informative than our previous try, although much more computationally expensive to generate.

This process is essentially a variant of "hard-negative mining" used broadly in contrastive learning. In "Hard negative examples are hard, but useful" [64], Hong Xuan et al. note that, when training using triplet loss, negatives should be hard, but not too hard, for best results. This is analogous to our upper and lower bounded filtering. In this setting, however, we benefit from being able to generate anchors for triples on demand and estimate the resulting hardness using the probability ratio, since our dataset is partially synthetic.

As is the case with everything involving a language model, this is not a perfect solution. When looking for these examples, we found one particularly confusing triple which ended up in the "goldilock's zone" even though the two theorems involved were identical modulo renaming the theorem and variables. We are doubtless that there are more.

2.4 MODEL EVALUATION

To evaluate our model, we run searches on a set of held-out test theorems and count how often the model can retrieve the intended label theorem within the top 10 theorems. We evaluate other embedding models in the same way to compare against our models. For our evaluations we provide 95% confidence interval error-bars on the proportion of intended theorems retrieved in the top 10 results as calculated by SciPy's [65] `binomtest`. We consider two of our models, both trained on the simple contrastive data and one also trained on the triplet data, against a suite of general purpose embedding models. We estimate that our models are somewhat better at retrieving intended theorem results than general purpose models on our human-written test searches regardless of whether they were also trained on the triplet dataset or not. Contrarily, we find that training on the triplet dataset seems to be required to outperform the general purpose models on the synthetic test searches. Our results are limited by factors described in Subsection 2.4.4.

2.4.1 Models Included in Evaluation

Here are the list of models we evaluate:

1. Our models.

(a) proofdb, trained on the phase 1 dataset (Subsection 2.3.2) , to move synthetic positive (THEOREM,SEARCH) pairs closer together and sampled negative pairs further away.

(b) proofdb-HN, the result of continued training on the phase 2 dataset (Subsection 2.3.3), consisting of triplets of (THEOREMA,THEOREMB,SEARCH), to move the search closer to theoremA than it is to theoremB, starting from 'proofdb' as a pretrained base.

2. Models featured on the SentenceTransformers pretrained models page [66], described as general purpose embedding models and trained on at least 1 billion training pairs.

   (a) all-mpnet-base-v2 [67], best overall performer on Sentence-Transformer's (selected) pretrained model list.

   (b) all-distilroberta-v1 [68]

3. Models featured on the SentenceTransformers pretrained models page [66], trained for retrieval tasks specifically.

   (a) multi-qa-mpnet-base-dot-v1 [38], best performer on the SentenceTransformer pretrained models page when evaluated on semantic search, second best overall.

   (b) multi-qa-distilbert-cos-v1 [69]

   (c) multi-qa-MiniLM-L6-v2 [70]

4. OpenAI's text embedding models (proprietary).

   (a) text-embedding-small

   (b) text-embedding-large

5. Models from the MTEB [71] leaderboard.

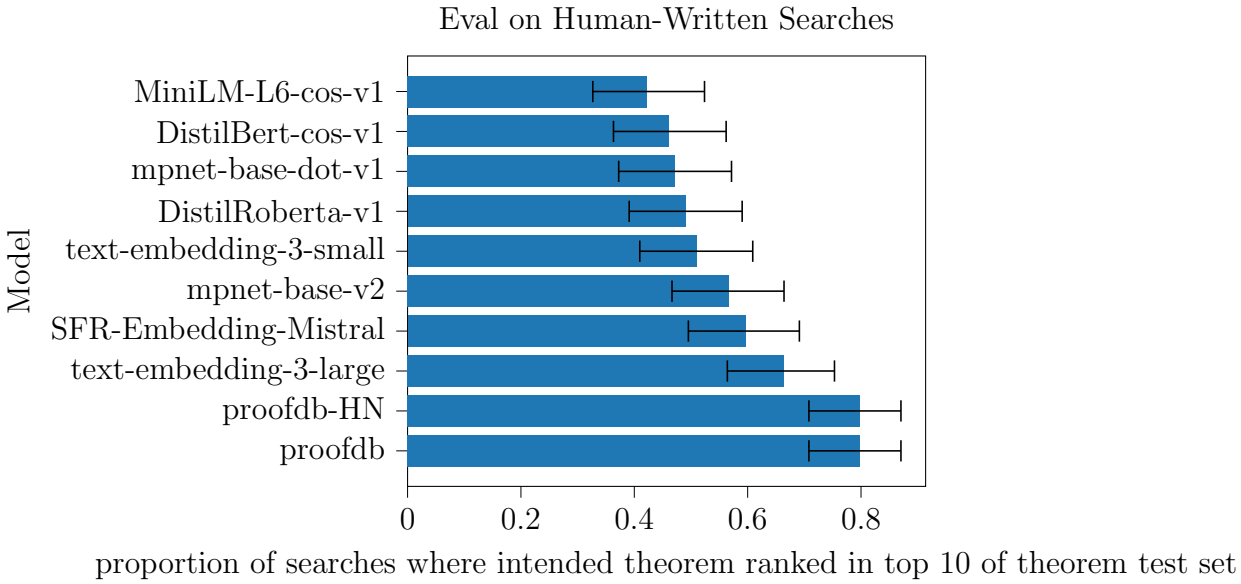
   (a) SFR-Embedding-Mistral [61], currently the best general purpose text retrieval model on the MTEB.

2.4.2 Human-Written Search Evaluation

Our models perform promisingly well on our small scale human written search dataset, even against much larger embedding models such as OpenAI's text embedding models (text-embedding-3-*), and SFR-Embedding-Mistral, which is currently ranked first on the Massive

Text Embedding Benchmark [71] for text retrieval. However, these test results have limited generalizability because they were written by one author on a small number of arbitrarily chosen theorems. The same author also provided the few-shot example used in the context of the positive example synthesis pipeline, so the generation of the training set may be disproportionately prone to the biases of the author. The human written searches used for this test are linked in Appendix A.1.

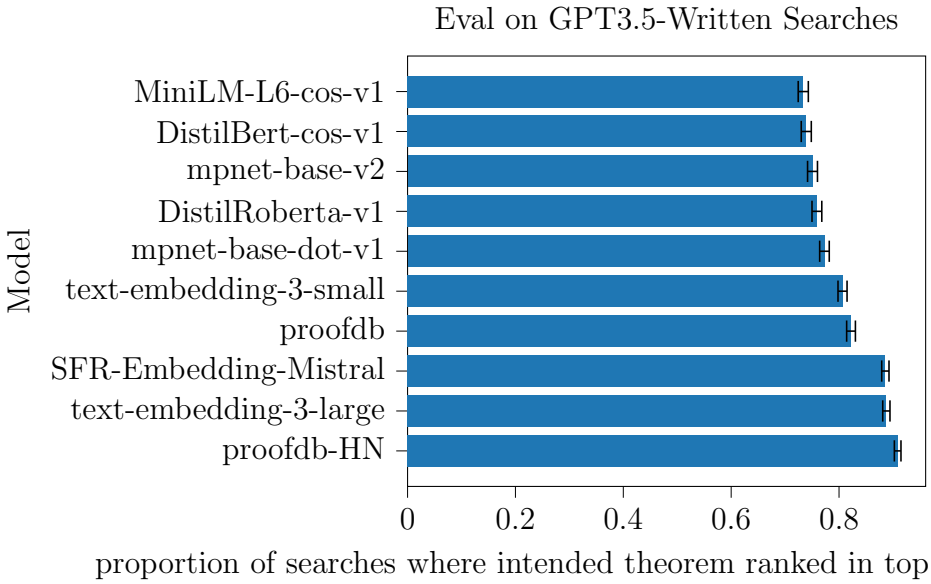
Figure 2.7: Comparison between our model and several general purpose models, evaluated by top-10 retrieval rates of intended theorems on a human written search set.



2.4.3 GPT3.5-Written Search Evaluation

For a larger scale test, we let GPT3.5 write searches for the theorems in our test split. Note that we did not give GPT3.5 an example so as to minimally contaminate the output with our preferences. The precise prompt for GPT3.5 is given in Appendix A.3, and the searches generated are given in Appendix A.1. For the most part our synthetic trial agrees with the small scale human trial. Notably, proofdb without hard-negative fine-tuning (the phase 2 or triplet dataset) does not perform as well as human written evaluation would indicate. We are unsure if this is a quirk of synthetic trials or a quirk of human trials, but we plan to conduct more thorough evaluation of human search preferences at a later date. Regardless, proofdb-HN performs well on both human-written searches and synthetic searches.

Figure 2.8: Comparison between our model and several general purpose models, evaluated by top-10 retrieval rates of intended theorems on a synthetically-generated search set.



2.4.4 Limitations

It should be noted that a limitation of both of these evaluations is that the test set may have theorems that are very similar to theorems in the training set: many Coq libraries prove results present in other libraries in only slightly different ways, so multiple similar variants of common theorems were likely split into both test and training sets. This can compromise the significance of the benchmark because the model may behave poorer on theorems markedly distinct from both the test and training set. Although, our models are relatively cheap to retrain and new training data can be synthetically generated, so the models can be updated to handle new libraries on demand, which slightly alleviates concerns about performance on out-of-distribution data.

2.5 CONCLUSION AND FUTURE WORK

We contributed natural language search datasets and trained a theorem embedding model. We designed multiple tools to flexibly meet the needs of end-users, whether that be mass-searches across the contents of an Opam switch through our web interface, or search over the instantaneous, live contents of a Coq session through our local client. We evaluated our model on both human-written and synthetic searches (though only one human was involved

in writing the human written searches) against several other embedding models and found that our models were competitive. We hope that ProofDB saves practitioners time in its current state and continues to improve. In the future, more interactive theorem provers such as Lean and Isabelle should be considered for inclusion into ProofDB and relevant datasets created. We think a thorough user study should guide future improvements to ProofDB and inform a less limited evaluation of search result quality. Finally, as ML continues to progress, we want our datasets to be used to train theorem search models starting from greater, more capable foundation models.

CHAPTER 3: PROOF REPAIR DATASET[11]

In this chapter we move on from theorem search and present work from Proof Repair Infrastructure for Supervised Models (PRISM) [1]—a dataset and benchmark suite for *proof repair* [16]. Proof repair is crucially important for reducing costs in large proof developments and for enabling the application of formal methods to broader and more diverse contexts. Unfortunately, data for proof repair is scarce and challenging to collect [72]. This chapter describes the repair data, reusable tools for building and extracting data from Coq projects, and the challenges we encountered along the way. The infrastructure described here is primarily responsible for producing the repair data for our eventual repair model in Chapter 4, but is also used throughout this thesis as it is of general use for interacting with Coq projects.

3.1 A PROOF REPAIR DATASET

The task that PRISM focuses on is proof repair. The data comprise aligned Git commits that correspond to existing changes in proof developments found on GitHub (Section 3.1.1). Success on the resulting benchmark is evaluated in terms of successful proof checking for repaired proofs (Section 3.1.2).

3.1.1 The Data: Aligned Git Commits

Projects were originally selected by querying GitHub's API for projects that contained Coq source code, had a file called "Makefile" in the project's root directory, and had at least 100 commits from which to mine repair data. Eventually, we also included projects from CoqGym [73] and filtered to projects that were listed in OCaml Package Manager (opam) repositories (opam acts as the primary distributor of Coq projects). We excluded projects that did not contain any proofs, or that had ulterior motives in their builds (e.g., projects that intended to test the performance of the Coq compiler `coqc`). We hope in the future to include additional projects, though this will require us to support more build environments and expand upon the work detailed in Section 3.2.2.

Within each Coq project, the data comprises a number of repair examples—that is, changes to definitions or proofs. A repair example is constructed by comparing a definition or proof before and after a change. Since sentences and files may be moved, renamed,

[11] This chapter contains previously published work [1] that we have permission to reproduce.

added, deleted, or otherwise altered between commits, they must first be *aligned* to ensure the right changes are compared. This means that Vernacular commands in one commit are assigned one-by-one to commands in another commit, where these assignments may cross file boundaries. Note that each command may not get a partner, indicating that it was either added or deleted. We describe this in more detail in Section 3.2.5.

After alignment, proof repair examples are constructed by partially applying changes, e.g., by omitting the changes to a proof that accompanied a change to the proposition. Thus, one pair of commits may give rise to multiple examples. The examples are compactly represented by commit hashes and diffs that indicate the state before and after a repair. This representation enables dissemination of the dataset without the accompanying projects, although we note supplementary tools for efficiently extracting project data will still be vitally important for eliminating redundant computations and effort in practice.

3.1.2 The Metrics: Proof Checking

Changes in proof developments that break proofs can be fixed in two ways: by repairing the proofs themselves or by repairing some other definition such as a program or specification [16]. PRISM includes both kinds of changes. We focus our benchmark suite on the former (repairing proofs), as the metric for success is immediately clear. We hope our benchmark suite will also be useful for the latter (repairing definitions), but we believe the problem of choosing a good metric for success for repairing definitions to be an open research problem.

Repairing Proofs We focus our benchmarks on the problem of repairing a proof script assuming that the statement of the repaired theorem is already known. In this case, checking the correctness of the repaired proof amounts to using Coq's kernel to proof check the type of the repaired proof against the type that represents the desired repaired theorem statement.

The proof checking metric is the same as that used for the standard CoqGym [73] proof generation benchmark suite for Coq. This metric is sound and complete (up to the correctness of Coq's kernel with nonterminating proof scripts designated as incorrect): any proof that checks with the desired type is a proof of the theorem the type encodes (**soundness**), and all proofs that prove that theorem will check with the desired type (**completeness**).

For this flavor of proof repair, we are able to take advantage of the fact that proof checking is a perfect oracle when the theorem statement is known. Perfect oracles have been hugely beneficial for existing ML work for proof generation [5] and for early symbolic work on proof repair when specifications do not change [74]. They continue to benefit ML for proof repair.

Repairing Definitions Helping users fix the definitions that the specification depends on—or the theorem statement itself—is also desirable. The REPLica user study, for example, found that 75% of the time proof engineers fixed a broken proof, they did so by fixing something else, like a program or specification [72]. Supporting this use case may actually be *more* helpful to proof engineers than supporting the original flavor of proof repair. Unfortunately, existing metrics are insufficient for measuring success on this task:

- The **proof checking** metric is insufficient when the repaired specification is unknown; showing that a proof type checks is not meaningful unless we know its intended type.
- The metric of **exact equality** with an expected repaired definition is too conservative, as there are many equivalent ways to state the same theorems or write the same definitions.
- Common notions of **definitional** or **propositional equality** in Coq are less conservative, but are still too far from complete.
- Pumpkin Pi [75] repairs definitions to be equivalent up to **univalent transport**, but checking this automatically is undecidable.
- Common natural language **distance metrics** like BLEU [76] are poor measures of success in code tasks [77], so we expect them to be inadequate for proofs.

If you choose to evaluate a model for repairing definitions, we recommend a conservative metric like exact or definitional equality to avoid the danger of chasing misleading benchmarks. We hope to eventually develop a suitable less conservative metric, particularly one that captures what makes a change "close to correct" for some suitable notion of correctness.

3.2 BUILDING THE PROOF REPAIR DATASET

We now take a step back and describe the processes behind our data collection efforts. As stated, the foundation of the dataset comprises open-source Coq projects. Mining the commits of these projects eventually yields examples of refactors or repairs. Each project is accompanied by per-commit metadata containing project dependencies, source URL, and build commands that is in parts manually curated and programmatically inferred. The process of generating repair data from a project comprises the following steps (see Figure 3.1):

- We obtain a *switch* (opam virtual environment) that satisfies as many of the project's dependencies as possible using the **Switch Manager (SwiM)** (Section 3.2.1).

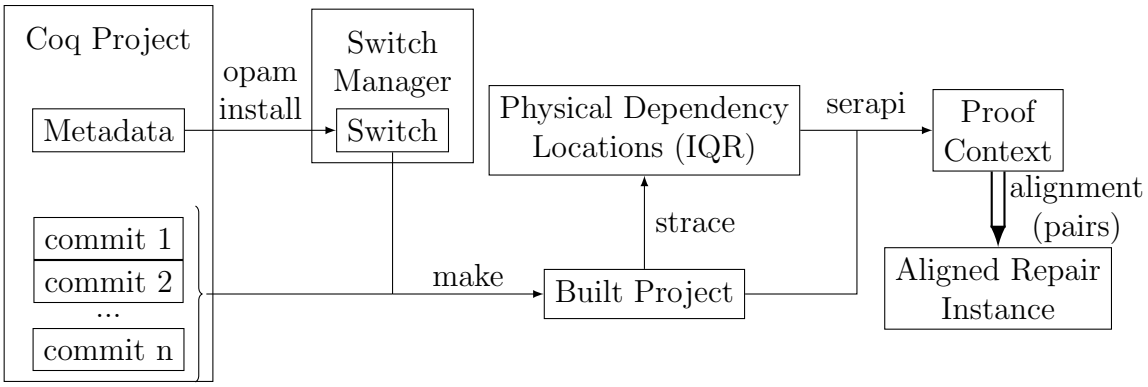


Figure 3.1: Process of extraction for a Coq project commit.

- Once we have a switch, we run the build command in the generated switch to produce a **Built Project** (Section 3.2.2).
- We `strace` the build process to scrape the **Physical Dependency Locations (IQR flags)** of each document in the project (Section 3.2.3).
- Using the IQR flags for each document in a built project along with the Coq serializer SerAPI [42], we extract a **Proof Context** corresponding to each intermediate proof state for the project by querying Coq's state during the execution of proofs (Section 3.2.4).
- Finally, we align changed proofs across commits and save those along with their intermediate proof contexts to arrive at an **Aligned Repair Instance** (Section 3.2.5).

Building this infrastructure was a significant undertaking with many challenges encountered along the way; we discuss these challenges in Section 3.3. Our hope is that the infrastructure we have built will make it easier to collect similar datasets in the future.

3.2.1 The Switch Manager

To extract information about projects like intermediate proof states, we must build them. This requirement is nontrivial because different projects can depend on different versions of Coq, the Ocaml compiler, or other dependencies.

To resolve dependencies and make it possible to build many different commits of one or more projects, we introduce a novel SwiM capability that works in tandem with opam to model the build environment for a given commit of a given project as a Python object. In

particular, the object models an opam switch, which is opam's representation of an isolated collection of installed packages.

This capability subverts the typical manual opam workflow to create and activate a switch. This manual workflow would be intractable at the scale of hundreds of commits for each of dozens of projects. With the SwiM, we can automate this functionality and extract a dataset at scale.

Several benefits arise from the SwiM's design. The SwiM enables build sandboxing by providing switch clones that last for just the duration of the commit's extraction, and it also minimizes the time needed to obtain a clone by maintaining a pool of switches across all threads with pre-installed packages, and choosing the one upon request that is closest to satisfying a commit's requirements. Implementation of this capability required reflection of opam's dependency formula parsing and evaluation logic from OCaml to Python.

As new commits are built, switches containing their dependencies are added to the managed pool of switches. Since switches range in size from hundreds of megabytes to a few gigabytes, a least-recently-used cache maintains the total disk consumption below an implicit limit by deleting stale, infrequently used switches.

3.2.2 Built Projects

Using the switch provided by the SwiM and a build command from the metadata, we may be able to build the project. Confounding issues that may prevent building include undefined opam variables within dependency formulas. In practice, we have so far seen a build failure rate of about 68%. We attempt to build each commit with seven different major versions of Coq ranging from 8.9 to 8.15 corresponding to versions of SerAPI that support capabilities we deemed necessary. Since Coq releases are rarely backwards compatible, many of the build failures can be explained by the fact that each commit can only be expected to build for the single Coq version for which it was written. Furthermore, since we pin one of the seven Coq versions in the switch supplied by the SwiM, conflicting version requirements may yield an opam command that has no solution. Consequently, some build errors are inevitable.

However, other errors are due to mistakes or missing information in the human-sourced metadata. We plan to address this latter class of build errors over time by fixing problems in the metadata through automated inference mechanisms. If the project build fails, we hope in the future to be able to recover proofs from the documents that built before the failure as well as subsequent independent proofs. We are also exploring ways to automatically recover from simple build errors such as dependency mismatches between the switch and the project's requirements by using the date the commit was made as a version hint.

3.2.3 Physical Dependency Locations (IQR Flags)

In order to run any of the Coq or SerAPI tools (e.g., `coqc`, `coqtop`, `sertop`) on a given Coq source file, one or more flags regularly need to be passed to these commands to specify the physical location of dependencies. These flags are described below:

- The `-I` flag allows a directory to be added to the OCaml loadpath.
- The `-Q` flag adds a physical directory to the loadpath and binds it to a given logical path.
- The `-R` flag acts like the `-Q` flag, but also makes subdirectories available recursively.

In publicly available Coq projects, these flags (referred to as "IQR" flags from here on) are specified in one or more build or configuration files. No single standardized approach for specifying IQR flags exists, making it difficult to automatically infer them from configuration and build files alone. While projects will be able to build successfully without our knowledge of these flags, we must infer them to use SerAPI tools in other stages of our framework.

Our solution to this problem builds off an approach developed in IBM's PyCoq [78]. Following PyCoq, we use `strace` to inspect the actual commands run during the build process for a Coq project. Each build command is captured and any present IQR flags are extracted using regular expressions. In some projects, build files may be nested, and IQR flags may specify physical paths that are relative to the nested directories. We need to ensure that the inferred IQR flags are relative to the project root directory, so before we store the inferred IQR flags, their paths are resolved to the project root directory.

3.2.4 Proof Contexts

Once the project has been built, the individual Coq source files are parsed into sentences and then interactively executed with `sertop` to capture intermediate proof states.

A parser (`sercomp`) is available through SerAPI, but it only works on Coq source files whose dependencies are already compiled, which prohibits its use in recoveries from partial builds. Furthermore, `sercomp` introduces significant redundant computation with respect to `sertop`. As a more efficient alternative, we developed a simple regular-expression-based "heuristic parser" to perform sentence extraction and approximate proof identification.

From `sertop`, we can collect thorough context from the document, like whitespace-normalized text, abstract syntax trees (ASTs), command types, and intermediate proof steps with goals and hypotheses. Each command is accompanied by inferred identifiers of the

command itself (e.g., an inductive type's name and constructors) and a list of fully-qualified identifiers referenced within the command, which enables models to more easily incorporate local context or apply graph-based approaches. Accompanying source code locations allow for accurate provenance of data and application of proposed repairs to appropriate destinations for testing.

3.2.5 Aligned Repair Instances

The last step in our data collection process is extracting proof repair examples from different versions of projects, accounting for the fact that definitions and proofs may be changed, moved, renamed, or deleted between commits. We must establish a robust mapping between Vernacular commands in a pair of commits that preserves some notion of command identity. Our objective is similar to that of the 'diff' utility, which describes changes made between files. Where our objective differs is that we seek to match Vernacular commands rather than lines, and we seek to do so within the entire project directory structure rather than a file.

A traditional order-preserving alignment between two sequences, e.g., the Smith-Waterman algorithm [79], is not quite an appropriate approach to resolve this issue as it cannot correctly align two independent definitions whose order has been reversed during a refactor (perhaps due to an introduced dependency). Therefore, we approach the problem as a bipartite matching or *assignment* between the unordered elements of two sets such that the overall similarity of matched elements is maximized. We can formally specify the desired assignment between two commits X and Y considered as respective sets of commands across one or more files as the solution to the following optimization problem:

$$\begin{aligned} &\underset{U,\,W}{\text{minimize:}} \sum_{u \in U} C(u, f(u)) \\ &\text{subject to: } f : U \leftrightarrow W \ \wedge \ U \subseteq X \ \wedge \ W \subseteq Y \ \wedge \ \|U\| = \min(\|X\|, \|Y\|) \end{aligned} \tag{3.1}$$

Here, $C(x, y)$ is a non-negative cost function that measures the *distance* between the commands x and y, and f is a bijection between subsets of X and Y– a partial alignment between commands of X and Y. To align as many commands as possible, the domain of f must have at least as many members of the smaller of X and Y, which is our final constraint above. This optimization is an instance of the well-known *assignment problem*, which one can solve exactly in polynomial time, e.g., with the Hungarian algorithm [80].

The optimization is parameterized by the choice of C, for which we choose a normalized

variant of the Levenshtein edit distance E:

$$C(x, y) = \frac{2E(x, y)}{\|x\| + \|y\| + E(x, y)}, \tag{3.2}$$

where $\|x\|$ and $\|y\|$ give the character lengths of x and y considered as text (not including proof bodies). This normalization is an instance of the biotope transform [81], which preserves the metric properties (such as the triangle inequality) of the edit distance. We further threshold the distance by a constant t such that $C_t(x, t) = \min\{C(x, y), t\}$, which also preserves metric properties [82]. After solving for f, commands x and y assigned to one another ($f(x) = y$) with a cost of t are considered to be unassigned (i.e., we determine that x was dropped between commits and y was added). We choose $t = 0.4$, which roughly corresponds to 50% of a command's text being changed before it is considered to have been dropped.

Solving this assignment problem for two entire commits can be costly: solving exactly is cubic complexity, and calculating the edit distance between all pairs of commands from both commits is necessarily quadratic complexity. Furthermore, the assignments produced may be somewhat spurious, especially in the event of multiple global optima. We mitigate these issues by applying the assignment problem only to those commands known to have changed in some manner between the commits according to their intersection with a (Git) 'diff'. The final resolution of the problem is thus somewhere in between alignment and assignment.

Once we determine an alignment, we create examples of proof errors by leaving out changes to individual proofs one at a time, thus providing the context for each change to a proof that required repair but not the repair itself. The left-out change to the proof then accompanies the error as a ground truth target for supervised learning.

3.3 CHALLENGES

The major challenges we faced in building this dataset and benchmark suite chiefly fall into two categories: Project Management (Section 3.3.1) and Parsing & Serialization (Section 3.3.2). For each of these categories, we discuss our experiences dealing with each stated challenge.

3.3.1 Project Management

One of the greatest barriers to building this dataset was the lack of a centralized archive for Coq proof data. In the absence of this centralized archive, we resorted to looser col-

lections of projects organized by package management. The package manager opam gives us a programmatic interface to build compatible environments for the dataset's constituent projects. However, it was designed to service individual developers using a few switches, whereas we must spin up *dozens* of switches efficiently. We thus had to reimplement and expand upon some of opam's capabilities. We faced three challenges in so doing:

1. Significant **build system variation** across different proof developments;
2. **Expressive dependencies** in opam packages that complicate efficient installs;
3. Insufficient caching of opam build artifacts that necessitated **copying switches** to avoid rebuilding the same packages.

Build System Variation Over the years, the recommended build system for Coq proof developments has been in flux. In 2019, for example, the Coq development team urged proof engineers to move their proof developments to Dune [83]. This effort did not fully succeed, and the documentation for the latest Coq version includes instructions for both Dune and the native Coq build system [83]. The native build system itself has also changed over time, losing compatibility with its previous versions. Because of this fragmented build infrastructure, we had to employ extremely abstract methods to extract arguments for SerAPI tools, namely by using `strace` to grab IQR flags passed to Coq's compiler `coqc` (described in Section 3.2.3) while making almost no assumptions about the process invoking `coqc`.

Expressive Dependencies The opam package manager provides a powerful and expressive syntax (package formulae) for packages to specify dependencies. Package formulae allow developers to restrict the versions of dependencies that can be installed, to conjunct and disjunct formulae into more complicated expressions, and to refer to variables declared elsewhere in the environment. This feature benefits the library developer that can precisely specify the environment for running code, but for our purposes it poses a challenge: packages can be picky about their environments and force opam to rebuild existing libraries. Since we need to install many versions of many packages, we need efficient ways to create or select compatible switches, which means interpreting these formulae. As a result, we reimplemented a majority of opam's package formula features, including parsing the custom grammar for package formulae and implementing package version comparison, to reason about which existing switches would require the least time to install a given package with opam.

Copying Switches To work with conflicting packages or different versions of the same packages, we must use different environments. The opam "switch" abstraction allows us to sandbox environments, but creating many switches incurs exorbitant overhead as each new switch rebuilds packages from source. Building a package *once* and deploying it in multiple switches is preferable, but many executables built by opam contain their absolute path as a hardcoded variable, which means they stop working if the name or location of the containing switch changes. That is, a built opam package only necessarily works in one switch. Our workaround is to copy switches and use the `bwrap` utility (which is also used internally in opam) to bind-mount the copied switch over the original such that the clone is in the original hardcoded location from the perspective of the running process. This solution allows copies of switches to act as if they are the original. Of course, handling these cloned switches requires extra bookkeeping and infrastructure, which the SwiM (Section 3.2.1) ultimately handles.

3.3.2 Parsing & Serialization

No matter how sophisticated the build system, we cannot get detailed data about individual proofs without parsing Coq files and serializing proof state to text. SerAPI [42] is the de facto standard for serializing Coq, providing a query protocol for exposing internal Coq data like definitions in the global environment, syntax trees, goals, types, and more. We used the CoqGym [73] Python wrapper as a starting point for our implementation, taking care to decouple it from CoqGym's custom versions of Coq and SerAPI since we need to support multiple versions of each coinciding with chosen projects' Git histories. This need to support multiple versions of Coq exacerbated challenges arising from gaps in SerAPI's query protocol, requiring us to implement workarounds using the most public and arguably stable interface Coq possesses: its Vernacular query commands. We faced four challenges related to parsing & serialization:

1. Executing a file one Coq sentence at a time requires accurately **parsing sentence boundaries**, but parsing requires execution: a catch-22.

2. **Identifying dependencies between commands** (e.g., which lemmas a theorem uses) is critical to providing locally relevant repairs but is not a capability provided by SerAPI.

3. **Determining the scope of a conjecture** is complicated by the potential presence of nested proofs/definitions and arbitrary grammar extensions.

Listing 3.1: A notation that breaks `CoqIDE`'s parser. This example was found in the Coq Discourse [84].

```
Notation "( a . b )" := (a, b).
Check (1 . 2).
```

4. SerAPI is experimental software, which leads to breaking **changes between versions**.

Parsing Sentence Boundaries A Coq statement or 'sentence' ends with a period (.), but Coq also uses the symbol for import paths and module members so that one cannot identify sentences in a file merely by splitting on periods. To further complicate matters, Coq boasts an extensible syntax that enables users to define syntax that allows periods to show up in even more situations. For example, Listing 3.1 defines syntax using a period that complicates sentence splitting to the point where the latest version of `CoqIDE`—the official editor for Coq—cannot correctly parse and run this code even though Coq can. We did not discover any public or officially supported mechanism to extract the sentences of a Coq document, which led us to develop the Python-based heuristic parser mentioned in Section 3.2.3 for simplicity and maximal portability between build environments.

Identifying Command Dependencies Identifying dependencies decomposes into two subproblems: detecting the definitions (if any) introduced by a given command and resolving referenced names unambiguously.

No SerAPI query resolves the first subproblem, nor is there any reliable syntactic clue in the text that generalizes across unforeseen grammar extensions. Instead, we rely upon parsing user-level feedback that notes the introduction of new identifiers (e.g., "X is defined") and Vernacular queries. Since feedback is not guaranteed for all definition types (particularly propositions, depending on the Coq version), we also monitor for changes in the set of all locally defined names yielded from Vernacular `Print All` command. One can thus reliably identify a command with names introduced immediately after its execution.

The second subproblem arises from the fact that identifiers within ASTs yielded from SerAPI are not necessarily fully qualified. Correcting this deficiency requires locating the identifiers within the AST and issuing a Vernacular `Locate` query for each one. Care must be taken to ensure that variables within local binders, patterns, or other sub-expressions do not get mistaken as any top-level definition that they may shadow. Given the lack of insight available into Coq's internal name resolution, the accuracy is ultimately limited by handcrafted scope rules. We also note one restriction on resolving globally bound identifiers:

Listing 3.2: A simple example showing that proofs may be interleaved and that multiple proofs (obligations) may be associated with one term.

```
Require Coq.Program.Tactics.
Set Nested Proofs Allowed.
Program Definition foo := let x := _ : unit in _ : x = tt.
Next Obligation. (* Start first obligation of foo *)
  Definition foobar : unit. (* Interject with new conjecture. *)
    exact tt.
    Next Obligation. (* Switch back to first obligation of foo *)
      exact tt.
    Qed. (* Finish proof of foo's first obligation *)
  Defined. (* Finish proof of foobar *)
Next Obligation. (* Start next obligation of foo *)
  simpl; match goal with ⊢ ?a = _ ⇒ now destruct a end.
Qed. (* foo is defined *)
```

if a definition shadows an existing one, then it cannot also use the shadowed one. Violation of this assumption is possible (consider a recursive function `nat` that expects arguments of type `nat`) but not expected to pose a significant risk as it is unlikely in the first place and would generally be considered poor practice. If the restriction is violated, then the shadowed definition will simply be mistaken for its shadower within the shadower's definition.

Determining Conjecture Scope Determining conjecture scope decomposes into two subproblems: attribution of proof steps to the correct conjecture and detection of proof (conjecture) completion. Each is complicated by potentially intermingled or nested proof steps as shown in Listing 3.2 and by the lack of a SerAPI query of the active conjecture's identity.

A Vernacular command—`Show Conjectures`—again provides the solution. This command lists the names of currently stated but unproved conjectures and by all observations is guaranteed to list the conjecture actively being proved first. We rely upon this presumed order to identify the current conjecture, accumulating proof steps in stacks per open conjecture. The method's accuracy depends upon the assumption that each conjecture enters proof mode once its first sentence is executed. The only known exceptions to this rule comprise Programs, which do not enter proof mode until their first Obligation's proof is begun.

Special handling is required to associate each Obligation with the correct Program since `Show Conjectures` reveals a unique name for each Obligation. However, the special handling means any grammar extension that defines its own Obligation or Program equivalents (e.g., multi-block proofs) cannot be serialized to the same level of accuracy. If any extension does

so, then each Obligation-equivalent is expected to be serialized as an unrelated theorem.

We rely upon detection of definitions to determine when and if a conjecture was proved, assuming that no conjecture emits an identifier before it is defined (i.e., before it is proved). Only subproofs (generally delimited by bullets and braces) are allowed to violate this rule. However, one cannot assume that the first detected definition in the midst of a proof corresponds to the conjecture, nor can one assume that the name of the conjecture once defined will actually match its name as returned by `Show Conjectures`.

We ultimately detect the completion of a proof by requiring two conditions: a change in the currently detected conjecture and the detection of a new definition. This rule necessarily invokes an additional assumption: a change in the current conjecture implies that either a new proof has begun or the current proof has ended (but not both). Since we assume that conjectures cannot emit identifiers before they are done, we deduce that the emission of an identifier upon the change of the current conjecture implies the completion of the prior one.

Finally, if the conjecture is aborted, then it will never be detected as a definition at all even though its proof has ended. We detect aborted proofs simply by checking the type of the command, assuming that no grammar extension defines `Abort` or `Abort All` equivalents.

Serialization and Version Changes SerAPI was in theory supposed to help with proof assistant versioning problems. In practice, though, SerAPI itself depends on the version of Coq, and we found we had to break the SerAPI abstraction barrier often as the Coq version changed. In other words, while SerAPI provides a convenient interface to expose certain Coq internals, those internals are not necessarily stable. For example, SerAPI had "can't-fix" bugs involving nested proofs because the serialization errors occur in the Coq codebase itself [85]. SerAPI itself has as of a few days ago been deprecated in favor of a new serializer [44, 45].

3.4 CONCLUSIONS

This chapter describes repair data, methodology for extracting it, and challenges in those processes. Future work could aim at making PRISM faster, more reliable, or compatible with more versions of Coq. We also might take a different stance and focus our efforts upstream, directly helping Coq and its related tools to enable extraction of metadata for ML purposes. One approach could be contributing lessons learned directly to actively developed serialization projects, such as the Coq Language Server Protocol implementation[45].

CHAPTER 4: PROOF REPAIR MODEL

This chapter is an extension of Chapter 3 wherein we utilize the repair data infrastructure to take steps towards practical automated proof automated repair. We present ProofDoctor, which can fix some proofs by updating broken theorems and tactics or, occasionally, even by synthesizing new proof strategies altogether. ProofDoctor decomposes to the following contributions:

1. Generated textual datasets derived from our work in Chapter 3 which can immediately be used to train a model. We produced both proof synthesis and proof repair datasets, opinionated to work with our intended repair methods (Section 4.1).

2. The first specialized proof repair model for recommending repairs (Section 4.2).

3. A model inference loop which can be used to generate and apply repairs to proofs in real projects. To complement our repair model, we utilize the search model of Chapter 2 to correct theorems hallucinated by the repair model to real theorems (Section 4.3).

We call the combined proof repair model and tooling ProofDoctor. A toy proof repair example is given in Figure 4.1 and we demonstrate repairs on real projects in Section 4.4.

4.1 DATA

While the contents of the training data are largely generated from the proof repair data infrastructure of Chapter 3, Chapter 3's output was meant to be model agnostic. In this section, we transform the data to turn it into concrete prompts that we train our repair model on.

Listing 4.1: A toy proof repair performed by ProofDoctor. We renamed `apple_something` and changed the name of the hypothesis from `H` to `A`, and the following fix was produced. The errors were committed so that the model couldn't simply read the diff of the errors we introduced.

```
 Theorem easy (A:Apples) : Oranges.
-Proof.
-  exact (apple_something H).
+Proof. (* This proof was automatically repaired. *)
+  exact (Apples_is_Oranges A).
 Qed.
```

Remark 4.1. The original PRISM[1] work that Chapter 3 is drawn from produces a larger repair dataset than we have. For the repair data used in this chapter we added stricter filters which excluded some data. Furthermore, we made changes to the pipeline itself, which required us to re-run the PRISM data pipeline. The modified data pipeline was both more fragile and was not run to completion, so we gathered fewer repair examples than PRISM's final data release had. Future work can improve this.

4.1.1 Prefix Alignment

An algorithm we make use of both at inference time and training time is the prefix alignment, which is a small change we made to a standard alignment algorithm. The standard sequence alignment algorithm is defined by the following recurrence for some (typically non-negative) cost function:

Figure 4.1: Recurrence cases for a standard alignment algorithm.

1. $\text{align}(nil, nil) = 0$
2. $\text{align}(nil, y :: T_y) := \text{cost}(nil, y) + \text{align}(nil, T_y)$
3. $\text{align}(x :: T_x, nil) := \text{cost}(x, nil) + \text{align}(T_x, nil)$
4. $$\text{align}(x :: T_x, y :: T_y) := \min \begin{cases} \text{cost}(x, y) + \text{align}(T_x, T_y) \\ \text{cost}(x, nil) + \text{align}(T_x, y :: T_y) \\ \text{cost}(nil, y) + \text{align}(x :: T_x, T_y) \end{cases}$$

For the cost function ($(x, y) \mapsto$ if $x = y$ then 1 else 0), the alignment algorithm returns the minimum number of replacements, insertions, and deletions required to make one of the input strings into the other. In practice, alignment can be implemented as a dynamic programming problem that runs in $O(|n||m|)$ for constant time cost functions. In addition to discovering the cost of the min-cost alignment, we can also obtain the actual min-cost alignment (the correspondence between elements of each sequence) by backtracking through the dynamic programming table to determine which choices were made. Many variations of this basic alignment algorithm exist, such as local alignment, which can find the minimum

cost alignment between one string and *any substring* of the other string. We perform a similar modification. Consider replacing case (2) above with the following:

$$\text{align}(nil, y :: T_y) := 0 \tag{4.1}$$

In other words, we do not penalize the algorithm for completely aligning the entirety of the first string before the second string is finished. The informal effect of this change is that we find the minimal alignment between the first input and *any prefix* of the second input. We call this prefix alignment. Prefix alignment is particularly relevant to our setting. Suppose we have a previously functioning proof (the old proof) and a partially repaired proof (a prefix of the new proof). By running prefix alignment between the new proof prefix and the entire old proof with an appropriate cost function, we obtain an alignment that signifies our *best guess* which prefix of the old proof corresponds to the incomplete new proof we're working on. This allows us to determine which steps we probably have and haven't changed so far in a proof, as well as guess what the next step should be if we were to continue our current proof using steps from the old proof.

4.1.2 Data Sampling

We make a few assumptions about the proof repair task which have consequences for the dataset and downstream inference tasks. First, we expect the majority of a proof to remain the same in a proof repair. If a proof must be mostly rewritten to reattain correctness, we consider the task mostly proof synthesis since the majority of the proof must be written ex nihilo. We throw out samples from proof 'repairs' of this nature. Second, we want to avoid spending time training the model to repair parts of proofs where the new proof does not change from the old proof, because these samples are trivial. We check to see if the prefix alignment algorithm of the previous section can correctly guess the next tactic to run in the new proof. If it can, we discard that sample as trivial. Since prefix alignment is not perfect, the particularities of the algorithm (conditions that cause it to succeed or fail) are baked into the dataset. To prevent this from impacting inference, we use the same prefix alignment during inference as a first guess, recreating the same distribution as the training data at inference time[12].

[12]This is also much faster than always running the model, especially for long chunks of unchanged tactics

4.1.3 Data Format

Our proof repair training data prompts the model with the following pieces of information, all of which can be extracted using the infrastructure of Chapter 3:

1. The 'git diff' of the project's ongoing repairs, which inform the model what changes/-fixes were already made.

2. The proof state as given by Coq. This is the usual information Coq would show to a user actively working on the proof, such as the hypotheses and variables in the local environment and the current goal at this point in the proof.

3. The most recently run tactics, plus extra information such as recommended tactics from Subsection 4.1.1. We also present the list of recently run tactics as a (prefix) diff from the old proof to emphasize the changes that had to be made.

A full textual example of a prompt is given in Appendix B.3. We also generate prompts for proof synthesis, but these are essentially just proof repair prompts without diff information and tactic recommendations. For both kinds of data we must be careful about the amount of tokens that all of the above information requires. We try to limit the total context length plus label length to 2048 tokens. When this isn't possible, we try truncating the end of the git diff and the oldest parts of the recently run tactics. However, we do not know of a good way to truncate the proof state, so when the proof state is too large to fit into the context, we give up on that sample.

The other half of a data sample is the predicted tactic, or label, for each prompt. We augment the label with certain annotations that improve interoperability with the environment and search tools at inference time. Specifically, we try to replace every reference to a theorem in the global environment with a textual 'lookup' of the following form: `<LOOKUP>theorem_name : theorem_type</LOOKUP>`, where `<LOOKUP>` and `</LOOKUP>` are special tokens inserted into the model and tokenizer before training. We describe the downstream usage of these lookups in Section 4.3.

4.1.4 Avoiding Test Split Contamination

Our model is derived from a pre-trained model, Llemma [17], which has already been trained on the source of Coq projects our dataset draws from. To avoid test set contamination, our test sets contain only repair examples and synthesis examples from Coq files whose names are not present in the list of Coq source files that Llemma was trained on. Since we

use this same process on the synthesis and repair portions of the dataset, we expect that the synthesis and repair datasets do not contaminate each other's test sets. This process may be imperfect, if, for example, a repair dataset test example comes from a file that was eventually renamed, and then that renamed file was in the Llemma training set.

Our test set may also be contaminated by data that Llemma's base models were trained on (Code Llama[59], Llama 2[58]). Unfortunately, the corpuses for these models were not released, so there is little we can do about this particular concern.

4.2 MODEL

4.2.1 Training

Our model can be thought of as extending a long chain of successive fine-tunings.

1. The Llama 2 [58] series of models was trained on a huge, private, 2 trillion token corpus of broad text.

2. The Code Llama [59] series was fine-tuned from the Llama series on a corpus of $\geq$500 billion tokens of programming related content.

3. The 7B parameter Llemma [17] model was fine tuned on 200 billion tokens from the Proof-Pile-2 dataset, which contains, among other math and proof content, Coq source code.

4. We fine-tuned a rank 192 QLORA [27] adapter on top of the 7 billion parameter LLemma model with 6,000 proof synthesis steps[13] in the format described in Subsection 4.1.3.

5. We continued training the adapter above on a half-and-half split of repair and synthesis examples comprising about 6,000 more steps, giving us the ProofDoctor model.

We used synthesis steps in training alongside repair steps because they were much more plentiful than repair steps and still contain valuable information about writing Coq proofs. In the next subsection we show evidence that pretraining on proof synthesis is helpful.

[13] essentially repair steps but without diffs and repair specific information

4.2.2 Test Losses and Pretraining Ablation

By removing the 6,000 steps of proof synthesis pre-training we can compare the test-set performance of the model with and without synthesis pretraining. We report losses for synthesis and repair test sets separately.

Figure 4.2: Repair/synthesis test losses for training with/without pretraining.

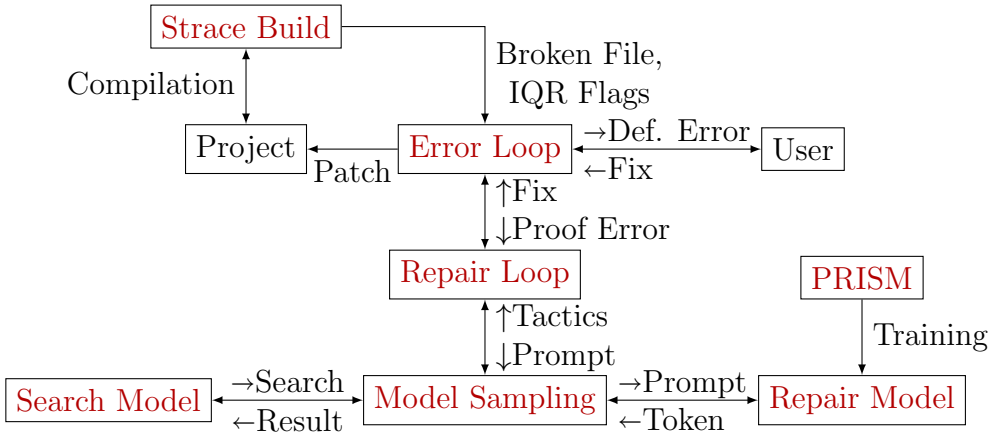
Model	Synthesis Loss	Repair Loss
No Pretrain	0.0642	0.00625
Pretrain	**0.0577**	**0.00589**

Our results show approximately a 10% decrease in loss for the test split of proof synthesis examples and approximately a 5% decrease in loss in the test split for proof repair examples when the model is pretrained on additional proof synthesis tasks. This suggests that improvements to proof synthesis models lead to improvements in proof repair generalization.

4.3 REPAIR TOOL

We illustrate the components of PROOFDOCTOR, describe their interconnections, and outline them below.

Figure 4.3: Components and connections of this work. If you're viewing this thesis as a PDF, the components of the diagram should be mostly **clickable**.



Strace Build This component comes directly from Chapter 3, described in 3.2.3. It runs a build command for a project and logs the system calls that the build process executes.

Eventually, the build process tries to compile code by invoking `coqc`, the Coq compiler binary. In order to do this, it issues a system call which references `coqc` as well as all of the arguments passed to the compiler. We keep track of the return code of the build process to see if it failed. If it did, we can report the last file it tried to compile to the error loop as broken.

Error Loop This component takes a broken file and determines whether the breakage is of the kind the repair model can fix (proof error) or if the error is in a definition (definitional error). If the error is in a proof, we determine the boundaries of the proof and send it to the repair loop. If it is a definitional error, we simply report the error to the user and let them fix it. The method underlying these determinations is described in Subsection 4.3.1. Once we obtain a fix from either the user or a functioning proof from the repair loop, we step through the rest of the file looking for more errors. If that succeeds, we look for errors in the rest of the project by invoking the strace build again.

Repair Loop Here, we try to repair a broken proof. We try to reuse the old proof as much as possible and ask our model sampling routine for help when we cannot proceed. We employ some proof-repair-centric heuristics, described in Subsection 4.3.3, to try to arrive at a repaired proof faster.

Model Sampling Given a prompt, we sample the repair model to produce several candidate tactics. The repair model might make a request to the search model at any point in writing a tactic. We resolve these requests immediately and let the repair model finish writing the tactic with the result in the context so that it is aware of the type of the theorem that is retrieved. The model sampling routine also caches the repair model's predictions so that subsequent tactic predictions can be written faster, especially if they have a shared prefix.

Search Model The search model is simply the `proofdb-HN` model from Chapter 2. The search model acts as the repair model's window to the global environment, allowing us to take advantage of local theorems that the repair model was not explicitly trained to use. The precise mechanisms by which this is done are described in Subsection 4.3.2.

Repair Model The repair model itself is a QLORA [27] fine-tuned Llemma [17] model trained on proof repair and proof synthesis. We comprehensively describe the training of the model in Section 4.2. The format of data and repair data is described in Section 4.1.

4.3.1 Locating Repair Targets

Only a fraction of the contents of Coq source files are proofs. Attempting to repair source that doesn't correspond to a proof is poorly defined– when we repair a proof, we only know the repair was successful when the new proof compiles. Thanks to proof irrelevant terms, we should expect that the new proof works just as well as the old proof, regardless of its content, but this principle does not hold true elsewhere. For instance, although any function that takes two naturals and returns one natural can be substituted for addition and Coq will still accept the definition, this will lead to later problems. We need a way to locate specifically broken proofs. Our inference loop is given, as arguments, the command that should be executed to try to build the broken project. When we execute it, we can trace the build command's calls to the Coq compiler using strace, just as we did in Section 3.2.3. This lets us simultaneously learn which file is currently breaking the build as well as determine the correct IQR flags required to step through that file.

Once we know the IQR flags and currently broken file, we can use python-coqtop (2.1.1) to step through the file. Currently, we use two passes through the file to learn what kind of error we're dealing with. The first pass steps through the file and automatically Admits any proof which would have been completed with a `Qed`[14]. If we encounter an error in this pass, it must be outside the proofs we can repair, so we report the error to the user and ask for help, since we cannot repair definitions. If the first pass completes without error, we assume that the reason the file is not compiling is because there's an error within a proof that we can try to repair. We find the line with the error, keeping track of the line which opened the most recently opened proof, and scan forward through the file until we find a `Qed`. We now know the beginning and end of the broken proof and can attempt to repair the proof.

4.3.2 Contextualizing Model Output in the Global Environment

In proof synthesis with large language models a common failure mode is hallucinating theorems that don't exist. In one study [86], 34.3% of broken LLM generated proofs were said to contain hallucinations by expert annotaters. In order to repair proofs which have broken or changed dependencies, the model will need to somehow ascertain information about the changed environment. Since Coq global environments typically contain thousands of theorems, it is not practical to simply provide a language model a complete serialized copy of the environment given our hardware limitations.

[14]Some proofs are terminated with `Defined`. These proofs are transparent and act like definitions, so depending on how we repair it we can break dependencies. Since we cannot guarantee that attempting to repair these will help, we don't initiate repairs on these proofs.

Remark 4.2. Other proof synthesis tools [87, 88, 89] select a portion of the environment using either neural or heuristic premise selection and provide their tactic generation models just these theorems to avoid the huge context of the full environment. We take a different approach here.

Instead, we use hallucination to our benefit. During training, the model is trained to 'annotate' attempts to use global theorems in Section 4.1.3. This gives us a way at inference time to process the language model's attempts to access the environment. For instance, we can check whether the model is attempting to access a theorem that actually exists or not. As the model sequentially predicts the tokens of a tactic, we perform the following conditional rewrites to augment the model's predictions with information about the environment:

1. If the model begins a lookup with the name of a theorem that really exists in the environment, we *supply* it with the actual type of that theorem. If the model is attempting to use a theorem whose type has changed, then this allows it to perceive the change in the theorem's type. This behavior is summarized in the following conditional rewrite rule:

Figure 4.4: Conditional rewrite rule describing the process of inserting the type of a theorem from the environment.

$$\frac{\text{“}\texttt{name}\text{” is the name of a theorem in the environment}}{\text{“}\texttt{...<LOOKUP>name : }\text{”} \longrightarrow_{\text{rewrite}} \text{“}\texttt{...<LOOKUP>name : }\text{”} + \text{type(“}\texttt{name}\text{”)} + \text{“}\texttt{</LOOKUP>}\text{”}}$$

2. If the model begins a lookup with the name of a theorem that doesn't exist in the environment, we let it hallucinate the type of that theorem. Once it is finished, we take the hallucinated name and type and plug it into the proof search model of Chapter 2 to find real theorems related to the hallucinated one. Then, we replace the model's output with the search model's closest match. This allows the model to utilize theorems from the environment it has never been trained to use. This behavior is summarized in the following conditional rewrite rule:

Figure 4.5: Conditional rewrite rule describing the process of searching for a theorem from the environment.

$$\frac{\text{“}\texttt{name}\text{” is not the name of a theorem in the environment} \qquad \text{“}\texttt{result}\text{” is the first search result for “}\texttt{Theorem name : type}\text{”}}{\text{“}\texttt{...<LOOKUP>name : type</LOOKUP>}\text{”} \longrightarrow_{\text{rewrite}} \text{“}\texttt{...<LOOKUP>result : }\text{”} + \text{type(“}\texttt{result}\text{”)} + \text{“}\texttt{</LOOKUP>}\text{”}}$$

Once the model is done predicting a tactic, we replace the `<LOOKUP>` tags with just the name of the theorem. These processing steps allow our model to perform some repairs that would be completely infeasible otherwise. For instance, when a proof is broken due to changes that originate from updates to external libraries, there is no information in the git diff about what changes have occurred. Without the contextualization above, if the name of a theorem is changed in a library or the arguments of a theorem were switched, the model would have no way to perceive this change and make the appropriate repairs without guessing the changes ex nihilo.

> **Remark 4.3.** Here we use the natural language search model to find related theorems. This may seem strange because it was not trained to do this. Although it may benefit from fine-tuning for repair applications, we posit that the natural language theorem search model of Chapter 2 should be able to find related theorems well without any modification. Further explanation and an example are given in Appendix B.2.

4.3.3 Repair Heuristics

Although our model is highly stochastic and cannot be trusted to make the correct choice the first time, there's nothing stopping us from evaluating multiple model outputs or exploring different branches of the tree of choices. Indeed, related works in proof synthesis usually operate over a tree of branching options rather than trying to write a proof in one shot [7, 89]. In the paragraphs below we explore heuristics afforded to us by the proof repair setting. These heuristics take advantage of the structure of the broken proof to guide the structure of the (presumed similar) repaired proof.

Falling Off of a Proof One of the simplest adaptations we add is detecting when we have apparently written far beyond the end of the original proof. We can use Prefix Alignment (Subsection 4.1.1) to see if it seems like our new proof contains and then extends the previous proof. When the model makes certain bad choices, it can irreparably change the goals that need to be proven (even making unprovable ones) or create several new goals that would take many tactic invocations to complete. The model doesn't know how or when to backtrack, so it is stuck with the consequences of these actions. Eventually, it might seem to be writing past the end of the original proof, yet still have several unproven obligations. When the model has "fallen far enough off of the proof", we consider it a failure and trigger a restart.

This adaptation is particularly limited. Occasionally, it prevents valid repairs from occurring when the nature of the repair is to prove additional obligations, and it happens that those repairs would occur at the very end of the original proof. Additionally, it does not completely solve the problem it set out to address: sometimes the model creates an unsolvable situation in the middle of following the original proof and never again runs anything resembling tactics from the original proof, causing prefix alignment to assume the current work is being done that corresponds to the middle of the old proof, so this adaptation doesn't trigger.

Taking Recommendations Prefix alignment creates a correspondence between our current attempted repair and the old proof. This lets us check what 'would have been run' in the old proof at the point in the old proof that corresponds in the current attempt. We call this a 'recommendation'. When a recommendation runs without error, we immediately take it. This is analogous to a developer compiling a proof to see what breaks, making changes around the line that is broken, then moving to the next line the compiler complains about by attempting to compile the proof again. In this proof repair setting we assume that the repairs do not change the majority of the proof, so this is a relatively accurate heuristic that dramatically improves the speed of repairs, since it allows us to rapidly step through through long chains of tactics that don't need to be changed.

This heuristic occasionally leads to problems, though they are not always irrecoverable for a sufficiently clever model. At the beginning of the proof, this heuristic will simply run tactics from the old proof until an error is encountered. Suppose the old proof sets up a hypothesis needed much later using a `assert` statement, but once we apply that hypothesis, we find that it does not work. It is now much too late to change the contents of the assert statement, and our model's only recourse is to `assert` the correct hypothesis immediately. However, the previous, incorrect assertion may have already been proven using the old proof's tactics, so if we prove the new assertion now we will no longer be following the structure of the old proof, which eliminates the benefit of proof repair over proof synthesis. Thankfully, this kind of situation does not seem to be common. Frequently we observe that the lines that must be changed cause some kind of error immediately rather than later down the line.

Pruning Choices by Avoiding Future Proof Breakage When the model is consulted to generate the next tactic in the proof, it might produce several tactics that actually run and progress the proof. Exploring all of the possible options is extremely expensive, so we'd like a notion of which options are better than others. We can use the repair model's internal notion of the probability of each tactic to weigh the options, but this isn't particularly well

calibrated for the task at hand. Another relatively easy way to pare down the options is to check how many tactics from the old proof *begin to work* when we select each one of the options. We call this the "future score". When the set of candidate tactics has varying future scores, that means that some of them can actually be thought of as avoiding breakages in the future, or equivalently that some of the candidates cause some future breakage that others avoid. Currently, we prune all choices that do not have a future score equivalent to the maximum future score among the set of candidates.

This heuristic has similar pitfalls to the previous heuristic. If very extensive changes need to be made to a proof, but methods to avoid making those changes that eventually result in an unsolvable proof exist, we may greedily avoid making changes until we get stuck.

4.4 EVALUATION

In this section we demonstrate the fundamental capabilities of the inference loop in synthetic examples through a series of case studies. We ran ProofDoctor on several projects (largely from coq-contribs) which did not compile on Coq 8.18 and discuss the results here. We use a fixed commit[15] of our tool and a 300 second timeout per proof.

> **Remark 4.4.** To avoid contamination between training/development and outcomes of case studies, we take the following precautions for projects presented as case studies:
>
> 1. The names of the libraries are dissimilar to those that are in our repair training set.
> 2. For examples we show here, we scan the prompts in the dataset for short identifying substrings from the proof to see if we have trained on the exact repair we're demonstrating.
> 3. We only show examples from projects we tried to repair *after* the methods and heuristics of ProofDoctor were solidified. We used some projects as iterative testing grounds for changes to heuristics, and for fairness we won't be showing those projects as case studies, since our adjustments were specifically meant to improve performance on those projects.

We ran our tool on several projects, but do not discuss all of them here. The choice of projects was arbitrary, and the number of proofs repaired should not be taken as statistically

[15] `74282d31eb107b060e5c9c76bced1241bf54cd0c`

representative.

4.4.1 fermat4

Fermat4 [90] is a Coq library which proves Fermat's last theorem for the case of n=4 and Diophantus' 20th problem. It was published in 2005 and now resides in coq-contribs. We partially repaired the specifications and proofs to compile in Coq 8.18.

Figure 4.6: Repair attempt info for fermat4.

Repair Attempt Link	https://github.com/tom-p-reichel/PR-fermat4/
Proofs Repaired	12
Proofs Admitted	28

Fixing Outdated References The nature of the successful repairs were almost entirely replacements of outdated theorems, such as in the following instance (spacing edited for clarity):

Listing 4.2: A typical ProofDoctor repair example.

```
 Lemma R_prime_wf : well_founded R_prime.
-Proof.
-  apply (well_founded_lt_compat _ f_Z R_prime); unfold R_prime, f_Z; intros;
-    apply Zabs_nat_lt; intuition.
+Proof. (* This proof was automatically repaired. *)
+  apply (well_founded_lt_compat _ f_Z R_prime); unfold R_prime, f_Z; intros;
+    apply Nat2Z.inj_lt ; intuition.
 Qed.
```

The only change made was the substitution of `Zabs_nat_lt` to `Nat2Z.inj_lt`, which causes the proof to compile again despite changes to the underlying specifications (a specification patch was made to `f_Z`, since it relied on functions that no longer existed) and under changes caused by updates to the standard library (`Zabs_nat_lt` no longer exists).

Unusual Workaround Most of the repairs for fermat4 were similar replacements of identifiers with contemporary replacements with the exception of the repair to `prime_dec_gen`, which was highly unusual:

Listing 4.3: An unusual ProofDoctor repair example.

```
 Lemma prime_dec_gen : forall a b : Z, 1 < b → b < a →
   (forall c : Z, b < c < a → rel_prime c a) → prime a ∨ ¬prime a.
-Proof.
-  intros a b; pattern b;
-    match goal with
-    | ⊢ (?p _) ⇒
-      simpl; case (Z_lt_dec 1 a); intro; try (right; red; intro; elim H2;
-      clear H2; intros; progress auto); apply (ind_prime p); intros;
-      case (rel_prime_dec x a); intro;
-        [ case (Z_eq_dec x 2); intro;
         (* several lines removed for brevity *)
-    end.
+Proof. (* This proof was automatically repaired. *)
+  intros; case (prime_dec a); intro; [ left | right ]; auto.
 Qed.
```

Here, ProofDoctor has discovered that there is a theorem called `prime_dec` in the standard library. This likely did not exist at the time this library was written, but it does now. `prime_dec` is almost a completed proof of the current goal, and the tool managed to circumvent the very long original, broken proof by directly invoking the standard library theorem[16]. This repair is notable because the model completely deviated from the structure of the original proof and demonstrated some small amount of "lateral problem solving".

$\mathcal{L}_{tac}$ One-Liners This library demonstrates a rather extreme version of a difficult edge case of the $\mathcal{L}_{tac}$ language: the majority of the proofs are written as 'one-liners'. In $\mathcal{L}_{tac}$, tactics can be chained through the use of semicolons and other combinators. These chained tactics constitute a single proof step. When tactics are chained we do not observe any of the intermediate states, which means we have less information to guide us when writing the tactics and, if we wish to follow the structure of the previous proof, we must write the correct proof all at once, which is a very large combinatoric search space for repair. Indeed, humans also consider one-line proofs to be difficult to read and maintain. The main failure mode of our tool here seems to be failing to initially repair the one-liner, though perhaps coming up with a step that advances the proof, then without the guidance of the old proof's structure, failing to make progress.

[16]It seems possible that fermat4 was eventually contributed to the standard library because some of the theorems in fermat4 now seem to exist under the same names in the modern Coq standard library.

Remark 4.5. In neural proof synthesis research, long $\mathcal{L}_{tac}$ one-liners are sometimes considered worse training data than a series of brief tactics. A process called linearization was used in Proverbot9001 [89] which broke down some series of semicolon-joined tactics into a longer series of independent tactics. This would be an appealing way for our model to interact with proofs as well, but we would like to make minimal changes to the proofs we repair, so we would want to reverse linearization after we repaired a proof. We trained on verbatim tactics written by developers because we are not aware of any exploration into reversing the linearization of a repaired proof, but this could be interesting future work.

4.4.2 distributed-reference-counting

Distributed reference counting [91] is a Coq library which proves correctness for several implementations of the titular distributed reference counting. It was published in 2000 and now resides in coq-contribs. We partially repaired the specifications and proofs within to compile in Coq 8.18.

Remark 4.6. Before we ran repair on this project, we tried to manually replace every invocation of `omega`, which is no longer a part of the standard library, with `lia` with a regex replace, since those kinds of repairs are not particularly interesting. An engineer would realistically take this step before using our tool, since it is a very common correction. We actually, erroneously, only made the substitution to one file, so the majority of proofs repaired for this project are the model fixing the remaining references to omega. We committed these changes before the repair tool was run so our repairs weren't provided to the model as a git diff in the prompt. Nonetheless, the resulting repairs had some interesting features.

Figure 4.7: Repair attempt info for distributed-reference-counting.

Repair Attempt Link	https://github.com/tom-p-reichel/PRM-distributed-reference-counting
Proofs Repaired	71
Proofs Admitted	3

Compulsed Substitution and Unnecessary Steps Several repairs resembling the following were made in distributed-reference-counting:

Listing 4.4: A ProofDoctor repair example with a junk tactic.

```
Remark add_reduce2_1 :
 forall x y z a : Z, x = a → (x + (y + z))%Z = (a + y + z)%Z.
- Proof.
- intros; omega.
+ Proof. (* This proof was automatically repaired. *)
+ intros; rewrite H; auto with *.
Qed.
```

Here, ProofDoctor replaced `omega` with the approximately equivalently powerful `auto with *`, plus a rewrite. However, the rewrite is entirely unnecessary: the proof continues to compile even without the rewrite. We hypothesize a particular behavior of our repair model we will call compulsed substitution. If a substitution exists in the git diff that seems to apply to the current line, the model usually tries to apply that substitution in the present location. This frequently results in successful proof repairs, but also might bring junk with it. The unnecessary tactic above (`rewrite H`) seems to have been propagated this way to at least 3 other repairs in that file. We confirm that the propagated rewrites are vestigial in the places it was copied by successfully recompiling the file with the rewrites removed. We suspect ProofDoctor added the unnecessary tactic once by chance and then compulsively substituted it into other proof repairs with similar circumstances.

Though compulsive substitution can lead to wasteful proofs, it does point towards the hypothesis that ProofDoctor reads and interprets the contents of the diff it is given, which is desirable. It also suggests we can guide proof repair, as in the next paragraph.

Guided Repair In the repairs above, ProofDoctor primarily used `auto with *` to replace the defunct `omega`, not `lia`, though both should work well. We reset the state of the distributed-reference-counting repository to the initial broken state and manually repaired a few proofs using `lia`. Performing manual repairs adds the diff to the model's prompt, which should suggest to ProofDoctor that it should use `lia`. We let repairs run for some time on this altered state and the resulting repairs used `lia` over 45 times while the repairs in which we did not initially suggest `lia` in the git diff used `lia` only twice. This suggests that ProofDoctor can be guided to enact certain kinds of repairs by giving it initial examples.

Proof Refactor distributed-reference-counting's proof repair attempt yielded another proof repair where the resulting proof was relatively novel from the broken proof (spacing edited for brevity):

Listing 4.5: An unusual ProofDoctor repair example.

```
 Theorem eq_message_dec : eq_dec Message.
-Proof.
- unfold eq_dec in ⊢ *; double induction a b; intros.
- auto.
+Proof. (* This proof was automatically repaired. *)
+ unfold eq_dec in ⊢ *.
- right; discriminate.
+ intros.
- right; discriminate.
+ decide equality.
- right; discriminate.
- case (eq_site_dec s0 s); intros.
- rewrite e; auto.
+ case (eq_site_dec s s0); intros.
+ auto.
- right; injection; auto.
+ right; auto.
- right; discriminate.
- right; discriminate.
- right; discriminate.
- auto.
 Qed.
```

The `double induction` tactic no longer exists, so ProofDoctor improvises and uses relevant automation (`decide equality`) to produce what turns out to be a much shorter proof than the original. This repair simultaneously alters the high-level structure of the proof and reuses bits of the original proof when relevant, demonstrating some flexibility in problem solving.

4.5 CONCLUSION AND FUTURE WORK

In this work we concretized PRISM [1]'s data to a model specific dataset, trained a model on the dataset, and implemented an end-user tool equipping the model with search capabili-

ties and other heuristics. We perform a cursory inspection of the repair tool's capabilities on real, held-out projects to demonstrate the tool can autonomously produce repair patches that fix references to outdated theorems and tactics as well as perform more sophisticated repairs on occasion. Evaluation over a large, diverse test set and user studies are important future work which should reveal deeper understanding of the model's strengths and weaknesses and motivation future improvements. A future but concrete use-case for ProofDoctor could be acting as a first responder to test suite failures for Coq libraries, automatically running upon test failure to attempt to patch broken proofs before a developer steps in. Many libraries have public test suite results, so dozens of third party libraries could be continuously monitored and any discovered repairs contributed as pull requests in an automatic fashion.

CHAPTER 5: RELATED WORK[17]

5.1 DATASETS & BENCHMARK SUITES

Our work on PRISM [1] culminated in a dataset of real repairs, but it was not the first ever collection of edits to Coq projects. The REPLica [72] user study collected incremental edit data from eight proof engineers over the course of a month. Due to difficulties recruiting participants, the dataset is too small for data-hungry ML tools. Additionally, the data collected by REPLica was not intended for training ML models, and did not collect features such as ASTs and proof states from all the sentences in the participant's projects.

A number of datasets and benchmark suites target *autoformalization*: the automatic translation of natural language mathematics to formal mathematics. Autoformalization datasets consisting of aligned natural and formal language include ProofNet [92] and the Isabelle Parallel Corpus [93]. MiniF2F [94] includes math Olympiad problems formalized in different proof assistants and is used as a benchmark for autoformalization and synthesis.

A few datasets and benchmark suites exist for proof synthesis, including CoqGym [73] for Coq and HOList [95] for HOL Light. These datasets include static data from fixed project versions. The distinguishing feature of PRISM is that it describes the project's history, which is necessary to produce repair examples.

We expect there is much that we can learn from ML for code, given the similarities between code and proofs. A summary of recent work in this space can be found in a survey paper on neurosymbolic programming [96]. Of particular interest for our work is the question of whether code distance metrics like CodeBLEU [97] will work well for formal proof.

In the field of software engineering, accessible datasets facilitate new research. For example, Defects4 [98] is a collection of bugs and patches in Java that is frequently used as a benchmark for program repair [99, 100, 101]. We hope that PRISM will spur new research in proof repair. We also hope that, by focusing on good benchmarks and metrics for success early on, we can avoid some of the methodology challenges faced in program repair [102].

5.2 THEOREM SEARCH

Our theorem search work co-exists with a large body of work in type-oriented search, such as Loogle [103] for theorems in Lean, Hoogle [104] for functions in Haskell, type-aware auto-complete for Coq [105], and built-in searches present in many interactive theorem

[17]This chapter contains previously published work [1] [40] that we have permission to reproduce.

provers [106]. Several of these tools focus on searching for fragments of types or type patterns using hand-tuned heuristics as opposed to natural language search. On the other hand, Moogle [107], released by Morph Labs, provides natural language search over Lean. Moogle was deemed useful in high profile formalization work by Terence Tao [108], but other than this, we are unaware of any publications or substantial evaluations associated with Moogle.

Alternatively, one could search an informal database of math rather than a formal database. Approach0.xyz stems from Wei Zhong's master's thesis [109] and contributes keyword and LaTeX aware search over a large collection of scraped proof-relevant webpages. A sophisticated and efficient but manual approach to matching mathematical expressions was used to determine theorem relevance, which might not adapt to unusual syntax as well as neural approaches do.

More generally, our theorem search model is an embedding model which embeds input text into semantically rich vectors. These embeddings can be used for clustering related texts, retrieving relevant information from a database (this work), or even individually as features for text classification. Specifically, our theorem search model is trained on synthetic data generated by a causal language model, emphasizing generation of challenging datasets to improve model performance. These strategies are common to other works such as SFR-Embedding-Mistral [61], which is currently ranked first on the MTEB [71] for text retrieval, as well as its predecessor E5-Mistral [62]. These models also use synthetic data and hard negative mining.

5.3 AUTOMATED PROOF SYNTHESIS

Proof synthesis can be thought of as a generalization of the proof repair task we explored where the repair information is unavailable. Due to this relationship, methods and improvements in neural proof synthesis are generally relevant to proof repair. Advances in ML have had a transformative effect on many fields, and theorem provers are not excluded. Examples of recent work on ML for synthesizing formal proofs include GPT-f [6] and HTPS [110] for Metamath and Lean; Proverbot9001 [89], ASTactic [73], Tactician [9], and DIVA [5] for Coq; and DeepHOL [95] for HOL Light. Also of note is recent work on autoformalization in Isabelle/HOL [111], Lean [92], and Coq [112]. More ML work for proofs can be found in QED at Large [113].

Our repair model uses a RAG [39] style approach to using theorems from the environment. A recent arXiv preprint from Saikat Chakraborty et al. [87], describes methods for automated proof synthesis for F* using similar RAG techniques, using their own custom-trained theorem retrieval model. Saikat Chakroborty et al attributes this method originally to LeanDojo [88].

In their method, a search over theorems from the environment is performed once at the beginning of each step, using the state of the proof as the search key, and then the top candidates are added to the context. A significant distinction between LeanDojo and Saikat et al's synthesis-search integration and our repair-search integration is that our model can run arbitrarily many searches while writing a single tactic with different criteria per search, whereas in LeanDojo's methodology only a single search, conditioned on the current state of the proof, can be run while each tactic is synthesized.

5.4 PROOF REPAIR

Proof repair is closely related to work in proof reuse [114, 115, 116], proof refactoring [117, 118, 119], and proof transformation [120]. These and other related topics in proof engineering have a long history, described in detail in the proof engineering survey paper QED at Large [113], as well as in the proof repair namesake thesis [16].

Proof repair can be viewed as program repair [121, 122] for proofs. There is a large amount of work on learning to repair programs, both symbolically (for example, in Getafix [123]) and neurally (for example, in Break-It-Fix-It [124]).

Although we believe our repair tool to be the first specialized neural proof repair tool, non-neural proof repair tools such as Pumpkin [74] and Sisyphus [125] exist which can reliably generate or partially generate repairs for specific kinds of changes, such as changes in the definitions of types or changes to a OCaml function for which Coq specifications have been proven, respectively.

Finally, Baldur [126] is a LLM-driven synthesis tool which attempts whole-proof generation for the Isabelle language. When a generated proof does not work, Baldur attempts a repair, which is another proof generation attempt augmented with information about the previous attempt and the error that occurred. This self-repair improves Baldur's proof synthesis success rate. However, Baldur was not specialized for proof repair on broken human-written proofs, only to repair its own attempts.

APPENDIX A: PROOF SEARCH APPENDIX

A.1 FULL LIST OF ARTIFACT LINKS

1. Live Web UI Demo (https://proofdb.tompreichel.com), availability not guaranteed!
2. Models & Datasets
   (a) Phase 1 training data (https://huggingface.co/datasets/tomreichel/proofdb-training-phase-1)
   (b) Phase 2 training data (https://huggingface.co/datasets/tomreichel/proofdb-training-phase-2)
   (c) Human-written test set (https://huggingface.co/datasets/tomreichel/proofdb_human_eval)
   (d) GPT-written test set (https://huggingface.co/datasets/tomreichel/proofdb-synthetic-eval)
   (e) ProofDB model (https://huggingface.co/tomreichel/proofdb)
   (f) ProofDB-HN model (https://huggingface.co/tomreichel/proofdb-HN)
3. Software Source
   (a) Web UI Source (https://github.com/tom-p-reichel/proofdb-webui/)
   (b) Client Source (https://github.com/tom-p-reichel/proofdb-webui-client)

A.2 SYNTHETIC DATA EXAMPLE

As an example, we will look at how our data pipeline generated positive synthetic search examples for the theorem Exists_vlookup from the stdpp [127] library, which is defined in the following way:

```
Lemma Exists_vlookup {A} (P : A -> Prop) {n} (v : vec A n) :
  Exists P (vec_to_list v) <-> exists i, P (v !!! i).
```
(A.1)

Firstly, we prompt a LLM (Code Llama 33B [59]) to give us a natural language description of the theorem given the source of the theorem and the surrounding source (including developer comments) as context. That prompt was as follows:

Listing A.1: Code Llama 33B prompt for explaining a theorem.

```
‘‘‘coq
{source}
‘‘‘
[INST] Given the context above, a portion of a Coq library, describe what the
    theorem or definition ‘{ident}‘ means intuitively. Here are some guidelines
    for a good response:
- Only describe the given theorem, e.g. don’t say that "P is true if and only if
    Q is true" if the theorem only says "if P then Q", even if you know that the
    stronger version is true.
- Try to preserve information about ‘{ident}‘ given in comments.
- Don’t try to re-prove the statement in your explanation.
- Try not to use symbols where words will suffice.
- Explain in plain english in a single paragraph.
[/INST]
```

For this theorem, the following description was generated:

Listing A.2: A generated theorem explanation.

```
The theorem ‘Exists_vlookup‘ states that there exists an element ‘x‘ in the
    vector ‘v‘ such that the proposition ‘P‘ is true for ‘x‘. In other words, it
    checks if there is at least one element in the vector that satisfies the given
     property. The theorem is written in a way that makes it easy to use with the
    ‘Exists‘ quantifier, which is often used in formal proofs to state that there
    exists at least one object with a certain property.
```

Then we prompt for concise searches a user might use in order to find this theorem, given the theorem's type and description as context. In order to get consistently well formatted output from the model, we found it was necessary to give a well-formatted example. The following searches were proposed by the model:

Listing A.3: Generated searches from the theorem explanation.

```
* "vector exists"
* "vector contains"
* "vector has element"
* "vector property"
* "vector quantifier"
* "vector exists property"
* "vector property exists"
```

```
* "vector contains property"
* "vector has property"
* "vector quantifier property"
* "vector property quantifier"
* "vector element property"
* "vector property element"
* "vector quantifier element"
* "vector element quantifier"
* "vector element property quantifier"
```

Finally, we perform certain preprocessing steps on the generated theorems to eliminate common failure modes of the model, such as quoting the theorem name verbatim.

A.3 VERBATIM PROMPT GIVEN TO GPT3.5 TO GENERATE TEST SYNTHETIC SEARCHES

Newlines added for readability.

Listing A.4: GPT3.5 prompt for generating synthetic searches.

```
Consider the following Coq theorem definition:

‘‘‘coq
{theorem}
‘‘‘

What are some searches a user might provide to a hypothetical natural language
    Coq theorem search engine that would best represent this theorem?
Your searches should be concise.
Don’t write full sentences.
There is no need to write in your search that you are looking for a ’Coq theorem’
     because the search engine only searches Coq theorems.
Give your response as a JSON object with a single key ’searches’ containing a
    list of strings.
```

A.4 BINARY EMBEDDING QUANTIZATION GUARANTEES PROOF SKETCH

What follows is a slightly more explicit sketch of the proof due to Xinyang Yi, Constantine Caramanis, and Eric Price, originally from their paper [50]. The proof we are sketching is

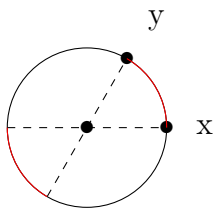
what they denote as Proposition 2.2. The scenario is as follows: we are trying to quantize embeddings in such a way that the angular distance between the vectors (and by extension the cosine similarity) is preserved, with high probability, up to a certain additive error. Suppose we have N^2 vectors we are trying to preserve. To quantize them, we generate n bits, each of which determined by which side of a randomly generated plane the point is on (Algorithm 1 from the paper). The planes all pass through the origin, so they cut any hypersphere in half. Let $d(x, y)$ denote the angular distance between two vectors normalized by dividing by π so that the maximum value is 1 and the minimum value is 0. Let $d_A(x, y)$ refer to the proportion of planes which separate x and y using the random set of planes A.

Lemma A.1. Pr[a randomly generated plane separates some arbitrary x and y] $= d(x, y)$
This is essentially equation (2.4) in the source paper, and explained there. We give a brief, intuitive explanation here.

First, if the angular distance between the points is 0, then no plane can separate them, so the probability of the plane separating them is 0, which is the angular distance between x and y, as desired.

Now consider the case where x and y are distinct points. Without loss of generality, we can assume they lie on the surface of a hypersphere, since their magnitudes don't affect the angular distance between them. Consider the unique 2D slice of the hypersphere which contains the distinct points x, y and the origin. There exists just one plane that touches x, y and the origin out of the infinite number of planes we can select, so it is almost never randomly selected, and therefore is inconsequential to the probability of separation. The rest of the planes we can select (almost all of them) will cut the diagram illustrated below in half at some random angle.

Figure A.1: Sketch of the slice containing x, y, and the origin.



We have established that our randomly selected plane cuts the diagram illustrated above in half almost surely. The probability that the random cut lands in the red region depicted above is just $d(x, y)$, which goes to 1 as the angular distance goes to π.

Lemma A.2.

$$\mathrm{E}[d_A(x, y)] = d(x, y) \tag{A.2}$$

This follows very quickly from the previous lemma. The expectation of indicator functions is just the probabilities of those indicator functions yielding 1, which in this case is $d(x, y)$ for all random planes. The expectation of the mean of indicator functions with identical probabilities is just that identical probability.

Theorem A.1.

$$\forall x, y \in K,\ P(|d_{\mathbf{A}}(x, y) - d(x, y)| \geq \delta) \leq 2e^{-2\delta^2|A|}|K|^2 \tag{A.3}$$

This corresponds to proposition 2.2 from the original paper.

This theorem states that probability that additive distortion introduced by quantization exceeds a threshold d for any pair of vectors in a set K falls exponentially with the number of planes used. First we'll consider a single arbitrary pair of vectors x, y and apply Hoeffding's inequality, which is as follows:

$$\text{When } X_i \text{ are independent random variables, } \Pr\left(\left|\sum_i X_i - \mathrm{E}\left(\sum_i X_i\right)\right| \geq \delta\right) \leq 2e^{-\frac{2\delta}{\Sigma_i r_i^2}} \tag{A.4}$$

Where r_i are the maximum value X_i can take minus the minimum value. Our X_i are whether each randomly selected plane separated x and y, and our r_i are all $\frac{1}{|A|}$, giving us:

$$P(|d_{\mathbf{A}}(x, y) - d(x, y)| \geq \delta) \leq 2e^{-2\delta^2(\sum_{i=0}^{|A|}\frac{1}{|A|^2})^{-1}} \tag{A.5}$$

The sum is equal to $\frac{1}{|A|}$, since it is $|A|$ times $\frac{1}{|A|^2}$, so:

$$\leq 2e^{-2\delta^2|A|} \tag{A.6}$$

Now we return to the case of bounding the probability of distortion for every pair of vectors $x, y \in K$. Suppose we have a dataset K of vectors. Consider the probability that any pair of vectors in our dataset has a distortion greater than δ:

$$\bigcup_{x,y\in K} P(|d_{\mathbf{A}}(x, y) - d(x, y)| \geq \delta) \tag{A.7}$$

Apply a union bound:

$$\leq |K|^2 P(|d_{\mathbf{A}}(x, y) - d(x, y)| \geq \delta) \tag{A.8}$$

Use our previous result:

$$\leq 2e^{-2\delta^2|A|}|K|^2 \tag{A.9}$$

Theorem A.2.

$$\frac{\log\left(|K|\sqrt{\frac{1}{P}}\right) + \frac{\log(2)}{2}}{\delta^2} \leq |A| \tag{A.10}$$

This is a relation between P, the probability that we observe a distortion greater than δ in the new distances between some pair of quantized vectors from K, the set of vectors to be quantized, and A, the set of planes to use. This was simply algebraically solved from the previous theorem. This is the expression we use to pick the number of planes given our tolerable levels of distortion and the probability we meet or exceed that level of distortion for any pair in the dataset.

APPENDIX B: PROOF REPAIR MODEL APPENDIX

B.1 FULL LIST OF ARTIFACT LINKS

1. Datasets
   (a) https://huggingface.co/datasets/tomreichel/PRISM-synthesis (Proof Synthesis Data)
   (b) https://huggingface.co/datasets/tomreichel/PRISM-repair (Proof Repair Data)
2. Models
   (a) https://huggingface.co/tomreichel/proof-synthesis-model (Proof Synthesis Model)
   (b) https://huggingface.co/tomreichel/proof-repair-model (Proof Repair Model
3. Repair Attempt Repos
   (a) https://github.com/tom-p-reichel/PRM-distributed-reference-counting
   (b) https://github.com/tom-p-reichel/PR-fermat4
4. Software
   (a) https://github.com/tom-p-reichel/proof-repair-tool (Proof Repair Tool)

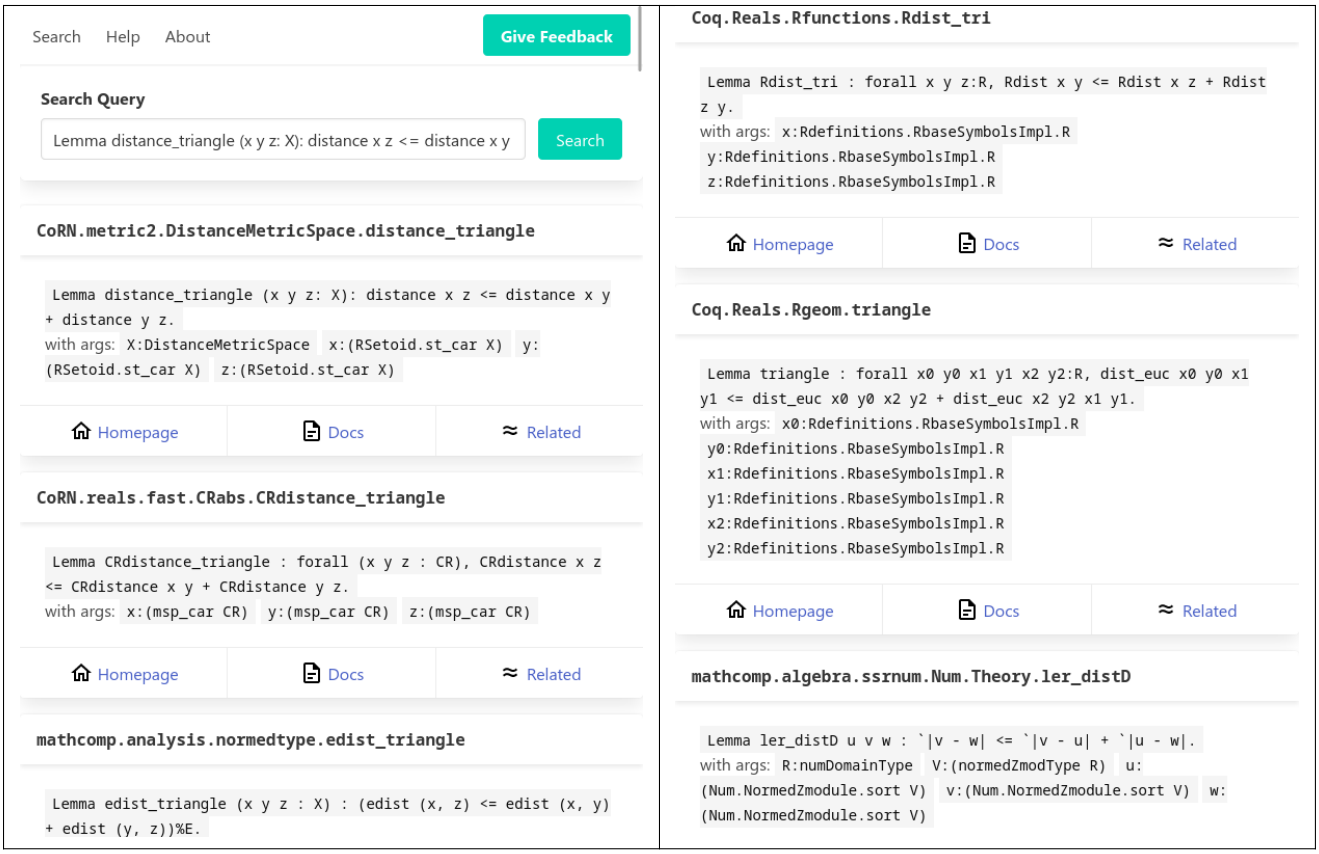
B.2 RELATED THEOREMS EXAMPLE

Even though our theorem search model was only trained to associate natural language searches with theorems, we can motivate an expectation that the distance between two theorems in the embedding space is also meaningful. The model was trained to move theorem embeddings closer to relevant natural language search embeddings, so we should expect, for a well-trained, generalizing model, that relevant theorem results are close to natural language searches in the embedding space. The embedding space is a real vector space and the relevant distance metric for retrieving search results is angular distance (corresponding to cosine similarity). In this geometry the triangle inequality holds, so we have that the angular distance between any two theorems is *upper bounded* by the sum of the angular distances from each theorem to any shared natural language search:

$$\begin{aligned}
&\forall search,\ \forall theorem_1,\ \forall theorem_2, \\
&\left(\mathrm{d}_{\text{angular}}\left(theorem_1,\ search\right) < \theta \ \ \wedge \ \ \mathrm{d}_{\text{angular}}\left(theorem_2,\ search\right) < \phi\right) \\
&\Rightarrow \mathrm{d}_{\text{angular}}(theorem_1,\ theorem_2) < \theta + \phi
\end{aligned} \tag{B.1}$$

In other words, **pairs of theorems which are both "good results" for some arbitrary search query must also be somewhat good results for each other when using the text of one of the theorem as a search query**. If we trust the model to retrieve "relevant" results for natural language queries for a particular definition of relevant, we may also believe that it can fetch theorems "relevant" to other theorem by the reasoning above. Examples of related theorems for a statement of the triangle inequality can be found in Figure B.1.

Figure B.1: Example of 6 'related theorems' on our web UI found by searching a statement of the triangle inequality.



B.3 REPAIR MODEL PROMPT EXAMPLE

What follows is the text fed to the model in a repair session for the AILS Coq project [128]. The 'commit diff' portion features the changes made to the repository, which includes proofs admitted and repaired by the tool so far. Then, the ongoing proof history describes the changes made so far from the old proof as well as 3 recommendations for tactics the model might want to try something similar to prefixed with '?' to distinguish them from the history. These recommendations are taken from the next tactics that would have been run in the old proof, as determined by prefix alignment. Next, it is given the Coq proof state. Finally, we prompt it to give us the next tactic.

Listing B.1: An example of a complete prompt fed to the repair model to generate a tactic.

```
Commit Diff:
‘‘‘
diff --git a/trajectory_def.v b/trajectory_def.v
index 8df1d99..bbc2cb8 100644
--- a/trajectory_def.v
+++ b/trajectory_def.v
@@ -53,9 +53,9 @@ Qed.
 Lemma init_tr_cond_x :
  forall (h : TypeSpeed) (t : R),
  derive_pt (fun t : R => (h * t)%R) t (init_tr_derivable_x h t) =
- (h * cos ((fun t : R => 0) t))%R.
+ (h * cos ((fun t : R => 0) t))%R. (* This proof was automatically repaired. *)
 intros; reg.
-rewrite cos_0; ring.
+rewrite cos_0 ; auto with *.
 Qed.

 Lemma init_tr_cond_y :
diff --git a/ycngftys.v b/ycngftys.v
index 2b37d96..849d138 100644
--- a/ycngftys.v
+++ b/ycngftys.v
@@ -56,7 +56,7 @@ reg; apply yi_derivable.
 Qed.

 (**********)
```

```
-Lemma d_u : forall eps : posreal, derivable (fun t : R => u eps t).
+Lemma d_u : forall eps : posreal, derivable (fun t : R => u eps t). (*
 intro.
 unfold u in |- *; unfold derivable in |- *; intro.
 reg.
@@ -71,7 +71,8 @@ apply Rplus_lt_compat_l.
 apply Rsqr_pos_lt.
 red in |- *; intro; assert (H0 := cond_pos eps); rewrite H in H0;
  elim (Rlt_irrefl _ H0).
-Qed.
+*)
+Admitted.

 Lemma fct_der2 :
  forall (eps : posreal) (t : R), derivable_pt (fun y : R => Rsqr (u eps y)) t.
@@ -266,8 +267,8 @@ reg.
 Qed.

 Lemma derive_vit :
- forall t : R, derive_pt (fun t : R => (vi * t)%R) t (vit_derivable t) = vi.
-intro; reg.
+ forall t : R, derive_pt (fun t : R => (vi * t)%R) t (vit_derivable t) = vi. (*
    This proof was automatically repaired. *)
+intro; reg. reflexivity.
 Qed.

 (**********)

‘‘‘

Ongoing Proof History:
‘‘‘
   intro.
   cut (derivable_pt (fun x : R => (vi intr * sin (thetat x - alphas x))%R) t).
   intro X.
   replace (derive_pt (fun y : R => (Rs y * derive_pt alphas y (cond_D1 alphas y)
       )%R) t (fct_der5 t)) with (derive_pt (fun x : R => (vi intr * sin (thetat x
        - alphas x))%R) t X).
```

```
   set (f := d2 alphas).
   cut (derivable_pt f t); [ intro X0 | apply (cond_D1 alphas) ].
   cut (derivable_pt thetat t); [ intro X1 | apply thetat_derivable ].
   reg.
   unfold fct_cte, minus_fct in |- *.
   replace (derive_pt f t (cond_D1 alphas t)) with (derive_pt f t X0); [ idtac |
      apply pr_nu ].
   replace (derive_pt thetat t (thetat_derivable t)) with (derive_pt thetat t X1)
      ; [ idtac | apply pr_nu ].
-  ring.
   unfold derive_pt in |- *.
   case X; case (fct_der5 t); simpl; intros.
   eapply uniqueness_limite.
+  *
-  unfold derivable_pt_abs in d0; apply d0.
+  unfold derivable_pt_lim in |- *; intros.
   unfold derivable_pt_lim in |- *; intros.
   unfold derivable_pt_abs in d; unfold derivable_pt_lim in d; elim (d eps H);
      intros.
   exists x1; intros.
-  do 2 rewrite <- Rs_alphas.
+  rewrite <- Rs_alphas .
?  apply H0; assumption.
?  fold alphas_p in |- *; reg.
?  apply (cond_D1 alphas).
```

```
Proof State:
1 focused goal (shelved: 1)

  intr : Trajectory
  t : R
  X : derivable_pt (fun x : R => (vi intr * sin (thetat x - alphas x))%R) t
  f := alphas : R -> R
  X0 : derivable_pt f t
  X1 : derivable_pt thetat t
  x : R
```

```
  d : forall eps : R,
      (0 < eps)%R ->
      exists delta : posreal,
        forall h : R,
        h <> 0%R ->
        (Rabs h < delta)%R ->
        (Rabs
           ((Rs (t + h) * derive_pt alphas (t + h) (cond_D1 alphas (t + h)) -
             Rs t * derive_pt alphas t (cond_D1 alphas t)) / h - x) < eps)%R
  x0 : R
  d0 : derivable_pt_abs
         (fun x : R => (vi intr * sin (thetat x - alphas x))%R) t x0
  eps : R
  H : (0 < eps)%R
  x1 : posreal
  H0 : forall h : R,
       h <> 0%R ->
       (Rabs h < x1)%R ->
       (Rabs
          ((Rs (t + h) * derive_pt alphas (t + h) (cond_D1 alphas (t + h)) -
            Rs t * derive_pt alphas t (cond_D1 alphas t)) / h - x) < eps)%R
  h : R
  H1 : h <> 0%R
  H2 : (Rabs h < x1)%R
  ============================
  (Rabs
     ((vi intr * (sin (thetat ?t - alphas ?t) + h) -
       Rs ?t * derive_pt alphas ?t (cond_D1 alphas ?t)) / h -
      (vi intr * cos (thetat t - f t) * proj1_sig X1 -
       vi intr * cos (thetat t - f t) * proj1_sig X0)) < eps)%R

‘‘‘
Next Tactic:
‘‘‘
```

REFERENCES

[1] T. Reichel, R. W. Henderson, A. Touchet, A. Gardner, and T. Ringer, "Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset," in *14th International Conference on Interactive Theorem Proving (ITP 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Naumowicz and R. Thiemann, Eds., vol. 268. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.26 pp. 26:1–26:20.

[2] "AI for Theorem Proving," 2016-2022. [Online]. Available: http://aitp-conference.org/

[3] "2nd MATH-AI Workshop at NeurIPS'22," 2021-2022. [Online]. Available: https://mathai2022.github.io/

[4] "Beyond Bayes: Paths Towards Universal Reasoning Systems," 2022. [Online]. Available: https://beyond-bayes.github.io/

[5] E. First and Y. Brun, "Diversity-driven automated formal verification," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)(22–27). Pittsburgh, PA, USA. https://doi.org/10.1145/3510003.3510138*, 2022.

[6] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," *CoRR*, vol. abs/2009.03393, 2020. [Online]. Available: https://arxiv.org/abs/2009.03393

[7] G. Lample, M.-A. Lachaux, T. Lavril, X. Martinet, A. Hayat, G. Ebner, A. Rodriguez, and T. Lacroix, "Hypertree proof search for neural theorem proving," 2022. [Online]. Available: https://arxiv.org/abs/2205.11491

[8] A. Agrawal, E. First, Z. Kaufman, T. Reichel, S. Zhang, T. Zhou, A. Sanchez-Stern, and T. Ringer, "Proofster." [Online]. Available: https://www.alexsanchezstern.com/papers/proofster.pdf

[9] L. Blaauwbroek, J. Urban, and H. Geuvers, "The tactician: A seamless, interactive tactic learner and prover for coq," in *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-53518-6_17 p. 271–277.

[10] D. Kühlwein, J. C. Blanchette, C. Kaliszyk, and J. Urban, "Mash: Machine learning for sledgehammer," in *Interactive Theorem Proving*, S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 35–50.

[11] M. N. Rabe and C. Szegedy, "Towards the automatic mathematician," in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds. Cham: Springer International Publishing, 2021, pp. 25–37.

[12] "Openai." [Online]. Available: https://openai.com/

[13] "Europroofnet." [Online]. Available: https://europroofnet.github.io/

[14] "Proof engineering, adaptation, repair, and learning for software (pearls)." [Online]. Available: https://sam.gov/opp/da84366306554cc981f37f703a78c698/view

[15] The Coq Development Team, "The Coq reference manual – release 8.19.0," https://coq.inria.fr/doc/V8.19.0/refman, 2024.

[16] T. Ringer, "Proof repair," Ph.D. dissertation, University of Washington, 2021.

[17] Z. Azerbayev, H. Schoelkopf, K. Paster, M. D. Santos, S. McAleer, A. Q. Jiang, J. Deng, S. Biderman, and S. Welleck, "Llemma: An open language model for mathematics," 2023.

[18] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, oct 2009. [Online]. Available: https://doi.org/10.1145/1592434.1592436

[19] B. Werner, "Une théorie des constructions inductives," Ph.D. dissertation, Paris Diderot University, 05 1994.

[20] W. A. Howard, "The formulae-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, H. Curry, H. B., S. J. Roger, and P. Jonathan, Eds. Academic Press, 1980.

[21] H. B. Curry, R. Feys, and W. Craig, "Combinatory logic, volume i," *Philosophical Review*, vol. 68, no. 4, pp. 548–550, 1959.

[22] D. Delahaye, "A Tactic Language for the System Coq," in *Proceedings of Logic for Programming and Automated Reasoning (LPAR)*, ser. LNCS/LNAI, vol. 1955, Réunion, France, Jan. 2000. [Online]. Available: https://hal.science/hal-01125070 pp. 85–95.

[23] G. Gonthier, A. Mahboubi, and E. Tassi, "A Small Scale Reflection Extension for the Coq system," Inria Saclay Ile de France, Research Report RR-6455, 2016. [Online]. Available: https://inria.hal.science/inria-00258384

[24] P. Villalobos, J. Sevilla, T. Besiroglu, L. Heim, A. Ho, and M. Hobbhahn, "Machine learning model sizes and the parameter gap," 2022. [Online]. Available: https://arxiv.org/abs/2207.02852

[25] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," 2022.

[26] T. Dettmers and L. Zettlemoyer, "The case for 4-bit precision: k-bit inference scaling laws," 2023.

[27] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," 2023.

[28] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, S. Biderman, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV, X. He, H. Hou, J. Lin, P. Kazienko, J. Kocon, J. Kong, B. Koptyra, H. Lau, K. S. I. Mantri, F. Mom, A. Saito, G. Song, X. Tang, B. Wang, J. S. Wind, S. Wozniak, R. Zhang, Z. Zhang, Q. Zhao, P. Zhou, Q. Zhou, J. Zhu, and R.-J. Zhu, "Rwkv: Reinventing rnns for the transformer era," 2023.

[29] T. Dao and A. Gu, "Transformers are ssms: Generalized models and efficient algorithms through structured state space duality," 2024.

[30] J. Liu, S. Min, L. Zettlemoyer, Y. Choi, and H. Hajishirzi, "Infini-gram: Scaling unbounded n-gram language models to a trillion tokens," 2024.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[32] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[33] Y. Zhao, Z. Lin, D. Zhou, Z. Huang, J. Feng, and B. Kang, "Bubogpt: Enabling visual grounding in multi-modal llms," 2023.

[34] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," 2023. [Online]. Available: https://arxiv.org/abs/2304.08485

[35] K. Li, Y. Wang, Y. He, Y. Li, Y. Wang, Y. Liu, Z. Wang, J. Xu, G. Chen, P. Luo, L. Wang, and Y. Qiao, "Mvbench: A comprehensive multi-modal video understanding benchmark," 2024. [Online]. Available: https://arxiv.org/abs/2311.17005

[36] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022.

[37] A. de Wynter, X. Wang, A. Sokolov, Q. Gu, and S.-Q. Chen, "An evaluation on large language model outputs: Discourse and memorization," *Natural Language Processing Journal*, vol. 4, p. 100024, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2949719123000213

[38] "multi-qa-mpnet-base-dot-v1." [Online]. Available: https://huggingface.co/sentence-transformers/multi-qa-mpnet-base-dot-v1

[39] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[40] T. Reichel and T. Ringer, "Proofdb: A prototype natural language coq search engine," to appear in 9th Conference on Artificial Intelligence and Theorem Proving (AITP), 2024.

[41] "Coq platform." [Online]. Available: https://github.com/coq/platform

[42] E. J. G. Arias, "SerAPI: Machine-Friendly, Data-Centric Serialization for COQ," HAL, Technical Report hal-01384408, 2016. [Online]. Available: http://dml.mathdoc.fr/item/hal-01384408/

[43] E. J. G. Arias and T. Martinez, "Pycoq." [Online]. Available: https://github.com/ejgallego/pycoq

[44] "Serapi 'classic mode' final release notice." [Online]. Available: https://github.com/ejgallego/coq-serapi/issues/252#issuecomment-1365510329

[45] E. J. G. Arias, "coq_lsp." [Online]. Available: https://github.com/ejgallego/coq-lsp

[46] P. Carrott, N. Saavedra, K. Thompson, S. Lerner, J. F. Ferreira, and E. First, "Coqpyt: Proof navigation in python in the era of llms," 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269614455

[47] "coqtop.py." [Online]. Available: https://github.com/coq/coq/blob/32533fb6fcdda3c085b25ff6a46a5dc3b3e50bd5/doc/tools/coqrst/repl/coqtop.py

[48] "Document the [.glob] file format." [Online]. Available: https://github.com/ejgallego/coq/commit/8770cc7040aafc566fa141fb29ebfab57d0aa6b9

[49] "Copilot+ pcs hardware requirement." [Online]. Available: https://support.microsoft.com/en-us/topic/copilot-pcs-hardware-requirements-35782169-6eab-4d63-a5c5-c498c3037364

[50] X. Yi, C. Caramanis, and E. Price, "Binary embedding: Fundamental limits and fast algorithm," 2019.

[51] N. Kasliwal, "Binary quantization - vector search, 40x faster." [Online]. Available: https://qdrant.tech/articles/binary-quantization/

[52] D. Koenig and A. Shakir, "64 bytes per embedding, yee-haw," 2024. [Online]. Available: https://www.mixedbread.ai/blog/binary-mrl

[53] N. Reimers. [Online]. Available: https://cohere.com/blog/int8-binary-embeddings

[54] Z. Ge, L. Kaushik, M. Omote, and S. Kumar, "Speed up Training with Variable Length Inputs by Efficient Batching Strategies," in *Proc. Interspeech 2021*, 2021, pp. 156–160.

[55] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever, "Deep double descent: Where bigger models and more data hurt," 2019. [Online]. Available: https://arxiv.org/abs/1912.02292

[56] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[57] S. Dash, I. Lyngaas, J. Yin, X. Wang, R. Egele, G. Cong, F. Wang, and P. Balaprakash, "Optimizing distributed training on frontier for large language models," 2023.

[58] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[59] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024.

[60] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, pp. 539–546 vol. 1.

[61] R. Meng, Y. Liu, S. R. Joty, C. Xiong, Y. Zhou, and S. Yavuz, "Sfr-embedding-mistral: Enhance text retrieval with transfer learning." [Online]. Available: https://blog.salesforceairesearch.com/sfr-embedded-mistral/#teacher-models-for-hard-negative-mining

[62] L. Wang, N. Yang, X. Huang, L. Yang, R. Majumder, and F. Wei, "Improving text embeddings with large language models," 2024.

[63] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2015. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2015.7298682

[64] H. Xuan, A. Stylianou, X. Liu, and R. Pless, "Hard negative examples are hard, but useful," 2021. [Online]. Available: https://arxiv.org/abs/2007.12749

[65] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[66] "Pretrained models – sentence-transformers documentation." [Online]. Available: https://sbert.net/docs/pretrained_models.html

[67] "all-mpnet-base-v2." [Online]. Available: https://huggingface.co/sentence-transformers/all-mpnet-base-v2

[68] "all-distilroberta-v1." [Online]. Available: https://huggingface.co/sentence-transformers/all-distilroberta-v1

[69] "multi-qa-distilbert-cos-v1." [Online]. Available: https://huggingface.co/sentence-transformers/multi-qa-distilbert-cos-v1

[70] "multi-qa-minilm-l6-v2." [Online]. Available: https://huggingface.co/sentence-transformers/multi-qa-MiniLM-L6-v2

[71] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, "Mteb: Massive text embedding benchmark," *arXiv preprint arXiv:2210.07316*, 2022. [Online]. Available: https://arxiv.org/abs/2210.07316

[72] T. Ringer, A. Sanchez-Stern, D. Grossman, and S. Lerner, "REPLica: REPL instrumentation for Coq analysis," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3372885.3373823 p. 99–113.

[73] K. Yang and J. Deng, "Learning to prove theorems via interacting with proof assistants," in *International Conference on Machine Learning (ICML)*, Long Beach, CA, USA, 2019. [Online]. Available: http://proceedings.mlr.press/v97/yang19a/yang19a.pdf

[74] T. Ringer, N. Yazdani, J. Leo, and D. Grossman, "Adapting proof automation to adapt proofs," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3167094 p. 115–129.

[75] T. Ringer, R. Porter, N. Yazdani, J. Leo, and D. Grossman, "Proof repair across type equivalences," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3453483.3454033 p. 112–127.

[76] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002. [Online]. Available: https://doi.org/10.3115/1073083.1073135 p. 311–318.

[77] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: how should we assess quality of the code generation models?" *arXiv preprint arXiv:2208.03133*, 2022.

[78] "pycoq." [Online]. Available: https://github.com/IBM/pycoq

[79] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022283681900875

[80] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: https://www.jstor.org/stable/2098689

[81] M. M. Deza and E. Deza, *Encyclopedia of Distances*, 3rd ed. Berlin: Springer Berlin Heidelberg, Oct. 2014. [Online]. Available: https://link.springer.com/book/10.1007/978-3-642-30958-8

[82] O. Pele and M. Werman, "Fast and robust earth mover's distances," in *2009 IEEE 12th International Conference on Computer Vision*, 2009, pp. 460–467.

[83] "Proposal: a custom build tool for coq projects." [Online]. Available: https://coq.discourse.group/t/proposal-a-custom-build-tool-for-coq-projects/239/2

[84] "Is there a full documentation of coq's grammar?" [Online]. Available: https://coq.discourse.group/t/is-there-a-full-documentation-of-coqs-grammar/647/10

[85] "Query ast returns empty result." [Online]. Available: https://github.com/ejgallego/coq-serapi/issues/117

[86] S. D. Zhang, T. Ringer, and E. First, "Getting more out of large language models for proofs," 2023.

[87] S. Chakraborty, G. Ebner, S. Bhat, S. Fakhoury, S. Fatima, S. Lahiri, and N. Swamy, "Towards neural synthesis for smt-assisted proof-oriented programming," 2024.

[88] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," 2023.

[89] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner, "Generating correctness proofs with neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1907.07794

[90] D. Delahaye and M. Mayero, "Diophantus' 20th problem and fermat's last theorem for n=4: Formalization of fermat's proofs in the coq proof assistant," 2005. [Online]. Available: https://arxiv.org/abs/cs/0510011

[91] L. Moreau and J. Duprat, "A construction of distributed reference counting," *Acta Informatica*, vol. 37, 03 2000.

[92] Z. Azerbayev, B. Piotrowski, H. Schoelkopf, E. W. Ayers, D. Radev, and J. Avigad, "Proofnet: Autoformalizing and formally proving undergraduate-level mathematics," 2023.

[93] A. Bordg, Y. A. Stathopoulos, and L. C. Paulson, "A parallel corpus of natural language and isabelle artefacts," in *7th Conference on Artificial Intelligence and Theorem Proving (AITP)*, 2022. [Online]. Available: http://aitp-conference.org/2022/abstract/AITP_2022_paper_8.pdf

[94] K. Zheng, J. M. Han, and S. Polu, "minif2f: a cross-system benchmark for formal olympiad-level mathematics," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=9ZPegFuFTFv

[95] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox, "Holist: An environment for machine learning of higher order logic theorem proving," in *ICML*, 2019.

[96] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, Y. Yue et al., "Neurosymbolic programming," *Foundations and Trends® in Programming Languages*, vol. 7, no. 3, pp. 158–243, 2021.

[97] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[98] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of existing faults to enable controlled testing studies for Java programs," in *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA, July 2014, tool demo. pp. 437–440.

[99] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic repair of real bugs: An experience report on the defects4j dataset," *CoRR*, vol. abs/1505.07002, 2015. [Online]. Available: http://arxiv.org/abs/1505.07002

[100] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3180155.3180233 p. 1–11.

[101] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397369 p. 101–114.

[102] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2771783.2771791 p. 24–36.

[103] J. Breitner, "Loogle!" [Online]. Available: https://loogle.lean-lang.org/

[104] N. Mitchell, "Hoogle." [Online]. Available: https://hoogle.haskell.org/

[105] H. Dalland, J. Israelsen, and S. Kristensen, "Expanding coq with type aware code completion," 2023. [Online]. Available: https://github.com/Jakobis/vscoqComparison/blob/main/Expanding_Coq_with_Type_Aware_Code_Completion.pdf

[106] "Vernacular commands." [Online]. Available: https://coq.inria.fr/doc/master/refman/proof-engine/vernacular-commands.html?highlight=search#coq:cmd.Search

[107] Morph, "Moogle: Semantic search over mathlib4," 2023. [Online]. Available: https://moogle.ai

[108] T. Tao, 2023. [Online]. Available: https://mathstodon.xyz/@tao/111360264941915326

[109] W. Zhong, "A novel similarity-search method for mathematical content in latex markup and its implementation," 2015. [Online]. Available: http://tkhost.github.io/opmes/thesis-ref.pdf

[110] G. Lample, M.-A. Lachaux, T. Lavril, X. Martinet, A. Hayat, G. Ebner, A. Rodriguez, and T. Lacroix, "Hypertree proof search for neural theorem proving," *arXiv preprint arXiv:2205.11491*, 2022.

[111] Y. Wu, A. Q. Jiang, W. Li, M. N. Rabe, C. Staats, M. Jamnik, and C. Szegedy, "Autoformalization with large language models," *arXiv preprint arXiv:2205.12615*, 2022.

[112] G. Cunningham, R. C. Bunescu, and D. Juedes, "Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs," 2023. [Online]. Available: https://arxiv.org/abs/2301.02195

[113] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, "QED at large: A survey of engineering of formally verified software," *CoRR*, vol. abs/2003.06458, 2020. [Online]. Available: https://arxiv.org/abs/2003.06458

[114] A. Felty and D. Howe, "Generalization and reuse of tactic proofs," in *Logic Programming and Automated Reasoning: 5th International Conference*, ser. LPAR '94. Berlin, Heidelberg: Springer, 1994, pp. 1–15.

[115] T. Ringer, N. Yazdani, J. Leo, and D. Grossman, "Ornaments for proof reuse in coq," in *Interactive Theorem Proving*, 2019.

[116] O. Boite, "Proof reuse with extended inductive types," in *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings.* Berlin, Heidelberg: Springer, 2004, pp. 50–65.

[117] K. Wibergh, "Automatic refactoring for agda," M.S. thesis, Chalmers University of Technology and University of Gothenburg, 2019.

[118] I. J. Whiteside, "Refactoring proofs," Ph.D. dissertation, University of Edinburgh, November 2013. [Online]. Available: http://hdl.handle.net/1842/7970

[119] V. Robert, "Front-end tooling for building and maintaining dependently-typed functional programs," Ph.D. dissertation, UC San Diego, 2018.

[120] F. Pfenning, "Proof transformations in higher-order logic," Ph.D. dissertation, Carnegie Mellon University Pittsburgh, 1987.

[121] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, jan 2018. [Online]. Available: https://doi.org/10.1145/3105906

[122] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3180155.3182526 p. 1219.

[123] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360585

[124] M. Yasunaga and P. Liang, "Break-it-fix-it: Unsupervised learning for program repair," in *International Conference on Machine Learning*. PMLR, 2021, pp. 11 941–11 952.

[125] K. Gopinathan, M. Keoliya, and I. Sergey, "Mostly automated proof repair for verified libraries," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591221

[126] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2303.04910

[127] "stdpp." [Online]. Available: https://gitlab.mpi-sws.org/iris/stdpp

[128] O. Desmettre, "Proof of ails algorithm," 2002. [Online]. Available: https://github.com/coq-contribs/ails/