Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset

Tom Reichel
University of Illinois Urbana-Champaign, IL, USA

R. Wesley Henderson
Radiance Technologies, Inc., Ruston, LA, USA

Andrew Touchet
Radiance Technologies, Inc., Ruston, LA, USA

Andrew Gardner[1]
Radiance Technologies, Inc., Ruston, LA, USA

Talia Ringer[1]
University of Illinois Urbana-Champaign, IL, USA

Abstract

We report on our efforts building a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset has been a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide recommendations for how the proof assistant community can address them. Our hope is to make it easier to build datasets and benchmark suites so that machine-learning tools for proofs will move to target the tasks that matter most and do so equitably across proof assistants.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its engineering → Software maintenance tools; Security and privacy → Logic and verification

Keywords and phrases proof repair, datasets, benchmarks, machine learning, formal proof

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.26

Supplementary Material
Text (Appendix): `https://dependenttyp.es/pdf/repairdataappendix.pdf`
Dataset (Sample): `https://doi.org/10.5281/zenodo.7935207`

Funding This research was developed with funding from the Defense Advanced Research Projects Agency. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

1 Introduction

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 [7, 4, 12], papers [26, 45, 37], tools [2, 13, 35], industrial research groups [47, 5], and funding opportunities [3, 6] centering on or prominently featuring ML for proofs. The surge in interest blurs the line between proofs and data so that any proof development, once released, may itself become data to improve proof automation for future proof developments.

[1] Co-senior authors

```
Lemma proc_rspec_crash_refines_op T (p : proc C_Op T)
  (rec : proc C_Op unit) spec (op : A_Op T) :
  (forall sA sC,
-    absr sA sC tt -> proc_rspec c_sem p rec (refine_spec spec sA)) ->
-   (forall sA sC, absr sA sC tt -> (spec sA).(pre)) ->
+    absr sA (Val sC tt) -> proc_rspec c_sem p rec (refine_spec spec sA)) ->
+   (forall sA sC, absr sA (Val sC tt) -> (spec sA).(pre)) ->
  (forall sA sC sA' v,
-    absr sA' sC tt ->
+    absr sA' (Val sC tt) ->
    (spec sA).(post) sA' v -> (op_spec a_sem op sA).(post) sA' v) ->
  (forall sA sC sA' v,
-    absr sA sC tt ->
+    absr sA (Val sC tt) ->
    (spec sA).(alternate) sA' v -> (op_spec a_sem op sA).(alternate) sA' v) ->
  crash_refines absr c_sem p rec (a_sem.(step) op)
    (a_sem.(crash_step) + (a_sem.(step) op;; a_sem.(crash_step))).
```

Figure 1 Changes made to a lemma by a participant in a recent user study of proof engineers, from the REPLica user study paper [52].

We in the proof engineering community have agency in how this surge of interest plays out. We can develop datasets and benchmark suites that steer the ML community toward the tasks that matter most. We can build infrastructure that makes it easy to develop those datasets and benchmark suites, or to work on those tasks. And we can build evaluation methodologies that measure success on those tasks in ways that truly matter, so that state-of-the-art results on benchmarks will transfer smoothly to real-world improvements in proof automation.

This paper takes a step in that direction. In particular, it presents the initial release of Proof Repair Infrastructure for Supervised Models (PRISM) – a dataset and benchmark suite for an important proof automation task in Coq: *proof repair* [49], which comprises the automatic correction of proofs in response to breaking changes in programs or specifications. 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 [52]. This paper highlights the challenges involved in collecting repair data with an emphasis on how the proof engineering community can adapt to those challenges as ML becomes increasingly relevant. Its contributions are the following:

1. an initial release of a Coq proof repair dataset and benchmark suite accessible to ML experts (Section 3),
2. reusable tools for building and extracting information from Coq projects (Section 4), and
3. a discussion of the challenges we encountered and how to overcome them (Section 5).

Our overarching goal is to build the infrastructure and proof assistant community support we need to steer the ML community toward the tasks that matter most (Section 2).

2 Overview

The necessity and difficulty of proof maintenance has been borne out empirically. A recent user study of eight intermediate through expert proof engineers showed that maintenance happened constantly for participating proof engineers during proof development [52] and that even experts sometimes gave up in the face of change [49].

Consider, for example, the change in Figure 1, in which a user study participant updated a lemma statement in response to a change in a dependency. As noted in the user study paper, this was part of a larger change, with 10 other definitions or lemma statements changing in analogous ways. Furthermore, this change broke at least five proofs, four of which the user study participant – an expert proof engineer – admitted or aborted rather than repair.

The ubiquity of maintenance and the challenges of repair have been largely neglected in ML tools for proofs. ML tools for proofs have instead historically fixated on tasks like predicting tactics or automatically formalizing natural language [13, 37, 45]. The lack of a good dataset and benchmark suite obstructs progress; what currently exists is not sufficient for training and evaluating proof repair models (see Section 6). If the datasets and benchmark suites are not fit for maintenance tasks, the ML community may neglect those tasks entirely, instead chasing state-of-the-art results on tasks for which existing benchmark suites suffice.

Our experiences interacting with ML experts and building datasets ourselves suggest that the choice of datasets and benchmark suites for a domain is not driven solely by what is likely to be useful – it is also driven by barriers imposed by infrastructure, lack of domain expertise, or social factors. Things we may take for granted, like parsing and checking proofs, can become prohibitive infrastructure challenges for ML experts.

In this paper, we describe our efforts to overcome these challenges and build a large proof repair dataset for Coq. We also discuss the barriers we do encounter, and we describe both how we overcome those barriers, and what kind of work the proof assistant community would need to put in to make it so that they cease to be barriers at all.

We find this work especially prudent given that the danger of chasing benchmarks that may not transfer to real life workflows has been realized quite dramatically in other domains, from incorrect patches to programs [46] to incorrect clinical interpretation of x-ray results [67]. Furthermore, these challenges can influence not just the tasks that the ML community chooses to tackle but even the very proof assistants for which the ML community chooses to build supporting tools. It is in the community's best interest to drive strong, practical results for useful tasks in a way that is equitable across proof assistants.

Our hopes are twofold. First, we hope that our dataset will be immediately useful for proof repair. Second, and perhaps more prudently, we hope our discussion of the challenges involved in building it will serve as a call for action to improve infrastructure. The proof engineering community can then ensure that ML experts focus on the automation tasks that matter most to the community, that they measure success on those tasks in ways that transfer smoothly to real-world usefulness, and that they do so equitably across proof assistants.

3 A Proof Repair Dataset

An initial release of the PRISM dataset and benchmark suite that we have assembled is publicly available (see Supplementary Material); we will continue to update the release with later versions as we mine more data. The task that PRISM focuses on is proof repair (Section 3.1). The data comprise aligned Git commits that correspond to existing changes in proof developments found on GitHub (Section 3.2). Success on the resulting benchmark is evaluated in terms of successful proof checking for repaired proofs (Section 3.3).

3.1 The Task: Proof Repair

In ML, a *task* refers to a high-level input/output specification of what is being learned. A dataset and benchmark suite typically organizes itself around a particular task while remaining agnostic to the details of the model implementation.

We define proof repair as an ML task with **inputs** comprising an old theorem statement, proof of the old theorem, and a new theorem statement; and **outputs** comprising a proof of the new theorem. Note that this particular task assumes that we already know how to repair the theorem statement and its dependencies. We could also consider a second task that allows the model to repair the specification itself. This second repair task would be harder to evaluate, so we do not focus our benchmark suite on it at this time, even though PRISM supports it.

Inputs. We aim to provide sufficient context in the data to support a wide range of ML approaches. At a high level, the input to the ML repair model is the entire state of a project where the approach dictates how much of this state (and in what form) actually reaches the model. More precisely, the input comprises the statement of the theorem whose proof should be repaired, any contextual definitions on which it depends, the step-by-step (and subgoals) goals and hypotheses for each sentence in the old proof, and known changes to the project up to and including changes in the theorem statement and its dependencies. For each of these components, we supply raw text representations, abstract syntax trees (ASTs), and identifiers. In the case of goals and hypotheses, serialized Coq kernel representations supply detailed internal proof states. Environmental dependencies such as Coq compiler versions are captured for errors induced by external application programming interface (API) changes.

Outputs. The ultimate output is the repaired proof, which takes the form of text that may be generated one sentence at a time, all at once, or through targeted modifications of existing sentences. In the case of supervised repair learning, we supply ground truth targets in the same form as the inputs: raw text, ASTs, etc. for the entire repaired project's state (compactly represented as a "diff" relative to the input). Sufficient context is provided in the data to programmatically execute up to the error in an interactive REPL such as `coqtop` or `sertop`, where one may apply reinforcement learning akin to CoqGym [65].

3.2 The Data: Aligned Git Commits

The training and testing data comprise aligned Git commits for a selection of realistic Coq projects. The initial release of PRISM spans just a few Coq projects and consists of roughly 200 unique changes. We are working to continue to grow PRISM in the short term to span the 60 Coq projects listed in the appendix, and in the long term to reach even more projects in the long term. An initial versions of the dataset and a summary of the projects we are considering for the next version can be found in supplementary material.

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 [65] 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 4.2.

Repair Examples. 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, 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 4.5.

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 compact 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.

Data Split. ML datasets and benchmark suites often include a data split between training, validation, and testing data. We do not commit to a split ahead of time, but we consider two different ways of splitting data: across projects and chronologically within projects. These two splits test two different kinds of generalization beyond the training data. We plan to include defaults for both splits in the final release of PRISM.

The first split – across projects – chooses distinct sets of projects to use for training, validation, and test data. This approach measures generalization of the learned model to new projects not seen at training time. The second split – chronologically within projects – uses the same set of projects for training and test data, but splits them by time of commit, so that training data includes earlier commits, and testing data includes later commits for the same projects. This approach measures generalization of the learned model to new changes within a given project, when the model was trained on older data for that project.

3.3 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 [49]. 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 [65] 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 [26] and for early symbolic work on proof repair when specifications do not change [53]. They continue to benefit ML for proof repair.

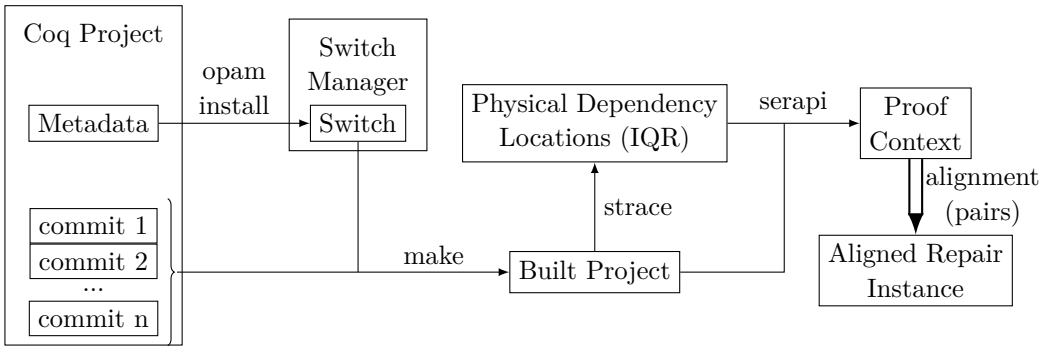

Figure 2 Process of extraction for a Coq project commit.

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 [52]. 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 [51] repairs definitions to be equivalent up to **univalent transport**, but checking this automatically is undecidable.
- Common natural language **distance metrics** like BLEU [42] are poor measures of success in code tasks [24], 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.

4 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 2):

- We obtain a *switch* (opam virtual environment) that satisfies as many of the project's dependencies as possible using the **Switch Manager (SwiM)** (Section 4.1).
- Once we have a switch, we run the build command in the generated switch to produce a **Built Project** (Section 4.2).
- We `strace` the build process to scrape the **Physical Dependency Locations (IQR flags)** of each document in the project (Section 4.3).

- Using the IQR flags for each document in a built project along with the Coq serializer SerAPI [8], 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 4.4).
- Finally, we align changed proofs across commits and save those along with their intermediate proof contexts to arrive at an **Aligned Repair Instance** (Section 4.5).

Building this infrastructure was a significant undertaking with many challenges encountered along the way; we discuss these challenges in Section 5. Our hope is that the infrastructure we have built will make it easier to collect similar datasets in the future.

4.1 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.

4.2 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.

4.3 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 [20]. 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.

4.4 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, 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.

4.5 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 [58], 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:

$$\begin{aligned} &\underset{U,\,W}{\text{minimize:}} \sum_{u \in U} C(u, f(u)) \\ &\text{subject to: } f : U \leftrightarrow W \ \wedge\ U \subseteq X \ \wedge\ W \subseteq Y \ \wedge\ \|U\| = \min(\|X\|, \|Y\|) \end{aligned} \tag{1}$$

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 [41].

The optimization is parameterized by the choice of C, for which we choose a normalized variant of the Levenshtein edit distance E:

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

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 [22], 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 [43]. 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.

5 Challenges

Collecting and building datasets and benchmark suites for many tasks is still extremely challenging, and it is challenging in a way that is not at all equitable across proof assistants. Here, we discuss our experiences dealing with challenges encountered during the creation of PRISM (Section 5.1), what we believe the Coq community can learn from other proof assistant communities (Section 5.2), and how the proof assistant community at large could address them more sustainably going forward (Section 5.3).

5.1 Our Experiences

The major challenges we faced in building this dataset and benchmark suite chiefly fall into two categories: Project Management (Section 5.1.1) and Parsing & Serialization (Section 5.1.2). For each of these categories, we discuss our experiences dealing with each stated challenge.

5.1.1 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 collections 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 [19]. 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 [19]. 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 4.3) 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.

Listing 1 A notation that breaks `CoqIDE`'s parser. This example was found in the Coq Discourse [17].

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

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 4.1) ultimately handles.

5.1.2 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 [8] 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 [65] 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.
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 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 4.3 for simplicity and maximal portability between build environments.

Listing 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 *)
```

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: 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 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 [28]. SerAPI itself has as of a few days ago been deprecated in favor of a new serializer [29, 18].

5.2 Other Proof Assistants

Here, we discuss features in other proof assistants in the context of our experiences above.

Project Management. In summary, we are not aware of an elegant and effective solution to package management for other proof assistants, but we believe Isabelle's rich archival culture sets a good example to follow. In Isabelle, the Archive of Formal Proofs (AFP) provides a highly centralized, standard host for proof developments and eases their association with metadata that may be useful for ML. The AFP also neatly versions proof developments for every official release of Isabelle and semantically groups them in different folders. At the time of writing, the AFP includes 725 proof developments [1], and it already forms the basis of a static dataset for Isabelle [32]. We suspect the AFP would also make a very strong basis for a proof repair dataset due to its neat versioning.

Agda possesses its own library management system, which it uses in combination with Hackage, the Haskell package repository. Anecdotally, researchers we have spoken to cite installation difficulties as a barrier to learning Agda. Lean also has its own package manager but lacks advanced features to address the problems we faced. Isabelle in general takes an IDE-centric approach to builds and other tooling [50, 60], but does include a notion of sessions that can inherit from other sessions. The underlying functionality the IDE is based on is also accessible in Scala and by command line. However, one of the authors has found that students learning Isabelle/HOL in a proof automation course struggle to understand how to build dependencies.

Parsing & Serialization. A particularly successful example of an interoperable proof system is MetaMath [39], whose syntax and semantics are so simple that its verifier has been reimplemented in under 1000 lines of Python [61]. This and its centralized proof database has made it a popular choice for ML experts as a benchmark for ML applications in theorem proving [37, 45] as all barriers to serialization can be avoided by modifying a very small parser/verifier. As a trade off, MetaMath does not have a comparable feature set to ITPs like Coq, Lean or Isabelle.

More complicated and featureful ITPs have more varied methods: PISA [33] is a bleeding edge Isabelle interaction and proof serialization tool written to support an ML experiment [32]. This complements `scala-isabelle` [57], an earlier, less ML-oriented tool which is also actively maintained. As for Lean, PACT [31] presented a dataset (LeanStep) aimed at ML applications that uses Lean's meta-programming facilities to serialize Lean. LeanStep's tools weigh under 1500 lines of code, which is light compared to line-counts for other serialization efforts.

5.3 Recommendations

Project Management. A strong archival effort is the best way forward, though even the best archival infrastructures for proof assistants fall short in multiple ways. For example, while Isabelle's AFP makes a natural data source for ML tools for proofs, it does not include any processes for informed consent – the datasets that build on it assume that all publicly available data is fair game. While this assumption is standard in ML for programs and proofs, it is not ideal; archival is a natural place to consider it.

In addition, though archival makes it possible to associate metadata with proof developments, little consideration is given to metadata that associates definitions and proofs *across versions* of a proof development. Presently, we rely on package management tools such as opam, which also posed challenges. Though a legitimate argument can be made that package management targets a very different use case from ours and that existing tools are sufficient for that use case, shared high-level libraries and tools on top of existing package managers and in support of bulk efforts like our own would be especially advantageous since such efforts are common when building ML datasets and tools. Nonetheless, package managers themselves warrant some improvement. For example, the problem we encountered of copying switches was due to poor caching of build dependencies, which itself was due to some degree to hard-coding of paths.

Parsing & Serialization. The great flexibility afforded Coq by its extensible grammar allows documents to be more human-friendly and readable, but the lack of syntactic assurances introduces major headaches for automated systems. Ideally, future languages will be structured to be machine-readable human/compiler-out-of-the-loop or to at least provide a public parsing API. For Coq, exposing the classification of a Vernacular command[1] in SerAPI would help substantially and obviate the need for many of the workarounds detailed above.

We also recommend a greater emphasis on backwards compatibility and backporting as several useful and even critical features that exist in newer versions of Coq or SerAPI were not suitable for our use. To this end, SerAPI is "still a research, experimental project, and it is expected to evolve considerably" [27] For instance, future plans rebase SerAPI on the language server protocol standard [29], which exposes features like document overviews that appear to list all the definitions in the file and the ability to fold proofs, implying that it has the capability to list theorems and gather the associated lines – one of our current challenges.

[1] See the `vernac_classification` type.

Overall. Based on our observations, we make the following broad recommendations for the proof assistant community going forward:

1. Work with the Isabelle community to learn how to build **centralized archives** as successful as the AFP for other proof assistants.
2. Include an **informed consent form** in any centralized archive that allows proof engineers to opt in or out of their developments being used for ML tools.
3. Determine what **kinds of metadata** within and across proof developments would ease the creation of ML tools for the tasks that matter most, make it easy to **track that metadata** inside of any centralized archive, and create standard ways of **associating that metadata** with proof developments even outside of centralized archives.
4. Establish or adopt **open standards** for managing proof developments and interfacing with external tools like modern IDEs.
5. Develop tools based on these standards to enable **extraction of metadata** relevant to ML tools from non-archival proof developments.
6. Help proof engineers **port legacy proof developments** to meet those standards, and continue to work on tools for **proof repair and reuse** that ease this burden.
7. Consider building **shared libraries and tools** optimized for the problems of bulk builds.
8. Consider limiting the scope of possible **dependency complexity**.
9. Improve **build caching** across multiple or bulk builds, for example by avoiding hard-coded paths seen in opam.
10. Consider opening conversations with **language developers and companies** inside and outside of verification about their solutions for package management, distribution, and release management, as these problems are pervasive across all software.

6 Related Work

Datasets & Benchmark Suites. The REPLica [52] 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. PRISM is less incremental, but we expect the final version of the dataset to be much larger. The REPLica data may make a useful supplement to PRISM.

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 [9] and the Isabelle Parallel Corpus [15]. MiniF2F [68] 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 [65] for Coq and HOList [11] 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 [16]. Of particular interest for our work is the question of whether code distance metrics like CodeBLEU [48] will work well for formal proof.

In the field of software engineering, accessible datasets facilitate new research. For example, Defects4 [34] is a collection of bugs and patches in Java that is frequently used as a benchmark for program repair [23, 59, 38]. 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 [46].

Proof Repair. The ML task that our dataset focuses on is proof repair, which is summarized in the namesake thesis [49]. There is not yet published work we are aware of for ML for proof repair, though we are aware of ongoing work by other teams in proof assistants other than Coq. We plan to train and evaluate at least two distinct proof repair models in Coq using PRISM, and we hope that PRISM makes it easy for others to do the same.

Proof repair is closely related to work in proof reuse [25, 54, 14], proof refactoring [63, 62, 55], and proof transformation [44]. These and other related topics in proof engineering have a long history, described in detail in the proof engineering survey paper QED at Large [50], as well as in the proof repair namesake thesis [49].

Proof repair can be viewed as program repair [40, 30] for proofs. There is a large amount of work on learning to repair programs, both symbolically (for example, in Getafix [10]) and neurally (for example, in Break-It-Fix-It [66]). This work may provide useful insights when building ML datasets, benchmark suites, and models for proof repair, though care must be taken to consider the differences between typical programs and formal proof developments [49].

Machine Learning for Proofs. 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 [45] and HTPS [36] for Metamath and Lean; Proverbot9001 [56], ASTactic [65], Tactician [13], and DIVA [26] for Coq; and DeepHOL [11] for HOL Light. Also of note is recent work on autoformalization in Isabelle/HOL [64], Lean [9], and Coq [21]. More ML work for proofs can be found in QED at Large [50]. Our main goal is to expand the scope of tasks covered in ML for proofs, reaching important tasks not previously explored.

7 Conclusions & Future Work

We have described the initial version of a novel dataset and benchmark suite for the ML task of proof repair centered around the Coq Proof Assistant. We expect later versions of the data to be significantly larger than any existing alternative, spanning years-long developments collected from a corpus of open-source Github repositories. We discussed challenges that we encountered during the creation of the dataset and ramifications for the proof engineering community going forward. The tools developed to overcome these challenges enable subsequent expansion of the dataset with supplementary Coq projects and are likely to be useful for creating and interfacing with datasets for other proof-related ML tasks including, for example, proof synthesis.

Moving forward, our immediate plans are to continue to grow the dataset, and to release the infrastructure we built for more general use. We also plan to use the dataset to build ML models for proof repair in Coq. We would also like to develop better metrics for measuring success at repairing definitions. Finally, we hope to work with the rest of the proof assistant community to address the many challenges we have highlighted, so that we may steer ML for proofs in the right direction.

References

1 Statistics - archive of formal proofs. URL: `https://www.isa-afp.org/statistics/`.

2 Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, and Talia Ringer. Proofster. URL: `https://www.alexsanchezstern.com/papers/proofster.pdf`.

3 Europroofnet. URL: `https://europroofnet.github.io/`.

4 2nd MATH-AI Workshop at NeurIPS'22, 2021-2022. URL: `https://mathai2022.github.io/`.

5 Openai. URL: https://openai.com/.

6 Proof engineering, adaptation, repair, and learning for software (pearls). URL: https://sam.gov/opp/da84366306554cc981f37f703a78c698/view.

7 AI for Theorem Proving, 2016-2022. URL: http://aitp-conference.org/.

8 Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for COQ. Technical Report hal-01384408, HAL, 2016. URL: http://dml.mathdoc.fr/item/hal-01384408/.

9 Zhangir Azerbayev, Bartosz Piotrowski, and Jeremy Avigad. ProofNet: A benchmark for autoformalizing and formally proving undergraduate-level mathematics problems. In *Second MATH-AI Workshop*, 2022. URL: https://mathai2022.github.io/.

10 Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360585.

11 Kshitij Bansal, Sarah M Loos, Markus N Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *ICML*, 2019.

12 Beyond Bayes: Paths Towards Universal Reasoning Systems, 2022. URL: https://beyond-bayes.github.io/.

13 Lasse Blaauwbroek, Josef Urban, and Herman 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*, pages 271–277, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53518-6_17.

14 Olivier 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*, pages 50–65, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-30142-4_4.

15 Anthony Bordg, Yiannos A Stathopoulos, and Lawrence C Paulson. A parallel corpus of natural language and isabelle artefacts. In *7th Conference on Artificial Intelligence and Theorem Proving (AITP)*, 2022. URL: http://aitp-conference.org/2022/abstract/AITP_2022_paper_8.pdf.

16 Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3):158–243, 2021.

17 Is there a full documentation of coq's grammar? URL: https://coq.discourse.group/t/is-there-a-full-documentation-of-coqs-grammar/647/10.

18 coq_lsp. URL: https://github.com/ejgallego/coq-lsp.

19 Proposal: a custom build tool for coq projects. URL: https://coq.discourse.group/t/proposal-a-custom-build-tool-for-coq-projects/239/2.

20 pycoq. URL: https://github.com/IBM/pycoq.

21 Garett Cunningham, Razvan C. Bunescu, and David Juedes. Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs, 2023. doi:10.48550/ARXIV.2301.02195.

22 Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, Berlin, 3 edition, October 2014. URL: https://link.springer.com/book/10.1007/978-3-642-30958-8.

23 Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *CoRR*, abs/1505.07002, 2015. arXiv:1505.07002.

24 Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *arXiv preprint arXiv:2208.03133*, 2022.

25 Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In *Logic Programming and Automated Reasoning: 5th International Conference*, LPAR '94, pages 1–15, Berlin, Heidelberg, 1994. Springer. doi:10.1007/3-540-58216-9_25.

26 Emily First and Yuriy 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.

27 General roadmap. URL: https://github.com/ejgallego/coq-serapi/issues/252.

28 Query ast returns empty result. URL: https://github.com/ejgallego/coq-serapi/issues/117.

29 Serapi 'classic mode' final release notice. URL: https://github.com/ejgallego/coq-serapi/issues/252#issuecomment-1365510329.

30 Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1219, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3182526.

31 Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *CoRR*, abs/2102.06203, 2021. arXiv:2102.06203.

32 Albert Jiang, Wenda Li, Jesse Han, and Wu Yuhuai. Lisa: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving (AITP)*, 2021.

33 Portal-to-isabelle. URL: https://github.com/albertqjiang/Portal-to-ISAbelle.

34 René Just, Darioush Jalali, and Michael 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*, pages 437–440, San Jose, CA, USA, July 2014. Tool demo.

35 Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. Mash: Machine learning for sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 35–50, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

36 Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving. *arXiv preprint arXiv:2205.11491*, 2022.

37 Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving, 2022. doi:10.48550/ARXIV.2205.11491.

38 Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin 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*, ISSTA 2020, pages 101–114, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3395363.3397369.

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

40 Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018. doi:10.1145/3105906.

41 James Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957. Publisher: Society for Industrial and Applied Mathematics. URL: https://www.jstor.org/stable/2098689.

42 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, USA, 2002. Association for Computational Linguistics. doi:10.3115/1073083.1073135.

43 Ofir Pele and Michael Werman. Fast and robust earth mover's distances. In *2009 IEEE 12th International Conference on Computer Vision*, pages 460–467, 2009. doi:10.1109/ICCV.2009.5459199.

44 Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University Pittsburgh, 1987.

45 Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. arXiv:2009.03393.

46 Zichao Qi, Fan Long, Sara Achour, and Martin 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*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2771783.2771791.

47 Markus N. Rabe and Christian Szegedy. Towards the automatic mathematician. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 25–37, Cham, 2021. Springer International Publishing.

48 Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. arXiv:2009.10297.

49 Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, 2021.

50 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *CoRR*, abs/2003.06458, 2020. arXiv:2003.06458.

51 Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 112–127, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454033.

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

53 Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 115–129, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167094.

54 Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for proof reuse in coq. In *Interactive Theorem Proving*, 2019.

55 Valentin Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. PhD thesis, UC San Diego, 2018.

56 Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks, 2019. doi:10.48550/ARXIV.1907.07794.

57 scala-isabelle. URL: https://dominique-unruh.github.io/scala-isabelle/.

58 T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. doi:10.1016/0022-2836(81)90087-5.

59 Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1–11, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180233.

60 Makarius Wenzel. Isabelle/jedit — a prover IDE within the PIDE framework. *CoRR*, abs/1207.3441, 2012. arXiv:1207.3441.

61 mmverify.py. URL: https://github.com/david-a-wheeler/mmverify.py.

62 Iain Johnston Whiteside. *Refactoring proofs*. PhD thesis, University of Edinburgh, November 2013. URL: http://hdl.handle.net/1842/7970.

63 Karin Wibergh. Automatic refactoring for agda. Master's thesis, Chalmers University of Technology and University of Gothenburg, 2019.

64 Yuhuai Wu, Albert Q Jiang, Wenda Li, Markus N Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *arXiv preprint arXiv:2205.12615*, 2022.

65 Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*, Long Beach, CA, USA, 2019. URL: `http://proceedings.mlr.press/v97/yang19a/yang19a.pdf`.

66 Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR, 2021.

67 John R. Zech, Marcus A. Badgeley, Manway Liu, Anthony B. Costa, Joseph J. Titano, and Eric Karl Oermann. Variable generalization performance of a deep learning model to detect pneumonia in chest radiographs: A cross-sectional study. *PLOS Medicine*, 15(11):e1002683, November 2018. `doi:10.1371/journal.pmed.1002683`.

68 Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL: `https://openreview.net/forum?id=9ZPegFuFTFv`.