Towards Mining Robust Coq Proof Patterns

Cezary Kaliszyk
The University of Melbourne
Melbourne, Australia
cezary.kaliszyk@unimelb.edu.au

Bach Le
The University of Melbourne
Melbourne, Australia
bach.le@unimelb.edu.au

Christine Rizkallah
The University of Melbourne
Melbourne, Australia
christine.rizkallah@unimelb.edu.au

Abstract

To reduce the human effort involved in maintaining Coq formal proof scripts, we discuss the software engineering program repair approaches and our plan to adapt them and apply them to proof repair. This talk proposes a mining approach on a recently published Coq dataset, that aims to adapt established software maintenance methodologies to benefit the area of proof maintenance. We would appreciate feedback from the Coq community on our planned approach.

1 Introduction

Even with the advanced features of modern proof assistants formalizing mathematical proofs require significant human effort. Having invested this effort, it is essential not to be required to redo large parts of the proofs every time a proof system changes or the proof libraries are improved. The importance of proof maintenance has already been observed with the creation of the first formal proof libraries in the eighties [2] and formally studied in the context of LCF proof systems in the nineties [6].

Usually, the responsibility for maintaining particular proofs stays with the original authors. However, some proof system communities have introduced different approaches for this. When a user of Isabelle wants to make a change that would break several people's developments, before the change is accepted they need to fix all broken parts of the library. Nevertheless, the responsibility for particular Isabelle/AFP [3] entries ultimately belongs to their original authors and they are sometimes asked to adapt their developments to the new versions.

Recently, Reichel et al. [14] have proposed a machine learning dataset for Coq intended for proof repair. Based on this rich dataset, we aim to bring software engineering methodologies that are used in software maintenance to inform proof engineering choices and guidelines as well as guide automatic program transformations, such as proof repair. It would be valuable to receive input from the Coq community on our planned methodology.

2 Manual Approaches to Making Proofs Maintainable

Tactical style proofs, predominantly used in Coq formalizations, are convenient for proof development as they enable Coq proof engineers to construct proofs interactively by applying a sequence of tactics. Such proofs are often particularly hard to maintain. This is because a small mismatch in a single step might mean that the whole later part of the proof requires significant adaptation. Additionally with new goals opened and closed by tactics, when fixing the proof it is necessary to figure out which parts of the tactic script corresponded to which part of the proof.

For this reason, explicitly stating as many sublemmas as possible and using them in shorter proofs helps proof maintainability. This approach is taken to the extreme by declarative proofs, where all intermediate steps are stated explicitly, as done for example in Isabelle/Isar [19]. In fact, certain kinds of tactical Coq proofs can be automatically translated to declarative proofs [10] where cuts are explicitly stated. A further study of the maintainability of such automatically translated proofs is necessary. A task related to proof maintenance is proof translation between proof systems, and declarative proofs are actually easier to translate across provers than tactical proofs [9].

Tactical style proofs are compiled using Coq's tactic compiler into a low-level representation of proofs called *proof terms*. Proof terms can also be manually written in Coq; effectively constructing proofs as terms that match with propositions as the types of these terms. Proof terms are checked using Coq's kernel for correctness. As opposed to tactic-based proofs that can obscure the underlying proof structure, proof terms both reflect and give control over the full explicit structure of the proof. Unlike tactical-style proofs, which are both hard to maintain as they obfuscate structure and typically require active maintenance across versions, proof terms tend to be more robust. For instance, in our personal experience, we have a decade-and-a-half-old manual proof term style formalisation [17, Appendix A] that has worked across various Coq versions over the years without requiring *any* maintenance. Moreover, thanks to the explicit structure proof terms are also more amenable to proof transformations and proof repair [15, 16]. Proof terms will, therefore, serve as the basis for our proof analysis.

3 Software Engineering Approach to Program Repair

Software bugs are prevalent and fixing them requires significant effort and resources, which in turn can substantially reduce developers' productivity. It can take days or even years for software defects to be repaired [4]. Automated bug

The International CoqPL Workshop, 2025, Denver, Colorado, United States
2024.

fixing, or automated program repair (APR), is now an active and exciting research area, which engages both academia and the software industry since its practicality was first realized in 2009 [18]. Real-world defects from large programs have been shown to be efficiently and effectively repaired by APR, e.g., the Heartbleed vulnerability was correctly repaired by a recent APR approach within a matter of minutes [13]. Notably, in 2018, Facebook announced the first-ever industrial-scale APR technique, namely GetAFix [1], developed by Facebook's headquarters-based research team in Menlo Park and widely used in-house. GetAFix was directly inspired by a recent research work [12]. More recently in 2023, APR was experimentally deployed in Bloomberg [20].

With the recent advances in Large Language Models (LLMs), the once futuristic idea of APR has further become closer to reality. Multiple APR solutions have been proposed, from leveraging program analysis to pure LLM-based prompt engineering. Approaches based on program analysis, e.g., symbolic reasoning such as [11, 13], reason about program semantics to synthesize patches. Symbolic reasoning is used to infer program specifications and then program synthesis is used to synthesize repair consistent with the inferred specifications. Approaches based on LLMs, deep learning, or data mining, such as [5, 8, 12], use syntactical patterns to search for repairs. The general idea in these approaches is that bug fixes often resemble their counterparts in the past and thus learning historical bug fix patterns is helpful to repair future bugs. While both of these approaches have shown promising results, there is still much room for improvement. That is, they rely heavily on test suites to validate the correctness of patches, and thus often produce plausible patches, i.e., patches that overfit to the test suite but do not generalize. Having more comprehensive or complete specifications would help APR overcome this issue in practice.

More recently, LLMs are also adopted for proof repair [7] and have shown promising results. Different from APR, proof repair has complete specifications which helps in part avoid the patch overfiting issue. It would be interesting to see how APR techniques can be transferred to the domain of proof repair, leveraging the benefit of having complete specifications to effectively fix broken proofs.

4 Proposed Methodology

This project focuses on mining proof datasets to learn robust proof patterns and proof repair patterns. We hope that this can help with gaining further insights on how to write proofs that are easy to maintain as well as in guiding proof transformations; be it so proofs can be automatically rewritten to more maintainable variants or for automated proof repair. Inspired by the process used in software engineering for program repair, we plan to proceed as follows.

Data collection Collecting data about proof transformations and proof repair was manually done by developers in the past. A recent Coq dataset encompasses formalisms across various Coq versions [14]. This can be used as a basis for our mining work but needs to be analysed for robust versus breaking-proof patterns.

Repair templates mining automatically mining proof repair templates based on the data collected. Automatic mining of proof repair templates via the collected data. Relying on the data collection of proof repairs made by human proof engineers based on existing data sets, this phase converts the proof repairs into a graph form that is amenable to graph mining techniques to mine discriminative graph patterns. This allows us to automatically mine frequently appearing repair patterns. To do so, we plan to follow the following steps.

1. We convert proofs before and after repair into abstract syntax trees (ASTs) and then represent the transformations that convert one AST to another in terms of a graph. To do this, we use tree-differencing techniques to generate the AST transformations. The tree differencing techniques originally supported traditional programming languages such as C/C++/Java. Similar differencing approaches apply to proofs.
2. We then convert the collected transformations that represent proof repairs into graphs that are amenable to discriminative graph mining techniques. We then use graph mining to automatically mine discriminative graph patterns and use the mined patterns to guide the proof repair.

Proof repair automatically applies the mined templates to repair proofs. This devises automated approaches to generate repairs via a feedback loop from Coq.

1. Generate repair candidates via mutations using the mined templates.
2. Validate repair candidates using Coq for feedback on the correctness of the repairs; in particular, where a repair breaks and where it succeeds.
3. Continually improve the repairs through a feedback loop until Coq accepts the repaired proof.
4. Note that by doing so, we get complete correctness guarantees for the proof by using Coq in the loop.

Robust proof pattern mining A similar approach to mining proof repair templates can be used to mine the dataset and identify resilient proof patterns. These can be used to guide automatic semantic-preserving proof transformations into such patterns.

References

[1] Bader, J., Scott, A., Pradel, M., and Chandra, S. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–27.

[2] Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., and Pąk, K. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *J. Autom. Reason. 61*, 1-4 (2018), 9–32.

[3] Blanchette, J. C., Haslbeck, M. W., Matichuk, D., and Nipkow, T.

Mining the archive of formal proofs. In *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings* (2015), M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, Eds., vol. 9150 of *Lecture Notes in Computer Science*, Springer, pp. 3–17.

[4] Böhme, M., and Roychoudhury, A. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 international symposium on software testing and analysis* (2014), pp. 105–115.

[5] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering 47*, 9 (2019), 1943–1959.

[6] Curzon, P. The importance of proof maintenance and reengineering. In *Int. Workshop on Higher Order Logic Theorem Proving and Its Applications* (1995).

[7] First, E., Rabe, M. N., Ringer, T., and Brun, Y. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023), pp. 1229–1241.

[8] Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., and Svyatkovskiy, A. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023), pp. 1646–1656.

[9] Kaliszyk, C., and Pąk, K. Declarative proof translation (short paper). In *10th International Conference on Interactive Theorem Proving (ITP 2019)* (2019), J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141 of *LIPIcs*, pp. 35:1–35:7.

[10] Kaliszyk, C., and Wiedijk, F. Merging procedural and declarative proof. In *Proc. of the Types for Proofs and Programs International Conference (TYPES'08)* (2008), S. Berardi, F. Damiani, and U. de'Liguoro, Eds., vol. 5497 of *LNCS*, Springer Verlag, pp. 203–219.

[11] Le, X.-B. D., Chu, D.-H., Lo, D., Le Goues, C., and Visser, W. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), pp. 593–604.

[12] Le, X. B. D., Lo, D., and Le Goues, C. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)* (2016), vol. 1, IEEE, pp. 213–224.

[13] Mechtaev, S., Yi, J., and Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering* (2016), pp. 691–701.

[14] Reichel, T., Henderson, R. W., Touchet, A., Gardner, A., and Ringer, T. Proof repair infrastructure for supervised models: Building a large proof repair dataset. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland* (2023), A. Naumowicz and R. Thiemann, Eds., vol. 268 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 26:1–26:20.

[15] Ringer, T. *Proof Repair.* PhD thesis, University of Washington, USA, 2021.

[16] Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., and Tatlock, Z. QED at large: A survey of engineering of formally verified software. *Found. Trends Program. Lang. 5*, 2-3 (2019), 102–281.

[17] Rizkallah, C. Proof representations for higher-order logic. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, December 2009. Available at https://people.eng.unimelb.edu.au/rizkallahc/publications/masters.pdf.

[18] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 364–374.

[19] Wenzel, M. Isar - A generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings* (1999), Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, and L. Théry, Eds., vol. 1690 of *Lecture Notes in Computer Science*, Springer, pp. 167–184.

[20] Williams, D., Callan, J., Kirbas, S., Mechtaev, S., Petke, J., Prideaux-Ghee, T., and Sarro, F. User-centric deployment of automated program repair at bloomberg. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (2024), pp. 81–91.