PROOF ENGINEERING TOOLS FOR A NEW ERA

TALIA RINGER

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:
TODO, Chair
TODO
TODO

Program Authorized to Offer Degree:
Computer Science & Engineering

ABSTRACT

PROOF ENGINEERING TOOLS FOR A NEW ERA

Talia Ringer

Chairs of the Supervisory Committee:
TODO
Computer Science & Engineering

Abstract will go here.

To my parents. Hope this fits on your fridge.

CONTENTS

ACKNOWLEDGMENTS

I've always believed the acknowledgments section to be one of the most important parts of a paper. But there's never enough room to thank everyone I want to thank. Now that I have the chance—where do I begin?

We got other wonderful feedback on the paper from Cyril Cohen, Tej Chajed, Ben Delaware, Jacob Van Geffen, Janno, James Wilcox, Chandrakana Nandi, Martin Kellogg, Audrey Seo, James Decker, and Ben Kushigian. And we got wonderful feedback on e-graph integration for future work from Max Willsey, Chandrakana Nandi, Remy Wang, Zach Tatlock, Bas Spitters, Steven Lyubomirsky, Andrew Liu, Mike He, Ben Kushigian, Gus Smith, and Bill Zorn. The Coq developers have for years given us frequent and efficient feedback on plugin APIs for tool implementation.

Dan Grossman, Jeff Foster, Zach Tatlock, Derek Dreyer, Alexandra Silva, the Coq community (Emilio J. Gallego Arias, Enrico Tassi, Gaëtan Gilbert, Maxime Dénès, Matthieu Sozeau, Vincent Laporte, Théo Zimmermann, Jason Gross, Nicolas Tabareau, Cyril Cohen, Pierre-Marie Pédrot, Yves Bertot, Tej Chajed, Ben Delaware, Janno), coauthors, Valentin Robert, my family, PLSE lab (especially Chandrakana Nandi oh my gosh), James Wilcox, Jasper Hugunin, Marisa Kirisame, Jacob Van Geffen, Martin Kellogg, Audrey Seo, James Decker, Ben Kushigian, Gus Smith, Max Willsey, Zach Tatlock, Steven Lyubomirsky, Andrew Liu, Mike He, Ben Kushigian, Bill Zorn, Anders Mörtberg, Conor McBride, Carlo Angiuli, Bas Spitters, UCSD Programming Systems group, Misha, PL Twitter, Roy, Vikram, Esther, Ellie, Mer, students, Qi, Saba.

1

INTRODUCTION

Motivation for verifying systems

Era of scale—enter proof engineering [2]

Looking back (Social Processes [1]), development has come a long way, but maintenance is still hard! And this is a problem in practice!

But missed opportunity: automation doesn't understand that proofs evolve

So we build automation that does, and we call this proof repair. Proof repair shows that there is reason to believe that verifying a modified system should often, in practical use cases, be easier than verifying the original the first time around.

Or, in other words (thesis statement): Changes in programs, specifications, and proofs carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change. A tool that automates this can save work for proof engineers relative to reference manual repairs in practical use cases.

Key technical bit: differencing and program transformations, taking advantage of the rich and structured language proofs are written in.

We implement this in a tool suite for Coq, get some sweet results.

Pave path to the next era of verification

READING GUIDE

How to read this thesis

Mapping of papers to chapters

Authorship statements for included paper materials, to credit coauthors

Expected reader background & where to find more info

2

MOTIVATING PROOF REPAIR

Before we talk more about proof repair, it helps to know what it's like to develop and maintain proofs to begin with, and what happens under the hood when you do that. This chapter gives you that context, then explains the high-level approach to proof repair that builds on that.

2.1 PROOF DEVELOPMENT

Cartoon version of development: program, spec, proof

Proof assistants: short overview of foundations & different options (survey paper), then say focus on Coq

Slightly less brief overview of Coq and its foundations and automation and so on (including proof terms), going through a running example of proof development in Coq

2.2 PROOF MAINTENANCE

Problem is when something changes—change something in running example

There are a lot of development processes people use to make proofs less likely to break to begin with (survey paper)

But still, even with these, the reality: This happens all the time (REPLICA)

And in fact not just after developing a proof, but during development too (REPLICA)

And breaks proofs even for experts (REPLICA)

And it's an extra big problem when you have a large development and the changes are outside of your control

Hence Social Processes

Why automation breaks, even with good development processes

Hence proof repair—smarter automation

2.3 PROOF REPAIR

Name inspired by program repair, but quite different as we'll soon see.

Recall thesis: Changes in programs, specifications, and proofs carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change. A tool that automates this can save work for proof engineers relative to reference manual repairs in practical use cases.

Proof repair accomplishes this using a combination of differencing and program transformations.

Differencing extracts the information from the change in program, specification, or proof.

The transformations then generalize that information to a more general fix for other proofs broken by the same change.

The details of applying the fix vary by the kind of fix, as we'll soon see.

Crucially, all of this happens over the proof terms in this rich language we saw in the Development section. This is kind of the key insight that makes it all work.

This is great because this language gives us so much information and certainty. This helps us with two of the biggest challenges from program repair. (generals related work)

But it's also challenging because this language is so unforgiving. Plus, in the end, we need these tactic proofs, not just proof terms. So we can't just reuse program repair tools. (generals related work)

So next two chapters will show two tools in our tool suite that work this way, how they handle these challenges, and how they save work.

3

PROOF REPAIR BY EXAMPLE

The first tool (PUMPKIN PATCH) focuses on changes in programs and specifications, though these changes are limited in scope as we'll see later.

What this tool does is, when programs and specifications change and this breaks a lot of proofs, it lets the proof engineer fix just one of those proofs. It then generalizes the example patch into something that can fix other proofs broken by the same change.

So in other words, the information from those changes is carried in the difference between the old and new version of the example patched proof. PUMPKIN PATCH generalizes that information.

Application can be automated in some cases at the end, or it can be manual.

The work saved is shown retroactively on case studies replaying changes from large proof devleopments in Git. Results for this tool are preliminary compared to what we'll see later, since this was the first prototype.

3.1 MOTIVATING EXAMPLE

PUMPKIN PATCH intro & automation, reimagined

3.2 APPROACH

Parts of PUMPKIN PATCH Motivating the Core, plus more

Like I mentioned earlier, this works using differencing and program transformations. And of course all of this happens over proof terms. Here's the system diagram.

Here, differencing thus looks at the difference between versions of the example patched proof for this information, and finds something called a patch candidate—which is localized to the context of the example, but not enough to fix other proofs broken by the change.

Then, program transformations generalize that candidate to a reusable proof patch, something that can fix other proofs broken by the same change. Application works with hint databases or is manual.

3.3 DIFFERENCING

parts of PUMPKIN PATCH Inside the Core, Testing Boundaries, Future Work

How differencing works in detail

Limitations and whether they're addressed in later tools yet or not

3.4 TRANSFORMATION

parts of PUMPKIN PATCH Inside the Core, Testing Boundaries, Future Work

How the four transformations work in detail

Limitations and whether they're addressed in later tools yet or not

3.5 IMPLEMENTATION

parts of PUMPKIN PATCH Inside the Core, plus more

3.5.1 *Tool Details*

3.5.2 *Workflow Integration*

3.6 SAVING WORK

PUMPKIN PATCH Case Studies

3.7 CONCLUSION

Rehashing thesis and how we do it

What we haven't accomplished yet at this point (parts of PUMPKIN PATCH future work), segue into next chapter

4

PROOF REPAIR ACROSS TYPE EQUIVALENCES

5

RELATED WORK

6

CONCLUSION

6.1 THE NEXT ERA: PROOF ENGINEERING FOR ALL

BIBLIOGRAPHY

[1] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 206–214, New York, NY, USA, 1977. ACM.

[2] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.