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

1 INTRODUCTION 3
2 MOTIVATING PROOF REPAIR 5
  2.1 Proof Development 5
  2.2 Proof Maintenance 5
  2.3 Proof Repair 6
3 PROOF REPAIR BY EXAMPLE 7
  3.1 Motivating Example 7
  3.2 Approach 7
  3.3 Differencing 8
  3.4 Transformation 8
  3.5 Implementation 8
  3.6 Results 8
  3.7 Conclusion 8
4 PROOF REPAIR ACROSS TYPE EQUIVALENCES 9
  4.1 Motivating Example 9
  4.2 Approach 9
  4.3 Differencing 10
  4.4 Transformation 10
  4.5 Implementation 10
  4.6 Results 10
  4.7 Conclusion 10
5 RELATED WORK 11
  5.1 Programs 11
  5.2 Proofs 11
6 CONCLUSIONS & FUTURE WORK 13

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 RESULTS

PUMPKIN PATCH Case Studies, key technical results

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

This extension to the suite adds support for a broad class of changes in datatypes, handling a large class of practical repair scenarios. What this tool (PUMPKIN Pi) does is, when datatypes change and this breaks a lot of proofs, it generalizes the change in datatype itself (possibly with some user input) so that it can automatically fix proofs broken by the change in datatype.

So in other words, the information from those changes is carried in the difference between the old and new version of the changed datatype, possibly with some user input.

PUMPKIN Pi generalizes that information and applies it automatically.

The work saved is shown on a lot of case studies (see Table from PUMPKIN Pi).

4.1 MOTIVATING EXAMPLE

PUMPKIN Pi motivating example

4.2 APPROACH

Parts of PUMPKIN Pi intro, problem definition, plus more

Like I mentioned earlier, this also 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 changed datatype, and finds something called a type equivalence. I'll explain that with examples. Sometimes differencing is automatic, and sometimes it's manual.

Then, program transformation ports proofs across the equivalence directly. So they take care of application.

4.3 DIFFERENCING

DEVOID 3.1 and 4.1, with some more general things from PUMPKIN Pi and more.

How differencing works in detail

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

4.4 TRANSFORMATION

Parts of PUMPKIN Pi Transformation, with DEVOID 3.2 and 4.2 as examples, plus some of the beautiful Carlo theory to explain why we go from equivalences to configurations and what that really means

How the transformation works in detail

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

4.5 IMPLEMENTATION

Parts of PUMPKIN Pi and DEVOID implementation, plus more

4.5.1 *Tool Details*

4.5.2 *Workflow Integration*

PUMPKIN Pi Decompiler and Implementation

4.6 RESULTS

PUMPKIN Pi Case Studies, key technical results

4.7 CONCLUSION

Rehashing thesis and how we do it

What we got here beyond what we had in PUMPKIN PATCH, segue into next chapter

5

RELATED WORK

5.1 PROGRAMS

Program Refactoring

Program Repair

Ornaments

Programming by Example

Differencing & Incremental Computation

5.2 PROOFS

Proof Reuse

Proof Refactoring

Proof Repair

Proof Design

Proof Automation

Transport

Parametricity

Refinement

6

CONCLUSIONS & FUTURE WORK

Reflect on thesis statement and explain how we got it exactly now that you know everything

But I want to spend the resst of this thesis talking about the next era of verification so I can write out a bunch of ideas for students who might want to work with me

THE NEXT ERA: PROOF ENGINEERING FOR ALL

Future Work from many papers, plus research statement, DARPA thoughts, plus more, but trimmed down a lot

What I want in the long run, how this all fits in, is a world of proof engineering for all. From research statement, three rings (four including experts in the center).

And what we have so far with my thesis is a world where it's easier for experts and a bit easier for practitioners, but there's still a lot left to go building on it.

So here are 12 short future project summaries that reach each of these tiers, building that world. Super please contact me if any of these seem fun to you.

Proof Engineering for Experts

Unifying theme: lateral reach. Some examples:

MORE PROOF ASSISTANTS Thoughts from PUMPKIN Pi on Isabelle/HOL, future work from PUMPKIN PATCH.

MORE CHANGES Version updates, isolating large changes (PUMPKIN PATCH), relations more general than equivalences (PUMPKIN Pi).

MORE STYLES ML for decompiler (PUMPKIN Pi, REPLICA) : more for diverse proof styles (PUMPKIN PATCH). Note that this is a WIP, but sketch out project, challenges, future ideas, expectations, evaluation a bit.

Proof Engineering for Practitioners

Unifying theme: usability. Some examples:

AUTOMATION More search procedures for automatic configuration, e-graphs from PUMPKIN Pi, custom unification heuristics.

INTEGRATION IDE & CI integration, HCI for repair.

EVALUATION repair challenge, user studies ideas (PUMPKIN PATCH, REPLICA, panel w/ Benjamin Pierce, QED at large). (maybe look for more ideas, this can be merged with integration if need be).

Proof Engineering for Software Engineers

Unifying theme: mixed methods verification, or the 2030 vision from Twitter thread. Some examples:

GRADUAL VERIFICATION A continuum from testing to verification, tools to help with that.

TOOL-ASSISTED PROOF DEVELOPMENT Tool-assisted development to follow good design principles for verificattion (James Wilcox conversation, final REPLICA takeaway).

SPECIFICATION INFERENCE Analysis to infer specs (TA1).

Proof Engineering for New Domains

Unifying theme: collaboration, new abstractions for new domains). Some examples:

MACHINE LEARNING Fairification & other ML correctness properties. Some stuff here but more.

CRYPTOGRAPHY Lots of stuff here but not thinking broadly enough. What about cryptographic proof systems? ZK and beyond. Recall email thread.

SOMETHING ELSE Look for more in survey paper, email, DARPA TAs, Twitter. Healthcare perhaps?

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.