Research Statement

TALIA RINGER

My long-term goal as a researcher is to make it easy for programmers of all skill levels across all domains to build formally verified software systems. With verification, programmers can guarantee the absence of costly or dangerous bugs in critical systems. This guarantee provides practical benefits: practitioners have found verified systems to be more robust and secure in deployment. My research focuses on *proof engineering*—technologies that make it easier to develop and maintain these verified systems—with an emphasis on maintenance.

My work makes it easier to maintain systems that have been verified using tools called *proof assistants*. To develop a verified system using a proof assistant, the proof engineer does three things:

(1) implements a program using a functional programming language,
(2) specifies what it means for the program to be correct, and
(3) proves that the program satisfies the specification.

The process of writing this proof is interactive: The proof engineer sends strategies (called *tactics*) to the proof assistant to solve an outstanding proof obligation, and the proof assistant responds with a new proof obligation. This continues until the proof assistant finds a proof that it can check. If the proof is correct, the program behaves as specified—it is *verified*.

My passion is making it easier to maintain verified systems. I am a **programming languages** and **verification** researcher, though my work draws parallels to software engineering. Much like software engineering scales programming to large systems, proof engineering scales verification to large systems. In recent years, proof engineers have verified operating system (OS) kernels, machine learning systems, distributed systems, constraint solvers, web browser kernels, compilers, and file systems. Nonetheless, while proof engineering—like software engineering—is about both development and maintenance, most work so far has focused on development. When it comes to *maintaining* these systems, proof engineering is decades behind software engineering. I have detailed this gap in a monograph that surveys the field of proof engineering [**Ringer** et al. 2019a].

My research introduces *proof repair*: a new approach to maintaining verified systems. It develops foundational results in dependent type theory (DTT) and leverages those results to build flexible and useful proof repair tools. These tools have already been used to automate difficult changes, and to support an industrial proof engineer writing proofs about an implementation of the TLS handshake protocol. This marks the first step toward my long-term research vision of verification for all—from beginners to experts across all domains of computer science.

Proof Repair

Proof repair automatically fixes broken proofs in response to changes in programs and specifications. For example, a proof engineer who optimizes an algorithm may change the program, but not the specification; a proof engineer who adapts an OS to new hardware may change both. Even a small change to a program or specification can break many proofs, especially in large systems. Changing a verified library, for example, can break proofs about programs that depend on that library.

Proof repair views broken proofs as bugs that a tool can patch, much like program repair does for unverified programs. Proof assistants are a good fit for program repair: A recent review of a widely used but often incorrect program repair approach recommends that program repair tools draw on extra information like specifications or example patches. Examples and specifications are rich and widely available in proof assistants; proof repair draws on both.

I have implemented a suite of proof repair tools for the Coq proof assistant called PUMPKIN PATCH. With PUMPKIN PATCH, when a program or specification changes and this breaks many proofs, the proof engineer need only fix one—or, in some cases, none—of the broken proofs. PUMPKIN PATCH has performed well on proof engineering benchmarks. It has also been used to help a professional proof engineer at Galois, Inc. integrate Coq with other tools and write proofs about an implementation of TLS. I continue to partner with Galois to support new use cases.

Automation for Change. My early work introduces proof repair and implements a Pumpkin Patch prototype [**Ringer** et al. 2018]. Proof repair reimagines *proof automation*: the interactive process of passing tactics to the proof assistant in order to search for a proof. When a program or specification changes and this breaks a proof, traditional proof automation searches for the fixed proof from scratch. Proof repair, in contrast, determines how the program or specification has changed, and uses that information to help fix the broken proof.

My key insight is that even a single example fix in the context of a particular proof can sometimes carry enough information to fix proofs in other contexts. That way, when a program or specification changes and this breaks proofs that depend on it, the proof engineer can sometimes fix just one of those broken proofs. A tool can then generalize that fix into a *reusable patch*, something that the proof engineer can use with traditional proof automation to fix other broken proofs as well. In some cases, if the fix comes from a library, a tool can generalize a fix to a proof that the library developer already made, so that the library-client developer need not fix any proofs at all.

On a technical level, this insight builds on the rich dependently typed language at the core of Coq. Each proof Coq produces is a *proof term* whose type is its specification. When the proof engineer fixes the example broken proof, the proof term changes. My novel proof search techniques find the difference between the original proof term and the fixed proof term, then transform that difference—itself a proof term—to lift it out of the context of the example proof.

These techniques are implemented in the Pumpkin Patch prototype. This prototype is demonstrated on three case studies: updating proofs in response to a change in the CompCert C compiler, porting proofs between two versions of a datatype from different solutions to an exercise from the Software Foundations textbook, and updating the Coq standard library.

The Changes that Matter. Pumpkin Patch aims to meet the needs of proof engineers. To help determine these needs, my collaborators and I launched a user study of eight Coq proof engineers [**Ringer** et al. 2020b]. We monitored every change to programs, specifications, and proofs that these proof engineers made over the course of a month. The changes that we found revealed patterns amenable to automation. We captured those patterns in the form of benchmarks for measuring the improvement of proof engineering tools.

To collect data granular enough to observe incremental changes, we instrumented Coq. Our data, benchmarks, and infrastructure are publicly available, and our instrumentation is part of the official Coq 8.10 release. Our findings have driven and continue to drive improvements to Pumpkin Patch and to a machine learning tool for proofs. There are plans for other researchers to use them to drive improvements to a property-based testing tool for proofs as well.

Automation for the Changes that Matter. My latest work takes Pumpkin Patch from a prototype to a flexible and powerful tool that meets the needs of proof engineers. The prototype was limited in scope and usability by its approach: it generalized an example proof fix into a reusable patch, then relied on the proof engineer to apply that patch. The key insight to building a flexible, powerful, and useful tool was to instead *reuse* the old proof to directly derive a new proof.

To bridge the gap between reuse and repair, I developed a proof term transformation that implements a powerful kind of reuse—technically known as *transport across equivalences*. My transformation is the only technique I am aware of that implements transport in a way that is suitable for repair and that does not rely on any axioms beyond those Coq assumes.

My research on this transformation started with an extension to Pumpkin Patch called Devoid [**Ringer** et al. 2019b]. Devoid automates a class of changes not handled by the original Pumpkin Patch that can be prohibitively difficult for proof engineers: extending a type (like a list) to track a function (like its length) at the type level (like a length-indexed vector). These extended types make it possible to rule out bad inputs (say, empty lists for a function that returns the first element) at compile-time. Recent work in DTT studies this class of changes—called *algebraic ornaments*—from types like lists to types like vectors. I designed and implemented a transformation that reuses proofs about types like lists to derive proofs about types like vectors—and so automatically repairs proofs broken by changes that correspond to algebraic ornaments.

I later generalized this transformation to repair proofs broken by any kind of change that can be described by an equivalence—not just algebraic ornaments, but also swapping and renaming constructors, changing inductive structure, and more [**Ringer** et al. 2020a].[1] To further improve usability, I mentored a student building a decompiler that translates the transformed proof terms into sequences of the tactics that proof engineers use for traditional proof automation. That way, proof engineers can continue to maintain automatically repaired proofs.

We extended Pumpkin Patch with both this transformation and the decompiler. Pumpkin Patch has since been used to support a benchmark from our user study, ease development with dependent types, port functions and proofs between unary and binary numbers, and support the proof engineer at Galois to write proofs and to interoperate between Coq and other verification tools more easily.

Other Work

The remainder of my work has focused on methods for developing and maintaining correct programs using lighter-weight verification tools.

Program Analysis for Access Control. One alternative to verification using a proof assistant is *program analysis*: automatically analyzing a program to determine some property of the program. My work on AUDACIOUS [**Ringer** et al. 2016] combines program analysis with a secure library to enforce a fine-grained permission model for mobile applications called *user-driven access control* (UDAC). AUDACIOUS is implemented for Android and available on Github. It is the first approach to UDAC that does not modify the OS.

Constraint Solvers for Test Input Generation. Another approach to checking program behavior is to write unit tests. During a research internship at Amazon in 2016, I developed Iorek [**Ringer** et al. 2017], a language and framework that helps software engineers generate test inputs. Iorek uses a constraint solver to generate test inputs that are sufficiently different for the sake of testing. Iorek increased test coverage in combination with a random testing tool, and helped identify bugs in real services being developed at Amazon.

Future Directions: Proof Engineering for All

My long-term research vision is a future where verification is accessible to all programmers, not just experts. I will address three increasingly broad audiences to bring this future to fruition: proof engineering practitioners, software engineers, and potential users in other domains.

Proof Engineering for Practitioners. Reaching practitioners means making both writing and maintaining proofs about programs much easier than the current state of the art. Automatic proof reuse and repair tools have come a long way toward this end, but they still target experienced proof engineers and require some manual effort. I plan to improve automation in order to make these tools easier for non-experts to use. To achieve this, I will integrate proof reuse and repair tools with *e-graphs*, a data structure that is used in constraint solvers and rewrite systems. Extending and leveraging these data structures will make it possible to elegantly support reuse and repair over large libraries when many changes occur at once, while imposing little additional effort on the proof engineer. In addition, it will make it simple for any proof engineer to extend these tools with optimizations, all while preserving correctness. The result will be proof reuse and repair tools that can help less experienced proof engineers write and maintain proofs.

Proof Engineering for Software Engineers. Even with good proof engineering technologies, it is not always economically feasible or even desirable to formally verify an entire system using a proof assistant. The strong guarantees of a proof assistant may be necessary only for the most critical parts of the system. Other techniques like testing, automated theorem proving, or program analysis may better serve parts of the system. This makes a strong case for *mixed methods verification*: verification using multiple techniques while guaranteeing that their composition preserves correctness. I advocated for this in **Ringer** et al. [2019a], and implemented one case of this at Galois: using Pumpkin Patch to help a proof engineer interoperate between a constraint solver and Coq. I

[1]The cited arXiv paper is an outdated version with weaker results. The current version is under double-blind submission.

plan to expand on this. The tools I build will, for example, interactively generalize test cases to specifications for use in a proof assistant. They will derive specifications and proofs from static analyses, prompting the proof engineer for additional input as needed. They will compile between solver-aided languages and proof assistants using a verified compiler, allowing for seamless and proven integration of these tools. In all cases, they will compose with Pumpkin Patch to refine specifications and fix proofs to make them more efficient or easier to reason about.

Proof Engineering for Domain Experts. Proof engineering technologies need new abstractions to better reach certain domains. For example, verified machine learning and image processing tools should encode and prove guarantees not just about the algorithms (like fairness), but also about the data (like bias), or how they interact (like robustness). I plan to work with researchers outside of programming languages to help them prove the properties about their systems that matter to them, and in doing so, discover new abstractions and build new technologies that help other researchers and programmers in those domains. These technologies will leverage my planned work on mixed methods verification to use different verification techniques as appropriate.

Proof Engineering for All. Formal verification has the potential to help programmers of all skill levels across many important domains build more secure and robust systems. I will continue to drive the improvement of proof engineering technologies to realize this potential.

REFERENCES

Talia Ringer, Dan Grossman, and Franziska Roesner. 2016. AUDACIOUS: User-Driven Access Control with Unmodified Operating Systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 204–216. https://doi.org/10.1145/2976749.2978344

Talia Ringer, Dan Grossman, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2017. A Solver-Aided Language for Test Input Generation. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 91 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133915

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019a. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281. https://doi.org/10.1561/2500000045

Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2020a. Proof Repair Across Type Equivalences. (2020). https://arxiv.org/abs/2010.00774 Under Submission.

Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020b. REPLica: REPL Instrumentation for Coq Analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. 99–113. https://doi.org/10.1145/3372885.3373823

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting Proof Automation to Adapt Proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. 115–129. https://doi.org/10.1145/3167094

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019b. Ornaments for Proof Reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 141. 26:1–26:19. https://doi.org/10.4230/LIPIcs.ITP.2019.26