PROOF REPAIR

TALIA RINGER

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

Doctor of Philosophy

University of Washington

2021

Reading Committee:
Dan Grossman, Chair
Zachary Tatlock
Rastislav Bodik

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

ABSTRACT

PROOF REPAIR

Talia Ringer

Chair of the Supervisory Committee:
Dan Grossman
Computer Science & Engineering

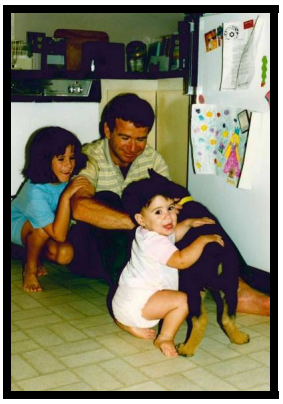
The days of verifying only toy programs are long gone. The last two decades have marked a new era of verification at scale, bringing strong guarantees to large and critical systems—an era of *proof engineering*. Proof engineering is for verified systems what software engineering is for unverified systems. Still, while proof engineering—like software engineering—is about both development and maintenance, most proof engineering technologies so far have focused on development. When it comes to *maintaining* these systems, proof engineering is decades behind software engineering.

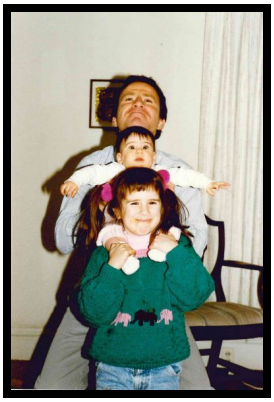
This thesis introduces *proof repair*: a new approach to maintaining verified systems. Proof repair reimagines the automation proof engineers typically use to interactively guide tools to search for a machine-checked proof. When a system changes and this breaks a proof about the system, traditional automation searches for the fixed proof from scratch. Proof repair, in contrast, is change-aware automation: it determines how the system has changed, and uses that information to help fix the broken proof.

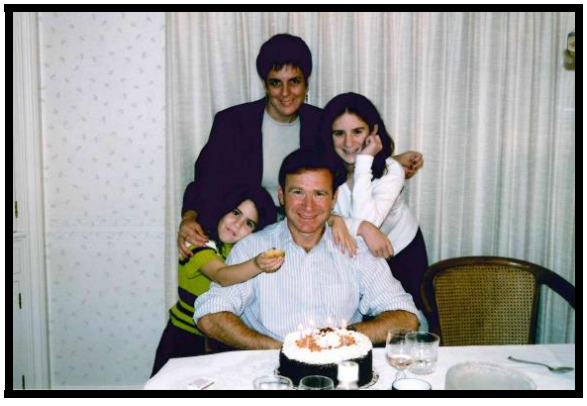
Proof repair in this thesis works by combining semantic differencing algorithms with program transformations. Importantly, both differencing and the transformations operate over low-level representations of proofs called *proof terms*. Thanks to the richness of these proof terms, differencing and the transformations can leverage new and existing results in dependent type theory. For example, one transformation externalizes univalent transport from homotopy type theory, leveraging novel transformations over equalities to make this possible.

This approach is realized inside of a proof repair tool suite for the Coq proof assistant. Case studies show both retroactively and by live use that this proof repair tool suite can save work for proof engineers on real proof developments.

To my family.







I love all of you.

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, though, I'm having trouble figuring out where I should begin.

LYMOR In the past, when I had trouble figuring out where to begin, I just followed my sister, *Lymor*. I followed her to Maryland for college, and when I got there, she told me to do research. I had no idea what "doing research" meant, but she told me that if I wanted to go to graduate school, I needed to do it. Honestly, I had no idea what graduate school was, either, but I did know it was something she was doing, and it sounded cool. So I did research.

This is the kind of thing that I think people often don't get to write about in acknowledgments sections. I don't think I could have had the opportunity to write this if not for her. How else was I supposed to know what research was, or that I should do it? That it would help me get into graduate school? Draw me to the world of programs and proofs? Instill in me the dream to become a professor? I start at Illinois in October, and I owe a lot of that to her.

So thanks, **Lymor**, for being such an amazing big sister ♡♡♡.

UNDERGRADUATE MENTORS & ADVISORS Once I got to Maryland, I did research with two wonderful math professors: *Kasso Okoudjou* and *Larry Washington*. These experiences were fun forays into the worlds of linear algebra and cryptology. They also helped me build the skills I needed to succeed in graduate school.

It took a couple of years at Maryland before I found my way from math into computer science (CS). I'm honestly thankful that I was afraid enough of physics and statistics to instead take CS to satisfy a breadth requirement for my math degree! But I'm also thankful that the undergraduate math advisor *Ida Chan* talked me into taking the more advanced sequence, so that it wasn't a dead end, and I could major in it later on. And I'm thankful that, shortly after, I had a chance to attend the second ever *Google* Computer Science Summer Institute (CSSI), where for the first time I felt empowered—so much that when I returned to Maryland, I decided to try to minor in CS. And I'm thankful that when I tried to minor in CS, *Brandi K. Adams* talked me into picking it up as a second major instead.

So thanks, **Kasso** and **Larry**, for the wonderful research opportunities. And thanks, **Ida**, **Brandi**, and everyone at **Google** CSSI, for giving me the confidence to major in CS.

JEFF & KRIS A pivotal semester for me at Maryland came during my senior year, when I took *Jeff Foster*'s advanced undergraduate programming languages (PL) class (with the awesome TA *Kris Micinski*). That whole class was amazing and got me super into PL. Like Jeff spent one of the lectures talking about the Curry-Howard Correspondence, which relates programs to proofs. I thought it was the coolest thing ever, even though I didn't really understand it that well at the time. It must have planted a seed or something, because I kept coming back to it again and again years later. And this thesis really gets to the soul of Curry-Howard, treats proofs just like the programs they really are, transforms them and evolves them over time. I'm obsessed.

Jeff is more than a great teacher, though—he's also a really great mentor. Like, the once-in-a-lifetime kind of mentor you're lucky to meet, who gives you selfless advice and helps you get to know *yourself* better, who is patient and kind and teaches you all of the things you'd have never even known to ask. It was thanks to Jeff that, one day in 2013, I found myself at a PL conference in Seattle talking to my eventual advisor. And it was thanks to Jeff that I knew anything at all about how to apply to and choose a graduate school. I spoke to Jeff every so often all throughout graduate school up to the very end—even talking to him about the faculty job search.

So thanks, **Jeff** (and **Kris**), for your fantastic teaching, advocacy, connections, and selfless advice over the last decade. Your students are lucky to learn from and work with you.

INDUSTRY MENTORS & MANAGERS During my time at Maryland, I worked as a software engineering intern at two companies: *Carr Astronautics* and *Amazon*. I continued to work as a software engineer at Amazon for three years after I graduated. My adventures as a software engineer at these two companies helped me learn to motivate useful research problems, build good tools, and collaborate with other people—skills that were essential to this thesis work.

At Amazon, my mentor *Musachy Barroso* and my manager *Ernesto Gonzalez* made my experiences so much fun that it was hard to leave. So it's maybe not surprising that I found myself back there for a research internship with *Serdar Tasiran* and *Daniel Schwartz-Narbonne* later on—and that was super valuable, too.

So thanks, all of my mentors at **Carr Astronautics** and **Amazon**, for the wonderful adventures in industry.

DAN After three years at Amazon, I left to work with my advisor *Dan Grossman* at the University of Washington. Dan is the best advisor. I'm really happy that I had a chance to work with him. It was a good work dynamic for me, since he always gave me the autonomy that I needed to explore the problems that I love. He did help me

push myself a few times—just only when I really, really needed to be pushed, and never any more than I needed.

Dan is also maybe the wisest and most patient person I've ever met, and I mean that. All through graduate school, he always gave me great advice—advice that I on many occasions rejected at first because I didn't want to hear it. When I rejected his advice, he just kind of patiently talked me through it. He recognized that I'm my own person, and so accepted that sometimes I just wouldn't listen in the end. But I often did listen in the end. And when I did, I'd often find myself looking back months later, thinking "oh yeah, he was right."

This thesis is a good example of that. Dan told me many times that I should actually put a lot of effort into this thesis, because people actually do read these, and they can be really useful. I fought that advice because I'd always thought of the thesis as some weird academic hazing ritual. But this past week I've spoken to potential students, and invariably I've ended up sending them a copy of this thesis. It's a really useful source of my work that has historical context and background information. So yeah, he was right. I apologize in advance if his future students have to read this, now.

Above all, though, it's important to note that Dan thought of the acronyms for three of my tools:

1. AUDACIOUS: Android User-Driven Access Control In Only User Space,
2. PUMPKIN PATCH: Proof Updater Mechanically Passing Knowledge Into New Proofs, Assisting The Coq Hacker, and
3. DEVOID: Dependent Equivalences Via Ornamenting Inductive Definitions.

This gave me an endless stream of jokes to tap into for my research talks all throughout graduate school.

So thanks, **Dan**, for the patience, wisdom, and dad jokes. I feel ready to be a professor and a dad.[1]

FRANZI AUDACIOUS is work that **Dan** and I did with *Franzi Roesner* during my first year. Franzi taught me a lot about how to write the niche parts of research papers. It was fun.

Thanks, **Franzi**, for your help becoming a better researcher!

UW PLSE The most amazing thing about my graduate school experience was being a part of the *UW PLSE* lab. I didn't realize what an absolute blessing it was to be a part of PLSE until one day the pandemic hit and the lab suddenly closed. If I could see everyone in PLSE right now, I would hug all of them. What an amazing group of

1 I'm not actually going to be a dad.

people, always there to support each other, to give feedback, to inspire new ideas, to chat, to be just amazing friends—I love PLSE.

One of the first people that I met in PLSE was *Chandrakana Nandi*, and she was there for me throughout my entire graduate school journey. It's not just that she gave me feedback on basically all of my papers and talks (she did). But also, she did things like—she sent me donuts when I was working on my PLDI paper recently. She helped me through the hardest year of my life. She even visited me in the hospital, once. She's just great. The most genuine and kind friend I could have had by my side for this journey. Really.

You know who else from PLSE is amazing? *Zach Tatlock*. Zach worked with me on the survey paper, taught me Coq, and inspired the original problem that got me interested in this thesis work. That's all fine and dandy, but he had to take that a step further and literally spend an hour of his week every week for an entire year helping me through a really hard time. Just because he's a kind and caring person, not because he got anything out of it. It's not just that I wouldn't have made it through graduate school without Zach's help; I honestly don't think I would *be here today* without Zach's help.

The same goes for *Alex Polozov*, who overlapped with me in PLSE for just one year, but ended up being one of the best friends I could have ever asked for. Alex gave me really great advice when life was really hard. Some of that advice saved my life. That's cool; you don't find friends like that just anywhere.

Sarah Chasins also gave me incredible advice all throughout graduate school; she is honestly the best listener I have ever met. My first year mentor *John Toman* humored my probably very weird early graduate school questions. My cohort—**Chandra**, *Chenglong Wang*, *Amanda Swearngin*, *Jared Roesch*, *Sam Elliott*, and *Bill Zorn*—was so much fun to share this journey with. My seniors—especially **Sarah**, **Alex Polozov**, *Doug Woos*, *James Wilcox*, *Stuart Pernsteiner*, *Konstantin Weitz*, and *Joe Redmond*—were such wonderful role models and friends to me throughout this journey. My juniors—especially *Max Willsey*, *Martin Kellog*, *Alex Sanchez-Stern*, *Gus Smith*, *Ben Kushigian*, *Steven Lyubomirsky*, *Jacob Van Geffen*, *Marisa Kirisame*, *Remy Wang*, *Melissa Hovik*, *Rashmi Mudduluru*, *Sorawee Porncharoenwase*, and *Krzysztof Drewniak*—helped me so much throughout, too. *Ras Bodik* joined **Dan** and **Zach** on my reading committee for this thesis. PLSE alumnus *Adrian Sampson* sent me the source files for this beautiful thesis format.

Above all, though, my students and research mentees—*Jasper Hugunin*, *Taylor Blau*, *RanDair Porter*, and *Nate Yazdani*—brought so much light and joy to my graduate school experience. They are the reason I'm so excited to become a professor.

So thank you so much to every person I've ever overlapped with in **UW PLSE**. I miss all of you, and I wish all of you nothing but joy and success for the rest of your lives. Please come visit me in Illinois!

FRIENDS Outside of **PLSE**, I'm really lucky to have met some amazing graduate school *friends* I can't imagine my life without. Not a day will pass when I won't miss *Roy Or-El*, *Vikram Iyer*, and *Esther Jang*—they were always there by my side when I needed them most, and were always so wonderful and supportive and caring and understanding. And I'm thankful to have met *Anna Kornfeld Simpson*, *Anne Spencer Ross*, *Karl Koscher*, *Dhruv Jain*, and *Jasper Tran O'Leary*.

I spent the better part of a year during graduate school in San Diego, with the *UCSD ProgSys* lab. Let me tell you, that lab gives **PLSE** a run for its money. Everyone in ProgSys made that year super fun and valuable, as did *Marcela Mendoza*, *Misha Kolmogorov*, and *Grace Uchida*.

My Dance Dance Revolution friends (yes, that's a thing) helped me forget about graduate school when I needed that; I'm especially lucky to know *Laura Chick*, *Ezgi Akgül*, *Chris Maines*, and *Melanie Walker*. My *Club Northwest* teammates—and especially my coach, *Tom Cotner*—gave me a healthy escape in a wonderful environment, as did my *Race Condition Running* (RCR) running buddies. My cousin *David Lasky*, my friend *Wade Gordon*, and everyone I met at *Chabad of Queen Anne* gave me a local support network in Seattle. My childhood best friend *Danielle Antosh* and my high school best friend *Erica Iantuono* always supported me from across the country, no matter how much time passed. **Esther**, *Ellie Berry*, *Mer Joyce*, and above all *Qi Cheng* brought so much light to a dark pandemic year.

So thank you so much to all of my **friends**, **Club Northwest** teammates, **RCR** running buddies, and everyone else who has been there for me these past six years. Nothing but love to all of you.

COMMUNITY The *Coq community* has given me so, so much feedback through every step of this process, especially *Matthieu Sozeau*, *Yves Bertot*, *Nicolas Tabareau*, *Jason Gross*, *Cyril Cohen*, *Tej Chajed*, *Emilio J. Gallego Arias*, *Enrico Tassi*, *Gaëtan Gilbert*, *Maxime Dénès*, *Vincent Laporte*, *Théo Zimmermann*, *Pierre-Marie Pédrot*, *Ben Delaware*, *Janno*, *Valentin Robert*, and *Robert Rand*. I am very lucky to be part of such a wonderful international research community, and I am looking forward to more trips to France once international trips resume.

The broader *PL community* has also been amazing. *Carlo Angiuli*, *Anders Mörtberg*, *Conor McBride*, and *Michael Shulman* really, really helped me navigate some of the more beautiful and challenging type theory that shows up in this thesis. Ideas from *Matthew Dwyer* and *Matt Might* are in the future work section of this thesis. *Bas Spitters*, *Jon Sterling*, *Bob Harper*, *Edward Z. Yang*, and *James Decker* all helped me with thesis-related work at some point. *Gerwin Klein* shepherded my final thesis-related paper, and did a really wonderful job. *Jonathan Aldrich*, *David van Horn*, *Michael Hicks*, *Emery Berger*, *Kenny Foner*, *Alexandra Silva*, *Lindsey Kuper*, *Nate Foster*, *Stephanie Weirich*, and many

others helped me give back to the community, and helped me when I needed guidance too. My mentor *Derek Dreyer* has been amazing.

The CS *Twitter community*—and especially the *PL Twitter community*—has not just been supportive, but has also directly contributed to my thesis work. I often take to Twitter to ask people to try out proof exercises related to my tools, or to brainstorm fun future work ideas, or to ask type theory questions, or to look for related work. Twitter followers *Nathanael* and *Quinn Wilton* thought of some of the medical devices that show up in the future work section of this thesis. *Hillel Wayne* offered to send me food as I wrapped up this section. *Rebecca Turner* showed me how to use the knowledge package that lets me highlight words, like **Rebecca**, and have them link to indexed definitions. *Ymir Vigfusson*, *Benjamin Lipp*, and *Daniel-Nikpayuk* found typos in screenshots I Tweeted. *Dionna Glaze*, *Jana Dunfield*, and *Amarin Phaosawasdi* also sent feedback. And *Michelle Lee* did a writing buddy log with me that motivated me to finish this thesis. Many others on Twitter—thousands of people—helped me over the last few years.

So thanks, everyone in the wonderful **Coq community**, and in the **PL community** more broadly. And thanks to everyone on **Twitter**, especially **PL Twitter**. You are all amazing people.

BELLE *Belle* is the best puppy. I cuddled with her early pandemic every night after work.

I love you, **Belle**, you cutie.

GRANDPA, SABA, & SAVTA In loving memory of *Grandpa*. I hope he would be proud. I never got to meet Grandma, but I hope she would be proud, too.

In loving memory of *Saba*, the most inspiring person in the world. I wish he'd stayed around for this, so I could explain my thesis work to him. I bet he'd understand it and enjoy it.

With all of my love for *Savta*, who makes sure I never go hungry or cold. And who, even if I'm not hungry or cold, makes sure I still know that she's worried I might be hungry or cold, and makes sure to feed me and tell me to wear a jacket anyways.

I miss you, **Grandpa**, and wish I could tell you what I'm up to now. I miss you, **Saba**, and wish I could share this thesis with you. **Savta**:

אני אוהבת אותך, סבתא. נתראה ביולי!

MOM & DAD I have the most amazing, supportive, and loving *parents* in the world. They're the reason I'm eating lunch as I write this. I can't imagine what graduate school would have been like without their constant unwavering support—or without their frequent gifts of coffee and food.

I love you so much, **Mom** & **Dad**.

1

INTRODUCTION

What would it take to empower programmers of all skill levels across all domains to formally prove the absence of costly or dangerous bugs in software systems—that is, to formally *verify* them?

Verification has already come a long way toward this since its inception. This is especially true when it comes to the scale of systems that can be verified. The seL4 [89] verified operating system (OS) microkernel, for example, is the result of a team effort spanning more than a million lines of proof, costing over 20 person-years. Given a famous 1977 critique of verification [51] (emphasis mine):

> *A sufficiently fanatical researcher* might be willing to devote *two or three years* to verifying a significant piece of software if he could be assured that the software would remain stable.

I could argue that, over 40 years, either verification has become easier, or researchers have become more fanatical. Unfortunately, not all has changed (emphasis still mine):

> But real-life programs need to be maintained and modified. There is *no reason to believe* that verifying a modified program is any easier than verifying the original the first time around.

This remains so difficult that sometimes, even experts give up in the face of change (Section 2.2.2).

This thesis aims to change that by taking advantage of a missed opportunity: tools for developing verified systems (Section 1.1) have no understanding of how these systems evolve over time, so they miss out on crucial information. This thesis introduces a new class of verification tools called *proof repair* tools (Section 1.2) that understand how software systems evolve, and use the crucial information that evolution carries to automatically evolve proofs about those systems (Section 1.3). This gives us **reason to believe** (Section 1.4).

1.1 DEVELOPING VERIFIED SYSTEMS

Proof repair falls under the umbrella of *proof engineering*: the technologies that make it easier to develop and maintain verified systems (Section 5.1). Much like software engineering scales programming to large systems, so proof engineering scales verification to large systems. In recent years, proof engineers have verified OS microkernels [89, 88], machine learning systems [146], distributed systems [168], constraint solvers [21], web browser kernels [82], compilers [98, 99, 93], file systems [28, 31, 30], and even a quantum optimizer [77]. Practitioners have found verified systems to be more robust and secure in deployment (Chapter 2).

Proof engineering focuses in particular on verified systems that have been developed using special tools called *proof assistants*. Examples of proof assistants include *Coq* [38], *Isabelle/HOL* [81], *HOL Light* [78], *HOL4* [117], *Agda* [7], *Lean* [97], and *NuPRL* [123]. The proof assistant that I focus on in this thesis will be the **Coq** proof assistant. A discussion of how this work carries over to other proof assistants is in Chapters 5 and 6.

To develop a verified system using a proof assistant like Coq, the proof engineer does three things:

1. implements a *program*,
2. *specifies* what it means for the **program** to be correct, and
3. *proves* that the **program** satisfies the **specification**.

This proof assistant then automatically checks this **proof** with a small trusted part of its system called the *kernel* [16, 17]. If the proof is correct, then the **program** satisfies its **specification**—it is **verified**.

1.2 THESIS

The challenge this thesis addresses is that **programs** and **specifications** change all of the time—and these changes can break many **proofs**. 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 a program that depends on that library—and that breaking change can be outside of the proof engineer's control.

In response to this challenge, this thesis introduces *proof repair*. Proof repair automatically fixes broken proofs in response to changes in programs and specifications. In other words, proof repair views these broken proofs as bugs that a tool can patch. In doing so, it shows that there *is* **reason to believe** that verifying a modified system should

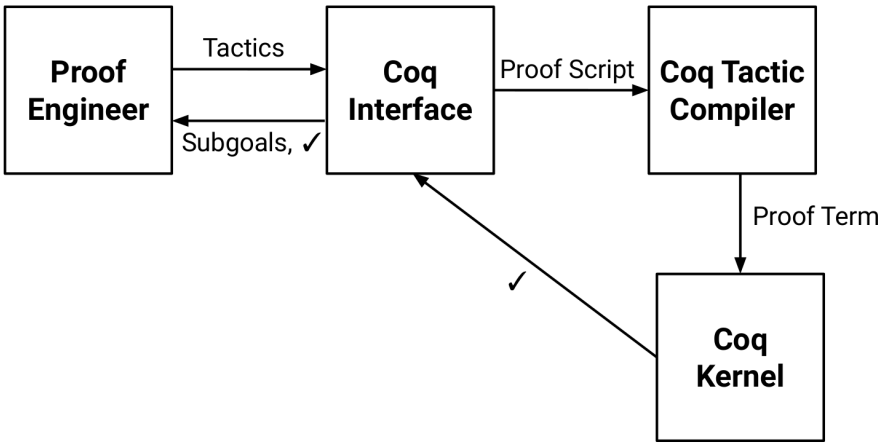


Figure 1: An illustration of the typical interactive workflow of using the **Coq proof assistant** to write a proof. The checkmark at the end represents correctness of the proof, which is communicated back to the proof engineer in the end.

often, in practical use cases, be easier than verifying the original the first time around, even when the proof engineer does not follow good development processes, or when change occurs outside of the proof engineer's control. More formally:

> ***Thesis***: Changes in programs, specifications, and proofs can 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.

1.3 APPROACH

My approach to proof repair operates over a low-level representation of proofs, one that carries useful structure and information. But that very structure can make it challenging for proof engineers to write proofs in that low-level representation to begin with, which is why proof engineers typically do not interact with it directly. Instead, proof engineers typically write proofs in a high-level representation.

Consider the typical proof engineering workflow in **Coq** (Figure 1). This workflow is interactive: To write a proof, the proof engineer passes Coq high-level search procedures called *tactics* (like `induction`), and Coq responds to each tactic by refining the current goal to some subgoal (like the goal for the base case). This loop of tactics and goals continues until no goals remain, at which point the proof engineer has constructed a sequence of tactics called a *proof script*—the high-level representation. To check the proof, Coq compiles that proof script down to a low-level representation called a *proof term*, then uses its

kernel to check that the proof term has the expected type. If the term has the expected type, Coq lets the proof engineer know in the end.

One major challenge for a proof repair tool is that it is not clear how to extract and generalize changes and apply them to fix proofs just by looking at the changes in high-level proof scripts that proof engineers make. The high-level language of tactics can abstract away important details of these changes. But the low-level language of proof terms can be brittle and challenging to work with. Crucially, though, the language is brittle and challenging precisely *because* it carries so much structure and gives such strong guarantees—two things that are very useful to a proof repair tool.

My approach to proof repair takes advantage of the structure and guarantees of the low-level language of proof terms, but produces something in the high-level language of proof scripts for proof engineers in the end. In particular, it uses *semantic differencing* algorithms over proof terms to extract information from a breaking change in a program, specification, or proof. It then combines those with program transformations over proof terms—called *proof term transformations*—to generalize and, in some cases, apply that information to fix other proofs broken by the same change. In the end, it uses a prototype decompiler to get from the low-level language back up to the high-level language, so that proof engineers can continue to work in that language going forward.

By working over the low-level language of proof terms, my approach to proof repair is able to systematically and with strong guarantees extract and generalize the information that breaking changes carry, then apply those changes to fix other proofs broken by the same change. But by later building up to the high-level language of proof scripts, my approach can in the end produce proofs that integrate more naturally with proof engineering workflows.

1.4 RESULTS

This thesis is divided into six chapters; chapters are further divided into sections. After motivating proof repair (Chapter 2), this thesis describes two different kinds of **proof repair** that validate the thesis: *by example* (Chapter 3) and *across equivalences* (Chapter 4). It concludes with a discussion of related work (Chapter 5), followed by a reflection on the thesis and how the results can inform the next era of proof engineering (Chapter 6).

The technical *results* of this thesis are threefold:

1. the *design* of semantic differencing algorithms & proof term transformations for repair (Sections 3.2, 3.3, 3.4, 4.2, 4.3, and 4.4),

2. an *implementation* of these algorithms and transformations inside of a proof repair tool suite (Sections 3.5 and 4.5), and

3. *case studies* to evaluate the tool suite on real proof repair scenarios (Sections 3.6 and 4.6).

Viewing the thesis statement as a theorem, the proof is as follows:

> Changes in programs, specifications, and proofs can carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change (by **design** and **implementation**). A tool that automates this can save work for proof engineers relative to reference manual repairs in practical use cases (by **case studies**).

DESIGN The **design** describes **semantic differencing** algorithms to extract information from breaking changes in verified systems, along with **proof term transformations** to generalize and apply the information to fix proofs broken by the change. The semantic differencing algorithms compare the old and new versions of a changed term or type, and from that find a **diff** that describes the change. The transformations then use that diff to **transform** some term to a more general fix. The details vary by the class of change supported. The design of these differencing algorithms and transformations is guided heavily by foundational developments in dependent type theory; the theory is sprinkled throughout as appropriate.

IMPLEMENTATION The **implementation** shows that in fact *a tool* can extract and generalize the information that changes carry, and then apply that information to fix other proofs broken by the same change. This implementation comes in the form of a proof repair tool suite for Coq called PUMPKIN PATCH (Proof Updater Mechanically Passing Knowledge Into New Proofs, Assisting the Coq Hacker). PUMPKIN PATCH is a suite of Coq *plugins*: extensions to Coq implemented in OCaml that can add new automation and syntax, and can define new terms. Notably, since all terms that plugins produce are checked by Coq in the end, PUMPKIN PATCH does not extend the *Trusted Computing Base* (TCB): the set of unverified components that the correctness of the proof development depends on [136]. In total, PUMPKIN PATCH is about 15000 lines of code, consisting of three plugins and a library that together bridge the gap between the theory supported by design and the practical proof repair needed for the case studies.

CASE STUDIES The *case studies* show that PUMPKIN PATCH can save work for proof engineers relative to reference manual repairs in practical use cases. In particular, the case studies in Chapter 3 show retroactively that a prototype implementation of proof repair **by example** *could have* saved work for proof engineers on major proof developments. The case studies in Chapter 4 show that proof repair **across type equivalences** *can* save and in fact *has already* saved work for proof engineers in practical use cases.

1.5 READING GUIDE

This thesis assumes some background in **proof engineering**, *type theory*, and (to a lesser extent) the **Coq proof assistant**. I strongly encourage readers of all backgrounds who would like more context to better understand this thesis look to my survey paper on proof engineering [136], which includes a detailed list of resources and is available for free on my website: `https://dependenttyp.es`.

I recommend that readers with less background on proof engineering, dependent type theory, or Coq take time to digest Chapter 2 before moving on—though I recommend that even Coq experts read Chapter 2! Chapters 3 and 4 get rather technical, so it is normal not to understand every detail. You may always contact me with questions.

Previously Published Material

While this thesis is self-contained, it centers material from two previously published papers:

- **Talia Ringer**, Nathaniel Yazdani, John Leo, and Dan Grossman. *Adapting Proof Automation to Adapt Proofs* [140]. Certified Programs and Proofs. 2018.
- **Talia Ringer**, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. *Proof Repair across Type Equivalences* [137]. Programming Languages Design and Implementation. 2021.

It also includes material from three other papers:

- **Talia Ringer**, Nathaniel Yazdani, John Leo, and Dan Grossman. *Ornaments for Proof Reuse in Coq* [141]. Interactive Theorem Proving. 2019.
- **Talia Ringer**, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. REPLica*: REPL Instrumentation for Coq Analysis* [138]. Certified Programs and Proofs. 2020.
- **Talia Ringer**, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. *QED at Large: A Survey of Engineering of Formally Verified Software* [136]. Foundations and Trends® in Programming Languages: Vol. 5: No. 2-3, pp 102-281. 2019.

The following is a map from each of these papers to corresponding sections, along with an explanation of what is new in this thesis and what is omitted. All of these papers are free on my website.

Adapting Proof Automation to Adapt Proofs: The bulk of Chapter 3 comes from this paper, though the content is significantly reorganized and reframed. The introduction and conclusion of Chapter 3 are fresh. Sections 3.2, 3.3, 3.4, and 3.5 include content beyond the original paper. Chapter 5 includes some related work from this paper, and Chapter 6 includes some future work from this paper.

Proof Repair across Type Equivalences: Parts of the introduction and Section 2.1.2 come from this paper. The bulk of Chapter 4 comes from this paper, though the content is likewise reorganized and reframed. The conclusion of Chapter 4 is fresh. Sections 4.2, 4.3, 4.4, and 4.5 all include content beyond the original paper. Chapter 5 includes some related work from this paper, and Chapter 6 includes some future work from this paper.

Ornaments for Proof Reuse in Coq: The example from Section 2.1 comes from this paper, though most of the text is new. Parts of Section 2.1.2 also come from this paper. Section 4.3 shows a simplified version of the differencing algorithm (originally called the search algorithm) from this paper. Section 4.6.2 includes the evaluation from this paper with additional context. Chapter 5 includes related work from this paper. This thesis retires the name of the tool from this paper (Devoid) in favor of the name of the generalized version of the tool from **Proof Repair across Type Equivalences** (Pumpkin Pi).

REPLica: Section 2.2.2 includes a few samples of this paper, and the abstract includes a few sentences from this paper.

QED at Large: Chapter 2 includes a few samples of this survey paper. Chapter 5 includes a large amount of related work from this paper.

Authorship Statements

The material in this thesis draws on work that I did with four student and postdoctoral coauthors: **Nathaniel Yazdani**, **RanDair Porter**, **Alex Sanchez-Stern**, and *Karl Palmskog*. Below is a summary of the contributions of each of those coauthors, indexed for later reference. The contributions of my faculty and professional coauthors—*John Leo*, **Dan Grossman**, **Zach Tatlock**, *Ilya Sergey*, *Milos Gligoric*, and *Sorin Lerner*—were of course also extremely valuable:

NATHANIEL YAZDANI I worked with **Nate** starting from when he was an undergraduate student. He contributed conceptually to all three proof repair papers his name appears on, helped with a number of the case studies, implemented important features on the critical path to success, and did some of the writing about his contributions. His contributions include:

1. a tool for preprocessing proof developments into a format suitable for repair,
2. higher-order transformations for applying proof term transformations over entire libraries, and
3. a key early insight about equality.

All three of these were necessary to scale proof repair to help real proof engineers in practical scenarios.

RANDAIR PORTER **RanDair** joined the project as an undergraduate. He implemented a prototype decompiler from proof terms to proof scripts, and wrote a description of the behavior of the decompiler that I built on in the corresponding paper. This decompiler was necessary for integrating proof repair tools with real proof engineering workflows, and it continues to inspire exciting work.

ALEX SANCHEZ-STERN **Alex** worked with me as a PhD student on the **REPLica** user study of proof engineers. He designed, implemented, and evaluated one of the two analyses in the user study paper. He also helped substantially in building the infrastructure necessary to deploy the user study, and wrote large sections of the paper. The user study and paper would not have happened without him.

KARL PALMSKOG **Karl** was a postdoctoral researcher when he joined me on the **QED at Large** survey paper. He wrote entire chapters of the survey paper. I could not have written that paper without him.

Pronouns

In this thesis, I use "I" to refer to work that I did, even though of course no work happens in a vacuum. I use the names of my coauthors like "**Nate**" or "**RanDair**" to refer to work that they did, when I operated primarily in an advisory role. When I collaborated with my coauthors, I name them and myself, like "**Nate** and I," and then (when not ambiguous) I use "we" thereafter. Throughout, I also use mathematical "we" to mean both myself and the reader.

When I discuss a rhetorical proof engineer who does not actually exist, like "the proof engineer," I always use "she"—this is a small attempt to seed the world with data that counteracts stereotypes. When preserving anonymity of a particular person, I always use singular "they." Otherwise, I use the person's pronouns.

Historical Note

At the time of writing, the Coq community is considering renaming the Coq proof assistant. There is a chance that the name of the proof assistant will be different for future readers.

2

MOTIVATING PROOF REPAIR

This thesis describes techniques and tools for automatically **repairing** broken **proofs** in a **proof assistant**. It focuses in particular on proofs about formally **verified programs**, though many of the techniques and tools carry over to mathematical proofs as well.

WHY VERIFY PROGRAMS? Formal verification of a program can improve actual and perceived reliability. It can help the programmer think about the desired and actual behavior of the program, perhaps finding and fixing bugs in the process [116]. It can make explicit which parts of the system are trusted, and further decrease the burden of trust as more of the system is verified.

One noteworthy program verification success story is the CompCert [98, 99] verified optimizing C compiler. Both the back-end and front-end compilation passes of CompCert have been verified, ensuring the correctness of their composition [85]. CompCert has stood up to the trials of human trust: it has been used, for example, to compile code for safety-critical flight control software [58]. It has also stood up to rigorous testing: while the test generation tool Csmith [170] found 79 bugs in GCC and 202 bugs in LLVM, it was unable to find any bugs in the verified parts of CompCert.

CompCert, however, was not a simple endeavor: the original development comprised approximately 35000 lines of code; functionality accounted for only 13% of this, while specifications and proofs accounted for the other 87%. This is not unusual for large proof developments. The initial correctness proofs for the seL4 OS microkernel, for example, consisted of 480000 lines of specifications and proofs [88]. **Proof engineering** technologies make it possible to develop verified systems at this scale. See **QED at Large** for a comprehensive overview of proof engineering.

WHY REPAIR PROOFS? **Proof repair**—the focus of this thesis—is a new proof engineering technology that focuses in particular on minimizing the burden of change as verified systems evolve over time. But for the sake of this chapter, I motivate proof repair not on a large verified system like a C compiler or an OS microkernel. Instead, I motivate it on a simple proof development: a list zip function

accompanied by a formal proof that it preserves the lengths of its inputs. This is a small example, but it is worth noting that large proof developments like compilers and OS microkernels are often made up of many of these small examples built on top of each other.

The proof assistant that I motivate this on is **Coq**, since this is the proof assistant that this thesis focuses on. I motivate proof repair in Coq using the small list zip example in three parts:

1. the workflow of and theory beneath *proof development* (Section 2.1),
2. some challenges of and approaches to *proof maintenance* (Section 2.2), and
3. the motivation for and approach to *proof repair* that follow (Section 2.3).

I will refer to the example and theory introduced in this chapter in later chapters, so it is good to at least skim this chapter regardless of Coq experience.

2.1 PROOF DEVELOPMENT

Before I motivate proof maintenance and repair, it helps to understand *proof development* to begin with. In the introduction, I briefly explained the workflow for using a **proof assistant** to develop a verified system, noting that the proof engineer:

1. implements a **program**,
2. **specifies** what it means for the **program** to be correct, and
3. **proves** that the **program** satisfies the **specification**.

In the **Coq** proof assistant, proof engineers implement programs in a rich functional programming language called *Gallina*. In fact, it is possible to use Gallina to write the program, the specification, *and* the proof—but writing the proof in Gallina can be challenging. Instead, proof engineers typically use Gallina to write only the program and specification, and write the proof interactively. I alluded to this when I explained the typical proof development workflow in Coq:

> To write a proof, the proof engineer passes Coq high-level search procedures called **tactics** (like `induction`), and Coq responds to each tactic by refining the current goal to some subgoal (like the goal for the base case). This loop of tactics and goals continues until no goals remain, at which point the proof engineer has constructed a sequence of tactics called a **proof script**—the high-level representation. To

```
zip {T1 T2} (l1 : list T1) (l2 : list T2) : list (T1 * T2) :=
  match l1, l2 with
  | nil, _ => nil
  | _, nil => nil
  | cons t1 tl1, cons t2 tl2 => cons (t1, t2) (zip tl1 tl2)
  end.
```

Figure 2: The list `zip` function, taken from an existing proof development [149]. The curly brace notation means that the type parameter `T` is implicit in applications.

> check the proof, Coq compiles that proof script down to a low-level representation called a **proof term**, then uses its **kernel** to check that the proof term has the expected type. If the term has the expected type, Coq lets the proof engineer know in the end.

The low-level language of proof terms in Coq is **Gallina**—the same rich functional programming language proof engineers use to write programs and specifications. The high-level language of proof scripts in Coq is a language called *Ltac* that I will soon describe.

In this thesis, I will not teach you all of Coq.[1] What I will do is motivate this workflow on an example (Section 2.1.1) and explain the theory beneath (Section 2.1.2).

2.1.1 *The Workflow*

For a moment, let us assume some primitives from the Coq standard library: the type `nat` of natural numbers, the type `list` of polymorphic lists, and the `length` function that computes the length of a list as a natural number. We will start by writing the list `zip` program, then specify what it means to preserve its length, and then finally write an interactive proof that shows that specification actually holds. In the end, Coq will check this proof and let us now that our proof is correct, so our `zip` function is verified.

PROGRAM Let our **program** be the list `zip` function, written in **Gallina** in Figure 2. The list zip function takes as arguments two lists l_1 and l_2 of possibly different types T_1 and T_2, and zips them together into a list of pairs $(T_1 * T_2)$. For example, if the inputs are a list of numbers and a list of characters, like:

```
l1 := cons 1 (cons 2 (cons 3 (cons 4 nil))).(* [1; 2; 3; 4] *)
l2 := cons "x" (cons "y" (cons "z" nil)).(* ["x"; "y"; "z"] *)
```

1 Good resources for learning more about Coq include the books Certified Programming with Dependent Types [32] and Software Foundations [132], and the survey paper **QED at Large**.

```
(* Weaker version of theorem *)
Theorem zip_preserves_length {T1 T2} :
  ∀ (l1 : list T1) (l2 : list T2),
    length l1 = length l2 →
    length (zip l1 l2) = length l1.

(* Stronger version of theorem *)
Theorem zip_preserves_length {T1 T2} :
  ∀ (l1 : list T1) (l2 : list T2),
    length (zip l1 l2) = min (length l1) (length l2).
```

Figure 3: Two possible specifications of a proof that `zip` preserves the length of the input lists.

then `zip` applied to those two lists returns a list of number-character pairs, like:

```
(* [(1, "x"); (2, "y"); (3, "z")] *)
cons (1, "x") (cons (2, "y") (cons (3, "z") nil)).
```

It is worth noting that the implementation of `zip` has to make some decision about how to behave when the input lists are different lengths—that is, what to do with the extra elements, like the extra `4` at the end of l_1. The decision that this implementation makes is to just ignore those extra elements.

Otherwise, the implementation is fairly standard. If l_1 is empty or, in other words, `nil` (first case), or if l_2 is `nil` (second case), then `zip` returns `nil`. Otherwise, `zip` combines the first two elements of each list into a pair (t_1, t_2), then sticks that in front of (using `cons`) the result of recursively calling `zip` on the tails of each list (`zip` tl_1 tl_2).

SPECIFICATION Once we have written our `zip` function, we can then **specify** what we want to prove about it: that the `zip` function preserves the lengths of the inputs l_1 and l_2. We do this by defining a type `zip_preserves_length` (Figure 3, top), which in Coq we state as a `Theorem`.[2] This theorem takes advantage of **Gallina**'s rich type system to quantify over all possible input lists l_1 and l_2. It says that if the lengths of the inputs are the same, then the length of the output is the same as the lengths of the inputs. Our proof will soon show that this type is inhabited, and so this statement is true.

It is worth noting that this step of choosing a specification is a bit of an art—we have some freedom when we choose our specification. We could just as well have chosen a different version of `zip_preserves_length` (Figure 3, bottom) that states that the length of the output is the *minimum* of the lengths of the inputs (using `min` from the Coq standard library). This is also true for our `zip` implementation, and in fact it is stronger—it implies the original theorem as well. But

2 We can also call this a `Lemma` if we'd like; these are equivalent in Coq.

regardless of which version we choose, we then get to the fun part of actually writing our proof.

PROOF As I mentioned earlier, it is possible to write **proofs** directly in **Gallina**—but this can be difficult. Instead, it is more common to write proofs interactively using the **tactic** language **Ltac**. Each tactic in Ltac is effectively a search procedure for a proof term, given the context and goals at each step of the proof. The way that this works is, after we state the theorem that we want to prove, say:

```
Theorem zip_preserves_length {T1 T2} :
  ∀ (l1 : list T1) (l2 : list T2),
    length l1 = length l2 →
    length (zip l1 l2) = length l1.
```

we then add one more word:

```
Proof.
```

then step down past that word inside of an Integrated Development Environment (IDE). The IDE then drops into an interactive proof mode. In that proof mode, it tracks the context of the proof so far, along with the goal we want to prove. After each tactic we add and step past, Coq responds by refining the goal into some subgoal and updating the context. We continue this until no goals remain. The **QED at Large** survey paper has a good overview of tactic languages in Coq and in other proof assistants, plus different interfaces and IDEs for writing proofs interactively, and screenshots of those interfaces in action.

In this case, after stepping past `Proof` in our IDE, our initial context (above the line) is empty, and our initial goal (below the line) is the original theorem:

```
______________________________________(1/1)
∀ (l1 : list T1) (l2 : list T2),
  length l1 = length l2 →
  length (zip l1 l2) = length l1.
```

We can start this proof with the introduction tactic `intros`:

```
  intros l1.
```

This is essentially the equivalent of the natural language proof strategy "assume arbitrary `l1`." That is, it moves the universally quantified argument from our goal into our context:

```
l1 : list T1
______________________________________(1/1)
∀ (l2 : list T2),
  length l1 = length l2 →
  length (zip l1 l2) = length l1.
```

From this state, we can induct over the input list (choosing names for variables Coq introduces in the inductive case):

```
  induction l1 as [|t1 tl1 IHtl1].
```

This breaks into two subgoals and subcontexts: one for the base case and one for the inductive case.

The base case:

```
______________________________________(1/2)
∀ (l2 : list T2),
  length nil = length l2 →
  length (zip nil l2) = length nil.
```

holds by reflexivity, which the `auto` tactic takes care of.

In the inductive case:

```
t1 : T1
tl1 : list T1
IHtl1 :
  ∀ (l2 : list T2),
    length tl1 = length l2 →
    length (zip tl1 l2) = length tl1
______________________________________(2/2)
∀ (l2 : list T2),
  length (cons t1 tl1) = length l2 →
  length (zip (cons t1 tl1) l2) = length (cons t1 tl1).
```

we again use `intros` and `induction`, this time to induct over l_2. This again produces two subgoals: one for the base case and one for the inductive case. The base case has an absurd hypothesis, which we introduce as `H` and then use `auto` to show that it implies our conclusion holds. The inductive case holds by simplification and rewriting by the inductive hypothesis $IHtl_1$.

After this, no goals remain, so our proof is done; we can write `Defined`.[3] What happens when we write `Defined` is that Coq compiles the **proof script** we have just written down to a **proof term**. Coq's **kernel** then checks that the type of this term is the theorem we have stated. Since it is, Coq lets us know that our proof is correct, so our `zip` function is verified.

Figure 4 shows the resulting proof script for this theorem (top), along with the corresponding proof term (bottom). As we can see, the proof term is quite complicated—I will explain what it means soon, in Section 2.1.2. The important thing to note for now is that the details of this low-level proof term do not matter much to us as proof engineers, since we can write the high-level proof script on the top instead. Even though this proof script is still a bit manual (for the sake of demonstration), it is much simpler than the low-level proof term.

Writing proofs using tactics does indeed make proof development easier than writing raw proof terms. But these highly structured proof terms carry a lot of information that is lost at the level of tactics. It is exactly that rich structure—the type theory beneath Gallina—that makes a principled approach to proof repair possible.

3 We can also write `Qed`. There is a subtle difference between the behavior of `Defined` and `Qed` that does not matter for the sake of this thesis; I usually favor `Defined`.

```
Theorem zip_preserves_length {T1 T2} :
  ∀ (l1 : list T1) (l2 : list T2),
    length l1 = length l2 →
    length (zip l1 l2) = length l1.
Proof.
  intros T1 T2 l1. induction l1[1] as [|t1 tl1 IHtl1].
  - auto.[2]
  - intros l2. induction l2[3] as [|t2 tl2 IHtl2].
    + intros H. auto.[4]
    + intros H. simpl.  rewrite IHtl1; auto.[5]
Defined.

zip_preserves_length :
  ∀ {T1} {T2} (l1 : list T1) (l2 : list T2),
    length l1 = length l2 →
    length (zip l1 l2) = length l1
:=
fun (T1 T2 : Type) (l1 : list T1) (l2 : list T2) =>
  list_rect[1] (fun (l1 : list T1) => ...)
    (fun (l2 :  list T2) _ => eq_refl)[2]
    (fun (t1 : T1) (tl1 : list T1) (IHtl1 : ...) (l2 : list T2) =>
      list_rect[3] (fun (l2 : list T2) => ...)
        (fun (H : ...) => eq_sym H)[4]
        (fun (t2 : T2) (tl2 : list T2) (IHtl2 : ...) =>
          fun (H : ...) => eq_rect_r ... eq_refl (IHtl1 ...)[5])
        l2[3])
    l1[1]
    l2.
```

Figure 4: An **Ltac proof script** (top) and a corresponding **Gallina proof term** (bottom) in **Coq** that shows that the list zip function preserves the lengths of the input lists. Some details of the proof term are omitted for simplicity. Corresponding parts are highlighted in the same color and annotated with the same number; the rest is boilerplate.

```
Inductive nat :=
| O : nat[1]
| S : nat → nat[2].
```

```
nat_rect :
  ∀ (P : nat → Type),
    P O[1] →
    (∀(n :  nat), P n → P (S n))[2] →
    ∀ (n : nat), P n.
```

Figure 5: The type of natural numbers `nat` in Coq defined inductively by its two **constructors** (left), and the type of the corresponding **eliminator** or induction principle `nat_rect` that Coq generates (right).

2.1.2 *The Theory Beneath*

Now that we have written a small proof development in Coq, let us take a step back and look at the theoretical foundations that make this possible. While proof scripts help humans like us write proofs, it is thanks to the proof terms these compile down to that Coq is able to check our proofs for us. These proof terms are in the rich functional programming language Gallina (Section 2.1.2.1). Gallina implements a rich type theory called the *Calculus of Inductive Constructions* (Section 2.1.2.2). This rich type theory makes it possible to write programs, specifications, and proofs in Coq, and have a small part of Coq—the **kernel**—check those proofs in the end.

2.1.2.1 *Gallina*

To see the power of Coq's **proof term** language **Gallina**, let us dissect our proof that `zip` preserves its length. Our proof development about `zip` uses the `nat` and `list` datatypes, as well as the `length` function. It also uses the equality type `=`. All of these can be found inside of the Coq standard library. But dissecting them already points to two important features of Gallina: *inductive types* and *intensionality*. Both of these arise in our proof development, and will continue to arise throughout this thesis.

INDUCTIVE TYPES Each of `nat` and `list` in Gallina is what is called an **inductive type**. An inductive type is defined by its *constructors*: the ways of constructing a term with that type. A `nat` (Figure 5, left), for example, is either `O` or the successor `S` of another `nat`; these are the two constructors of `nat`.

Every inductive type in Gallina comes equipped with an *eliminator* (also called an induction principle) that the proof engineer can use to write functions and proofs about the datatype. For example, the eliminator for `nat` (Figure 5, right) is the standard induction principle

```
Inductive list {T : Type} :=
| nil :  list T[1]
| cons :
   T → list T → list T.[2]
```

```
list_rect {T : Type} :
  ∀ (P : list T → Type),
    P nil[1] →
    ((∀(t :  T) (tl :  list T),
      P tl → P (cons t tl))[2] →
    ∀ (l : list T), P l.
```

Figure 6: The type of polymorphic lists `list` in Coq defined inductively by its two **constructors** (left), and the type of the corresponding **eliminator** or induction principle `list_rect` that Coq generates (right).

```
Fixpoint length {T} l :=
  match l with
  | nil => O[1]
  | cons t tl =>
     S (length tl)[2]
  end.
```

```
Definition length {T} l :=
  list_rect T (fun _ => nat)
    O[1]
    (fun t tl (length_tl :  nat) =>
      S length_tl)[2]
    l.
```

Figure 7: The list `length` function, defined by pattern matching and recursion (left) and using the eliminator `list_rect` (right).

for natural numbers, which Coq calls `nat_rect`.[4] This eliminator states that a statement `P` (called the inductive *motive*) about the natural numbers holds for every number if it holds for `O` in the base case and, in the inductive case, assuming the *inductive hypothesis* that it holds for some `n`, it also holds for the successor `S n`.

A `list` (Figure 6, left) is similar to a `nat`, but with two differences: `list` is polymorphic over some type `T` (so we can have a list of natural numbers, for example, written `list nat`), and the second constructor adds a new element of the type `T` to the front of the list. Otherwise, `list` also has two constructors, `nil` and `cons`, where `nil` represents the empty list, and `cons` sticks a new element in front of any existing list. Similarly, the eliminator for `list` (Figure 6, right) looks like the eliminator for `nat`, but with an argument corresponding to the parameter `T` over which `list` is polymorphic, and with an additional argument corresponding to the new element in the inductive case.

One interesting thing about the types of these eliminators `list_rect` and `nat_rect` is that they include universal quantification over all inputs, written ∀. Gallina's type system is expressive enough to include universal quantification over inputs, as we will soon see.

We can use these eliminators to write not just proofs, but also functions, like the `length` function (Figure 7, right). For functions, though, it is more standard to instead use pattern matching and

4 For technical reasons that are not important to this thesis, Coq actually defines *three* eliminators: `nat_rect`, `nat_rec`, and `nat_ind`. For the sake of this thesis, it is sufficient to imagine that these are all the same.

```
zip {T1} {T2} (l1 : list T1) (l2 : list T2) : list (T1 * T2) :=
  list_rect (fun _ : list T1 => list T2 → list (T1 * T2))
    (fun _ => nil)
    (fun t1 tl1 (zip_tl1 : list T2 → list (T1 * T2)) l2 =>
      list_rect (fun _ : list T2 => list (T1 * T2))
        nil
        (fun t2 tl2 (_ : list (T1 * T2)) =>
          cons (t1, t2) (zip_tl1 tl2))
        l2)
    l1
    l2.
```

Figure 8: The list `zip` function from Figure 2, translated to use eliminators.

recursion with a syntactic guard condition,[5] like the `length` function from the Coq standard library (Figure 7, left). Both of these functions behave the same way, but the function on the left is perhaps a bit easier to understand from a traditional programming background: the `length` of the empty list `nil` is `O`, and the length of any other list is the successor (`S`) of the result of recursively calling `length` on everything but the first element of the list. Indeed, `list_rect`—like all eliminators in Coq—is a constant that refers to a function itself defined using pattern matching and recursion with a syntactic guard condition. In fact, eliminators are equally expressive to pattern matching and recursion with a syntactic guard [64, 34].[6]

For the sake of this thesis, however, I will assume *primitive eliminators*: eliminators that are a part of the core syntax and theory itself, rather than being defined by way of pattern matching and recursion. Likewise, when I show Gallina code, from this point forward, I will favor functions that apply eliminators rather than pattern matching, like the `length` function from Figure 7 on the right.

To handle practical code that uses pattern matching and recursion, I preprocess the code using a tool by **Nate**. This preprocessing tool transforms functions that use simple pattern matching and recursion to instead use Coq's automatically generated eliminators (Section 4.5.1.3). Figure 8, for example, shows the `zip` function after running the preprocessing tool. In the rest of this thesis, I skip the step of running this preprocessing tool in examples, though the corresponding code invokes it explicitly.

5 Coq does not support arbitrary recursion. The syntactic guard on recursion forces recursive functions in Coq to terminate. **QED at Large** explains ways to reason about possibly nonterminating functions in spite of this.

6 What may be confusing, though, is that *Coq's* automatically generated eliminators are *not* as expressive Coq's pattern matching and recursion, though it *is* possible to manually define sufficiently expressive eliminators in Coq. This is an artifact of Coq's history.

INTENSIONALITY Gallina is based on what is called an **intensional** type theory: a type theory that distinguishes between equalities that hold by reduction (*definitional equality*), and those that hold by proof (*propositional* equality). That is, two terms `t` and `t'` of type `T` are definitionally equal if they reduce[7] to the same normal form, and propositionally equal if there is a proof that `t = t'` using the inductive equality type `=` at type `T`. Definitionally equal terms are necessarily propositionally equal, but the converse is not in general true.

The inductive equality type also shows up inside of our proof development—that is what the `=` sign in our theorem statement means. The `=` type has exactly one constructor, `eq_refl` (reflexivity), which can be applied to two terms exactly when those terms are definitionally equal. Propositional equality is more general, though, because it comes equipped with an eliminator `eq_rect` (or the reverse, `eq_rect_r`) that encodes rewriting: if `t = t'`, and some motive `P` holds on `t`, then the motive must also hold on `t'`. It is possible to use this eliminator to prove equalities by sequences of rewrites, substituting equal terms for one another until the goal holds by reflexivity.

BACK TO OUR PROOF With that in mind, we can now look back at the proof script and corresponding proof term in Figure 4. There is a correspondence between the proof script and the proof term, highlighted in the same color: The proof term is a function from the context to a body that proves the goal type. Every call to `induction` in the proof script shows up as an application of the `list` eliminator `list_rect`, with the cases corresponding to the appropriate arguments of the eliminator. The first call to `auto` compiles down to `eq_refl`, the constructor for the inductive equality type. The second call to `auto` compiles down to `eq_sym`, the proof of symmetry of equality in the Coq standard library. Rewrites compile down to applications of `eq_rect_r`, an eliminator over the inductive equality type.

Still, the proof terms can be difficult for a proof engineer to write and understand. In Section 4.5, I will introduce a prototype decompiler by **RanDair** from proof terms back up to proof scripts. This decompiler will make it possible for PUMPKIN PATCH to work over highly structured Gallina terms, but produce Ltac proof scripts that the proof engineer can use going forward.

2.1.2.2 *Calculus of Inductive Constructions*

The type theory that Gallina implements is CIC_ω, or the **Calculus of Inductive Constructions**. CIC_ω is based on the *Calculus of Con-*

7 Reduction uses a sequence of predefined reduction rules, which are described by various Greek letters. β-reduction, for example, has the standard meaning. δ-reduction unfolds constants. Chapter 10 of Certified Programming with Dependent Types [32] includes a nice summary of reduction and its relationship to **definitional equality**.

$\langle i \rangle \in \mathbb{N}$, $\langle v \rangle \in \text{Vars}$, $\langle s \rangle \in \{ \text{Prop}, \text{Set}, \text{Type}\langle i \rangle \}$

$$
\begin{aligned}
\langle t \rangle ::= \ & \langle v \rangle \\
| \ & \langle s \rangle \\
| \ & \Pi\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \\
| \ & \lambda\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \\
| \ & \langle t \rangle\ \langle t \rangle
\end{aligned}
$$

Figure 9: Syntax for CoC_ω with (from top to bottom) variables, sorts, function types, functions, and application.

$$
\begin{aligned}
\langle t \rangle ::= \ & \ldots \\
| \ & \text{Ind}\ (\langle v \rangle : \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\} \\
| \ & \text{Constr}\ (\langle i \rangle, \langle t \rangle) \\
| \ & \text{Elim}(\langle t \rangle, \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\}
\end{aligned}
$$

Figure 10: CIC_ω is CoC_ω with **inductive types**, inductive **constructors**, and **primitive eliminators**.

structions (CoC), a variant of the lambda calculus with two kinds of universally quantified function types: *polymorphism* (types that depend on types) and *dependent types* (types that depend on terms) [40]. CoC with an infinite universe hierarchy—basically a trick for logical consistency—is called CoC_ω.[8] The syntax for CoC_ω is in Figure 9. Note that whereas in Gallina we represent universal quantification over terms or types with $\forall$, here we represent it with Π, as is standard.

CIC_ω extends **CoC_ω** with **inductive types** [41]; the syntax for CIC_ω (building on syntax from an existing paper [153]) is in Figure 10, and the typing rules are standard and omitted. As in Gallina, inductive types are defined by their constructors and eliminators. Consider the inductive type `list` of polymorphic lists that we saw in Figure 6 (fixing a type parameter `T`), this time in CIC_ω:

```
Ind (list T : Type) {
  list T¹,
  T → list T → list T²
}
```

where the `nil` constructor type is the zeroth constructor, and the `cons` constructor type is the first constructor. Accordingly, the terms:

```
Constr (0, list T)¹
```

and:

```
Constr (1, list T)²
```

refer to the constructors `nil` and `cons`, respectively.

8 The need for the infinite universe hierarchy is too distracting for me to explain in detail here, but it is also not important for this thesis.

As in Gallina, inductive types like `list` come associated with **eliminators**. Unlike in Gallina, here we truly assume **primitive eliminators**—that these eliminators do not reduce at all. Instead, we represent them explicitly with the `Elim` construct. Thus, to eliminate over a list `l` with motive `P`, we write:

```
Elim (l, P) {
  f0[1],
  f1[2]
}
```

where functions:

```
f0 :  P (Constr (0, list T))[1]
```

and:

```
f1 :  Π (t :  T) (l :  list T) (IHl :  P l) .
       P ((Constr (1, list T)) t l)[2]
```

prove the base and inductive cases, respectively. When `l`, `P`, f_0, and f_1 are arbitrary, this statement has the same type as `list_rect` in Gallina.

CONVENTIONS Throughout this thesis, I assume the existence of an inductive **propositional** equality type = with constructor `eq_refl` and eliminator `eq_rect`. I also assume an inductive type Σ for existential or refinement types, with constructor $\exists$ and projections π_l and π_r. For simplicity of presentation, when I write terms in CIC_ω, I assume variables are names, and that all names are fresh.

I often present **differencing** algorithms and **transformations** over CIC_ω relationally, using a set of judgments; to turn these relations into algorithms, prioritize the rules by running the derivations in order, falling back to the original term when no rules match. I use $\vec{t}$ and $\{t_1, \ldots, t_n\}$ to denote lists of terms; the default derivation for a list of terms is to run the derivation on each element of the list individually.

FROM CIC_ω BACK TO GALLINA Gallina implements CIC_ω, but with some important differences. Three differences are especially relevant: The first is that Gallina lacks **primitive eliminators**, as I mentioned earlier. The second is that Gallina has constants that define terms—later on, this will help with building optimizations for proof repair tools. The third is that variables in Gallina are de Bruijn indices rather than names—the implementation handles this discrepancy.

Otherwise, a proof repair tool for Gallina can harness the power of CIC_ω. In particular, Π makes it possible to quantify over both terms and types, so that we can state powerful theorems and prove that they hold. Inductive types make it possible to write proofs by induction. Both of these constructs mean that terms in Gallina are extremely structured, and as we will soon see, that structure makes a proof repair tool's job much easier.

FROM GALLINA BACK TO LTAC This very same structure that helps proof repair tools can be difficult for proof engineers to work with, which is why proof engineers typically rely on **Ltac tactics**. Chapter 4 will introduce a prototype decompiler by **RanDair** from **Gallina** back up to **Ltac**, so that proof repair tools can suggest tactics in the end.

Tactics more generally are a form of *proof automation*, or tools that automatically search for proofs. This proof automation makes it much simpler to develop proofs to begin with. But it turns out this proof automation is a bit naive when it comes to *maintaining* proofs as programs and specifications change over time. Proof repair is a new form of proof automation for maintaining proofs: it uses the rich type information carried by proof terms to automatically fix broken proofs in response to change.

2.2 PROOF MAINTENANCE

What does it mean to *maintain* a **verified** system? Like all software systems, verified systems evolve over time. The difference is that, for verified systems, the proofs must evolve alongside the rest of the system (Section 2.2.1). Proof engineers typically use development processes to make proofs less likely to break in the face of these changes. Still, even with good development processes, breaking changes happen all the time, even for experts (Section 2.2.2). All of this points to a need for change-aware proof automation—that is, **proof repair**.

2.2.1 *Breaking Changes*

As verified systems evolve over time, both **programs** and **specifications** can change. Either of these changes can break existing **proofs**.

Consider the example from Section 2.1.1. We had two choices for the specification of `zip_preserves_length`. We chose the weaker specification on the top of Figure 3. This gave us some freedom in how we implemented our `zip` function. At some point, however, we may wish to change `zip`, and update our proof so that it still holds. Alternatively, we may wish to port our development to use the stronger specification on the bottom of Figure 3. We may even wish to use a datatype more expressive than `list`, as I will show you in Chapter 4. Any of these changes can break proofs in our proof development.

CHANGING OUR PROGRAM Since we chose the weaker specification of `zip_preserves_length`, we are free to change how our `zip` function from Figure 2 behaves on edge cases, when the lengths of input lists are not equal. Suppose we change our `zip` function to always return `nil` in those cases, by just returning the old behavior when the lengths are equal, and otherwise returning `nil`. To do this, we rename our old `zip` function to be `zip_same_length`. We then define a new

`zip` function that breaks into those two cases, calling `zip_same_length` when the lengths are equal, and otherwise returning `nil`:

```
zip {T1} {T2} (l1 : list T1) (l2 : list T2) : list (T1 * T2) :=
  sumbool_rect (fun _ => list (T1 * T2))
    (fun (_ : length l1 = length l2) =>
      zip_same_length l1 l2)
    (fun (_ : length l1 = length l2) → False) =>
      nil)
    (eq_dec (length l1) (length l2)).
```

where `sumbool_rect` is an eliminator that lets us break into these two cases, and `eq_dec` says that equality is decidable over natural numbers (that is, any two numbers are either equal or not equal).

Our theorem `zip_preserves_length` still holds, but after changing our program, the *proof* that it holds breaks. We can fix it by adding the highlighted tactics:

```
Proof.
  intros.  unfold zip.
  induction (eq_dec (length l1) (length l2)); try contradiction.
  simpl.  revert a.  revert H. revert l2.
  induction l1 as [|t1 tl1 IHtl1].
  - auto.
  - intros l2. induction l2 as [|t2 tl2 IHtl2].
    + intros H. auto.
    + intros H. simpl. rewrite IHtl1; auto.
Defined.
```

If we have many proofs about `zip`, they may break in similar ways, and require similar patchwork. This can be painful!

CHANGING OUR SPECIFICATION Suppose we instead wish to switch to use the stronger specification on the bottom of Figure 3, and keep our `zip` function the same. We can then update our proof accordingly, but after changing this specification, other proofs may break. For example, if we had proven a lemma for the `cons` case:

```
Lemma zip_preserves_length_cons {T1 : Type} {T2 : Type} :
  ∀ (l1 : list T1) (l2 : list T2) (t1 : T1) (t2 : T2),
    length l1 = length l2 →
    length (zip (cons t1 l1) (cons t2 l2)) = S (length l1).
```

that followed by `zip_preserves_length`:

```
Proof.
  intros T1 T2 l1 l2 t1 t2 H.
  simpl. f_equal.
  rewrite zip_preserves_length; auto.
Defined.
```

then after the change, this proof would break.

We would have two choices to fix it:

1. leave our specification alone, and fix our proof, or
2. strengthen the specification of the broken proof.

In the first case, leaving our specification alone, we could write this patched proof instead (with the difference highlighted):

```
Proof.
  intros T1 T2 l1 l2 t1 t2 H.
  simpl. f_equal.
  rewrite ← min_id.  rewrite H at 2.
  apply zip_preserves_length; auto.
Defined.
```

The extra tactics correspond to an extra proof obligation: we must now show that `length l1 = min (length l1) (length l2)`. This holds by the lemma `min_id` from the Coq standard library, combined with the hypothesis that says that `length l1= length l2`.

Alternatively, in the second case, strengthening the specification:

```
Lemma zip_preserves_length_cons {T1 : Type} {T2 : Type} :
  ∀ (l1 : list T1) (l2 : list T2) (t1 : T1) (t2 : T2),
    length (zip (cons t1 l1) (cons t2 l2)) =
    S (min (length l1) (length l2)).
```

we could leave the proof alone:

```
Proof.
  intros T1 T2 l1 l2 t1 t2.
  simpl. f_equal.
  apply zip_preserves_length; auto.
Defined.
```

But this could continue to break other downstream proofs that depend on `zip_preserves_length_cons`, causing a cascading effect of change. And this sort of cascading effect is precisely why the challenges of change are exacerbated at scale, affecting even experts.

2.2.2 *Even Experts are Human*

Proof engineers often use development processes to work around some of the challenges of change to begin with (Section 5.1.2). For example, they might use information hiding techniques [168, 87] to abstract away the details of `zip` or of `zip_preserves_length`, so that the burden of change becomes just showing that the changed program or proof still implements that abstraction. Similarly, they may build their own custom tactics that hide the details of `zip` or `zip_preserves_length` behind the automation itself, so that the burden of change becomes just fixing the automation.

But even with good development processes, proof engineers change programs and specifications all the time—and this does break proofs, even for experts. **Alex** and I found evidence of this in our user study of Coq proof engineers. For this user study, Alex and I built and deployed a Coq **plugin** called **REPLica** (REPL Instrumentation for Coq Analysis). REPLica listens to the Read Eval Print Loop (REPL)—a simple loop that all user interaction with Coq passes through—to collect data that the proof engineer sends to Coq during development.

```
 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 11: Patches to a lemma by an expert proof engineer, from the REPLica user study.

We used REPLica to collect a month's worth of granular data on the proof developments of 8 intermediate to expert Coq proof engineers. We visualized and analyzed this data to classify hundreds changes to programs and specifications, and fixes to broken proofs. The resulting data, analyses, and proof repair benchmarks are publicly available with the proof engineers' consent.[9]

We found that changes to programs and specifications were often formulaic and repetitive. For example, Figure 11 shows an example change by an expert proof engineer. In this change, the proof engineer wrapped two arguments into a single application of `Val` in three different hypotheses of a lemma. This change did not occur in isolation: the proof engineer patched 10 other definitions or lemmas in similarly, wrapping arguments into an application of `Val`.

We also found that changes to programs and specifications did break proofs, even for expert proof engineers. The proof engineers most often (75% of the time) fixed broken proofs by stepping up above those proofs in the UI and fixing something else, like a specification. That is, development and maintenance were in reality tightly coupled.

But sometimes, proof engineers did not successfully fix proofs broken by changes in programs and specifications. For example, for the change in Figure 11, the expert proof engineer admitted or aborted (that is, gave up on) the proofs of four of the five broken lemmas after this change. In other words, right now, even experts sometimes just give up in the face of change. This is empirical evidence of a problem reported by proof engineers tasked with maintaining large proof developments like CompCert or seL4 (Section 5.1.3).

9 http://github.com/uwplse/analytics-data

The reason that experts still struggle with change is that design principles all rely on proof engineers having the right foresight to choose the right abstractions in the right places, and hide the right information behind them. But proof engineers do not always have perfect foresight. They may write proofs that depend on details like the names of variables or the names of lemmas, much like our proofs in Section 2.2.1. Or they may choose abstractions to hide information, but those may not be the abstractions they still want after the change, or they may hide the wrong information behind the abstractions. Worse, breaking changes may happen outside of the proof engineers' control, in libraries upon which their proof developments depend.

In other words, even experts are human. And with traditional proof automation, the burden of change largely falls on those very humans. This is because traditional proof automation considers only the current state of programs, specifications, and proofs. It has no way of representing changes in programs, specifications, and proofs over time. So when traditional proof automation breaks in response to change, it cannot help the proof engineer fix the broken proof. But proof repair—smarter proof automation—can.

2.3 PROOF REPAIR

Proof repair is a new form of **proof automation** that automatically fixes broken proofs in response to change. Unlike traditional proof automation, proof repair views programs, specifications, and proofs as fluid entities. When a program or specification changes and this breaks proofs, proof repair extracts information from those changes, generalizes it, and applies it to fix proofs broken by the change.

The name of proof repair is inspired by program repair [114, 63], or automatically fixing bugs in programs. But my proof repair tools work differently from program repair tools, using a combination of **semantic differencing** algorithms and **proof term transformations** (Section 2.3.1). All of this happens over low-level proof terms in **Gallina**—and this is the key to success (Section 2.3.2).

2.3.1 *How Proof Repair Works*

Recall my **thesis**:

> Changes in programs, specifications, and proofs can 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.

My proof repair tools are the tools that automate this, using a combination of semantic differencing and proof term transformations. The

differencing algorithms compare the old and new version of the program, specification, or proof that has changed, and from that extract the information carried by the change. The **transformations** then generalize that information and, in some cases, apply it to fix other proofs broken by the same change. The details of all of this vary by the kind of breaking change, as I will demonstrate in Chapters 3 and 4.

The way that this works is quite different from the way that program repair tools typically work. A number of program repair tools work by running tests or the programs themselves, and many use fitness functions to identify candidate patches that are almost correct [114]. But there are not natural analogues of this in the world of proofs: there are often no tests, it is not possible to just run the proof, and there is not a natural fitness function that describes what it means for a patch to a proof to be almost correct.

In addition, proof engineers write proofs in this high-level language of **tactics**, **Ltac**. Each of these tactics is really a search procedure for a **proof term**, so it is not straightforward to apply typical program repair techniques to identify the next search procedure when a proof breaks. Instead, proof repair tools can look down at the low-level language of proof terms, **Gallina**. But this is difficult: the type theory **CIC$_\omega$** beneath Gallina is so rich that Gallina itself is quite unforgiving. That is, even very small changes can produce proof terms that no longer type check. But—and this is the key to proof repair—the unforgiving nature of Gallina actually turns out to be *a good thing*.

2.3.2 *The Key to Proof Repair*

The key to proof repair in this thesis is using the structure and information carried by Gallina proof terms. In other words, differencing operates over Gallina terms, and is guided by their semantics to narrow down the search space—it is **semantic differencing**. The transformations use the result of differencing to transform some proof term to a more general patch (in Chapter 3) or the patched proof itself (in Chapter 4)—they are **proof term transformations**.

This approach circumvents two of the biggest challenges in program repair: gathering enough information to efficiently search for a patch, and knowing when that patch is actually correct (Section 5.2). Thanks to the rich type theory beneath Gallina, changes in programs, specifications, and proofs carry *so much information* that my tools can use to search for a patch, and proofs provide *so much certainty* that the patch my tools find is correct in the end. This is why proof repair works.

But this approach presents its own challenges, like how to deal with the unforgiving nature of proof terms, and how to produce friendly proof scripts in the end. So Chapters 3 and 4 will show two tools that instantiate this approach, and describe how these tools tackle these challenges. Each chapter will introduce a tool that supports a different

class of changes. The first tool (Chapter 3) implements proof repair **by example**, while the second tool (Chapter 4) implements proof repair **across equivalences**. Chapters 3 and 4 will follow a parallel structure:

- **Motivating Example** (Sections 3.1 and 4.1): an example that motivates the supported class of changes.
- **Approach** (Sections 3.2 and 4.2): a high-level description of how the approach in Section 2.3.1 is instantiated.
- **Differencing** (Sections 3.3 and 4.3): detailed explanations of the corresponding differencing algorithms.
- **Transformations** (Sections 3.4 and 4.4): detailed explanations of the corresponding proof term transformations.
- **Implementation** (Sections 3.5 and 4.5): a description of the implementation of the approach as a tool for Coq.
- **Results** (Sections 3.6 and 4.6): results from case studies and experiments that show the tool can save work for proof engineers.
- **Conclusion** (Sections 3.7 and 4.7): a conclusion and reflection on how the thesis is validated so far.

Enjoy.

HISTORICAL NOTE Chapter 3 draws on the 2018 work that introduced the original proof repair vision, while Chapter 4 draws on the more mature work that followed. This difference in maturity permeates the **results** of each chapter:

- **Design**: Chapter 3 describes its algorithms at a high level, as ad hoc heuristics that sometimes struggle with undecidability. It defines judgments for these algorithms as interfaces, but does not provide the derivations. Chapter 4 formalizes more elegant algorithms, building on insights from Chapter 3, and cleanly separating the decidable and undecidable parts.
- **Implementation**: Chapter 3 describes a *prototype* Coq **plugin**. This prototype includes little automation for *applying* patches, integrates poorly into proof engineering workflows, and does not properly unfold constants (δ-reduce). Chapter 4 details technologies that address these limitations, some of which can be used with the Chapter 3 prototype if desired.
- **Case studies**: Chapter 3 shows only that a tool *could have helped* proof engineers retroactively on a few repair scenarios over small proofs of a fixed style. Chapter 4 shows that a tool *can help* and in fact *has helped* proof engineers on a variety of real repair scenarios, supporting larger proof developments (about 10000 LOC) in an order of seconds, regardless of proof style.

3

PROOF REPAIR BY EXAMPLE

The first tool in the **Pumpkin Patch proof repair plugin** suite is the namesake Pumpkin Patch prototype plugin. To prevent confusion, when I refer to the Pumpkin Patch prototype and not to the tool suite as a whole, I will abbreviate it as Pumpkin.

Pumpkin implements **proof repair by example**—so called because of its resemblance to programming by example [71]. In this approach to proof repair, the proof engineer provides an *example* of how to patch a proof in response to a breaking change. A tool then generalizes the example patched proof into a *reusable patch* that the proof engineer can use to fix other proofs broken by that change. In this way, proof repair by example is a new form of **proof automation** that accounts for how breaking changes in programs and specifications are sometimes reflected in the patches to the proofs they break.

In other words, in the frame of the thesis, proof repair by example extracts information from changes in proofs, then generalizes it to information corresponding to changes in the programs and specifications that broke those proofs to begin with (Section 3.2). This extraction and generalization works at the level of proof terms, through a combination of **semantic differencing** algorithms over proof terms (Section 3.3) and semantics-aware **proof term transformations** (Section 3.4). **Pumpkin** automates this process (Section 3.5). **Case studies** show retroactively that Pumpkin could have saved work for proof engineers on major proof developments (Section 3.6).

3.1 MOTIVATING EXAMPLE

To motivate proof repair by example, consider a commit from the Coq 8.7 release [110]. This commit redefined injection from integers to reals (Figure 12), or `IZR`. The change in `IZR` broke 18 proofs in the standard library, plus many proofs in client proof developments. The Coq standard library developer who committed the change fixed most of the broken proofs, but failed to fix some of them—once again, an expert giving up in the face of change.

The developer then made an additional 12 commits to address the change in `coq-contribs`, a regression suite of projects that the Coq standard library developers maintain as Coq versions change. Many

```
IZR (z:Z) : R :=
  match z with
  | Z0 => 0
  | Zpos n => INR (Pos.to_nat n)
  | Zneg n => - INR (Pos.to_nat n)
  end.
```

```
IZR (z:Z) : R :=
  match z with
  | Z0 => 0
  | Zpos n => IPR n
  | Zneg n => - IPR n
  end.
```

Figure 12: Old (left) and new (right) definitions of `IZR` in Coq. The old definition applies injection from naturals to reals and conversion of positives to naturals; the new definition applies injection from positives to reals. In contrast with most terms shown in this thesis, this term uses pattern matching and recursion rather than **primitive eliminators**.

of these changes were simple. For example, the developer wrote a lemma that describes the change:

```
Lemma INR_IPR : ∀p, INR (Pos.to_nat p) = IPR p.
```

The developer then used this lemma to fix broken proofs within the standard library. For example, one proof broke on this line:

```
rewrite Pos2Nat.inj_sub by trivial.✗
```

It succeeded with the lemma:

```
rewrite ← 3!INR_IPR, Pos2Nat.inj_sub by trivial.✓
```

These changes were outside-facing: proof engineers had to make similar changes to their own proofs when they updated client proof developments from Coq 8.6 to Coq 8.7. The Coq standard library developer could have updated some tactics to account for this, but it would have been impossible to account for every tactic that proof engineers could have used.

Furthermore, while the library developer responsible for the changes knew about the lemma `INR_IPR` describing the change, proof engineers of client proof developments did not. Proof engineers had to determine how the definition had changed themselves, and how to address the change in their broken proofs—perhaps by reading documentation or by talking to the Coq standard library developers.

This problem is what motivated the original vision for proof repair by example. The idea was to build a tool that could determine how the definition has changed for the proof engineer. It could then analyze changes in the standard library and in `coq-contribs` that resulted from the change in definition (in this case, rewriting by the lemma). It could extract a reusable patch from those changes, which it could automatically apply within broken user proofs. The proof engineer would never have to consider how the definition had changed.

The Pumpkin prototype presented in this chapter took the first steps toward realizing this vision.

3.2 APPROACH

Proof repair by example takes advantage of the fact that an example patch to a broken proof can carry enough information (like the rewrite by `INR_IPR`) to fix other proofs broken by the same change (like the broken client proofs).

To repair proofs by example with PUMPKIN (Section 3.2.1), the proof engineer modifies a single proof script to provide an *example patched proof*: an example of how to adapt a proof to a change. PUMPKIN extracts the information that example carries into a *patch candidate*: a function that describes the change in the example patched proof, but that is localized to the context of the example, and not yet enough to fix other proofs broken by the change. PUMPKIN then generalizes that candidate into a **reusable patch**: a function that can be used to fix other broken proofs broken by the same change, which PUMPKIN defines as a Gallina term.

The PUMPKIN prototype focuses on finding reusable patches to proofs in response to certain changes in the content of programs and specifications (Section 3.2.2). It does this using a combination of **semantic differencing** and **proof term transformations**: Differencing (Section 3.2.3) looks at the difference between versions of the example patched proof for this information, and finds the candidate. The proof term transformations (Section 3.2.4) then modify that candidate to produce the reusable proof patch. All of this happens over proof terms in Gallina, since tactics may hide necessary information.

In other words, looking back to the **thesis** statement, the information corresponding to changes in programs and specifications shows up in the difference between versions of the example patched proof. PUMPKIN can extract and generalize that information.

3.2.1 *Workflow: Repair by Example*

The interface to PUMPKIN is exposed to the proof engineer as a *command*. Commands in Coq are similar to **tactics**, except that they can occur outside of the context of proofs, and they can define new terms. Plugins like PUMPKIN can extend Coq with new commands. In this case, PUMPKIN extends Coq with a new command called `Patch Proof`, with the syntax:

```
Patch Proof old_proof new_proof as patch_name.
```

where `old_proof` and `new_proof` are the old and new versions of the **example patched proof**, and `patch_name` is the desired name of the **reusable proof patch**.[1] This invokes the PUMPKIN plugin, which searches for a reusable proof patch and defines it as a new term if

1 Section 3.5 describes an alternative interface for PUMPKIN with Git integration.

```
find_patch(old_proof, new_proof) :=
  diff types of old_proof and new_proof for goals
  diff terms old_proof and new_proof for candidates
  if there are candidates then
    transform candidates
    if there is a patch then return patch
  return failure
```

Figure 13: Search procedure for a reusable proof patch in PUMPKIN.

successful. All terms that PUMPKIN defines are type checked in the end, so PUMPKIN does not extend the **TCB**.

When the proof engineer calls `Patch Proof`, this invokes the proof patch search procedure in Figure 13. The search procedure starts by differencing the *types* of `old_proof` and `new_proof` (that is, the theorems they prove). The result that it finds is the *goal type*: the type that the reusable proof patch should have. It then differences the *terms* `old_proof` and `new_proof` directly to identify patch candidates, which are themselves proof terms. Finally, it transforms those patch candidates directly into a reusable patch. If it finds a reusable patch with the goal type, it succeeds and defines it as a term.

To demonstrate this workflow, consider the change from the theorem `old` to the slightly stronger theorem `new` in Figure 14. Changing `old` to `new` can break proofs that used to successfully apply `old`:

```
apply old.✓
```

so that they fail after migrating to `new`:

```
apply new.✗
```

When we call:

```
Patch Proof old new as patch.
```

PUMPKIN invokes the search procedure, which differences `old` and `new` to infer the **goal type** for the **patch**. Here, it infers the following goal:

```
∀ (n m p : nat),
  n <= m →
  m <= p →
  n <= p →
  n <= p + 1
```

which maps from the conclusion of `new` back to the conclusion of `old` in a common context. It then differences the terms `old` and `new` to identify **candidate proof patches** (Section 3.2.3), then transforms those candidates to a **reusable proof patch** with that type (Section 3.2.4), which it defines as a new term `patch`. This is something that we can use to fix other proofs broken by this change, either by applying it with traditional **proof automation**:

```
apply patch. apply new.✓
```

or by using the automation in Section 3.5.

```
1  Theorem old:
2    ∀ (n m p : nat),
3      n <= m →
4      m <= p →
5      n <= p + 1.
6  Proof.
7    intros. induction H0.
8    - auto with arith.
9    - constructor. auto.
10 Qed.
11
12 old (n m p : nat)
13   (H : n <= m)
14   (H0 : m <= p)
15 :=
16   le_ind
17     m
18     (fun p0 => n <= p0 + 1)
19     (le_plus_trans n m 1 H)
20     (fun (m0 : nat) _
21       (IHle : n <= m0 + 1) =>
22         le_S n (m0 + 1) IHle)
23     p
24     H0.
```

```
1  Theorem new:
2    ∀ (n m p : nat),
3      n <= m →
4      m <= p →
5      n <= p.
6  Proof.
7    intros. induction H0.
8    - auto with arith.
9    - constructor. auto.
10 Qed.
11
12 new (n m p : nat)
13   (H : n <= m)
14   (H0 : m <= p)
15 :=
16   le_ind
17     m
18     (fun p0 => n <= p0)
19     H
20     (fun (m0 : nat) _
21       (IHle : n <= m0) =>
22         le_S n m0 IHle)
23     p
24     H0.
```

Figure 14: Two proofs with different conclusions (top) and the corresponding proof terms (bottom). Highlighted lines correspond to the change in theorem conclusion (top) and the difference in terms that correspond to a patch (bottom).

3.2.2 *Scope: Changes in Content*

The search procedure in Figure 13 searches for patches to proofs broken by changes in the *content* of programs and specifications. For example, PUMPKIN can support the change in Figure 14, since some of the content (the conclusion of the theorem) changes, but the structure remains identical. In general, the PUMPKIN prototype supports only changes that do not add, remove, or rearrange any hypotheses.

The search procedure can be instantiated to different classes of change in the content of programs and specifications. Thus, before running the search procedure, PUMPKIN infers an *instance* of the search procedure from the example change.[2] This instance customizes the highlighted lines for an entire class of changes: it determines what to diff on lines 1 and 2, and what transformations to run to achieve what goal on line 4.

Figure 14 used the instance for a change in the conclusion of a theorem. Given two such proofs of theorems:

```
∀ x, H x → P x
∀ x, H x → Q x
```

for any `x`, `H`, `P`, and `Q`, PUMPKIN searches for a patch with goal type:

```
∀ x, H x → Q x → P x
```

In total, the PUMPKIN prototype currently implements six instances of the search procedure. Section 3.5.1.1 explains these instances, and Section 3.6 demonstrates some of them on real proof developments.

3.2.3 *Differencing: Candidates from Examples*

Differencing operates over terms and types. Differencing tactics would be insufficient, since tactics and hints may mask information helpful to finding patches. For example, for the change in Figure 14, the tactics in the proofs of `old` and `new` are identical, even though the proof terms they compile down to are not. This is why differencing looks at the change in terms to extract the patch candidates.

In the end, differencing identifies the semantic difference between the old and new versions of the proof terms for the example patched proof. At a high level, the semantic difference is the difference between two terms that corresponds to the difference between their types (see Section 3.3). The details of the semantic difference and where differencing looks to find it vary by instance of the search procedure.

Consider a simplified version of the example in Figure 14, looking only at the base case (line 19):

```
old_proof := le_plus_trans n m 1 H : n <= m + 1.
new_proof := H : n <= m.
```

2 In the original 2018 PUMPKIN PATCH paper, I called this a *configuration*. In this chapter, I rename configuration to **instance** everywhere to avoid overloading this term, since I unfortunately used this word again for something different later.

For this change, PUMPKIN uses the instance for changes in conclusions:

```
1: diff theorem conclusions of old_proof and new_proof for goals
2: diff function bodies of old_proof and new_proof for candidates
3: if there are candidates then
4:     transform candidates
```

When this instance of the search procedure is invoked, semantic differencing first identifies the difference in their types, here the respective **motives** (line 18):

```
(fun p0 => n <= p0 + 1)
(fun p0 => n <= p0)
```

applied to `m` (line 17). This produces the **candidate goal type**:

```
n <= m → n <= m + 1
```

Differencing then diffs the terms for a function that has that type, here (line 19):

```
fun (H : n <= m) => le_plus_trans n m 1 H
```

This is a **patch candidate**. This candidate is close, but it is not yet a **reusable patch**. In particular, this candidate maps base case to base case (it is applied to `m`); the patch should map conclusion to conclusion (it should be applied to `p`). This is where the proof term transformations will come in.

SUMMARY In summary, differencing has the following specification:

- **Inputs**: the **example patched proof** given by `old_proof` and `new_proof`, a set of options corresponding to the **instance** of the search procedure `instance`, and a final **goal type** `goal`, assuming:
  - the change from `old_proof` to `new_proof` is in the class of changes supported by `instance`.
- **Outputs**: a list of terms `candidates` of **patch candidates**, and a **candidate goal type** `candidate_goal`, guaranteeing:
  - each term in `candidates` has type `candidate_goal`.

PUMPKIN infers the instance of the procedure and the candidate goal type from the change in the example patched proof, so the proof engineer does not have to provide this information. PUMPKIN could in theory infer the wrong instance or the wrong candidate goal type, but this would not sacrifice soundness—it would mean only that the patch procedure would either fail to produce a patch, or produce a patch that is not useful. All terms that PUMPKIN produces type check.

3.2.4 *Transformations: Patches from Candidates*

Differencing produces **patch candidates** that are localized to a particular context according to the inferred goal for that change, but do

not yet generalize to other contexts. The **transformations** take each candidate and try to modify it to produce a term that *does* generalize. If they succeed, PUMPKIN has found a **reusable patch**.

Consider once more the example in Figure 14. The candidate patch that differencing found has this type:

```
candidate :=
  fun (H : n <= m) => le_plus_trans n m 1 H
: n <= m →
  n <= m + 1.
```

in an environment with fixed `n` and `m`. The reusable patch that PUMPKIN is looking for, however, should have this type:

```
∀ n m p,
  n <= m →
  m <= p →
  n <= p →
  n <= p + 1.
```

as this is the **goal** that PUMPKIN inferred for this **instance**. The transformations that PUMPKIN runs will attempt to transform the candidate into a patch with that type.

The details of which transformations to run vary by instance. There are four transformations that turn candidates into reusable patches:

1. *patch specialization* to arguments,
2. *patch generalization*[3] of arguments or functions,
3. *patch inversion* to reverse a patch, and
4. *lemma factoring* to break a term into parts.

Each instance chooses among these transformations strategically based on the structure of the proof term.

For Figure 14, we can instantiate *transform* with two transformations:

```
1: diff conclusions of the theorems of old_proof and new_proof for goals
2: diff bodies of the proof terms for candidates
3: if there are candidates then
4:     generalize and then specialize candidates
```

That is, first, PUMPKIN generalizes the candidate by `m` (line 17), which lifts it out of the base case:

```
fun n0 n m p H0 H1 =>
  (fun (H : n <= n0) => le_plus_trans n n0 1 H)
: ∀ n0 n m p,
    n <= m →
    m <= p →
    n <= n0 →
    n <= n0 + 1.
```

3 In the original paper, I called this *patch abstraction*. But I later learned that **generalization** has a technical meaning in the automated theorem proving world, and that the technical meaning coincides beautifully with the technical meaning in the world of proof assistants. So I renamed it for this thesis.

PUMPKIN then specializes this generalized candidate to `p` (line 23), the argument to the conclusion of `le_ind`. This produces a patch:

```
patch n m p H0 H1 :=
  (fun (H : n <= p) => le_plus_trans n p 1 H)
: ∀ n m p,
    n <= m →
    m <= p →
    n <= p →
    n <= p + 1.
```

which has the goal type, so PUMPKIN is done.

This simple example uses only two transformations. The other transformations help turn candidates into patches in similar ways, all guided by the structure of the proof term. I will describe these transformations more in Section 3.4.

SUMMARY In summary, the transformations together have the following specification:

- **Inputs**: the inputs and outputs of differencing, assuming:
  - the assumptions and guarantees from differencing hold.
- **Outputs**: a term `patch` that is the **reusable proof patch**, guaranteeing:
  - `patch` has the inferred final **goal type** `goal` for the change.

When these transformations fail, or when the list of candidates that differencing returns is empty, PUMPKIN simply fails to return a patch. As with differencing, it is possible that a mistake in the implementation leads to a final goal type is not useful to the proof engineer, but this cannot sacrifice soundness: every patch PUMPKIN produces type checks with the goal type in the end.

3.3 DIFFERENCING

Differencing is aware of and guided by the semantics of Coq's rich proof term language **Gallina**—that is what makes it a **semantic differencing** algorithm. This means that differencing can take advantage of the structure and information carried in every proof term, thanks to Gallina's rich type theory **CIC$_\omega$**. The rich structure of terms helps guide differencing for each instance of the search procedure, while the rich information in their types helps ensure correctness in the end.

Consider once again the example from Figure 14, but this time not just the base case. Both versions of the proof are inductive proofs using the same **eliminator**, with slightly different **motives**. Accordingly, differencing knows that there are two places to look for candidates, namely the base case (line 19) and the inductive case (lines 20-22). Differencing breaks each inductive proof into these cases, then recursively calls itself for each case. In the base case, it finds the candidate

from Section 3.2.3. Since this candidate has the candidate goal type (here, the goal type specialized to the base case), differencing knows it has successfully found a candidate.

The rich type information proof terms carry helps prevent exploration of syntactic differences that are not meaningful. For example, in the inductive case of the proof term from Figure 14, the **inductive hypothesis** `IHle` (line 21) changes:

```
... (IHle : n <= m0 + 1) ...
... (IHle : n <= m0) ...
```

Notably, though, the type of `IHle` changes for *any* two inductive proofs over `le` with different conclusions. A syntactic differencing component may identify this change as a candidate. My semantic differencing algorithms know that they can ignore this change. This section describes the design of these algorithms. Section 3.5.1.2 describes the implementation in PUMPKIN.

Differencing recurses over the structure of two terms t_a and t_b in a common environment Γ. When it recurses, it extends Γ with common assumptions, then differences subterms. In each case, it carries a goal type G, and returns a list of patch candidates $\vec{t}$ that each have that goal type. That is, we can view it as a judgment $\Gamma \vdash (t_a, t_b, G) \Downarrow_d \vec{t}$, where in the end, for every t in $\vec{t}$, $\Gamma \vdash t : G$. The details of this vary by subterm-instance combination.

IDENTITY The simplest patch is the identity patch. When two terms are **definitionally equal**, differencing infers that the goal is identity, and returns a singleton list containing only the identity function instantiated to the appropriate type.

APPLICATION When one proof term is a function application, for example:

$$\Gamma \vdash (f\ t_a,\ t_b,\ G) \Downarrow_d \vec{t}$$

differencing checks to see if t_b is in $f\ t_a$. That is, it searches for a subterm of $f\ t_a$ that is definitionally equal to t_b. This is how differencing can identify the candidate for the base case of Figure 14 (line 19). It is also a core building block that other differencing heuristics rely on.

When both proof terms are function applications:

$$\Gamma \vdash (f_a\ t_a,\ f_b\ t_b,\ G) \Downarrow_d \vec{t}$$

and the previous heuristic fails, differencing may recurse into both the functions and the arguments, search for patches, and then compose the results. How to compose those results varies by instance of the search procedure.

FUNCTIONS The treatment of functions depends on whether a hypothesis or a conclusion has changed. When recursing into the body of two functions, each with a hypothesis of the same type:

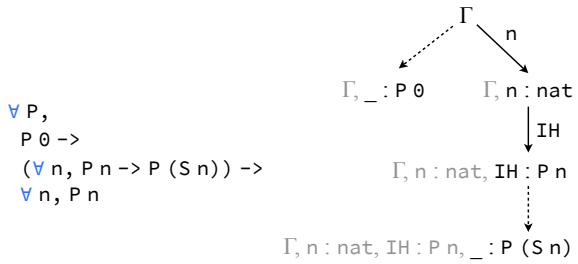


Figure 15: The type of (left) and tree for (right) the eliminator of `nat`. The solid edges represent hypotheses, and the dotted edges represent the proof obligations for each case in an inductive proof.

$$\Gamma \vdash (\lambda(t_a : T).b_a,\ \lambda(t_b : T).b_b,\ G) \Downarrow_d \vec{t}$$

differencing assumes that the conclusion has changed. That is, it assumes that t_a and t_b are the same, adds one of them to a common environment, and differences the body:

$$\Gamma,\ t_a : T \vdash (b_a,\ b_b[t_a/t_b]) \Downarrow_d \vec{b}$$

It then filters those candidates $\vec{b}$ to only those with an adjusted goal type $G\ t_a$, then wraps each candidate b in $\vec{b}$ in a function in the end:

$$\lambda(t_a : T).b$$

with type G.

When a hypothesis type has changed:

$$\Gamma \vdash (\lambda(t_a : T_a).b_a,\ \lambda(t_b : T_b).b_b,\ G) \Downarrow_d \vec{t}$$

differencing acts similarly, but it substitutes the changed hypothesis type in the body in order to recurse into a well-typed environment. It also has some additional logic to remove hypotheses that need not show up in the goal type.

ELIMINATORS Recall that **inductive types** in CIC$_\omega$ come equipped with **primitive eliminators**. The semantic differencing algorithms views inductive types as *trees* that represent these eliminators. In these trees, every node is a type context, and every edge is an extension to that type context with a new term. Correspondingly, type differencing (to identify goal types) compares nodes, and term differencing (to find candidates) compares edges.

The key benefit to this model is that it provides a natural way to express inductive proofs, so that differencing can efficiently identify candidates. Consider, for example, searching for a patch between conclusions of two inductive proofs of theorems about the natural numbers:

$$\text{Elim}(\text{nat},\ P)\ \{f_0,\ f_S\}$$
$$\text{Elim}(\text{nat},\ Q)\ \{g_0,\ g_S\}$$

with goal type:

$$Q \to P$$

Differencing looks in both the base case and in the inductive case for candidates. In each case, differencing diffs the terms in the dotted edges of the tree for the eliminator of `nat` (Figure 15) to try to find a term that maps between conclusions of that case:

$$\Gamma \vdash (f_0,\ g_0,\ Q\ 0 \to P\ 0) \Downarrow_d \vec{t_0}$$
$$\Gamma \vdash (f_S,\ g_S,\ \Pi(n : \text{nat}).Q\ (S\ n) \to P\ (S\ n)) \Downarrow_d \vec{t_S}$$

where `O` is shorthand for `Constr (0, nat)`, and `S` is shorthand for `Constr (1, nat)`. In the inductive case, differencing also knows that the change in the type of the **inductive hypothesis** is not semantically relevant (it occurs for any change in the inductive **motive**). Furthermore, it knows that the inductive hypothesis cannot show up in the patch itself, since the goal type does not reference the inductive hypothesis, so it attempts to remove any occurrences of the inductive hypothesis in any candidate.

When differencing finds a candidate, it knows Q and P as well as the arguments 0 or $S\ n$. This makes it simple for PUMPKIN to later query the transformations for the final patch, with type $Q \to P$.

3.4 TRANSFORMATION

The **proof term transformations** together transform a **patch candidate** into a **reusable proof patch**. At a high level, these transformations adapt the candidate to the context of the goal type that PUMPKIN infers. As with differencing, the transformations are aware of and guided by the semantics of Gallina's type theory CIC$_\omega$. This section describes the design of these transformations. Section 3.5.1.3 describes the implementation.

The transformations together recurse over the structure of each term in the list of candidates $\vec{t}$ in an environment Γ, and adapt that candidate to some new context in a goal-directed manner. In the end, if successful, they produce a reusable proof patch p with type G, where G is the inferred goal type. That is, we can view the high-level composition of transformations as a single judgment $\Gamma \vdash (\vec{t},\ G) \Downarrow_t p$, where in the end, $\Gamma \vdash p : G$. The details vary by transformation, and the details of which transformations run at all and in what order to reach the goal vary by the instance of the search procedure.

SPECIALIZATION Sometimes, candidates are too general. **Specialization** takes a candidate that is too general, and specializes it to arguments as determined by the difference in terms. To find a patch for Figure 14, for example, PUMPKIN specialized the candidate to `p`.

Specialization takes a single patch candidate, some arguments, and a reduction strategy, and returns a new candidate. It first applies the

function to the argument, then applies the reduction strategy on the result. The default reducer, for example, uses $\beta\iota$-reduction in Coq—two of the **definitional equality** reductions.[4] Section 3.5.1.3 describes other reducers. The only requirement for a reducer is that the end result should be definitionally equal to the original.

Depending on the procedure instance and the step in the process, the transformed candidate may be the reusable patch, or it may just be an intermediate candidate. It is the job of the patch finding procedure to provide both the candidate and the arguments, and to determine which transformation to run next, if applicable.

GENERALIZATION In other cases, a patch candidate is too specific. **Generalization** takes a candidate that is too specific and generalizes it. We saw this for the example in Figure 14 as well: to go from the candidate that PUMPKIN found in the base case to the eventual reusable patch, PUMPKIN generalized the candidate by `m` (before applying **specialization**).

There are two kinds of generalization. The first generalizes candidates that map between types that share a common argument, like:

$$Q\ t\ \rightarrow P\ t$$

by the common argument:

$$\Pi(t' : T),\ Q\ t' \rightarrow P\ t'$$

where T is the type of t. The second generalizes candidates that map between types that share a common function, like:

$$P\ t'\ \rightarrow P\ t$$

by the common function:

$$\Pi\ (Q : T),\ Q\ t' \rightarrow Q\ t$$

where T is the type of Q.

Generalization takes a patch candidate, the goal type, and the function arguments or function by which to generalize. It first wraps the candidate inside of a lambda from the type of the term by which to generalize. Then, it substitutes terms inside the body with the generalized term. It continues to do this until there is nothing left to generalize, then filters results by the goal type.

Consider, for example, generalizing the candidate from Figure 14 by m (represented in CIC$_\omega$ syntax this time, but using shorthand for Gallina constants that were defined in the example):

```
λ (H : le n m) . le_plus_trans n m (S O) H
: le n m →
  le n (plus m (S O)).
```

where `le` is an inductive type, and `le_plus_trans` and `plus` are both functions in the current context. The first step wraps this in a lambda from some `nat`, the type of `m`:

4 ι-reduction is basically β-reduction for inductive types.

```
λ (n₀ : nat) (H : le n m) . le_plus_trans n m (S O) H
: Π (n₀ :  nat),
    le n m →
    le n (plus m (S O)).
```

The second step substitutes `n₀` for `m`:

```
λ (n₀ : nat) (H : le n n₀) . le_plus_trans n n₀ (S O) H
: Π (n₀ : nat),
    le n n₀ →
    le n (plus n₀ (S O)).
```

In general, generalization is undecidable, as the terms and types may be reduced, so that the common function or argument does not appear explicitly. That is, generalization fundamentally relies on a kind of unification. This poses a challenge for generalization in the second step—substitution.

To handle this challenge, generalization uses a list of *substitution strategies*[5] to determine what subterms to substitute. In this case, the simplest strategy works: the tool replaces all terms that are convertible to the concrete argument `m` with the generalized argument `n0`, which produces a single candidate. Type checking this candidate confirms that it is a patch. In some cases, the simplest strategy is not sufficient, even when it is possible to generalize the term. Section 3.5.1.3 describes a sample of other strategies.

It is the job of the patch finding procedure to provide the candidate and the terms by which to generalize. In addition, the implementation of each search procedure instance includes a list of strategies. The instance for changes in conclusions, for example, starts with the simplest strategy, and moves on to more complex strategies only if that strategy fails. This design makes generalization simple to extend with new strategies and simple to call with different strategies for different instances, or even as an optimization for the proof engineer.

INVERSION Sometimes, when two types are **propositionally** equal, candidate patches may appear in the wrong direction. For example, consider two list lemmas:[6]

```
old : ∀ {T} l' l, length (l' ++ l) = length l' + length l.
new : ∀
     {T} l' l, length (l' ++ l) = length l' + length (rev l).
```

If PUMPKIN searches the difference in proofs of these lemmas for a patch from the conclusion of `new` to the conclusion of `old`, it may find a candidate *backwards*:

```
candidate {T} l' l (H : old l' l) :=
  eq_rect_r ... (rev_length l)
: ∀ {T : Type} (l' l : list T),
    old l' l → new l' l.
```

5 I originally called these *abstraction strategies*, but this is more accurate.

6 It is too difficult to port this entire example, including the proof, to CIC$_\omega$ for demonstration. So I leave it in Gallina and Ltac, but still use it to give intuition for inversion.

The transformation can **invert** this to get the patch:

```
patch {T} l' l (H : new l' l) :=
  eq_rect_r ... (eq_sym (rev_length l))
: ∀ {T : Type} (l' l : list T),
    new l' l → old l' l.
```

We can then use this patch to port proofs. For example, if we add this patch to a hint database [1], we can port this proof:

```
Theorem app_rev_len {T : Type} : ∀ (l l' : list T),
  length (rev (l' ++ l)) = length (rev l) + length (rev l').
Proof.
  intros. rewrite rev_app_distr. apply old.✓
Defined.
```

to this proof:

```
Theorem app_rev_len {T : Type} : ∀ (l l' : list T),
  length (rev (l' ++ l)) = length (rev l) + length (rev l').
Proof.
  intros. rewrite rev_app_distr. apply new.✓
Defined.
```

Rewrites like `candidate` are *invertible*: we can invert any rewrite in one direction by rewriting in the opposite direction. In contrast, it is not possible to invert the patch PUMPKIN found for Figure 14.

When a candidate is invertible, patch inversion exploits symmetry to try to reverse the conclusions of a candidate patch. It first factors the candidate using the factoring transformation, then calls the primitive inversion function on each factor, then finally folds the resulting list in reverse. The primitive inversion function exploits symmetry. For example, propositional equality is symmetric, so the transformation can invert any application of the equality eliminators. I will explain this more in Section 3.5.1.3.

FACTORING The other transformations sometimes need help breaking a large function into smaller subterms. This can break other problems, like generalization, into smaller subproblems. It is also necessary to invert certain terms, since the inverse of:

$$g \circ f \ : \ X \to Z$$

where:

$$g : Y \to Z$$
$$f : X \to Y$$

for arbitrary non-dependent types X, Y, and Z, is of course:

$$f^{-1} \circ g^{-1} \ : \ Z \to X$$

This shows up often for inverting sequences of rewrites, as I will show in Section 3.5.1.3. To invert a term like this, PUMPKIN identifies the factors `[f; g]`, inverts each factor to `[f`$^{-1}$`; g`$^{-1}$`]`, then folds and applies the inverse factors in the opposite direction.

The **factoring** transformation looks within a term for its factors. For the term above, it returns both factors: `f` and `g`. In this case, factoring

takes the composite term and `X` as arguments. It first searches as deep as possible for a term of type `X` → `Y` for some `Y`. If it finds such a term, then it recursively searches for a term with type `Y` → `Z`. It maintains all possible paths of factors along the way, discarding any paths that cannot reach `Z`. It does not yet support paths where `Y` depends on `X`.

Factoring will necessarily sometimes fail, as the general problem of finding all possible nontrivial factors of a function is undecidable: there are always infinitely many factors if factors can compose to identity. Determining all possible *nontrivial* factors—those that contain no subsets of factors that compose to identity—reduces to detecting whether an arbitrary function is extensionally equal to the identity function at a fixed but arbitrary type, which is itself undecidable.[7] Factoring, like generalization, relies on a kind of unification to attempt this problem in spite of undecidability.

3.5 IMPLEMENTATION

The source code of the PUMPKIN PATCH proof repair plugin suite is on Github.[8] It includes the source code of the PUMPKIN prototype plugin, extended with a number of new features. The thesis release supports Coq 8.8, with Coq 8.9.1 support in a branch.

This section describes the implementation of the PUMPKIN prototype (Section 3.5.1), along with some new features that go beyond the original prototype and help with workflow integration (Section 3.5.2), plus an evaluation of the boundaries of the PUMPKIN prototype that still stand (Section 3.5.3). The interested reader can follow along in the repository.

3.5.1 *Tool Details*

The implementation (Section 3.5.1.1) of the procedure from Figure 13 in Section 3.2.1 starts with a preprocessing step which compiles the proof terms to trees (like the tree in Figure 15).[9] The implementation always maintains pointers to easily switch between the tree and AST

7 **Edward Z. Yang** wrote a cute proof of this on Twitter: Let the function be an arbitrary Turing complete program parameterized by fuel, returning the number of steps taken when it does not terminate within that number of steps, or otherwise returning zero when it does terminate. Then determining whether or not the function is extensionally equal to identity reduces to determining whether or not the program terminates.

8 Latest version: `http://github.com/uwplse/PUMPKIN-PATCH`.
Stable thesis release: `https://github.com/uwplse/PUMPKIN-PATCH/tree/v1.0`.
Coq 8.9.1 branch: `https://github.com/uwplse/PUMPKIN-PATCH/tree/8.9.1`.
2018 release: `https://github.com/uwplse/PUMPKIN-PATCH/releases/tag/cpp18`.

9 Representing proof terms directly as trees in the implementation rather than just in the theory is one of my biggest regrets three years later. This representation is useful for understanding how differencing works over inductive types, but implementing it adds clutter that makes PUMPKIN difficult to maintain. My later work represents proof terms as terms, and avoids this representation.

representations of the terms. Differencing (Section 3.5.1.2) operates over trees, while the transformations (Section 3.5.1.3) operate directly over the terms those trees represent. PUMPKIN has no impact on the **TCB** (Section 3.5.1.4).

3.5.1.1 *The Procedure*

The PUMPKIN prototype exposes the patch finding procedure (`patcher.ml4`) to users through the Coq command `Patch Proof`. After compiling to trees (`evaluation.ml`), PUMPKIN automatically infers which **instance** of the search procedure to use from the example change.

Internally, PUMPKIN represents search procedure instances as sets of options, which it passes to the procedure. The procedure uses these options to determine how to compose components (for example, whether to generalize candidates) and how to customize components (for example, whether semantic differencing should look for an intermediate lemma). In total, the PUMPKIN prototype currently implements six instances. The first five correspond to changes in:

1. conclusions of theorems,
2. hypotheses of theorems,
3. dependent arguments to constructors of inductive types,
4. conclusions of constructors of inductive types, and
5. cases of fixpoints.

The final instance is useful for proof optimization (Section 3.5.2). The support for these changes is limited in expressiveness and power; more information on limitations in scope can be found in the repository. Extending PUMPKIN with a new instance of the search procedure amounts to extending key functions in the implementation with a case corresponding to the new instance.

3.5.1.2 *Differencing*

As noted in Section 3.3, **differencing** (`differencing.ml`) operates over trees. Differencing uses the structure of these trees to prioritize semantically relevant differences. At the lowest level, it calls a primitive differencing function which checks if it can substitute one term within another term to find a function between their types.

The differencing component is *lazy*: it compiles terms into trees one step at a time. It then *expands* each tree as needed to find candidates (`expansion.ml`). For example, differencing two functions for a patch between conclusions:

```
fun (t : T) => b
fun (t' : T) => b'
```

Differencing introduces a single term of type `T` to a common environment, then expands and recursively diffs the bodies `b` and `b'` in that environment.

3.5.1.3 *Transformation*

Pumpkin implements the four **transformations** from Section 3.4: **specialization**, **generalization**, **inversion**, and **factoring**. These transformations operate directly over terms in Gallina. The Pumpkin procedure chooses among them by search procedure instance, and in the end checks the type of the patch to ensure that is has the goal type. Pumpkin also determines what arguments to pass to each transformation based on the instance. As a bonus, Pumpkin exposes commands that correspond to each of these transformations (`patcher.ml4`), so that proof engineers can call them outside of the proof patching procedure.

SPECIALIZATION **Specialization** (`specialize.ml`) takes a patch candidate and some arguments, all of which are Gallina terms. It also takes a custom reducer (`reducers.ml`) as a higher-order function that reduces a Coq term. It applies the candidate function to the arguments, then reduces the result using the supplied reducer.

The default specializer reduces the result using Coq's `Reduction.nf_betaiotazeta` function. Other reducers include one that does not reduce at all, one that removes unnecessary applications of the identity function, one that does weak head reduction, one that completely normalizes terms, and one that δ-reduces to unwrap constants. There are also higher-order combinators for reducers, including chain reduction with errors, chain reduction without errors, and reduction over the types of supplied terms. This makes it possible for search procedure instances to highly customize specialization. I expose these reducers and combinators in the Coq plugin library.

GENERALIZATION **Generalization** (`abstraction.ml`) takes a patch candidate, the goal type, and the arguments or function by which to generalize, all as Gallina terms. It also takes a list of substitution strategies to determine what subterms to substitute, and how. After generalizing the candidate to a lambda term, it substitutes subterms inside of the body with the generalized argument using the supplied list of substitution strategies, in order.

The simplest strategy replaces all terms convertible to a particular concrete argument with the supplied generalized argument. In some cases, this strategy is not sufficient. It may be possible to produce a patch only by generalizing *some* of the subterms convertible to the argument or function (see Section 3.5.3.2), or the term may not contain any subterms convertible to the argument or function at all.

I implement several strategies to account for this. The combinations strategy, for example, tries all combinations of substituting only

some of the convertible subterms with the generalized argument. The pattern-based strategy substitutes subterms that match a certain pattern with a term that corresponds to that pattern. Some strategies reduce before generalizing, and some do not. I expose these strategies in the PUMPKIN OCaml API (`abstracters.mli`).

INVERSION **Inversion** (`inverting.ml`) takes as input a patch candidate as a Gallina proof term, and tries to reverse the conclusion of its type. It works by first factoring the candidate, then exploiting symmetry properties to invert those factors, then finally composing the inverted factors in the opposite order.

For example, recall that the equality eliminator in Gallina is `eq_rect`. The `rewrite` tactic in Ltac often compiles down to an application of `eq_rect`. Since equality is symmetric, the Coq standard library also comes equipped with an inverse function `eq_rect_r`, related to `eq_rect` by symmetry of equality:

```
eq_rect_r A x P (H : P x) y (H0 : y = x) :=
  eq_rect x (fun y0 : A => P y0) H y (eq_sym H0)
```

When inversion encounters an `eq_rect` in one of its factors, it reverses it by applying symmetry of equality, effectively producing an application of `eq_rect_r`. The opposite direction works similarly.

If inversion does not recognize any type symmetry properties it can exploit in a factor, it strategically swaps subterms in the factor and type checks the result to see if it is an inverse. This essentially amounts to an ad hoc attempt to discover symmetry properties.

FACTORING **Factoring** (`factoring.ml`) takes as input a Gallina term, and attempts to break it into factors. When it succeeds, it returns the factors as a list of terms; otherwise, it returns the empty list.

Factoring works by searching with a term for factors. Consider factoring a sequence of rewrites:

```
t (a b c d: nat) (H: a = b) (H0: b = c) (H1: c = d) : a = d :=
  eq_rect_r
    (fun (a0 : nat) => a0 = d)
    (eq_rect_r (fun (b0 : nat) => b0 = d) H1 H0)
    H.
```

into two independent rewrites:

```
f (a b c d: nat) (H: a = b) (H0: b = c) (H1: c = d) : b = d :=
  eq_rect_r (fun (b0 : nat) => b0 = d) H1 H0.

g (a b c d: nat) (H: a = b) (H0: b = c) (H1: b = d) : a = d :=
  eq_rect_r (fun (a0 : nat) => a0 = d) H1 H.

t (a b c d: nat) (H: a = b) (H0: b = c) (H1: c = d) : a = d :=
  g a b c d H H0 (f a b c d H H0 H1).
```

To discover `f` and `g`, factoring starts by assuming all of the hypotheses of `t`, then searching as deep as possible within the conclusion of `t`

for a term of type `Y` for some `Y` (here, the conclusion of `f`, with type `b = d`). It then assumes `Y`, and recursively factors the term it gets from substituting in that hypothesis for `f` (here, it assumes `b = d`, and substitutes to derive the term `g`). It repeats this until it is able to reach the conclusion type (here `a = d`, the type of the conclusion of `g`), at which point it has found the only possible path of factors, and it is done. It returns these factors as Gallina terms.

3.5.1.4 *Trusted Computing Base*

A common concern for proof developments is an increase in the **TCB**. PUMPKIN takes this into consideration. In particular, PUMPKIN is implemented as a Coq **plugin**, and Coq type checks all terms that plugins produce. Since PUMPKIN does not modify the type checker, it cannot produce an ill typed term. PUMPKIN also does not add any axioms, and so does not increase the TCB.

3.5.2 *Extensions*

Since releasing the PUMPKIN prototype, I have extended it with many features for better integration into **proof engineering** workflows. This section summarizes three early extensions: Git integration, preliminary support for applying patches, and proof optimization.

GIT INTEGRATION PUMPKIN PATCH exposes a Git interface to PUMPKIN called PUMPKIN-git [139]. PUMPKIN-git makes it possible to call PUMPKIN's `Patch Proof` command by command line over Git commits. To call PUMPKIN-git, the proof engineer simply runs the (command line) command:

```
pumpkin-git example_proof file.v -rev rev
```

This searches for a patch corresponding to the change in `example_proof` in `file.v` compared to the revision `rev` of the local repository. It will then prompt the proof engineer with the patch it finds, and either overwrite the file (with consent) or otherwise save the results to a temporary file. There are many options to control the behavior of PUMPKIN-git, all of which can be found in the repository.

PATCH APPLICATION For the PUMPKIN prototype, the differencing algorithm and proof term transformations extract and generalize information from example patched proofs in the form of reusable patch, but do not yet help apply those patches automatically. Since implementing the prototype, I have extended PUMPKIN-git with preliminary support for patch application via hint generation. The interface is:

```
pumpkin-git example_proof_id file.v -rev rev -hint
```

This places the generated patch in a hint database in Coq. Coq applies hints in its hint databases automatically, in some cases taking care of

the changes at the proof script level that the proof engineer would have to make to use these patches.

PROOF OPTIMIZATION Five of the implemented search procedure instances correspond to changes, but the sixth is special: it corresponds to the *absence* of change. This makes it possible to reuse the PUMPKIN infrastructure to optimize proofs to automatically remove extra calls to induction. See `Optimization.v` for more information.

3.5.3 *Testing Boundaries*

In this section, I explore the boundary between what the semantic differencing and transformation implementations in the PUMPKIN prototype can and cannot handle. It is precisely this boundary that informs how to improve the implementations.

To evaluate this boundary, I tested the PUMPKIN prototype on a suite of 50 pairs of proofs (Section 3.5.3.1). I designed 11 of these pairs to succeed, then modified their proofs to produce the remaining 39 pairs that try to stress the core functionality of the tool. I learned the following from the pairs that tested PUMPKIN's limitations:

1. **The failed pairs drive improvements.**
   PUMPKIN failed on 17 of 50 pairs. These pairs inform how to improve differencing and transformations in the future. (Section 3.5.3.2)

2. **The pairs reveal potential substitution strategies.**
   PUMPKIN produced an exponential number of candidates in 5 of 50 pairs. New substitution strategies would dramatically reduce the number of candidates. (Section 3.5.3.2)

3. **PUMPKIN was fast, but it could be even faster.**
   The slowest successful patch took 48 ms. The slowest failure took 7 ms. Simple changes could make PUMPKIN more efficient. (Section 3.5.3.3)

3.5.3.1 *Patch Generation Suite*

I wrote a suite[10] of 50 pairs of proofs, proving a total of 11 pairs of theorems. I wrote these proofs myself since there was no existing benchmark suite to work with. I used the following methodology:

1. Choose theorems `old` and `new`.

2. Write similar inductive proofs of `old` and `new`.

3. Modify the proof of `old` to produce more pairs.

10 `http://github.com/uwplse/PUMPKIN-PATCH/blob/cpp18/plugin/coq/Variants.v`

```
fun n m p (H : n <= m) (H0 : m <= p) =>
  le_S n p ... (* proof of stronger lemma *)
: ∀ n m p,
    n <= m →
    m <= p →
    n <= S p.

fun n m p (H : n <= m) (H0 : m <= p) =>
  le_plus_trans n p 1 ... (* proof of stronger lemma *)
: ∀ n m p,
    n <= m →
    m <= p →
    n <= p + 1.
```

Figure 16: Two proof terms `old` (top) and `new` (bottom) that contain the same proof of a stronger lemma.

4. Search for patches from `new` to `old`.

5. If possible, search for patches from `old` to `new`.

My goal was to determine what changes to proofs stress the components and how to use that information to drive improvements. I focused on differences in conclusions, the most supported **instance** of the search procedure at the time. Since PUMPKIN operates over terms, I removed redundant proof terms, even if they were produced by different tactics. I controlled the first pair of proofs of each pair of theorems for features I had not yet implemented at the time, like nested induction, changes in hypotheses, and generalizing `omega` terms. These features sometimes showed up in later proofs (for example, after moving a rewrite); I kept these proofs in the suite, since isolated changes to supported proofs that introduce unsupported features can inform future improvements.

3.5.3.2 *Three Challenges*

PUMPKIN found patches for 33 of the 50 pairs. 28 of the 33 successes did not stress PUMPKIN at all: PUMPKIN found the correct candidate immediately and was able to generalize it in one try. The pairs that PUMPKIN failed to patch and the successful pairs that stressed generalization reveal key information about how to improve differencing and the transformations. I walk through three examples below.

A CHALLENGE FOR DIFFERENCING For one pair of proofs of theorems with propositionally equal conclusions (Figure 16), **differencing** failed to find candidates in either direction. These proofs both contain the same proof of a stronger lemma; PUMPKIN found patches from this lemma to both `old` and `new`, but it was unable to find a patch between `old` and `new`. A patch may show up deep in the difference between

`le_plus_trans` and `le_S`, but even if we δ-reduce (unfold the definition of) `le_plus_trans`, this is not obvious:

```
le_plus_trans n m p (H : n <= m) :=
  (fun lemma : m <= m + p =>
    trans_contra_inv_impl_morphism
      PreOrder_Transitive
      (m + p)
      m
      lemma)
  (le_add_r m p)
  H.
```

This points to two difficulties in finding patches: Knowing when to δ-reduce terms is difficult; exploring the appropriate time for reduction may produce patches for pairs that Pumpkin currently cannot patch. Furthermore, finding patches is more challenging when neither theorem has a conclusion that is as strong as possible.

A CHALLENGE FOR INVERSION For one pair of proofs with propositionally equal conclusions, Pumpkin found a patch in one direction, but failed to **invert** it:

```
fun n m p (_ : n <= m) (_ : m <= p) (H1 : n <= p) =>
  gt_le_S n (S p) (le_lt_n_Sm n p H1)
: ∀ n m p,
    n <= m →
    m <= p →
    n <= p →
    S n <= S p.
```

Inversion was unable to invert this term, even though an inverse does exist. To invert this, inversion needs to know to δ-reduce `gt_le_S`:

```
gt_le_S n m :=
  (fun (H : ∀ n0 m0, n0 < m0 → S n0 <= m0) => H n m) ...
: ∀ n m,
    n < m →
    S n <= m.
```

It then needs to swap the hypothesis with the conclusion in `H` to produce the inverse:

```
gt_le_S^-1 n m :=
  (fun (H : ∀ n0 m0, S n0 <= m0 → n0 < m0) => H n m) ...
 : ∀ n m,
     S n <= m →
     n < m.
```

Inversion currently swaps subterms when it is not aware of any symmetry properties about the inductive type. However, it does not know when to δ-reduce function definitions. Furthermore, there are many possible subterms to swap; for inversion to know to only swap the subterms of `H`, it must have a better understanding of the structure of the term. Both of these are ways to improve inversion.

A CHALLENGE FOR GENERALIZATION **Generalization** produced an exponential number of candidates when generalizing a patch candidate with this type:

```
∀ n n0,
  (fun m => n <= max m n0) n →
  (fun m => n <= max n0 m) n
```

The goal was to generalize by `n` and produce a patch with this type:

```
∀ m0 n n0,
  n <= max m0 n0 →
  n <= max n0 m0.
```

The difficulty was in determining which occurrences of `n` to generalize. The component needed to generalize only the highlighted occurrences:

```
fun n n0 (H0 : n <= max n0 n) =>
  @eq_rect_r
    nat
    (max n0 n)
    (fun n1 => n <= n1)
    H0
    (max n n0)
    (max_comm n n0)
```

The simplest substitution strategy failed, and a more complex strategy succeeded only after producing exponentially many candidates. While this did not have a significant impact on time, this makes a good case for semantics-aware substitution strategies. In this case, we know from the type of the candidate and the type of `eq_ind_r` that these two hypothesis types are equivalent (similarly for the conclusions):

```
(fun m => n <= max m n0) n
(fun n1 => n <= n1) (max n0 n)
```

The tool could search recursively for patches to find two patches that bridge the two equivalent types:

```
p1 := fun n => max n0 n
p2 := fun n => max n n0
```

Then the candidate type is exactly this:

```
∀ n n0,
  (fun n1 => n <= n1) (p2 n) →
  (fun n1 => n <= n1) (p1 n)
```

Generalization should thus generalize the highlighted subterms and the terms that have types constrained by those subterms. This would produce a patch in one candidate:

```
fun m0 n n0 (H0 : n <= max n0 m0) =>
  @eq_ind_r
    nat
    (max n0 m0)          (* p1 m0 *)
    (fun n1 => n <= n1) (* P *)
    H0                   (* : P (p1 m0) *)
    (max m0 n0)          (* p2 m0 *)
    (max_comm m0 n0)     (* : p1 m0 = p2 m0 *)
```

This strategy would find a patch for one of the pairs that PUMPKIN failed to generalize. This is a natural future direction.

3.5.3.3 *Performance*

PUMPKIN performed well for all pairs, and when it failed, it failed fast. The slowest success took 48 ms, and the slowest failure took 7 ms.[11] Though proof terms were small ($\leq$ 67 LOC), I found this promising.

3.6 RESULTS

To show how PUMPKIN could have saved work for proof engineers, I used the PUMPKIN prototype on three **case studies** to emulate three motivating scenarios from real proof developments:

1. **Updating definitions** within a project
   (CompCert, Section 3.6.1)
2. **Porting definitions** between libraries
   (Software Foundations, Section 3.6.2)
3. **Updating proof assistant versions**
   (Coq Standard Library, Section 3.6.3)

The code I chose for these scenarios demonstrated different classes of changes. For each case, I describe how PUMPKIN configures the procedure to use differencing and transformations for that class of changes. My experiences with these scenarios suggest that patches are useful and that both differencing and the transformations are effective and flexible.

IDENTIFYING CHANGES I identified commits from popular projects that demonstrated each scenario. I emulated each scenario as follows:

1. *Replay* an example proof update for PUMPKIN.
2. *Search* the example for a patch using PUMPKIN.
3. *Apply* the patch to fix a different broken proof.

My goal was to simulate incremental use of a repair tool, at the level of a small change or a commit that follows best practices. I favored commits with changes that I could isolate, and that fit into the scope of changes supported by the PUMPKIN prototype. When isolating examples for PUMPKIN, I replayed changes from the bottom up, as if I was making the changes myself. This means that I did not always make the same change as the user. For example, the real change from Section 3.6.1 updated multiple definitions; I updated only one.

11 i7-4790K, at 4.00 GHz, 32 GB RAM

```
Record int : Type :=
  mkint {
    val: Z;
    range: 0 <= val < modulus
  }.
```

```
Record int : Type :=
  mkint {
    val: Z;
    range: -1 < val < modulus
  }.
```

Figure 17: Old (left) and new (right) definitions of `int` in CompCert.

3.6.1 *Updating Definitions*

Coq programmers sometimes make changes to definitions that break proofs within the same project. To emulate this use case, I identified a CompCert commit [100] with a breaking change to `int` (Figure 17). I used PUMPKIN to find a patch that corresponds to the change in `int`. The patch PUMPKIN found fixed broken inductive proofs.

REPLAY I used the proof of `unsigned_range` as the example for PUMPKIN. The proof failed with the new `int`:

```
Theorem unsigned_range:
  ∀ (i : int),
    0 <= unsigned i < modulus.
Proof.
  intros i. induction i using int_ind; auto.
```

✗

I replayed the change to `unsigned_range`:

```
  intros i. induction i using int_ind. simpl.  omega.
```

✓

SEARCH I used PUMPKIN to search the example for a patch that corresponds to the change in `int`. It found a patch with this type:

```
∀ (z : Z),
  -1 < z < modulus →
  0 <= z < modulus
```

APPLY After changing the definition of `int`, the proof of the theorem `repr_unsigned` failed on the last tactic:

```
Theorem repr_unsigned:
  ∀ (i : int),
    repr (unsigned i) = i.
Proof.
  ... apply Zmod_small; auto.
```

✗

Manually trying `omega`—the tactic which helped us in the proof of `unsigned_range`—did not succeed. I added the patch that PUMPKIN found to a hint database. The proof of the theorem `repr_unsigned` then went through:

```
... apply Zmod_small; auto.
```

✓

```
Fixpoint bin_to_nat (b: bin) :=
  match b with
  | B0 => O
  | B2 b' =>
      2 * (bin_to_nat b')
  | B21 b' =>
      1 + 2 * (bin_to_nat b')
  end.
```

```
Fixpoint bin_to_nat (b: bin) :=
  match b with
  | B0 => O
  | B2 b' =>
      (bin_to_nat b') +
      (bin_to_nat b')
  | B21 b' =>
      S ((bin_to_nat b') +
         (bin_to_nat b'))
  end.
```

Figure 18: Definitions of `bin_to_nat` for Users A (left) and B (right). Note that `bin_to_nat` uses fixpoints rather than **primitive eliminators**, unlike most terms in this thesis.

Instance

This scenario used the search procedure instance for changes in constructors of an inductive type. Given such a change:

```
Inductive T := ... | C : ... → H → T
Inductive T' := ... | C : ... → H' → T'
```

PUMPKIN differences two inductive proofs of theorems:

```
∀ (t : T), P t
∀ (t : T'), P t
```

for an isomorphism[12] between the constructors:

```
... → H → H'
... → H' → H
```

The proof engineer can apply these patches within the inductive case that corresponds to the constructor `C` to fix other broken proofs that induct over the changed type. PUMPKIN uses this search procedure instance for changes in constructors:

1: **diff** inductive constructors for **goals**
2: **diff** and **transform** to recursively search for changes in conclusions of the corresponding case of the proof
3: **if** there are **candidates** **then**
4: try to **invert** the **patch** to find an isomorphism

3.6.2 *Porting Definitions*

Proof engineers sometimes port theorems and proofs to use definitions from different libraries. To simulate this, I used PUMPKIN to port two solutions [4, 14] to an exercise in Software Foundations to each use the other solution's definition of the fixpoint `bin_to_nat` (Figure 18). I demonstrate one direction; the opposite was similar.

12 If PUMPKIN finds just one implication, it returns that.

REPLAY I used the proof of `bin_to_nat_pres_incr` from User A as the example for PUMPKIN. User A cut an inline lemma in an inductive case and proved it using a rewrite:

```
assert (∀ a, S (a + S (a + 0)) = S (S (a + (a + 0)))).
- ... rewrite ← plus_n_O. rewrite → plus_comm.
```

When I ported the solution by User A to use User B's definition of `bin_to_nat`, the application of this inline lemma failed. I changed the conclusion of the lemma and removed the corresponding rewrite:

```
assert (∀ a, S (a + S a) = S (S (a + a))).
- ... rewrite → plus_comm.
```

SEARCH I used PUMPKIN to search for a patch that corresponds to the change in `bin_to_nat`. It found an isomorphism:

```
∀ P b, P (bin_to_nat b) → P (bin_to_nat b + 0)
∀ P b, P (bin_to_nat b + 0) → P (bin_to_nat b)
```

APPLY After porting to User B's definition, a rewrite in the proof of the theorem `normalize_correctness` failed:

```
Theorem normalize_correctness:
  ∀ (b : bin),
    nat_to_bin (bin_to_nat b) = normalize b.
Proof.
  ... rewrite → plus_0_r.✗
```

Attempting the obvious patch from the difference in tactics—rewriting by `plus_n_O`—failed. Applying the patch that PUMPKIN found fixed the broken proof:

```
... apply patch_inv. rewrite → plus_0_r.✓
```

In this case, since I ported User A's definition to a simpler definition,[13] PUMPKIN found a patch that was not the most natural patch. The natural patch would have been to *remove* the rewrite. This did not occur when I ported User B's definition, which suggests that in the future, a proof repair tool may help inform novice users which definition is simpler: it can factor the proof, then inform the user if two factors are inverses. My Magic tutorial plugin[14] implements a prototype of this, based on lessons from this case study.

Instance

This scenario used the search procedure instance for changes in cases of a fixpoint. Given such a change:

```
Fixpoint f ... := ... | g x
Fixpoint f' ... := ... | g x'
```

13 User A uses `*`; User B uses `+`. For arbitrary `n`, `2 * n` and `n + n` are not **definitionally equal**, since `2 * n` reduces to `n + (n + 0)`, which does not reduce any further.

14 `https://github.com/uwplse/magic`

```
Definition divide p q :=
  ∃ r, p * r = q.
```

```
Definition divide p q :=
  ∃ r, q = r * p.
```

Figure 19: Old (left) and new (right) definitions of `divide` in Coq.

PUMPKIN differences two versions of proofs of the theorems:

```
∀ ..., P (f ...)
∀ ..., P (f' ...)
```

for an isomorphism that corresponds to the change:

```
∀ P, P x → P x'
∀ P, P x' → P x
```

The proof engineer can apply these patches to fix other broken proofs about the fixpoint.

The key feature that differentiates these from the patches we have encountered so far is that these patches hold for *all* `P`; for changes in fixpoint cases, the procedure generalizes candidates by `P`, not by its arguments. PUMPKIN uses this search procedure instance for changes in fixpoint cases:

1: **diff** fixpoint cases for goals
2: **diff** and **transform** to recursively search within an intermediate lemma for a change in conclusions
3: **if** there are **candidates** **then**
4: **specialize** and **factor** the **candidate**
generalize the factors by functions
try to **invert** the **patch** to find an isomorphism

For the prototype, I require the user to cut the intermediate lemma explicitly and to pass its type and arguments. In the future, an improved semantic differencing component can infer both the intermediate lemma and the arguments: it can search within the proof for some proof of a function that is applied to the fixpoint.

3.6.3 *Updating Proof Assistant Versions*

Coq sometimes makes changes to its standard library that break backwards compatibility. To test the plausibility of using a patch finding tool for proof assistant version updates, I identified a breaking change in the Coq standard library [102]. The commit changed the definition of `divide` prior to the Coq 8.4 release (Figure 19). The change broke 46 proofs in the standard library. I used PUMPKIN to find an isomorphism that corresponds to the change in `divide`. The isomorphism PUMPKIN found fixed broken proofs.

REPLAY I used the proof of `mod_divide` as the example for PUMPKIN. The proof broke with the new `divide`:

```
Theorem mod_divide:
  ∀ a b,
    b~=0 →
    (a mod b == 0 ↔ (divide b a)).
Proof.
  ... rewrite (div_mod a b Hb) at 2.✗
```

I replayed changes to `mod_divide`:

```
  ... rewrite mul_comm.  symmetry.
  rewrite (div_mod a b Hb) at 2.✓
```

SEARCH I used PUMPKIN to search within the example patched proof for a patch that corresponds to the change in `divide`. It found an isomorphism:

```
∀ r p q, p * r = q → q = r * p
∀ r p q, q = r * p → p * r = q
```

APPLY The proof of the theorem `Zmod_divides` broke after rewriting by the changed theorem `mod_divide`:

```
Theorem Zmod_divides:
  ∀ a b,
    b<>0 →
    (a mod b = 0 ↔ ∃ c, a = b * c).
Proof.
  ... split; intros (c,Hc); exists c; auto.✗
```

Adding the patches PUMPKIN found to a hint database made the proof go through:

```
  ... split; intros (c,Hc); exists c; auto.✓
```

Instance

This scenario used the search procedure instance for changes in dependent arguments to constructors. PUMPKIN differences two versions of a proof that apply the same constructor to different dependent arguments:

```
... (C (P x)) ...
... (C (P' x)) ...
```

for an isomorphism between the arguments:

```
∀ x, P x → P' x
∀ x, P' x → P x
```

The proof engineer can apply these patches to patch proofs that apply the constructor (here, to fix proofs that apply `divide` to some `r`).

So far, we have encountered changes of this form as arguments to an induction principle; in this case, the change is an argument to a constructor. A patch between arguments to an induction principle maps directly between conclusions of the new and old theorem without induction; a patch between constructors does not. For example, for `divide`, we can find a patch with this form:

```
∀x, P x → P' x
```

However, without using the induction principle for `exists`, we can't use that patch to prove this:

```
(∃ x, P x) → (∃ x, P' x)
```

This changes the goal type that semantic differencing determines.

PUMPKIN uses this search procedure instance for changes in constructor arguments:

1: **diff** constructor arguments for **goals**
2: **diff** and *transform* to recursively search within those arguments for changes in conclusions
3: **if** there are candidates **then**
4: **generalize** the **candidate**
factor and try to **invert** the **patch** to find an isomorphism

For the prototype, the model of constructors for the semantic differencing component is limited, so PUMPKIN asks the user to provide the type of the change in argument (to guide line 2). Extending semantic differencing may help remove this restriction.

3.7 CONCLUSION

This **thesis** set out to show that:

> changes in programs, specifications, and proofs can 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.

With PUMPKIN, so far, it is fair to say that:

> changes in the content of programs, specifications, and proofs can carry information that a tool can extract, generalize, and sometimes apply to fix other proofs broken by the same change (Sections 3.2, 3.3, and 3.4). A tool that automates this (Section 3.5) could have saved work for proof engineers relative to reference manual repairs in a few practical use cases (Section 3.6).

Or, informally, there is *some* **reason to believe** that verifying a modified system *could have* been easier than verifying the original the first time around, in *a few* practical use cases.

This is progress, but it is not quite there yet. As I have shown you throughout this chapter, the PUMPKIN prototype is too limited in both theory and implementation. Most notably, the PUMPKIN prototype has limited support for patch application and supports a narrow

class of changes in an ad hoc manner. And as I mentioned earlier, without considering the extension from the next chapter, the PUMPKIN prototype includes very little support for tactics.

The next chapter will introduce a repair tool that supports a broad, complementary class of changes beyond that supported by PUMPKIN alone. In parallel, it will introduce new technologies that address many of the limitations seen in this chapter. In doing so, it will show how the thesis holds on a broad class of changes, with more principled and better integrated support for patch application and tactic generation. It will show how all of this helps proof engineers in the real world—not just retroactively, but in real time.

4

PROOF REPAIR ACROSS TYPE EQUIVALENCES

This chapter presents the Pumpkin P*i* extension to the **Pumpkin Patch proof repair plugin** suite.[1] Pumpkin Pi is a plugin that supports proof repair across a broad class of changes in datatypes called *type equivalences* (Section 4.2.2), thereby supporting a large class of practical repair scenarios. Proof repair **across type equivalences** with Pumpkin Pi makes progress on two challenges that **Pumpkin** had left open:

1. Pumpkin supported a very limited classes of changes in datatypes, namely those that do not change structure. Similar tools developed since still supported only a predefined set of changes [142, 165]. As the **REPLica** user study showed, these were not informed by the current needs of proof engineers.

2. Pumpkin had only preliminary integration with typical proof engineering workflows. Similar tools developed since likewise lacked support for workflow integration [136, 142], or imposed additional proof obligations like always proving relations corresponding to changes [151].

CHALLENGE 1: FLEXIBLE TYPE SUPPORT The case studies in Section 4.6—summarized in Table 1 on page 97—show that Pumpkin Pi is flexible enough to support a wide range of proof repair use cases. In general, Pumpkin Pi can support any change described by an equivalence—a scope that even includes changes like adding indices to datatypes. Pumpkin Pi takes the equivalence in a deconstructed form that I call a *configuration*. The configuration expresses to the proof term transformation how to translate terms defined over the old version of a type to refer only to the new version, and how to do so without breaking **definitional equality**. The proof engineer can write this configuration in Coq and feed it to Pumpkin Pi (*manual configuration* in Table 1), configuring Pumpkin Pi to support the change.

CHALLENGE 2: WORKFLOW INTEGRATION Research on workflow integration for proof repair tools is in its infancy. Pumpkin Pi is built

1 I annotate each claim in this chapter to which code is relevant with a circled number like ①. These circled numbers are links to code, and are detailed in a guide that can be found here: `https://github.com/uwplse/pumpkin-pi/blob/v2.0.0/GUIDE.md`.

```
Inductive list T : Type :=
| nil : list T
| cons : T → list T → list T.
```

```
Inductive list T : Type :=
| cons : T → list T → list T
| nil : list T.
```

Figure 20: A change from the old version (left) to the new version (right) of `list`. Recall that `list` is an inductive datatype that is either empty (the `nil` constructor), or the result of placing an element in front of another `list` (the `cons` constructor). The change swaps these constructors (orange).

with workflow integration in mind. For example, PUMPKIN Pi produces suggested proof scripts (rather than proof terms) for repaired proofs, a challenge highlighted in the previous chapter, in other existing work [142], and in **QED at Large**. In addition, PUMPKIN Pi implements search procedures that automatically discover configurations and prove the equivalences they induce for four different classes of changes (*automatic configuration* in Table 1), decreasing the burden of proof obligations imposed on the proof engineer. My partnership with an industrial proof engineer has informed other changes to further improve workflow integration (Sections 4.5 and 4.6).

BRINGING IT TOGETHER In the frame of the thesis, proof repair across type equivalences is a new form of proof automation that extracts general information from breaking changes in the *datatypes* that programs, specifications, and proofs refer to, then applies that information to fix any program, specification, or proof broken by that change (Section 4.2). This extraction, generalization, and application works at the level of proof terms, through a combination of novel semantic differencing algorithms over datatypes (Section 4.3) and a configurable proof term transformation (Section 4.4). PUMPKIN Pi automates this process, with additional support for manual configuration by proof engineers and for integration with typical **proof engineering** workflows (Section 4.5). Case studies show that PUMPKIN Pi can save and in several cases *has* saved work for proof engineers on major proof developments and on changes that matter (Section 4.6).

4.1 MOTIVATING EXAMPLE

Consider a simple example of using PUMPKIN Pi: repairing proofs after swapping the two constructors of the `list` datatype (Figure 20). This is inspired by a similar change from a user study of proof engineers (Section 4.6). Even such a simple change can cause trouble, as in the proof of the lemma `rev_app_distr` from the Coq standard library (Figure 21). This lemma says that appending (`++`) two lists and reversing (`rev`) the result behaves the same as appending the reverse of the second list onto the reverse of the first list. The proof script

```
Lemma rev_app_distr {A} :
  ∀ (x y : list A),
    rev (x ++ y) = rev y ++ rev x.
Proof. (* by induction over x and y *)
  induction x as [| a l IHl].
  (* x nil:  *) induction y as [| a l IHl].
  (* y nil:  *) simpl. auto.
  (* y cons *) simpl. rewrite app_nil_r; auto.
  (* both cons:  *) intro y. simpl.
  rewrite (IHl y). rewrite app_assoc; trivial.
Defined.
```

Figure 21: The proof of the lemma `rev_app_distr` from the Coq standard library. Comments mine for clarity.

works by induction over the input lists `x` and `y`: In the base case for both `x` and `y`, the result holds by reflexivity. In the base case for `x` and the inductive case for `y`, the result follows from the existing lemma `app_nil_r`. Finally, in the inductive case for both `x` and `y`, the result follows by the inductive hypothesis and the existing lemma `app_assoc`.

When we change the `list` type, this proof no longer works. To repair this proof with PUMPKIN Pi, we run this command:

```
Repair Old.list New.list in rev_app_distr.
```

assuming the old and new list types from Figure 20 are in modules `Old` and `New`. This suggests a proof script that succeeds (in light blue to denote PUMPKIN Pi produces it automatically):

```
Proof. (* by induction over x and y *)
  intros x. induction x as [a l IHl| ]; intro y0.
  - (* both cons:  *) simpl. rewrite IHl. simpl.
    rewrite app_assoc. auto.
  - (* x nil:  *) induction y0 as [a l H| ].
    + (* y cons:  *) simpl. rewrite app_nil_r. auto.
    + (* y nil:  *) auto.
Defined.
```

where the dependencies (`rev`, `++`, `app_assoc`, and `app_nil_r`) have also been updated automatically ①. If we would like, we can manually modify this to something that more closely matches the style of the original proof script:

```
Proof. (* by induction over x and y *)
  induction x as [a l IHl|].
  (* both cons:  *) intro y. simpl.
  rewrite (IHl y). rewrite app_assoc; trivial.
  (* x nil:  *) induction y as [a l IHl|].
  (* y cons:  *) simpl. rewrite app_nil_r; auto.
  (* y nil:  *) simpl. auto.
Defined.
```

We can even repair the entire list module from the Coq standard library all at once by running the `Repair module` command ①. When we are done, we can get rid of `Old.list`.

The key to success is taking advantage of Coq's structured proof term language: Recall that Coq compiles every **proof script** to a **proof term** in the rich functional programming language **Gallina**—PUMPKIN Pi repairs that term. PUMPKIN Pi then decompiles the repaired proof term (with optional hints from the original proof script) back to a suggested proof script that the proof engineer can maintain.

In contrast, updating the poorly structured proof script directly would not be straightforward. Even for the simple proof script above, grouping tactics by line, there are $6! = 720$ permutations of this proof script. It is not clear which lines to swap since these tactics do not have a semantics beyond the searches their evaluation performs. Furthermore, just swapping lines is not enough: even for such a simple change, we must also swap arguments, so that:

```
induction x as [|a l IHl].
```

becomes:

```
induction x as [a l IHl|].
```

Valentin Robert's thesis [142] describes the challenges of repairing tactics in detail. PUMPKIN Pi's approach circumvents this challenge.

4.2 APPROACH

PUMPKIN Pi can do much more than permute constructors. Given an equivalence between types A and B, PUMPKIN Pi repairs functions and proofs defined over A to instead refer to B. It does this in a way that allows for removing references to A, which is essential for proof repair, since A may be an old version of an updated type.

The proof engineer can use PUMPKIN Pi (Section 4.5.2) to automatically repair proofs in response to a broad class of changes in datatypes. PUMPKIN Pi in particular repairs proofs in response to changes that correspond to *type equivalences* [154], or pairs of functions that map between two types (possibly with some additional information) and are mutual inverses (Section 4.2.2).[2] Looking back to the **thesis** statement, the information shows up in the difference between versions of the changed datatype. With **automatic configuration**, PUMPKIN Pi can extract and generalize that information to a type equivalence, then apply it to fix other broken proofs. With **manual configuration**, the *proof engineer* extracts and generalizes that information herself, but PUMPKIN Pi can still apply the result to fix other broken proofs.

Like the original **PUMPKIN** prototype, PUMPKIN Pi also does this using a combination of **differencing** and **proof term transformations**.

2 Every equivalence induces something called an *adjoint* equivalence [154], and those adjoint equivalences can be nicer to work with. **Jasper** proved this fact for me in a way that does not rely on any axioms beyond those assumed by Coq (23), and **Nate** used that proof to build machinery for PUMPKIN Pi to derive the adjoint equivalence from the equivalence itself (10).

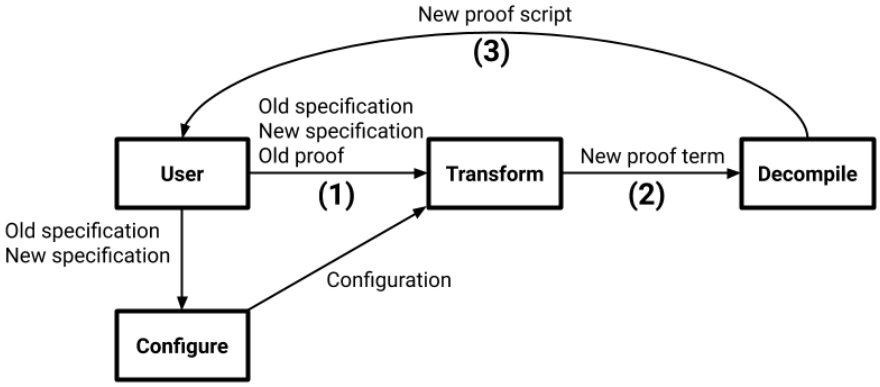


Figure 22: The workflow for PUMPKIN Pi.

The differencing algorithms (Section 4.2.3) run in response to a breaking change in a datatype that corresponds to a type equivalence. When they succeed, the diff that they find is that type equivalence. The proof engineer can also pass the type equivalence to PUMPKIN Pi directly, effectively doing differencing by hand. In either case, the proof term transformation (Section 4.2.4) then transforms a proof term defined over the old version of the datatype directly to a proof term defined over the new version of the datatype. PUMPKIN Pi further supports proof script integration and other features for better workflow integration (Section 4.5), with real payoffs for proof engineers (Section 4.6).

4.2.1 *Workflow: Configure, Transform, Decompile*

PUMPKIN Pi extends PUMPKIN PATCH with a new **command** called `Repair`, with the syntax:

```
Repair old_type new_type in old_proof.
```

where `old_type` and `new_type` are the old and new versions of the changed datatype, and `old_proof` is the old version of the proof broken by that change in datatype. This invokes the PUMPKIN Pi plugin, which updates the old version of the proof to some new version of the proof defined over the new version of the datatype, then defines it as a new proof term and suggests a corresponding new proof script if successful. All terms that PUMPKIN Pi defines are type checked in the end, so PUMPKIN Pi does not extend the **TCB**.

Figure 22 shows how this comes together when the proof engineer invokes PUMPKIN Pi:

1. The proof engineer **Configure**s PUMPKIN Pi, either manually or automatically.
2. The configured **Transform** transforms the old proof term into the new proof term.
3. **Decompile** suggests a new proof script.

```
swap T (l : Old.list T)
  : New.list T
:=
  Old.list_rect T
    (fun _ => New.list T)
    New.nil
    (fun t _ (IHl: New.list T) =>
      New.cons T t IHl)
    l.

Lemma section:
  ∀ T (l : Old.list T),
    swap⁻¹ T (swap T l) = l.
Proof.
  intros T l. symmetry.
  induction l as [ |t l0 H].
  - auto.
  - simpl. rewrite ← H. auto.
Qed.
```

```
swap⁻¹ T (l : New.list T)
  : Old.list T
:=
  New.list_rect T
    (fun _ => Old.list T)
    (fun t _ (IHl: Old.list T) =>
      Old.cons T t IHl)
    Old.nil
    l.

Lemma retraction:
  ∀ T (l : New.list T),
    swap T (swap⁻¹ T l) = l.
Proof.
  intros T l. symmetry.
  induction l as [t l0 H| ].
  - simpl. rewrite ← H. auto.
  - auto.
Qed.
```

Figure 23: Two functions between `Old.list` and `New.list` (top) that form an equivalence (bottom).

There are currently four search procedures for **automatic configuration** implemented in Pumpkin Pi (see Table 1 on page 97). **Manual configuration** makes it possible for the proof engineer to configure the transformation to any equivalence, even without a search procedure.

The breaking change to Figure 20 in Section 4.1, for example, used automatic configuration. When we invoked Pumpkin Pi:

```
Repair Old.list New.list in rev_app_distr.
```

it invoked a search procedure that differences `Old.list` and `New.list`, then transformed `rev_app_distr` to use `New.list` in place of `Old.list`. In the end, it suggested a proof script that we could use going forward.

4.2.2 *Scope: Type Equivalences*

Pumpkin Pi automatically repairs proofs in response to changes in types that correspond to **type equivalences**. When a type equivalence between types A and B exists, we can say that those types are **equivalent** (denoted $A \simeq B$). Figure 23 shows a type equivalence between the two versions of `list` from Figure 20 that Pumpkin Pi discovered and proved automatically ①.

To give intuition for what kinds of changes can be described by equivalences, I preview two changes. See Table 1 on page 97 for more.

FACTORING OUT CONSTRUCTORS Consider changing the type `I` to the type `J` in Figure 24. `J` can be viewed as `I` with its two constructors `A` and `B` pulled out to a new argument of type `bool` for a single

```
Inductive I :=
| A : I
| B : I.
```

```
Inductive J :=
| makeJ : bool → J.
```

Figure 24: The old type `I` (left) is either `A` or `B`. The new type `J` (right) is `I` with `A` and `B` factored out to `bool` (orange).

```
Inductive list T : Type :=
| nil : list T
| cons : T →
    list T →
    list T.
```

```
Inductive vector T : nat → Type :=
| nil : vector T 0
| cons : T → ∀ (n : nat),
    vector T n →
    vector T (S n).
```

Figure 25: A vector (right) is a list (left) indexed by its length (orange). Vectors effectively make it possible to enforce length invariants about lists at compile time.

constructor. With PUMPKIN Pi, the proof engineer can repair functions and proofs about `I` to instead use `J`, as long as she configures PUMPKIN Pi to describe which constructor of `I` maps to `true` and which maps to `false`. This information about constructor mappings induces an equivalence `I` $\simeq$ `J` across which PUMPKIN Pi repairs functions and proofs. File ② shows an example of this, mapping `A` to `true` and `B` to false, and repairing proofs of De Morgan's laws.

ADDING A DEPENDENT INDEX At first glance, the word *equivalence* may seem to imply that PUMPKIN Pi can support only changes in which the proof engineer does not add or remove information. But equivalences are more powerful than they may seem. Consider, for example, changing a list to a length-indexed vector (Figure 25). Since Σ`(l:list T).length l = n` $\simeq$ `vector T n`, PUMPKIN Pi can repair functions and proofs about lists to functions and proofs about vectors of particular lengths ③. From the proof engineer's perspective, after moving from `list` to `vector`, to fix her functions and proofs, she must prove invariants about the lengths of her lists. PUMPKIN Pi makes it easy to separate out that proof obligation, then automates the rest.

A more formal result about the expressiveness of equivalences holds inside of *homotopy type theory*: a type theory with *univalence*, which states that **equivalence** is equivalent to **propositional equality**. Homotopy type theory makes it possible to construct quotient types; with the help of these quotient types, it is possible to form an equivalence from a relation, even when the relation is not an equivalence [9]. A less general result holds in Coq, which lacks quotient types: it is possible to achieve a similar outcome and use PUMPKIN Pi for changes that add or remove information whenever those changes can be expressed as equivalences between Σ types or sum types. With some creativity, this can even support adding new constructors to types, though not yet in a way that saves work proof engineers.

4.2.3 *Differencing: Equivalences from Changes*

Differencing in PUMPKIN Pi is what configures the proof term transformation to a particular **type equivalence**. By default, when the proof engineer invokes the `Repair` command, differencing runs automatically. For example, when we invoked:

```
Repair Old.list New.list in rev_app_distr.
```

in Section 4.1, differencing discovered and proved the equivalence in Figure 23. In total, PUMPKIN Pi currently implements four search procedures for **automatic configuration**; I will explain these more in Sections 4.3 and 4.5. Each search procedure automates differencing for an entire class of changes that can be described by type equivalences. In the end, it returns a configuration that corresponds to the equivalence (see Section 4.3), along with the equivalence itself.

Manual configuration makes it possible for the proof engineer to skip differencing, so that PUMPKIN Pi is not limited by the search procedures currently implemented, nor by the undecidability of differencing arbitrary types. Manual configuration supports any change that can be described by a type equivalence. To configure PUMPKIN Pi manually, the proof engineer can pass the configuration corresponding to the equivalence to PUMPKIN Pi's `Configure Repair` command before invoking `Repair`. This is what I did for the change in Figure 24 ②.

SUMMARY In summary, differencing has the following specification:

- **Inputs**: types A and B, assuming:
  - there is a **type equivalence** describing the change from A to B (possibly with some new information), and
  - the corresponding change is in a class currently supported by a search procedure for **automatic configuration**.
- **Outputs**:
  - functions `f` and `g`,
  - proofs `section` and `retraction`, and
  - a configuration c,

  guaranteeing:
  - `f` and `g` form an **equivalence** that describes the change from A to B (possibly with some new information),
  - `section` and `retraction` prove that `f` and `g` form an **equivalence**, and
  - c is a decomposition of the equivalence.

The new information for the change in Figure 25, for example, is the length of the list. Differencing discovers the equivalence corresponding

to this change, as well as a configuration c that is a decomposition of this equivalence. Section 4.3 describes what it means for c to be a decomposition of the equivalence, and proves that this is possible for any equivalence. Manual configuration, on the other hand, takes in the configuration directly. In either case, the transformation uses this configuration to automatically repair broken proofs.

4.2.4 *Transformation: Transport with a Twist*

PUMPKIN Pi repairs proofs in response to these changes by implementing and automating a kind of proof reuse known as *transport* from **homotopy type theory**, but in a way that is suitable for repair. In homotopy type theory, transport is essentially a specialized **eliminator** for rewriting across **equivalences**. In particular, it takes a term t and produces a term t' that is the same as t modulo an equivalence $A \simeq B$. If t is a function, then t' behaves the same way modulo the equivalence; if t is a proof, then t' proves the same theorem the same way modulo the equivalence.

The details of transport in homotopy type theory can be found in the homotopy type theory book [154], and in Section 5.1.4. Transport is realizable as a function in homotopy type theory—that is, it is *internal*—precisely because of **univalence**. But univalence is a property that Coq's type theory **CIC$_\omega$** lacks! It is possible to finitely approximate internal transport in Coq using a special framework [150], but this sometimes introduces axioms, thereby extending the **TCB**. Instead, the PUMPKIN Pi transformation assumes a univalent *metatheory* in which to interpret its specification, but does not introduce any axioms to Coq—that is, it implements transport *externally*.

In this thesis, when transport across $A \simeq B$ takes t to t', I say that t and t' are *equal up to transport* across that equivalence (denoted $t \equiv_{A \simeq B} t'$).[3] In Section 4.1, the original append function `++` over `Old.list` and the repaired append function `++` over `New.list` that PUMPKIN Pi produces are equal up to transport across the equivalence from Figure 23, since (by `app_ok` ①):

```
∀ T (l1 l2 : Old.list T),
  swap T (l1 ++ l2) = (swap T l1) ++ (swap T l2).
```

The original `rev_app_distr` is equal to the repaired proof up to transport, since both prove the same thing the same way up to the equivalence, and up to the changes in `++` and `rev`.

In **univalent** type theories, transport works by applying the functions that make up the equivalence to convert inputs and outputs between types. Even if we had univalence, this approach would not be suitable for repair, since it would not make it possible to remove the

3 This notation should be interpreted in a **univalent** metatheory. Note also that, for equivalent A and B, there can be many equivalences $A \simeq B$. Equality up to transport is across a *particular* equivalence, but I erase this in the notation.

old type A. PUMPKIN Pi implements transport externally, in a way that allows for removing references to A—by **proof term transformation**.

SUMMARY In summary, the transformation has the following specification:

- **Inputs**:
  - types A and B,
  - the **configuration** c, and
  - a proof term t,

  assuming c is a decomposition of a **type equivalence** describing the change from A to B (possibly with some new information).
- **Outputs**: a proof term t', guaranteeing:
  - t' refers to B in place of A, and
  - t and t' are equal up to **transport** along the **equivalence** formed by c.

This specification glazes over a few important issues, most notably that B may refer to A (so PUMPKIN Pi has specialized termination logic), and that it may be desirable to port only *some* instances of A to B (but PUMPKIN Pi by default ports *all* instances). I discuss these more in Sections 4.4 and 4.5.

4.3 DIFFERENCING

Differencing—whether done by the tool (**automatic configuration**) or by the proof engineer (**manual configuration**)—identifies and proves a **type equivalence**. But differencing further decomposes that equivalence into a form called a **configuration**. The configuration is the key to building a proof term transformation that implements transport in a way that is suitable for repair.

Each configuration is a deconstruction of a particular equivalence $A \simeq B$. In particular, it deconstructs the equivalence into things that talk about A, and things that talk about B. It does so in a way that hides details specific to the equivalence, like the order or number of arguments to an **eliminator** or type.

At a high level, the configuration helps the transformation achieve two goals:

1. preserve equality up to transport across the equivalence between A and B (Section 4.3.1), and
2. produce well-typed terms (Section 4.3.2).

To differencing, this configuration is a pair of pairs:

```
((DepConstr, DepElim), (Eta, Iota))
```

each of which corresponds to one of the goals: `DepConstr` and `DepElim` define how to transform *dependent constructors* and *dependent eliminators*, thereby preserving the equivalence, and `Eta` and `Iota` define how to transform η-expansion and ι-reduction of dependent constructors and dependent eliminators, thereby producing well-typed terms.

The connection between the configuration parts and constructors and eliminators is an analogy—though one with formal meaning by way of category theory (Section 4.3.3.1). Configurations and equivalences are equally expressive: every configuration induces an equivalence, and every equivalence induces a configuration (Section 4.3.3.2). Each search procedure for automatic configuration produces both the configuration and the equivalence that it induces (Section 4.3.4). Manual configuration takes as input the configuration directly.

All terms that I introduce in this section are in **CIC$_\omega$** with **primitive eliminators**. Section 4.5 describes how I scale this from CIC$_\omega$ to Coq.

4.3.1 *Preserving the Equivalence*

To preserve the equivalence, the configuration maps terms over A to terms over B by viewing each term of type B as if it were an A. This way, the transformation in Section 4.4 can replace values of A with values of B, and inductive proofs about A with inductive proofs about B, all without changing the order or number of arguments.

The two configuration parts responsible for this are `DepConstr` and `DepElim` (**dependent constructors** and **dependent eliminators**). These describe how to construct and eliminate A and B, wrapping the types with a common inductive structure. The transformation requires the same number of dependent constructors and cases in dependent eliminators for A and B, even if A and B themselves are **inductive types** with different numbers of constructors (A and B need not even be inductive; see Sections 4.3.3 and 4.6).

For the `list` change from Section 4.1, the configuration that Pumpkin Pi discovers uses the dependent constructors and eliminators in Figure 26. The dependent constructors for `Old.list` are the normal constructors with the order unchanged, while the dependent constructors for `New.list` swap the order of constructors. Similarly, the dependent eliminator for `Old.list` is the normal eliminator for `Old.list`, while the dependent eliminator for `New.list` swaps cases.

As the name hints, these constructors and eliminators can be dependent. For example, let B be the type of vectors of some arbitrary, unspecified length, packed into a Σ type:

```
Σ(n : nat).vector T n
```

```
(* nil *)
DepConstr(0, Old.list T)
  : Old.list T
:= Constr(0, Old.list T).

(* cons *)
DepConstr(1, Old.list T) t l
  : Old.list T
:= Constr (1, Old.list T) t l.

(* induction over lists *)
DepElim(l, P) { p_nil, p_cons }
  : P l
:= Elim(l, P) { p_nil, p_cons }.
```

```
(* nil *)
DepConstr(0, New.list T)
  : New.list T
:= Constr(1, New.list T).

(* cons *)
DepConstr(1, New.list T) t l
  : New.list T
:= Constr(0, New.list T) t l.

(* induction over lists *)
DepElim(l, P) { p_nil, p_cons }
  : P l
:= Elim(l, P) { p_cons, p_nil }.
```

Figure 26: The dependent constructors and eliminators for old (left) and new (right) `list`, with the difference in orange.

Pumpkin Pi can port proofs across the equivalence between this choice of B and `list T` ③. The dependent constructors Pumpkin Pi discovers for B pack the index into an existential, like:

```
DepConstr(0, B) : B :=
  ∃ (Constr(0, nat)) (Constr(0, vector T)).
```

and the eliminator it discovers eliminates the projections:

```
DepElim(b, P) { f_0 f_1 } : P (∃ (π_l b) (π_r b)) :=
  Elim(π_r b, λ(n : nat)(v : vector T n).P (∃ n v)) {
    f_0,
    (λ(t : T)(n : nat)(v : vector T n).f_1 t (∃ n v))
  }.
```

In both these examples, the interesting work moves into the configuration: the configuration for the first swaps constructors and cases, and the configuration for the second maps constructors and cases over `list T` to constructors and cases over `Σ(n : nat).vector T n`. That way, the transformation need not add, drop, or reorder arguments—it can truly be generic over type equivalences. Furthermore, both examples use **automatic configuration**, so differencing in Pumpkin Pi's **Configure** component discovers `DepConstr` and `DepElim` from just the types A and B, taking care of even the difficult work.

4.3.2 *Producing Well-Typed Terms*

The other configuration parts `Eta` and `Iota` deal with producing well-typed terms, in particular by **transporting** equalities. Recall that CIC_ω is an **intensional** type theory, and so distinguishes between **definitional** and **propositional** equality. When a datatype changes, sometimes, definitional equalities defined over the old version of that type must become propositional. A naive proof term transformation may fail to generate well-typed terms if it does not account for this.

```
                                Inductive positive :=
                                | xI : positive → positive
                                | xO : positive → positive
                                | xH : positive.

Inductive nat :=                Inductive N :=
| O : nat                       | N0 : N
| S : nat → nat.                | Npos : positive → N.
```

Figure 27: A unary natural number `nat` (left) is either zero (`O`) or the successor of some other natural number (`S`). A binary natural number `N` (right) is either zero (`N0`) or a positive binary number (`Npos`), where a positive binary number is either `1` (`xH`), or the result of shifting left and adding `1` (`xI`) or `0` (`xO`). While `nat` and `N` are equivalent, they have different inductive structures. Consequentially, **definitional** equalities over `nat` may become **propositional** over `N`.

Otherwise, if the transformation transforms a term $t : T$ to some $t' : T'$, it does not necessarily transform T to T' [151].

`Eta` and `Iota` describe how to transport equalities. More formally, they define η-expansion and ι-reduction of A and B, which may be propositional rather than definitional, and so must be explicit in the transformation. η-expansion describes how to expand a term to apply a constructor to an eliminator in a way that preserves propositional equality, and is important for defining dependent eliminators [121]. ι-reduction (β-reduction for inductive types) describes how to reduce an elimination of a constructor [120].

The configuration for the change from lists to vectors of some length has propositional `Eta` over B. It uses η-expansion for Σ:

```
Eta(B) := λ(b : B).∃ (π_l b) (π_r b).
```

which is propositional and not definitional in Coq. Thanks to this, we can forego the assumption that our language has primitive projections (definitional η for Σ).

Each `Iota`—one per constructor—describes and proves the behavior of ι-reduction for `DepElim` on the corresponding case. This is needed, for example, to port proofs about unary numbers `nat` to proofs about binary numbers `N` (Figure 27). While we can define `DepConstr` and `DepElim` to induce an equivalence between them (5), we run into trouble reasoning about applications of `DepElim`, since proofs about `nat` that hold by reflexivity do not necessarily hold by reflexivity over `N`. For example, in Coq, while `S (n + m) = S n + m` holds by reflexivity over `nat`, when we define `+` with `DepElim` over `N`, the corresponding theorem over `N` does not hold by reflexivity.

To transform proofs about `nat` to proofs about `N`, we must transform **definitional** ι-reduction over `nat` to **propositional** ι-reduction over `N`. For our choice of `DepConstr` and `DepElim`, ι-reduction is definitional over `nat`, since a proof of:

```
Π P p0 pS (n : nat),
  DepElim(DepConstr(1, nat) n, P) { p0, pS } =
  ps n (DepElim(n, P) { p0, pS }).
```

holds by reflexivity. `Iota` for `nat` in the `S` case is a rewrite by that proof by reflexivity ⑤, with type:

```
Π P p0 pS (n : nat) (Q: P (DepConstr(1, nat) n) → s),
  Iota(1, nat, Q) :
    Q (ps n (DepElim(n, P) { p0, pS })) →
    Q (DepElim(DepConstr(1, nat) n, P) { p0, pS }).
```

In contrast, ι for `N` is propositional, since the theorem:

```
Π P p0 pS (n : N),
  DepElim(DepConstr(1, N) n, P) { p0, pS } =
  ps n (DepElim(n, P) { p0, pS }).
```

no longer holds by reflexivity. `Iota` for `N` is a rewrite by the propositional equality that proves this theorem ⑤, with type:

```
Π P p0 pS (n : N) (Q: P (DepConstr(1, N) n) → s),
  Iota(1, N, Q) :
    Q (ps n (DepElim(n, P) { p0, pS })) →
    Q (DepElim(DepConstr(1, N) n, P) { p0, pS }).
```

By replacing `Iota` over `nat` with `Iota` over `N`, the transformation replaces rewrites by reflexivity over `nat` to rewrites by propositional equalities over `N`. That way, `DepElim` behaves the same over `nat` and `N`.

Taken together over both A and B, `Iota` describes how the inductive structures of A and B differ. The transformation requires that `DepElim` over A and over B have the same structure as each other, so if A and B *themselves* have the same inductive structure (if they are *ornaments* [108]), then if ι is definitional for A, it will be possible to choose `DepElim` with definitional ι for B. Otherwise, if A and B (like `nat` and `N`) have different inductive structures, then definitional ι over one would become propositional ι over the other.

4.3.3 *Specifying Correct Configurations*

Choosing a configuration necessarily depends in some way on the proof engineer's intentions: there can be infinitely many equivalences that correspond to a change, only some of which are useful (for example ⑦, any A is equivalent to `unit` refined by A). And there can be many configurations that correspond to an equivalence, some of which will produce terms that are more useful or efficient than others (consider `DepElim` converting through several intermediate types).

Thankfully, while it is not possible to control for intentions, it *is* possible to specify what it means for a chosen configuration to be correct. In particular, the analogy tying the **configuration** to **constructors** and **eliminators** has meaning in terms of what in category theory are known as *initial algebras of endofunctors*, and correctness—that configurations and equivalences are isomorphic—follows by Lambek's

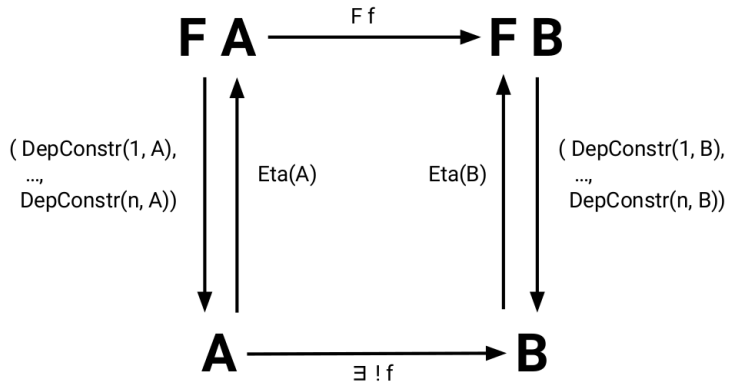


Figure 28: The categorical representation of a configuration for equivalent types A and B in terms of initial algebras.

theorem (Section 4.3.3.1). A more syntactic version of this claim can be used to specify and prove correctness of configurations in a **univalent** metatheory, or on an ad hoc basis in **CIC**$_\omega$ (Section 4.3.3.2).

HISTORICAL NOTE When I met with **Michael Shulman** over coffee a few years ago, he said that a preliminary version of this work "feels like univalence," "feels like coherence," and "feels like an endofunctor." All three were correct! But I did not understand the connection to endofunctors by way of Lambek's until a few months before writing this thesis. **Carlo Angiuli** and **Anders Mörtberg** identified this connection, and **Carlo** explained it to me. It is quite beautiful!

But it is also quite preliminary. I hope this connection as presented in Section 4.3.3.1 will help communicate some of the categorical intuition behind why all of this works, but I will not be surprised if I get some of the details wrong. Please regard this as a bit speculative, unlike the syntactic presentation in Section 4.3.3.2, which is already published.

4.3.3.1 *Categorical Intuition for Correctness*

Configurations have meaning in terms of **initial algebras** [122], which in **homotopy type theory** represent inductive types [154]. This is why the configuration is most natural when the types A and B are inductive. But the configuration is in fact more general than that—it can support any two equivalent types A and B (by Lambek's theorem).

Figure 28 shows this for an arbitrary configuration for an equivalence between A and B. Here, $F\ A$ is the inductive structure of A. `DepConstr` maps from $F\ A$ to A, and `Eta` maps from A back to $F\ A$. `DepElim` (not pictured explicitly) is the arrow corresponding to `DepConstr` defined over the analogous diagram lifted to $A \to s$ for some sort s (also not pictured explicitly); by uniqueness of `f`, any `f` must pass through a function isomorphic to `DepElim`. `Iota` (also not

pictured explicitly) is used in the proof that the diagram commutes. The configuration parts for B are similar.

For example, fixing A as `nat`, $F\ A$ is `1 + nat`: the sum of the unit type and `nat`. Going from $F\ A \to A$, the left injection maps to the `O` constructor, and the right injection maps to the `S` constructor. The inverse is essentially[4] the identity function. Any f must pass through the eliminator for `nat`, which would show up explicitly in place of the constructors in the diagram lifted to `nat` $\to s$. The diagram commutes trivially, since the `S` case of the eliminator reduces.

This diagram gives intuition for how the configuration splits up an arbitrary equivalence between A and B into parts that talk about A and B separately. All the transformation in Section 4.4 does is follow the arrows in the diagram to get from A directly to B. But in order to do that, it needs to handle the nuances of **definitional equality** and **dependent types** in CIC$_\omega$—for example, by explicitly representing and porting the proof that the diagram commutes. The syntactic presentation in the next section handles those nuances.

4.3.3.2 *Correctness, Syntactically*

The categorical definition may help with some of the intuition, but it does not help with validating correct configurations. Fortunately, it is also possible to specify syntactically what it means for a chosen configuration to be correct: Fix a configuration. Let `f` be the function that uses `DepElim` to eliminate A and `DepConstr` to construct B, and let `g` be similar. Figure 29 specifies the correctness criteria for the configuration. These criteria relate `DepConstr`, `DepElim`, `Eta`, and `Iota` in a way that preserves equivalence coherently with equality.

EQUIVALENCE To preserve the equivalence (Figure 29, top), together `DepConstr` and `DepElim` must form an equivalence (`section` and `retraction` must hold for `f` and `g`). `DepConstr` over A and B must be equal up to transport across that equivalence (`constr_ok`), and similarly for `DepElim` (`elim_ok`). Intuitively, `constr_ok` and `elim_ok` guarantee that the transformation correctly transports dependent constructors and dependent eliminators, as doing so will preserve equality up to transport for those subterms. This makes it possible for the transformation to avoid applying `f` and `g`, instead porting terms from A directly to B.

EQUALITY To ensure coherence with equality (Figure 29, bottom), `Eta` and `Iota` must prove η and ι. That is, `Eta` must have the same definitional behavior as the dependent eliminator (`elim_eta`), and must behave like identity (`eta_ok`). Each `Iota` must prove and rewrite along the simplification (*refolding* [25]) behavior that corresponds to a case

4 Some differences in the type theory make this not quite perfect.

```
(* -------------------- Equivalence -------------------- *)

section: Π (a : A) . g (f a) = a.
retraction: Π (b : B), f (g b) = b.

constr_ok:
  ∀ j x⃗_A x⃗_B,
    x⃗_A ≡_{A≃B} x⃗_B →
    DepConstr(j, A) x⃗_A ≡_{A≃B} DepConstr(j, B) x⃗_B.

elim_ok:
  ∀ a b P_A P_B f⃗_A f⃗_B,
    a ≡_{A≃B} b →
    P_A ≡_{(A→s)≃(B→s)} P_B →
    (∀ j, f⃗_A[j] ≡_{ξ(A,P_A,j)≃ξ(B,P_B,j)} f⃗_B[j]) →
    DepElim(a, P_A) f⃗_A ≡_{(Pa)≃(Pb)} DepElim(b, P_B) f⃗_A.


(* ---------------------- Equality ---------------------- *)

elim_eta(A): Π a P f⃗, DepElim(a, P) f⃗ : P (Eta(A) a).
eta_ok(A): Π (a : A), Eta(A) a = a.

iota_ok(A):
  ∀ j P f⃗ x⃗ (Q: P(Eta(A) (DepConstr(j, A) x⃗)) → s),
    Iota(A, j, Q) :
      Q (DepElim(DepConstr(j, A) x⃗, P) f⃗) →
      Q (rew ← eta_ok(A) (DepConstr(j, A) x⃗) in
        (f⃗[j]...(DepElim(IH_0, P) f⃗)...(DepElim(IH_n, P) f⃗)...)).
```

Figure 29: Correctness criteria for a configuration to ensure that the transformation preserves equivalence (top) coherently with equality (bottom, shown for A; B is similar). `f` and `g` are defined in text. s, $\vec{f}$, $\vec{x}$, and $\vec{\mathtt{IH}}$ represent sorts, **eliminator** cases, **constructor** arguments, and **inductive hypotheses**. $\xi\ (A, P, j)$ is the type of `DepElim(A, P)` at `DepConstr(j, A)` (similarly for B). `rew` is shorthand for applying the equality eliminator.

of the dependent eliminator (`iota_ok`). This makes it possible for the transformation to avoid applying `section` and `retraction`.

CORRECTNESS With these correctness criteria for a configuration, we get the completeness result (proven in Coq ⑧) that every equivalence induces a configuration. We also obtain an algorithm for the soundness result that every configuration induces an equivalence. Both of these are what we would expect from Lambek's theorem, which states that the initial algebra and the equivalence are isomorphic to one another.

The algorithm to prove `section` is as follows (`retraction` is similar): replace `a` with `Eta(A) a` by `eta_ok(A)`. Then, induct using `DepElim` over A. For each case j, the proof obligation is to show that `g ( f a)` is equal to `a`, where `a` is `DepConstr(A, j)` applied to the non-inductive arguments (by `elim_eta(A)`). Expand the right-hand side using `Iota(A, j)`, then expand it again using `Iota(B, j)` (destructing over each `eta_ok` to apply the corresponding `Iota`). The result follows by definition of `g` and `f`, and by reflexivity.

EQUIVALENCES FROM CONFIGURATIONS The algorithm above is essentially what differencing uses for each search procedure to generate functions `f` and `g` for the automatic configurations ⑨, and also generate proofs `section` and `retraction` that these functions form an equivalence ⑩. To minimize dependencies, PUMPKIN Pi does not produce proofs of `constr_ok` and `elim_ok` directly, as stating these theorems cleanly would require either a special framework [150] or a univalent type theory [154]. If the proof engineer wishes, it is possible to prove these in individual cases ⑧, but this is not necessary in order to use PUMPKIN Pi.

4.3.4 *Search Procedures*

PUMPKIN Pi implements four search procedures for **automatic configuration** ⑥:

1. algebraic ornaments,
2. unpacking Σ types,
3. swapping constructors, and
4. moving between nested pairs and records.

As a courtesy to the reader, in this section, I detail just the first search procedure as an example. In Section 4.5, I will briefly describe the other search procedures, as well as what is needed to extend PUMPKIN Pi with new search procedures. I will also explain how the search procedures are implemented.

ALGEBRAIC ORNAMENTS The first search procedure discovers equivalences that correspond to *algebraic ornaments*. An algebraic ornament relates an **inductive type** to an indexed version of that type, where the new index is fully determined by a fold over A (I call this fold the *indexer*). For example, `vector` is exactly `list` with a new index of type `nat`, where the new index is fully determined by the `length` function (recall Figure 25 on page 67). The equivalence that we already saw in Section 4.2.2 follows from this:

```
Σ(l : list T).length l = n ≃ vector T n
```

Alternatively, a list is equivalent to a vector of *some* length:

```
list T ≃ Σ(n : nat).vector T n
```

As usual, this equivalence is made up of two functions `f` and `g`, along with proofs `section` and `retraction`. In addition, for algebraic ornaments, there is a proof of this theorem:

```
Π (l : list T), length l = π_l (f l)
```

which states that the `length` function is *coherent* with this equivalence.

In Section 4.6, I will show you a case study moving from lists to length-indexed vectors. Nominally this works by porting functions and proofs along the equivalence from Section 4.2.2, but in practice this works by chaining two automatic configurations with some human input. The first configuration uses a search procedure that discovers the equivalence between lists and vectors of *some* length above, as well as the proof of coherence. The second configuration uses a search procedure that discovers how to unpack vectors of some length to vectors of a *particular* length. The first configuration nicely demonstrates how differencing works, so let us look at it in detail.

DIFFERENCING FOR ALGEBRAIC ORNAMENTS Assume **inductive types** A and A_I, related by an **algebraic ornament** with the index of type I. In the scope of this thesis, further assume that A and I are not indexed (**dependent**) types.[5] Then there is a **type equivalence**:

```
A ≃ Σ(i : I).A_I i
```

In addition, there is an **indexer**, which is a unique fold:

```
indexer : A → I.
```

that projects the lifted index:

```
coherence : Π (a : A), indexer a = π_l (f a).
```

5 The original paper that this is from lets all of A, A_I, and I be indexed inductive types, with the new index of type I appearing anywhere within the list of indices of A_I; the implementation makes the same decision. I felt that this was very important to show in detail when I wrote that paper, since indices are often omitted, even though handling them is one of the trickiest parts of implementing an algorithm like this. In this thesis, however, I decided to simplify the presentation and assume the types are not indexed, and so the new index is the only index of A_I. I do recommend checking out the original paper if you would really like to implement something like this over indexed types—it is formalized for those who are *sufficiently fanatical*.

$$
\begin{aligned}
&A \;:=\; \mathrm{Ind}(Ty_A : \mathrm{s}_A)\{\mathrm{C}_{A_1}, \ldots, \mathrm{C}_{A_n}\} \\
&A_I \;:=\; \mathrm{Ind}(Ty_{A_I} : (\Pi(i : I).\mathrm{s}_{A_I}))\{\mathrm{C}_{A_{I_1}}, \ldots, \mathrm{C}_{A_{I_n}}\} \\
&B \;:=\; \Sigma(i : I).A_I\, i \\
\\
&\mathrm{P}_A \;:=\; \Pi(a : A).\mathrm{s}_A \\
&\mathrm{P}_{A_I} \;:=\; \Pi(i : I)(a_i : A_I\;\; i).\mathrm{s}_{A_I} \\
\\
&\forall 1 \leq j \leq n, \\
&\quad \mathrm{E}_{A_j}\;\; (p_A : \mathrm{P}_A) \;:=\; \xi(A,\;\; p_A,\;\; j) \\
&\quad \mathrm{E}_{B_j}\;\; (p_{A_I} : \mathrm{P}_{A_I}) \;:=\; \xi(A_I,\;\; p_{A_I},\;\; j)
\end{aligned}
$$

Figure 30: Common definitions. Here, $\xi\ (A,\ p_A,\ j)$ is the type of `Elim(A, p`$_A$`)` at `Constr(j, A)` (similarly for A_I).

Following existing work, I call this equivalence the *ornamental promotion isomorphism* [91]; when it holds and a **coherent** indexer exists, I say that A_I is an **algebraic ornament** of A.

In their original form, ornaments are a programming mechanism: given a type A, an ornament determines some new type A_I. Differencing inverts this process for algebraic ornaments: given types A and A_I, it searches for the configuration that induces the ornamental promotion isomorphism between them. This is possible for algebraic ornaments precisely because the indexer is extensionally unique. For example, all possible indexers for `list` and `vector` must compute the length of a list; if we were to try doubling the length instead, we would not be able to satisfy the equivalence, since no lists would map to the vectors of odd lengths.

COMMON DEFINITIONS The algorithm assumes a function `new` that determines whether a hypothesis in a case of the eliminator type of A_I is new. Figure 30 contains other common definitions, the names for which are reserved: Input type A expands to an inductive type with constructors $\{\mathrm{C}_{A_1}, \ldots, \mathrm{C}_{A_n}\}$. P_A denotes the type of the motive of the eliminator of A, and each E_{A_j} denotes the type of the eliminator for the jth constructor of A. Analogous names are also reserved for input type A_I. The type B is A_I at some index of type I.

For historical reasons, differencing generates the equivalence first, then derives the configuration, rather than the other way around. It builds on three intermediate steps: one to generate each of `indexer`, `f`, and `g`. It then uses that to build the configuration. Figure 31 shows the algorithm for generating `indexer`. The algorithms for generating `f` and `g` are similar; Figure 32 shows only the derivations for generating `f` that are different from those for generating `indexer`, and the derivations for generating `g` are omitted.

$\boxed{\Gamma \vdash (T_A,\ T_{A_I}) \Downarrow_{i_m} t}$

INDEX-MOTIVE

$$\frac{}{\Gamma \vdash (A, A_I) \Downarrow_{i_m} \lambda(a : A).I}$$

$\boxed{\Gamma \vdash (T_A,\ T_{A_I}) \Downarrow_{i_c} t}$

INDEX-CONCLUSION

$$\frac{}{\Gamma \vdash (p_A\ a,\ p_{A_I}\ i\ a_i) \Downarrow_{i_c} i}$$

INDEX-HYPOTHESIS

$$\frac{\begin{array}{l}\text{new } n_{A_I}\ b_{A_I} \\ \Gamma,\ n_{A_I} : t_{A_I} \vdash (\Pi(n_A : t_A).b_A,\ b_{A_I}) \Downarrow_{i_c} t\end{array}}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_{A_I} : t_{A_I}).b_{A_I}) \Downarrow_{i_c} t}$$

INDEX-IH

$$\frac{\Gamma \vdash (A, A_I) \Downarrow_{i_m} p \qquad \Gamma,\ n_A : p\ a \vdash (b_A,\ b_{A_I}[n_A/i]) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : p_A\ a).b_A,\ \Pi(n_{A_I} : p_{A_I}\ i\ b).b_{A_I}) \Downarrow_{i_c} \lambda(n_A : p\ a).t}$$

INDEX-PROD

$$\frac{\Gamma,\ n_A : t_A \vdash (b_A,\ b_{A_I}[n_A/n_{A_I}]) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_{A_I} : t_{A_I}).b_{A_I}) \Downarrow_{i_c} \lambda(n_A : t_A).t}$$

$\boxed{\Gamma \vdash (T_A,\ T_{A_I}) \Downarrow_{i} t}$

INDEX-IND

$$\frac{\begin{array}{l}\Gamma \vdash (A,\ A_I) \Downarrow_{i_m} p \\ \Gamma,\ p_A : \mathrm{P}_A,\ p_{A_I} : \mathrm{P}_{A_I} \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{A_{I_1}}\ p_{A_I}), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{A_{I_n}}\ p_{A_I})\} \Downarrow_{i_c} \vec{f}\end{array}}{\Gamma \vdash (A,\ A_I) \Downarrow_{i} \lambda(a : A).\mathrm{Elim}(a, p)\vec{f}}$$

Figure 31: Differencing for the indexer function.

4.3.4.1 *Differencing for the Indexer*

Differencing generates the `indexer` by traversing the types of the eliminators for A and A_I in parallel using the algorithm from Figure 31, which consists of three judgments: one to generate the **motive**, one to generate each case, and one to compose the motive and cases.

GENERATING THE MOTIVE The $(T_A,\ T_{A_I}) \Downarrow_{i_m} t$ judgment consists of only the derivation INDEX-MOTIVE, which computes the indexer motive from the types A and A_I (expanded in Figure 30). It does this by constructing a function from A to I. Consider `list` and `vector`:

```
list T := Ind (Ty_A : s) { ... }
vector T := Ind (Ty_B : Π(n :  nat).s) { ... }
```

For these types, INDEX-MOTIVE computes the motive:

```
Γ ⊢ (list T, vector T) ⇓_im λ (l : list T) . nat
```

which is the motive for the length function.

GENERATING EACH CASE The $\Gamma \vdash (T_A,\ T_{A_I}) \Downarrow_{i_c} t$ judgment generates each case of the indexer by traversing in parallel the correspond-

ing cases of the eliminator types for A and A_I. It consists of four derivations: INDEX-CONCLUSION handles base cases and conclusions of inductive cases, while INDEX-HYPOTHESIS, INDEX-IH, and INDEX-PROD recurse into products.

INDEX-HYPOTHESIS handles each new hypothesis that corresponds to a new index in an **inductive hypothesis** of an inductive case of the eliminator type for A_I. It adds the new index to the environment, then recurses into the body of only the type for which the index already exists. For example, in the inductive case of `list` and `vector`: `new` determines that `n` is the new hypothesis. INDEX-HYPOTHESIS then recurses into the body of only the `vector` case:

```
Π (l : list T) (IH_l : p_A l), p_A (cons t_l l)
Π (v : vector T n) (IH_v : p_A_I n v), p_A_I (S n) (cons n t_l l)
```

INDEX-PROD is next. It recurses into product types when the hypothesis is neither a new index nor an inductive hypothesis. Here, it runs twice, recursing into the body and substituting names until it hits the inductive hypothesis for both types:

```
Π (IH_l : p_A l), p_A (cons t_l l)
Π (IH_v : p_A_I n l), p_A_I (S n) (cons n t_l l)
```

INDEX-IH then takes over. It substitutes the new motive in the inductive hypothesis, then recurses into both bodies, substituting the new inductive hypothesis for the index in the eliminator type for A_I. Here, it substitutes the new motive for p_A in the type of IH_l, extends the environment with IH_l, then substitutes IH_l for `n`, so that it recurses on these types:

```
p_A (cons t_l l)
p_A_I (S IH_l) (cons IH_l t_l l)
```

Finally, INDEX-CONCLUSION computes the conclusion by taking the new index of the application of the motive p_{A_I}, here `S` IH_l. In total, this produces a function:

```
Γ ⊢ (Π (l : list T) (IH_l : p_A l), p_A (cons t_l l),
       Π (v : vector T n) (IH_v : p_A_I n v), p_A_I (S n) (cons n t_l l))
  ⇓_ic λ (t_l : T) (l : list T) (IH_l : (λ (l : list T) . nat) l) . S IH_l
```

that computes the length of `cons t l`.

COMPOSING THE RESULT The $\Gamma \vdash (T_A, T_{A_I}) \Downarrow_i t$ judgment consists of only INDEX-IND, which identifies the motive and each case using the other two judgments, then composes the result. In the case of `list` and `vector`, this produces a function:

```
Γ ⊢ (list T, vector T)
  ⇓_i λ (l : list T).
       Elim(l, λ (l : list T) . nat) {
         0,
         λ (t_l : T) (l : list T) (IH_l : (λ (l : list T) . nat) l) . S IH_l
       }
```

that computes the length of a list.

$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{f_m} t}$

F-MOTIVE
$$\frac{\Gamma \vdash (A,\ A_I) \Downarrow_i \pi}{\Gamma \vdash (A,\ A_I) \Downarrow_{f_m} \lambda(a : A).A_I\ (\pi\ \vec{i_a}\ a)}$$

$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{f_c} t}$

F-CONCLUSION
$$\frac{}{\Gamma \vdash (p_A\ a,\ p_{A_I}\ i\ a_i) \Downarrow_{f_c} a_i}$$

F-IH
$$\frac{\begin{array}{l}\Gamma \vdash (A,\ A_I) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ A_I) \Downarrow_{f_m} p \\ \Gamma,\ n_A : p\ a \vdash (b_A,\ b_{A_I}[n_A/a_i][\pi\ a/i]) \Downarrow_{p_c} t\end{array}}{\begin{array}{l}\Gamma \vdash (\Pi(n_A : p_A\ a).b_A,\ \Pi(n_{A_I} : p_{A_I}\ i\ a_i).b_{A_I}) \\ \quad \Downarrow_{p_c} \lambda(n_A : p\ a).t\end{array}}$$

$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_f t}$

F-IND
$$\frac{\begin{array}{l}\Gamma \vdash (A,\ A_I) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ A_I) \Downarrow_{f_m} p \\ \Gamma,\ p_A : \mathrm{P}_A,\ p_{A_I} : \mathrm{P}_{A_I} \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{A_{I_1}}\ p_{A_I}), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{A_{I_n}}\ p_{A_I})\} \Downarrow_{p_c} \vec{f}\end{array}}{\Gamma \vdash (A,\ A_I) \Downarrow_p \lambda(a : A).\exists\ (\pi\ a)\ (\mathrm{Elim}(a, p)\vec{f})}$$

Figure 32: Differencing for `f`.

4.3.4.2 *Differencing for the Configuration*

As mentioned earlier, for historical reasons, differencing in PUMPKIN Pi discovers the equivalence parts `f` and `g` first, then uses those functions to discover the configuration, rather than the other way around. It also proves that these functions form an equivalence, and that the indexer is coherent with the equivalence.

DISCOVERING THE EQUIVALENCE PARTS Figure 32 shows the interesting derivations for the judgment $(T_A,\ T_B) \Downarrow_f t$ that searches for `f`: F-MOTIVE identifies the motive as B with a new index (which it computes using `indexer`, denoted by metavariable π). When F-IH recurses, it substitutes the inductive hypothesis for the term rather than for its index, and it substitutes the new index (which it also computes using `indexer`) inside of that term. F-CONCLUSION returns the entire term, rather than its index. Finally, F-IND not only recurses into each case, but also packs the result into an **existential**.

The omitted derivations to difference for `g` are similar, except that the domain and range are switched. Consequentially, `indexer` is never needed; G-MOTIVE removes the index rather than inserting it, and G-IH no longer substitutes the index. Additionally, G-HYPOTHESIS adds the hypothesis for the new index rather than skipping it, and G-IND eliminates over the projection rather than packing the result.

DERIVING THE CONFIGURATION `DepConstr` and `DepElim` over A are just the standard constructors and eliminators for A. To derive

`DepConstr` over B, differencing takes each constructor of A, applies `f` to the conclusion of that constructor, and normalizes the result. It then ports the hypotheses of the resulting constructor to use B in place of A, and drops the remaining applications of `f` and the indexer in the body, replacing them instead with the projections π_r and π_l, respectively.

For example, earlier, letting A be lists, A_I be vectors, and B be vectors of some length, I noted that the empty constructor of B packs the constructor of A_I into an existential:

```
DepConstr(0, B): B := ∃(Constr(0, nat)) (Constr(0, vector T)).
```

This is the same as applying the function `f` that PUMPKIN Pi derives to `DepConstr(0, A)`, and then normalizing the result. On the other hand, the `cons` constructor over B not just packs the result into an existential, but also takes B itself as an argument rather than A:

```
DepConstr(1, B) : T → B → B :=
  λ (t : T) (l : B) .
    ∃ ((Constr (1, nat)) (π_l l))
      ((Constr (1, vector T)) t (π_l l) (π_r l))).
```

To derive this, differencing applies `f` to the conclusion of the constructor of A and normalizes the result:

```
λ (t : T) (l : A) .
  ∃ ((Constr (1, nat)) (π l))
    ((Constr (1, vector T)) t (π l) (f l))).
```

It then lifts the hypothesis of type `list T`, and removes remaining references to the indexer π and to `f`, replacing them instead with the projections π_l and π_r.

Deriving `DepElim` over B works similarly, except that it lifts not just the hypotheses of types A, but also the motive and inductive hypotheses. This produces the dependent eliminator from earlier:

```
DepElim(s, P) { f_0 f_1 } : P (∃ (π_l s) (π_r s)) :=
  Elim(π_r s, λ(n : nat)(v : vector T n) . P (∃ n v)) {
    f_0,
    (λ (t : T) (n : nat) (v : vector T n) . f_1 t (∃ n v))
  }.
```

Differencing discovers `Eta` and `Iota` directly. For any **algebraic ornament**, `Eta` is the standard η-expansion for Σ types:

```
Eta(B) := λ(b : B).∃ (π_l b) (π_r b).
```

Each `Iota` follows by rewriting by reflexivity, since A and A_I have the same inductive structure.

PROVING CORRECTNESS In the end, PUMPKIN Pi generates proofs of `section` and `retraction`, as well as `coherence`. This proves the correctness property that the configuration induces an equivalence, thereby increasing confidence in the output of differencing. The proof of **coherence** follows by reflexivity, thanks to the construction of `f` applying the **indexer** as the left projection. The proofs of `section` and `retraction` follow from the algorithm presented earlier.

$$\boxed{\Gamma \vdash t \Uparrow t'}$$

DEP-ELIM
$$\frac{\Gamma \vdash a \Uparrow b \qquad \Gamma \vdash p_a \Uparrow p_b \qquad \Gamma \vdash \vec{f}_a \Uparrow \vec{f}_b}{\Gamma \vdash \mathrm{DepElim}(a,\ p_a)\vec{f}_a \Uparrow \mathrm{DepElim}(b,\ p_b)\vec{f}_b}$$

DEP-CONSTR
$$\frac{\Gamma \vdash \vec{t}_a \Uparrow \vec{t}_b}{\Gamma \vdash \mathrm{DepConstr}(j,\ A)\ \vec{t}_a \Uparrow \mathrm{DepConstr}(j,\ B)\ \vec{t}_b}$$

ETA
$$\frac{}{\Gamma \vdash \mathrm{Eta}(A) \Uparrow \mathrm{Eta}(B)}$$

IOTA
$$\frac{\Gamma \vdash q_A \Uparrow q_B \qquad \Gamma \vdash \vec{t_A} \Uparrow \vec{t_B}}{\Gamma \vdash \mathrm{Iota}(j,\ A,\ q_A)\ \vec{t_A} \Uparrow \mathrm{Iota}(j,\ B,\ q_B)\ \vec{t_B}}$$

EQUIVALENCE
$$\frac{}{\Gamma \vdash A \Uparrow B}$$

CONSTR
$$\frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{t} \Uparrow \vec{t'}}{\Gamma \vdash \mathrm{Constr}(j,\ T)\ \vec{t} \Uparrow \mathrm{Constr}(j,\ T')\ \vec{t'}}$$

IND
$$\frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{C} \Uparrow \vec{C'}}{\Gamma \vdash \mathrm{Ind}(Ty : T)\vec{C} \Uparrow \mathrm{Ind}(Ty : T')\vec{C'}}$$

APP
$$\frac{\Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash f t \Uparrow f' t'}$$

ELIM
$$\frac{\Gamma \vdash c \Uparrow c' \qquad \Gamma \vdash Q \Uparrow Q' \qquad \Gamma \vdash \vec{f} \Uparrow \vec{f'}}{\Gamma \vdash \mathrm{Elim}(c, Q)\vec{f} \Uparrow \mathrm{Elim}(c', Q')\vec{f'}}$$

LAM
$$\frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \lambda(t : T).b \Uparrow \lambda(t' : T').b'}$$

PROD
$$\frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \Pi(t : T).b \Uparrow \Pi(t' : T').b'}$$

VAR
$$\frac{v \in \mathrm{Vars}}{\Gamma \vdash v \Uparrow v}$$

Figure 33: Transformation for transporting terms across $A \simeq B$ with configuration `((DepConstr, DepElim), (Eta, Iota))`.

4.4 TRANSFORMATION

At the heart of PUMPKIN Pi is a configurable **proof term transformation** that automatically **transports** proofs across **equivalences** ④. Figure 33 shows this proof term transformation. The transformation $\Gamma \vdash t \Uparrow t'$ takes some term t defined over the old version of a type to a new term t' defined over the new version of the type. It is parameterized over equivalent types A and B (EQUIVALENCE) as well as the configuration for that equivalence. It assumes η-expanded functions. It implicitly constructs an updated context Γ' in which to interpret t', but this is not needed for computation.

The proof term transformation is (perhaps deceptively) simple by design: it moves the bulk of the work into the configuration, and represents the configuration explicitly. This configuration either comes from automatic configuration (like the search procedure in the previ-

```
(* 1: original *)
λ (T: Type) (l m: Old.list T) .
 Elim(l, λ(l: Old.list T).Old.list T → Old.list T)) {
   (λ m . m),
   (λ t _ IHl m . Constr(1, Old.list T) t (IHl m))
 } m.

(* 2: unified *)
λ (T: Type) (l m: A) .
 DepElim(l, λ(l: A).A → A)) {
   (λ m . m)
   (λ t _ IHl m . DepConstr(1, A) t (IHl m))
 } m.

(* 3: transformed *)
λ (T: Type) (l m: B) .
 DepElim(l, λ(l: B).B → B)) {
   (λ m . m)
   (λ t _ IHl m . DepConstr(1, B) t (IHl m))
 } m.

(* 4: reduced *)
λ (T: Type) (l m: New.list T) .
 Elim(l, λ(l: New.list T).New.list T → New.list T)) {
   (λ t _ IHl m . Constr(0, New.list T) t (IHl m)),
   (λ m . m)
 } m.
```

Figure 34: Swapping cases of the append function, from top to bottom, the input term: 1) unmodified, 2) unified with the configuration, 3) ported to the new version of the type, and 4) reduced to the output.

ous section), or directly from the proof engineer. Of course, in both of these cases, typical proof terms in Coq do not apply these configuration terms explicitly. PUMPKIN Pi does some additional work using *unification heuristics* to get real proof terms into this format before running the transformation (Section 4.4.1). It then runs the proof term transformation, which transports proofs across the equivalence that corresponds to the configuration, replacing A with B (Section 4.4.2).

4.4.1 *Unification Heuristics*

The transformation does not fully describe the search procedure for transforming terms that PUMPKIN Pi implements. Before running the transformation, PUMPKIN Pi unifies subterms with particular A (fixing parameters and indices), and with applications of configuration terms over A. The transformation then transforms configuration terms over A to configuration terms over B. Reducing the result produces the output term defined over B.

Figure 34 shows this with the list append function ++ from Section 4.1. To update ++ (top), PUMPKIN Pi unifies `Old.list T` with A, and `Constr` and `Elim` with `DepConstr` and `DepElim` (second from the top). After unification, the transformation recursively substitutes B for A, which moves `DepConstr` and `DepElim` to construct and eliminate over the updated type (second from the bottom). This reduces to a term with swapped constructors and cases over `New.list T` (bottom).

In this case, unification is straightforward. This can be more challenging when configuration terms are dependent. This is especially pronounced with definitional `Eta` and `Iota`, which typically are implicit (reduced) in real code. To handle this, PUMPKIN Pi implements custom *unification heuristics* for each search procedure that unify subterms with applications of configuration terms, and that instantiate **parameters** and **dependent** indices in those subterms ⑥. The transformation in turn assumes that all existing parameters and indices are determined and instantiated by the time it runs.

PUMPKIN Pi falls back to Coq's unification for manual configuration and when these custom heuristics fail. When even Coq's unification is not enough, PUMPKIN Pi relies on proof engineers to provide hints in the form of annotations ⑤.

The unification heuristics largely move the burden of undecidability outside of the details of the proof term transformation. Most notably, since unification heuristics are abstracted from the transformation itself, this makes it relatively simple to hook in a human-assisted workflow using annotations. Still, the PUMPKIN Pi transformation does struggle sometimes with termination. I describe this more in Section 4.5.

4.4.2 *Specifying a Correct Transformation*

The implementation of this transformation in PUMPKIN Pi produces a term that Coq type checks, and so does not add to the **TCB**. As PUMPKIN Pi is an engineering tool, there is no need to formally prove the transformation correct, though doing so would be satisfying. The goal of such a proof would be to show that if $\Gamma \vdash t \Uparrow t'$, then t and t' are equal up to transport, and t' refers to B in place of A. The key steps in this transformation that make this possible are porting terms along the configuration (DEP-CONSTR, DEP-ELIM, ETA, and IOTA). The rest is straightforward. For metatheoretical reasons, without additional axioms, a proof of this theorem in Coq can only be approximated [150]. It would be possible to generate per-transformation proofs of correctness, but this does not serve an engineering need.

4.5 IMPLEMENTATION

Like the **PUMPKIN** prototype, **PUMPKIN Pi** is also included in the **PUMPKIN PATCH plugin** suite by default. As with PUMPKIN, the latest version supports Coq 8.8, with Coq 8.9.1 support in a branch. This section describes the implementation of the core functionality of the PUMPKIN Pi plugin (Section 4.5.1), along with important features for workflow integration (Section 4.5.2). The interested reader can follow along in the repository.

4.5.1 *Tool Details*

The implementation (Section 4.5.1.1) of the `Repair` command looks up the configuration for the types, invoking a search procedure for automatic configuration if relevant. The configuration, whether obtained manually or automatically, is cached for future calls to `Repair`. The implementations of the differencing search procedures for automatic configuration (Section 4.5.1.2) have freedom over the details of how they are implemented, as long as they return configurations in the end. The transformation (Section 4.5.1.3) operates directly over proof terms in Gallina, with the cached configuration also defined as proof terms in Gallina. As with PUMPKIN, the PUMPKIN Pi extension to PUMPKIN PATCH does not extend the **TCB** in any way (Section 4.5.1.4).

4.5.1.1 *The Command*

The implementation of PUMPKIN Pi exposes the repair workflow to proof engineers through the `Repair` command. This invokes the workflow from Figure 22 on page 65. When the proof engineer invokes the **command**:

```
Repair A B in old_proof as new_proof.
```

The first step—CONFIGURE—looks up A and B in a cache of configurations. If it finds one, it uses that; otherwise, it runs the differencing search procedures for **automatic configuration**. If that fails, it prompts the proof engineer to supply a configuration **manually**. Proof engineers can supply manual configurations by defining them as Gallina terms and providing them to the `Configure Repair` command.

Once PUMPKIN Pi has found a configuration, the second step—TRANSFORM—runs the transformation. It transforms `old_proof` into some new proof term, then defines the new proof term as `new_proof`. It is also possible to ask PUMPKIN Pi to automatically generate a name. The final step—DECOMPILE—runs in the end to suggest a proof script.

There are a few other useful commands in PUMPKIN Pi that are detailed in the repository. Notably, the `Repair Module` command uses higher-order functions written by **Nate** to operate over entire modules at once and define new, repaired versions of those modules. The `Preprocess` command converts functions to use eliminators, so that the transformation can assume **primitive eliminators**. The commands `Lift` and `Lift Module` skip the decompilation step, when only the term is desired in the end. Several options make it possible to control whether PUMPKIN Pi generates correctness proofs, as well as how aggressively it runs optimizations. The repository includes more information on all of these.

4.5.1.2 *Differencing*

As mentioned in Section 4.3.4, **differencing** inside of PUMPKIN Pi implements four search procedures for automatic configuration ⑥:

1. **algebraic ornaments**,
2. unpacking Σ types,
3. swapping constructors, and
4. moving between nested pairs and records.

I already detailed the first of these in Section 4.3.4. The second is often used in combination with the first, to get from equivalences like:

$$A \simeq \Sigma(i : I).A_I\ \ i$$

to equivalences like:

$$\Sigma(a : A).\pi\ \ a = i \simeq A_I\ \ i$$

where π is the indexer. The first two configurations are used in combination, for example, to repair proofs in response to the change from lists to vectors of a particular length in Figure 25, as long as the proof engineer proves the missing length invariant. This differencing algorithm is implemented by simply instantiating the types A and B of a generic configuration that can be defined inside of Coq directly.

The third configuration—invoked in the example in Section 4.1—constructs a swap map that maps the constructors of A to the swapped constructors of B, then otherwise runs an algorithm much like the algorithm for algebraic ornaments. When there are multiple possible swap maps, it returns an ordered list of possible mappings to choose from, and prompts the proof engineer to select one. It also lets the proof engineer supply the swap map directly, and from that can derive the configuration, the equivalence, and the proof. The fourth search procedure is similar to the search procedure for algebraic ornaments, but with some additional logic to handle nested tuples.

All search procedures have the historical detail of defining the equivalence first, then deriving the configuration from it, rather than the other way around. For simplicity, the search procedures currently implemented inside of Pumpkin Pi also make some syntactic assumptions about the input and output types. The details of these restrictions can be found inside of the repository.

Defining new search procedures comes with a lot of freedom, as long as the search procedures produce a configuration in the end. I found it simple to add configuration four in a matter of a few days in response to a request by an industrial proof engineer, reusing existing functions defined in other search procedures. I would not expect the average proof engineer to be able to do the same, and I have not yet asked anyone else to write a search procedure to gauge the effort involved for others. It is notable that manual configuration is always an option when search procedures do not exist, but the results in Section 4.6 suggest that automatic configuration is very helpful for saving work for proof engineers, and so worth implementing for useful classes of equivalences.

4.5.1.3 *Transformation*

The implementation ④ of the **transformation** from Section 4.4 operates over terms in Gallina. It takes as input two types A and B, along with a configuration that induces an equivalence between them. A and B may not necessarily be the inputs to differencing—in the case of algebraic ornaments, for example, differencing takes A and A_I as inputs, but the transformation takes A and B as inputs, where B is $\Sigma(i : I).A_I\ i$.

The unification heuristics run before the transformation ⑥, and in the end return which transformation derivation to run, and with which arguments ⑫. For the most part, the derivations are implemented in a way that corresponds to the derivations in Figure 33 on page 85, but with a few differences highlighted below.

FROM CIC$_\omega$ TO GALLINA The implementation ④ of the transformation handles language differences to scale from **CIC$_\omega$** to Gallina. For example, recall that the transformation assumes **primitive elimi-**

nators, while Gallina implements eliminators using pattern matching and fixpoints. To handle terms that use these features, **Nate** implemented a `Preprocess` command in PUMPKIN Pi that translates these terms into corresponding eliminator applications. This command can preprocess a definition (like `length` from Figure 7 on page 17) or an entire module (like `List`, as shown in `ListToVect.v`) for lifting. It currently supports fixpoints that are structurally recursive on only immediate substructures.[6] To translate such a fixpoint, it first extracts a motive, then generates each case by partially reducing the function's body under a hypothetical context for the constructor arguments. The implementation documents this and other language differences.

OPTIMIZATIONS The implementation of the transformation includes a number of cases that correspond to optimizations not found in the original transformation. For example, the transformation assumes that all terms are fully η-expanded. Sometimes, however, η-expansion is not necessary. For efficiency, rather than fully η-expand ahead of time, PUMPKIN Pi η-expands lazily, only when it is necessary for correctness. The `LazyEta` optimization implements this optimization. The code denotes all optimizations explicitly (4) and explains them in detail in comments (12). Section 4.5.2.2 describes some other optimizations included for the sake of integration with proof engineering workflows.

TERMINATION When a subterm unifies with a configuration term, this suggests that PUMPKIN Pi *can* transform the subterm, but it does not necessarily mean that it *should*. In some cases, doing so would result in nontermination. For example, if B is a refinement of A, then the transformation can always run EQUIVALENCE over and over again, forever. PUMPKIN Pi thus includes some simple termination checks in the code (12).

INTENT Even when termination is guaranteed, whether to transform a subterm depends on the proof engineer's intent. That is, PUMPKIN Pi automates the case of porting *every* A to B, but proof engineers sometimes wish to port only *some* As to Bs. PUMPKIN Pi has some support for this using an interactive workflow (13), with plans for automatic support in the future.

4.5.1.4 *Trusted Computing Base*

As with PUMPKIN, PUMPKIN Pi is implemented as a Coq plugin, and produces terms that Coq type checks in the end. PUMPKIN Pi does not

6 This is enough to preprocess many practical terms, including the entire `List` module. But it is not as general as it could be [64, 34]. A more general translation may help PUMPKIN Pi support more terms, and discussions with Coq developers a couple of years ago suggested that the implementation of such a translation building on work from the equations [147] plugin was in progress. I do not know the current status.

modify the type checker, and furthermore does not add any axioms, so it does not increase the **TCB**. This is perhaps even more notable for PUMPKIN Pi, since all other implementations of **transport** across equivalences that I am aware of at least sometimes add axioms. Since PUMPKIN Pi implements transport as a proof term transformation, it is able to circumvent this and not increase the TCB in any way.

4.5.2 *Workflow Integration*

So far, I have described the core functionality of PUMPKIN Pi. But PUMPKIN Pi has many additional features for the sake of integration with typical proof engineering workflows. Most notable of these is the decompiler from Gallina to Ltac (Section 4.5.2.1), which helps PUMPKIN Pi produce a suggested proof script that the proof engineer can maintain in the end. This and other features help PUMPKIN Pi reach real proof engineers (Section 4.5.2.2).

4.5.2.1 *Decompiling to Tactics*

Transform produces a **proof term**, while the proof engineer typically writes and maintains proof scripts made up of **tactics**. PUMPKIN Pi improves usability thanks to the realization that, since Coq's proof term language **Gallina** is very structured, it is possible to decompile these Gallina terms to suggested **Ltac** proof scripts for the proof engineer to maintain.

Decompile implements a prototype of this translation (11): it translates a proof term to a suggested proof script that attempts to prove the same theorem the same way. Note that this problem is not well defined: while there is always a proof script that works (applying the proof term with the `apply` tactic), the result is often qualitatively unreadable. This is the baseline behavior to which the decompiler defaults. The goal of the decompiler is to improve on that baseline as much as possible, or else suggest a proof script that is close enough to correct that the proof engineer can manually massage it into something that works and is maintainable.

Decompile achieves this in two passes: The first pass decompiles proof terms to proof scripts that use a predefined set of tactics. The second pass improves on suggested tactics by simplifying arguments, substituting tacticals, and using hints like custom tactics and decision procedures.

FIRST PASS: BASIC PROOF SCRIPTS The first pass takes **Gallina** terms and produces tactics in **Ltac**. Ltac can be confusing to reason about, since Ltac tactics can refer to Gallina terms, and the semantics of Ltac depends both on the semantics of Gallina and on the implementation of proof search procedures written in OCaml. To give a sense of how the first pass works without the clutter of these details,

$\langle v \rangle \in$ Vars, $\langle t \rangle \in \text{CIC}_\omega$

$\langle p \rangle$ ::= intro $\langle v \rangle$
| rewrite $\langle t \rangle$ $\langle t \rangle$
| symmetry
| apply $\langle t \rangle$
| induction $\langle t \rangle$ $\langle t \rangle$ { $\langle p \rangle, \ldots, \langle p \rangle$ }
| split { $\langle p \rangle, \langle p \rangle$ }
| left
| right
| $\langle p \rangle$. $\langle p \rangle$

Figure 35: Qtac syntax.

I start by defining a mini decompiler that implements a simplified version of the first pass. I then explain how **RanDair** scaled this to the implementation.

The mini decompiler takes **CIC$_\omega$** terms and produces tactics in a mini version of Ltac which I call Qtac. The syntax for Qtac is in Figure 35. Qtac includes hypothesis introduction (`intro`), rewriting (`rewrite`), symmetry of equality (`symmetry`), application of a term to prove the goal (`apply`), induction (`induction`), case splitting of conjunctions (`split`), constructors of disjunctions (`left` and `right`), and composition (`.`). Unlike in Ltac, `induction` and `rewrite` take a motive explicitly (rather than relying on unification), and `apply` creates a new subgoal for each function argument.

The semantics for the mini decompiler $\Gamma \vdash t \Rightarrow p$ are in Figure 36 (assuming $=$, `eq_sym`, $\wedge$, and $\vee$ are defined as in Coq). As with the real decompiler, the mini decompiler defaults to the proof script that applies the entire proof term with `apply` (Base). Otherwise, it improves on that behavior by recursing over the proof term and constructing a proof script using a predefined set of tactics.

For the mini decompiler, this is straightforward: Lambda terms become introduction (Intro). Applications of `eq_sym` become symmetry of equality (Symmetry). Constructors of conjunction and disjunction map to the respective tactics (Split, Left, and Right). Applications of equality eliminators compose symmetry (to orient the rewrite direction) with rewrites (Rewrite), and all other applications of eliminators become induction (Induction). The remaining applications become apply tactics (Apply). In all cases, the decompiler recurses, breaking into cases, until only the Base case holds.

While the mini decompiler is very simple, only a few small changes were needed for **RanDair** to move this to Coq. The generated proof term of `rev_app_distr` from Section 4.1, for example, consists only of induction, rewriting, simplification, and reflexivity (solved by `auto`). Figure 37 shows the proof term for the base case of `rev_app_distr` alongside the proof script that Pumpkin Pi suggests. This script is fairly low-level and close to the proof term, but it is already something

$\boxed{\Gamma \vdash t \Rightarrow p}$

INTRO
$$\frac{\Gamma,\ n : T \vdash b \Rightarrow p}{\Gamma \vdash \lambda(n : T).b \Rightarrow \text{intro } n.\ p}$$

SYMMETRY
$$\frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \texttt{eq\_sym}\ H \Rightarrow \text{symmetry. } p}$$

SPLIT
$$\frac{\Gamma \vdash l \Rightarrow p \qquad \Gamma \vdash r \Rightarrow q}{\Gamma \vdash \text{Constr}(0,\ \wedge)\ l\ r \Rightarrow \text{split}\{p, q\}.}$$

LEFT
$$\frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \text{Constr}(0,\ \vee)\ H \Rightarrow \text{left. } p}$$

RIGHT
$$\frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \text{Constr}(1,\ \vee)\ H \Rightarrow \text{right. } p}$$

REWRITE
$$\frac{\Gamma \vdash H_1 : x = y \qquad \Gamma \vdash H_2 \Rightarrow p}{\Gamma \vdash \text{Elim}(H_1,\ P)\{x,\ H_2,\ y\} \Rightarrow \text{symmetry. rewrite } P\ H_1.\ p}$$

INDUCTION
$$\frac{\Gamma \vdash \vec{f} \Rightarrow \vec{p}}{\Gamma \vdash \text{Elim}(t,\ P)\ \vec{f} \Rightarrow \text{induction } P\ t\ \vec{p}}$$

APPLY
$$\frac{\Gamma \vdash t \Rightarrow p}{\Gamma \vdash ft \Rightarrow \text{apply } f.\ p}$$

BASE
$$\frac{}{\Gamma \vdash t \Rightarrow \text{apply } t}$$

Figure 36: Qtac decompiler semantics.

```
fun (y0 :  list A)[1] =>
  list_rect[2] _ _  (fun a l H[2] =>
    eq_ind_r[3] _ eq_refl[4] (app_nil_r (rev l) (a::[]))[3])
    eq_refl[5]
    y0[2]

- intro y0.[1] induction y0 as [a l H|].[2]
  + simpl.  rewrite app_nil_r.[3] auto.[4]
  + auto.[5]
```

Figure 37: Proof term (top) and decompiled proof script (bottom) for the base case of `rev_app_distr` (Section 4.1), with corresponding terms and tactics grouped by color & number.

that the proof engineer can step through to understand, modify, and maintain. There are few differences from the mini decompiler needed to produce this, for example handling of rewrites in both directions (`eq_ind_r` as opposed to `eq_ind`), simplifying rewrites, and turning applications of `eq_refl` into `reflexivity` or `auto`.

SECOND PASS: BETTER PROOF SCRIPTS The implementation of **Decompile** first runs something like the mini decompiler, then modifies the suggested tactics to produce a more natural proof script (11). For example, it cancels out sequences of `intros` and `revert`, inserts semicolons, and removes extra arguments to `apply` and `rewrite`. It can also take tactics from the proof engineer (like part of the old proof script) as hints, then iteratively replace tactics with those hints, checking for correctness. This makes it possible for suggested scripts to include custom tactics and decision procedures.

FROM QTAC TO LTAC The mini decompiler assumes more predictable versions of `rewrite` and `induction` than those in Coq. **Decompile** includes additional logic to reason about these tactics (11). For example, Qtac assumes that there is only one `rewrite` direction. Ltac has two rewrite directions, and so the decompiler infers the direction from the motive.

Qtac also assumes that both tactics take the **motive** explicitly, while in Coq, both tactics infer the motive automatically. Consequentially, Coq sometimes fails to infer the correct motive. To handle induction, the decompiler strategically uses `revert` to manipulate the goal so that Coq can better infer the motive. To handle rewrites, it uses `simpl` to simplify the goal before rewriting. Neither of these approaches is guaranteed to work, so the proof engineer may sometimes need to tweak the suggested proof script appropriately. Even if **RanDair** passes Coq's induction principle an explicit motive, Coq still sometimes fails due to unrepresented assumptions. Long term, using another tactic like `change` or `refine` before applying these tactics may help with cases for which Coq cannot infer the correct motive.

FROM CIC$_\omega$ TO GALLINA Scaling the decompiler to Gallina introduces let bindings, which are generated by tactics like `rewrite in`, `apply in`, and `pose`. **Decompile** implements (11) support for the tactics `rewrite in` and `apply in` similarly to how it supports `rewrite` and `apply`, except that it ensures that the unmanipulated hypothesis does not occur in the body of the let expression, it swaps the direction of the rewrite, and it recurses into any generated subgoals. In all other cases, it uses `pose`, a catch-all for let bindings.

FORFEITING SOUNDNESS While there is a way to always produce a correct proof script, by default, **Decompile** deliberately forfeits soundness to suggest more useful tactics. For example, it may suggest `induction`, but leave motive inference to the proof engineer. I have found these suggested tactics easier to work with (Section 4.6). In the case the suggested proof script is not quite correct, it is still possible to use the generated proof term directly. **RanDair** has also implemented a simplified sound version of the decompiler in a branch.

PRETTY PRINTING After decompiling proof terms, the implementation of **Decompile** pretty prints the result (11). Like the mini decompiler, **Decompile** represents its output using a predefined grammar of Ltac tactics, albeit one that is larger than Qtac, and that also includes tacticals. It maintains the recursive proof structure for formatting. PUMPKIN Pi keeps all output terms from **Transform** in the Coq environment in case the decompiler does not succeed. Once the proof engineer has the new proof, she can remove the old one.

4.5.2.2 *Reaching Real Proof Engineers*

The goal of workflow integration is to reach real proof engineers. Many of my design decisions in implementing PUMPKIN Pi were informed by my partnership with an industrial proof engineer (Section 4.6). For example, the proof engineer rarely had the patience to wait more than ten seconds for PUMPKIN Pi to port a term, so I implemented optional aggressive caching, even caching intermediate subterms encountered while running the transformation (14). I also added a cache to tell PUMPKIN Pi not to δ-reduce certain terms (14). With these caches, the proof engineer found PUMPKIN Pi efficient enough to use on a code base with tens of thousands of lines of code and proof.

The experiences of proof engineers also inspired new features. For example, I implemented a search procedure to generate custom eliminators to help reason about types like Σ`(l : list T).length l = n` by reasoning separately about the projections (15). I added informative error messages (22) to help the proof engineer distinguish between user errors and bugs. **Nate** implemented machinery for whole module processing to handle entire libraries at once. These features helped with workflow integration.

Class	Conf.	Examples	Sav.	Repair	Search
Alg. Ornaments	♡♡♡	List to Vec., hs-to-coq ③	☺	Pi, UP	Pi
		List to Vec., Std. Lib. ⑯	☺	Pi, UP	Pi
Unpack Sigma	♡♡♡	Vec., hs-to-coq ③	☺	Pi, UP	Pi
Tuples & Records	♡♡♡	Simple Records ⑬	☺	Pi, UP	Pi
		Param. Records ⑰	☺	Pi, UP	Pi
		Industrial Use ⑱	☺	Pi, UP	Pi
Perm. Constructors	♡♡♡	List, Std. Lib. ①	☺	Pi, UP	Pi
		Modify PL, **REPLica** ①	😐	Pi, UP	Pi
		Large Ambig. Enum ①	😐	Pi, UP	Pi
Add Constructors	♡♡	Extend PL, **REPLica** ⑲	☹	Pi	Pi (part)
Factor Constructors	♡	Reviewer ②	☺	Pi, UP	None
Perm. Hypotheses	♡	**Anders** ⑳	☹	Pi, UP	None
Change Structure	♡	Unary to Bin., MB ⑤	😐	Pi, MB	None
		Vec. to Fin., **Anders** ㉑	☺	Pi	None

Table 1: Some changes using PUMPKIN Pi (left to right): class of changes, kind of configuration (♡♡♡ if automatic, ♡♡ if mixed automatic and manual, and ♡ if manual), examples, whether using PUMPKIN Pi saved development time relative to reference manual repairs (☺ if yes, 😐 if comparable, ☹ if no), and Coq tools I know of that support repair along (Repair) or automatic proof of (Search) the equivalence corresponding to each example. Besides PUMPKIN Pi (Pi), tools considered are the Univalent Parametricity (UP) white-box transformation [151] and the tool from Magaud & Bertot 2000 [105] (MB). PUMPKIN Pi is the only one that suggests tactics. More nuanced comparisons to these and more are in Chapter 5.

4.6 RESULTS

PUMPKIN Pi is flexible and useful. It can help and in fact has helped proof engineers save work on a variety of real proof repair scenarios (Section 4.6.1). In addition, the approach taken has measurable benefits in terms of both work savings and performance relative to a comparable tool for the class of changes on which **Nate** and I have done an extended evaluation (Section 4.6.2).

4.6.1 PUMPKIN P*i Eight Ways*

This section summarizes eight case studies using PUMPKIN Pi, corresponding to the eight rows in Table 1. These **case studies** highlight PUMPKIN Pi's flexibility in handling diverse scenarios, the success of automatic configuration for better workflow integration, the preliminary success of the prototype decompiler, and clear paths to better serving proof engineers. Detailed walkthroughs are in the code.

ALGEBRAIC ORNAMENTS: LISTS TO PACKED VECTORS PUMPKIN Pi implements a search procedure for automatic configuration of **algebraic ornaments**, detailed in Section 4.3.4. In file ③, I used this to port functions and a proof from lists to vectors of *some* length, since `list T` $\simeq$ Σ`(n : nat).vector T n`. The decompiler helped me write proofs in the order of hours that I had found too hard to write by hand, though the suggested tactics did need massaging.

UNPACK SIGMA TYPES: VECTORS OF PARTICULAR LENGTHS In the same file ③, I then ported functions and proofs to vectors of a *particular* length, like `vector T n`. I supported this in PUMPKIN Pi by chaining the previous change with an automatic configuration for unpacking Σ types. By composition, this transported proofs across the equivalence from Section 4.2.2.

Two tricks helped with workflow integration for this change: 1) have the search procedure view `vector T n` as Σ`(v : vector T m). n = m` for some `m`, then let PUMPKIN Pi instantiate those equalities via **unification heuristics**, and 2) generate a custom **eliminator** for combining list terms with length invariants. The resulting workflow works not just for lists and vectors, but for any algebraic ornament, automating otherwise manual effort. The suggested tactics were helpful for writing proofs in the order of hours that I had struggled with manually over the course of days, but only after massaging. More effort is needed to improve tactic suggestions for **dependent types**.

TUPLES & RECORDS: INDUSTRIAL USE An industrial proof engineer at the company Galois has been using PUMPKIN Pi in proving correct an implementation of the TLS handshake protocol. Galois had been using a custom solver-aided verification language to prove correct C programs, but had found that at times, the constraint solvers got stuck. They had built a compiler that translates their language into Coq's specification language Gallina, that way proof engineers could finish stuck proofs interactively using Coq. However, due to language differences, they had found the generated Gallina programs and specifications difficult to work with.

The proof engineer used PUMPKIN Pi to port the automatically generated functions and specifications to more human-readable functions and specifications, wrote Coq proofs about those functions and specifications, then used PUMPKIN Pi to port those proofs back to proofs about the original functions and specifications. So far, they have used at least three automatic configurations, but they most often used an automatic configuration for porting compiler-produced anonymous tuples to named records, as in file ⑱. The workflow was a bit nonstandard, so there was little need for tactic suggestions. The proof engineer reported an initial time investment learning how to use PUMPKIN Pi, followed by later returns.

```
Inductive Term : Set :=
| Var : Id → Term
| Int : Z → Term
| Eq : Term → Term → Term
| Plus : Term → Term → Term
| Times : Term → Term → Term
| Minus : Term → Term → Term
| Choose : Id → Term → Term.
```

```
Inductive Term : Set :=
| Var : Id → Term
| Bool : Id → Term
| Eq : Term → Term → Term
| Int : Z → Term
| Plus : Term → Term → Term
| Times : Term → Term → Term
| Minus : Term → Term → Term
| Choose : Id → Term → Term.
```

Figure 38: A simple language (left) and the same language with two swapped constructors and an added constructor (right).

PERMUTE CONSTRUCTORS: MODIFYING A LANGUAGE The swapping example from Section 4.1 was inspired by benchmarks from the REPLICA user study of proof engineers. A change from one of the benchmarks is in Figure 38. The proof engineer had a simple language represented by an inductive type `Term`, as well as some definitions and proofs about the language. The proof engineer swapped two constructors in the language, and added a new constructor `Bool`.

This case study and the next case study break this change into two parts. In the first part, I used PUMPKIN Pi with automatic configuration to repair functions and proofs about the language after swapping the constructors ①. With a bit of human guidance to choose the permutation from a list of suggestions, PUMPKIN Pi repaired everything, though the original tactics would have also worked, so there was not a difference in development time.

ADD NEW CONSTRUCTORS: EXTENDING A LANGUAGE I then used PUMPKIN Pi to repair functions after adding the new constructor in Figure 38, separating out the proof obligations for the new constructor from the old terms ⑲. This change combined manual and automatic configuration. I defined an inductive type `Diff` and (using partial automation) a configuration to port the terms across the equivalence `Old.Term + Diff` $\simeq$ `New.Term`. This resulted in case explosion, but was formulaic, and pointed to a clear path for automation of this class of changes. The repaired functions guaranteed preservation of the behavior of the original functions.

Adding constructors was less simple than swapping. For example, PUMPKIN Pi did not yet save us time over the proof engineer from the user study; fully automating the configuration would have helped significantly. In addition, the repaired terms were (unlike in the swap case) inefficient compared to human-written terms. For now, they make good regression tests for the human-written terms—in the future, I hope to automate the discovery of the more efficient terms, or use the refinement framework CoqEAL [36] to get between proofs of the inefficient and efficient terms.

FACTOR OUT CONSTRUCTORS: EXTERNAL EXAMPLE The change from Figure 24 came at the request of an anonymous reviewer. I supported this using a manual configuration that described which constructor to map to `true` and which constructor to map to `false` ②. The configuration was very simple for me to write, and the repaired tactics were immediately useful. The development time savings were on the order of minutes for a small proof development. Since most of the modest development time went into writing the configuration, I expect time savings would increase for a larger development.

PERMUTE HYPOTHESES: EXTERNAL EXAMPLE The change in ⑳ came at the request of **Anders**, a cubical type theory expert. It shows how to use PUMPKIN Pi to swap two hypotheses of a type, since `T1` $\to$ `T2` $\to$ `T3` $\simeq$ `T2` $\to$ `T1` $\to$ `T3`. This configuration was manual. Since neither type was inductive, this change used the generic construction for any equivalence. This worked well, but necessitated some manual annotation due to the lack of custom **unification heuristics** for manual configuration, and so did not yet save development time, and likely still would not have had the proof development been larger. Supporting custom unification heuristics would improve this workflow.

CHANGE INDUCTIVE STRUCTURE: UNARY TO BINARY In ⑤, I used PUMPKIN Pi to support a classic example of changing inductive structure: updating unary to binary numbers, as in Figure 27. Binary numbers allow for a fast addition function, found in the Coq standard library. In the style of Magaud & Bertot 2000 [105], I used PUMPKIN Pi to derive a slow binary addition function that does not refer to `nat`, and to port proofs from unary to slow binary addition. I then showed that the ported theorems hold over fast binary addition.

The configuration for `N` used definitions from the Coq standard library for `DepConstr` and `DepElim` that had the desired behavior with no changes. `Iota` over the successor case was a rewrite by a lemma from the standard library that reduced the successor case of the eliminator that I used for `DepElim`:

```
N.peano_rect_succ : ∀ (P : N → Type) pO pS (n : N),
  N.peano_rect P pO pS (N.succ n) =
  pS n (N.peano_rect P pO pS n).
```

The need for nontrivial `Iota` comes from the fact that `N` and `nat` have different inductive structures. By writing a manual configuration with this `Iota`, it was possible to instantiate the PUMPKIN Pi transformation to the transformation that had been its own tool.

While porting addition from `nat` to `N` was automatic after configuring PUMPKIN Pi, porting proofs about addition took more work. Due to the lack of unification heuristics for manual configuration, I had to annotate the proof term to tell PUMPKIN Pi that implicit casts in the inductive cases of proofs were applications of `Iota` over `nat`. These

annotations were formulaic, but tricky to write. Unification heuristics would go a long way toward improving the workflow.

After annotating, I obtained automatically repaired proofs about slow binary addition, which I found simple to port to fast binary addition. I hope to automate this last step in the future using CoqEAL. Repaired tactics were partially useful, but failed to understand custom eliminators like `N.peano_rect`, and to generate useful tactics for `Iota`; both of these are clear paths to more useful tactics. The development time for this proof with PUMPKIN Pi was comparable to reference manual repairs by external proof engineers. Custom unification heuristics would help bring returns on investment for experts.

4.6.2 *Evaluation: External Transport*

PUMPKIN Pi implements transport across equivalences **externally**, in a way that is suitable for repair. A bonus benefit of external transport is that, for some classes of changes, the resulting terms are small and compute efficiently relative to those derived via **internal** transport. To evaluate this, **Nate** and I compared PUMPKIN Pi to a tool that approximates internal transport across equivalences in Coq, using **algebraic ornaments** as an example. We used PUMPKIN Pi to automatically discover and transport functions and proofs along the equivalences corresponding to these ornaments for two scenarios:

1. Single Iteration: from binary trees to sized binary trees
2. Multiple Iterations: from binary trees to binary search trees to AVL trees

At the time, the decompiler was not yet implemented, so we focused on proof terms. For comparison, we also used the ornaments that PUMPKIN Pi discovered to transport terms using an internal approximation of transport in Coq from UP [150]. PUMPKIN Pi produced faster functions and smaller terms, especially when composing multiple iterations of repair. In addition, PUMPKIN Pi imposed little user burden, and the equivalences it discovered proved useful to UP.

HISTORICAL NOTE These days, the authors of UP call the internal approximation of transport used for comparison in this evaluation the *black-box* transformation. After this evaluation and several discussions with the authors, UP introduced the external implementation of transport used for comparison in the case studies in Table 1, though without support for arbitrary equivalences. At the time of the evaluation, PUMPKIN Pi also did not yet support arbitrary equivalences (though it now does), but the UP black-box transformation did (and still does). The development of both the UP white-box transformation and the generalized PUMPKIN Pi transformation happened with frequent

conversations between the authors of both papers, and doubtlessly involved mutual influence. It was wonderful, and involved multiple trips to France. Chapter 5 discusses UP in more detail.

SETUP The code is in the `eval` folder of the repository. For each scenario, **Nate** ran PUMPKIN Pi to search for the **ornamental promotion isomorphism**, and then transported functions and proofs along it using both PUMPKIN Pi and UP. He noted the amount of user interaction (Section 4.6.2.1), and together we measured the performance of transported terms (Section 4.6.2.2). To test the performance of transported terms, we tested runtime by taking the median of ten runs using `Time Eval vm_compute` with test values in Coq 8.8.0, and we tested size by normalizing and running `coqwc` on the result.[7]

In the first scenario, Nate transported traversal functions along with proofs that their outputs are permutations of each other from binary trees (`tree`) to sized binary trees (`Sized.tree`). In the second scenario, he transported the traversal functions to AVL trees (`avl`) through four intermediate types (one for each new index), and he lifted a search function from BSTs (`bst`) to AVL trees through one intermediate type. Both scenarios considered only full binary trees.

To fit `bst` and `avl` into algebraic ornaments, we used boolean indices to track invariants. While the resulting types are not the most natural definitions, this scenario demonstrates that it is possible to express interesting changes to structured types as algebraic ornaments, and that lifting across these types in PUMPKIN Pi produces efficient functions.

4.6.2.1 *User Experience*

For each intermediate type in each scenario, **Nate** used PUMPKIN Pi to discover the equivalence. This was enough for PUMPKIN Pi to lift functions and proofs with no additional proof burden and no additional axioms. To use UP without PUMPKIN Pi, he would have had to prove the equivalence by hand, but instead he was able to use the equivalence generated by PUMPKIN Pi. In addition, to use UP, he had to prove univalent parametricity of each inductive type; these proofs were small, but required specialized knowledge. To lift the proof of the theorem `pre_permutes` using UP, he had to prove the univalent parametric relation between the unlifted and lifted versions of the functions that the theorem referenced; this pulled in the functional extensionality axiom, which was not necessary using PUMPKIN Pi.

In the second scenario, to simulate the incremental workflow PUMPKIN Pi requires, he transported along each intermediate equivalence to each intermediate type, then unpacked the result. For example, the ornament from `bst` to `avl` passed through an intermediate type; He transported `search` to use this type first, unpacked the result, and

7 i5-5300U, at 2.30GHz, 16 GB RAM

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Pi	0.0	0.0	0.0	3.0 (1.00x)	35.0 (0.95x)
	UP	0.0	1.0	27.0	486.5 (162.17x)	8078.5 (218.33x)

Figure 39: Median runtime (ms) of the original (`tree`) and transported (`Sized.tree`) `preorder` over ten runs with test inputs ranging from about 10 to about 10000 nodes.

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Pi	71.5	71.0	69.0	75.0 (25.00x)	109.0 (2.95x)
	UP	1.0	11.0	152.0	2976.5 (992.17x)	56636.5 (1530.72x)
`search`	**Unlifted**	0.0	0.0	2.0 (1.00x)	3.0 (1.00x)	29.0 (1.00x)
	Pi	12.0	14.0	12.0 (6.00x)	15.0 (5.00x)	50.0 (1.72x)
	UP	1.0	5.0	67.0 (33.50x)	1062.0 (354.00x)	15370.5 (530.02x)

Figure 40: Median runtime (ms) of the original (`tree`) and transported (`avl`) `preorder`, plus the original (`bst`) and transported (`avl`) `search`, over ten runs with inputs ranging from about 10 to about 100000 nodes.

then repeated this process. In this scenario, using UP differently or using PUMPKIN Pi with a manual configuration could have saved some work relative to the workflow chosen, since with those workflows, it is possible to skip the intermediate type;[8] PUMPKIN Pi with automatic configuration is best fit where an incremental workflow is desirable.

4.6.2.2 *Performance*

Relative to the UP black-box transformation, PUMPKIN Pi produced faster functions. Figure 39 summarizes runtime in the first scenario for `preorder`, and Figure 40 summarizes runtime in the second scenario for `preorder` and `search`. The `inorder` and `postorder` functions performed similarly to `preorder`. The functions PUMPKIN Pi produced imposed modest overhead for smaller inputs, but were tens to hundreds of times faster than the functions that UP produced for larger inputs. This performance gap was more pronounced over multiple iterations.

PUMPKIN Pi also produced smaller terms: in the first scenario, 13 vs. 25 LOC for `preorder`, 12 vs. 24 LOC for `inorder`, and 17 vs. 29 LOC for `postorder`; and in the second scenario, 21 vs. 120 LOC for `preorder`, 20 vs. 119 LOC for `inorder`, 24 vs. 125 LOC for `postorder`, and 31 vs. 52 LOC for `search`. In the first scenario, the transported proof of `pre_permutes` using PUMPKIN Pi was 85 LOC; the transported proof of `pre_permutes` using UP was 1463184 LOC.

8 The performances of the terms that the UP black-box transformation produces are sensitive to the equivalence used; for a 100 node tree, this alternate workflow in UP produced a search function hundreds of times slower and traversal functions thousands of times slower than the functions that PUMPKIN Pi produced. The lifted proof of `pre_permutes` using UP failed to normalize with a timeout of one hour.

I suspect Pumpkin Pi provided these performance benefits because it directly transformed **eliminators**, whereas the UP black-box transformation implemented transport in a standard way, defining transported functions in terms of the original functions. The multiple iteration case in particular highlights this, since UP's black-box approach makes lifted terms much slower and larger as the number of iterations increases, while Pumpkin Pi's approach does not.

4.7 CONCLUSION

The Pumpkin Pi plugin extends the Pumpkin Patch plugin suite with support for a broad class of changes in datatypes. It also supports patch application in a principled manner, is built with workflow integration including tactics in mind, and can save and in fact already has saved work for proof engineers in practical use cases. At this point, it is fair to say that my **thesis** holds:

> Changes in programs, specifications, and proofs can carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change (Sections 3.2, 3.3, 3.4, 4.2, 4.3, and 4.4). A tool that automates this (Sections 3.5 and 4.5) can save work for proof engineers relative to reference manual repairs in practical use cases (Sections 3.6 and 4.6).

And so there really is **reason to believe**.

I will talk about what that means for proof engineers—and what I believe the next era of verification can look like—in Chapter 6. But first, I will back up a bit and talk about related work.

5

RELATED WORK

Proof repair can be viewed as program repair (Section 5.2) for the domain of proof engineering (Section 5.1).

5.1 PROOF ENGINEERING

Proof repair falls into the domain of **proof engineering**, or the technologies that make it easier to develop and maintain systems **verified** using **proof assistants**. Proof repair in this thesis is implemented for Coq, but there are many other proof assistants to choose from, some with different implications for proof repair (Section 5.1.1). In contrast with proof repair, most work on proof maintenance focuses on designing proofs to be robust to change to begin with (Section 5.1.2). While proof repair is new, it builds on work in proof evolution (Section 5.1.3), proof reuse (Section 5.1.4), and other kinds of proof automation (Section 5.1.5). **QED at Large** contains a detailed overview of all of these proof engineering technologies and more.

5.1.1 *Proof Assistants*

Proof engineering as defined in **QED at Large** considers proof assistants that satisfy the *de Bruijn criterion* [16, 17], which requires that they produce proof objects that a small proof-checking **kernel** can check. This includes **Coq** [38]—the proof assistant that this thesis focuses on—but also other proof assistants like **Isabelle/HOL** [81], **HOL Light** [78], **HOL4** [117], **Agda** [7], **Lean** [97], and **NuPRL** [123]. These proof assistants have different foundations (in the case of Coq, **CIC$_\omega$**), different notions of proof objects (in the case of Coq, proof terms in **Gallina**), and different automation built on top of those proof objects (in the case of Coq, proof scripts made up of **Ltac tactics**). Differences in proof assistants along these dimensions have different implications for proof repair.

FOUNDATIONS The foundations that make up proof assistants vary by proof assistant. The foundations of Coq, for example, build on the **intensional** type theory **CIC$_\omega$**. CIC$_\omega$ is *intuitionistic* [76] (or **constructive**) in that proofs in CIC$_\omega$ do not assume the *law of the excluded middle*

(**LEM**), which states that for any proposition, either that proposition is true, or its negation is true. CIC_ω also does not assume *double negation elimination* (**DNE**), which states that the negation of the negation is the original proposition or type. Because of this, in CIC_ω, it is not possible in general to prove existence by proving that nonexistence implies a contradiction. Instead, one must supply a witness to the existential.[1]

The proof assistants Isabelle/HOL [81], HOL4 [117], and HOL Light [78] are built on *classical* foundations in that they assume both **LEM** and **DNE**. The proof assistants Agda and Lean are built on **constructive** foundations that are similar to Coq, though with some minor differences. Lean in particular further assumes an axiom called *uniqueness of identity proofs* (**UIP**), which states that all proofs of **propositional equality** at a given type are equal. As I detail in **QED at Large**, this axiom is incompatible with the **univalence** axiom from homotopy type theory [154], which states that **equivalence** is equivalent to propositional equality. *Cubical type theory*—of which there are two flavors [35, 10]—gives univalence a constructive interpretation, so that it is no longer an axiom. Implementations of cubical type theory include *RedPRL* [135] and *Cubical Agda* [6].

Proof repair in this thesis assumes **constructive** rather than **classical** foundations. The core techniques should largely transfer to other proof assistants built on constructive foundations, like Agda and Lean. It is possible that the particular transformations may have to account for differences, like the presence of **UIP** in Lean, which may make the general correctness statement of the Pumpkin Pi transformation—one that relies on **univalence** at the level of the metatheory—less meaningful. The techniques transfer with a bit more resistance to proof assistants built on classical foundations, since classical proofs may omit information helpful for repair.

PROOF OBJECTS In proof assistants that satisfy the **de Bruijn criterion**, the **proof object** is the certificate that the **kernel** can check for a given proof to make sure that it proves a given theorem. In Coq, proof objects are proof terms in **Gallina**; the kernel verifies these terms by checking their types. As **Karl** notes in **QED at Large**, producing *explicit proof objects* and checking them is just one of the two dominant approaches to satisfying the de Bruijn criterion—the approach followed by **Coq**, **Agda**, and **Lean**. The other approach is to produce *ephemeral proof objects* that are correct by construction [15]—an approach followed by **Isabelle/HOL**, **HOL Light**, and **HOL4**.

Proof repair in this thesis assumes **explicit** rather than **ephemeral** proof objects. One possible way to apply the same techniques to proof assistants with ephemeral proof objects is to follow the following workflow:

1 Perhaps notably, though, the witness to the existential can be that **DNE** *provably* holds for a particular instance. This is true for decidable domains in CIC_ω.

1. Reify ephemeral proof objects to be explicit.
2. Apply differencing and transformations over those objects.
3. Decompile the transformed proof objects to automation.

Section 6.2 describes a potential specific instance of this workflow for **Isabelle/HOL**.

AUTOMATION Typically, layers of **proof automation** act as an interface between the proof engineer and the **kernel**. In Coq, for example, proof engineers write proof scripts interactively using **tactics**. Proof engineers can combine existing tactics, or write their own, either in the general-purpose programming language OCaml or in the tactic language **Ltac**.

In contrast, in **Isabelle/HOL**, proof engineers commonly write proofs in Isabelle/Isar [163, 157], a high-level language for structuring and composing propositions, facts, and proof goals. Proof engineers can also sometimes mix languages for automation. For example, Coq proof engineers can write proofs using the high-level proof language SSReflect [67, 66], or they can even mix SSReflect with Ltac tactics. Isabelle/HOL proof engineers can write automation in the general-purpose language Standard ML—which was originally introduced specifically for developing proof automation [69]—or with the Ltac-inspired tactic language Eisbach [107]. **Agda** proof engineers typically rely only on reflection [50, 72, 155] and not on any additional language to automate their proofs.

The approach to proof repair taken in this thesis is independent of the kind of automation used, or the language used to implement it. All that needs to change to transfer this approach to a different tactic or proof language, for example, is the implementation of the decompiler. Still, some kinds of automation may produce proof terms that are larger or more difficult to manipulate, or may make decompilation especially difficult.

Better languages for automation may help circumvent some of the difficulties of repairing tactics directly, without relying a decompiler. Or, they may help simplify the implementation of the decompiler. The Ltac successor Ltac2 [129], for example, is much more structured than Ltac, and may help ease proof repair in Coq in the future.

5.1.2 *Proof Design*

Much work focuses on designing proofs to be robust to change, rather than fixing broken proofs. This can take the form of robust abstractions or robust automation.

ROBUST ABSTRACTIONS Proof engineers often use abstractions to make proofs less likely to break to begin with. Examples of this

include using information hiding techniques [168, 87] or any of the structures [33, 148, 144] for encoding interfaces in Coq. CertiKOS [70] introduces the idea of a deep specification to ease verification of large systems. Design principles for specific domains (like formal metatheory [13, 48, 49]) can also make verification more tractable. Design and repair are complementary: design requires foresight, while repair can occur retroactively. Repair can help with changes that occur outside of the proof engineer's control, or with changes that are difficult to protect against even with informed design.

ROBUST AUTOMATION Another approach to robust design is to use heavy **proof automation**, for example through custom program-specific **tactics** [32] or general-purpose hammers [22, 128, 84, 42]. The degree to which proof engineers rely on automation varies, as seen in the data from the REPLICA user study. Automation-heavy proof engineering styles localize the burden of change to the automation, but can result in proof terms that are large and slow to type check, and tactics that can be difficult to debug. While these approaches are complementary, more work is needed for proof repair tools to better support developments in this style.

5.1.3 *Proof Evolution*

The need to evolve proofs to changes was raised as a barrier for verification of real software systems as far back as 1977 [51]. This barrier impacted real developments. A review [55] of the evolution of the seL4 verified OS microkernel [89], for example, notes that while customizing the kernel to different environments may be desirable, "the formal verification of seL4 creates a powerful disincentive to changing the kernel." Another paper [101] motivates and describes updates to the initial CompCert memory model that include changes in specifications, automation, and proofs [37].

As I have shown in this thesis, breaking changes are not always in the proof engineer's control. *Proof refactoring* [164] tools—meant to help proof engineers redesign proof developments—can also help proof engineers repair proofs in response to breaking changes. These tools are the proof engineering equivalent of *program refactoring* tools, or tools that restructure code in a way that preserves semantics [124]. Some proof refactoring tools developed in parallel with my thesis work can be viewed as **proof repair** tools. This section describes refactoring tools that work at the level of proof scripts and proof terms; PUMPKIN PATCH is the only tool suite for proof evolution I am aware of that supports both.

PROOF SCRIPTS A few proof refactoring tools operate directly over proof scripts: POLAR [53] refactors proof scripts in languages based

on Isabelle/Isar [157], CoqPIE [143] is an *Integrated Development Environment* (IDE) with support for simple refactorings of Ltac scripts, and Tactician [5] is a refactoring tool for proof scripts in HOL Light that focuses on refactoring proofs between sequences of tactics and tacticals. This approach is not tractable for more complex changes [142].

Some proof refactoring tools focus on specific refactoring tasks that are common in proof development. For example, Levity [24] is a proof refactoring tool for an old version of Isabelle/HOL that automatically moves lemmas to maximize reuse. The design of Levity is informed by experiences with two large proof developments. Levity addresses problems that are especially pronounced in the domain of proof refactoring, such as the context-sensitivity of proof scripts. Levity has seen large scale industrial use.

PROOF TERMS There is little work on refactoring proof terms directly. This is the main focus of Chick [142], which refactors terms in a language similar to **Gallina**. Chick was developed in parallel to the PUMPKIN prototype, and both tools influenced one another. Consequentially, Chick and PUMPKIN PATCH have similar workflows: both take example changes supplied by the proof engineer, use differencing algorithms to determine the changes to make elsewhere, and then apply the changes they find. Chick supports insertion, deletion, modification, and permutation of subterms. Chick does this using a syntactic algorithm that handles only simple transformations, and so presents itself primarily as a proof refactoring tool.

Another term-based refactoring tool is the refactoring tool RefactorAgda [165] for a subset of **Agda** terms. RefactorAgda supports many changes, including changing indentation, renaming terms, moving terms, converting between implicit and explicit arguments, reordering subterms, and adding or removing constructors to or from types; it also documents ideas for supporting other refactorings, such as adding and removing arguments and indices to and from types. Both Chick and RefactorAgda support primarily syntactic changes and operate solely over proof terms.

5.1.4 *Proof Reuse*

Proof repair is ultimately a form of *proof reuse*—applying software reuse principles to proof assistants in order to repurpose existing proofs as much as possible. Like software reuse, proof reuse leverages design principles and language constructs. In addition, the interactive nature of proof assistants naturally leads to a class of proof reuse technologies less explored in the software reuse world: automated tooling. Proof repair falls into the class of automated tooling for proof reuse. This section describes other automated tooling for proof reuse that may be useful to a future proof repair tool.

EXTENDING INDUCTIVE TYPES PUMPKIN PATCH is yet to save proof engineers work when it comes to extending **inductive types** with new **constructors**. In the future, it may help to draw on early work in proof reuse for extending inductive types. For example, a 2004 paper [23] describes a tactic to adapt proof obligations to changes in inductive types. Soon after, a 2006 paper [115] provides a high-level description of a possible method to synthesize missing proofs for those new obligations using a type reconstruction algorithm, though it is not currently implemented.

PROOF TRANSFORMATION Proof repair in this thesis combines **differencing** with **proof term transformations**. The idea of proof term transformations dates back to at least 1987 [131]. Any proof reuse tool that works by proof term transformation can, in theory, be used for repair, especially when coupled with something like the PUMPKIN Pi decompiler.

The PUMPKIN prototype implements a kind of proof **generalization**. This is a common kind of proof term transformation that arose in the context of proof assistants in the 1990s [73, 92, 133]. Coq's `generalize` tactic does basic syntactic generalization [39]. Both PUMPKIN and a tool [83] for generalizing theorems in **Isabelle/HOL** implement more complex transformations for generalization.

Magaud & Bertot 2000 [105] implement a proof term transformation between unary and binary numbers that fits into a PUMPKIN Pi configuration. The expansion algorithm from the paper describing this transformation may help guide the design of better unification heuristics for PUMPKIN Pi, in particular when identifying applications of **definitional** `Eta` and `Iota`.

The refinement framework CoqEAL [36] transforms functions across relations in Coq, and these relations can be more general than PUMPKIN Pi's equivalences. However, while PUMPKIN Pi supports both functions and proofs, CoqEAL supports only simple functions due to the problem with **definitional equality** that `Iota` addresses. CoqEAL may be most useful to chain with PUMPKIN Pi to get faster functions, or to help support better workflows for changes that do not correspond to equivalences.

One of the automatic configurations in PUMPKIN Pi automates discovery of and transport across equivalences that correspond to **algebraic ornaments**. This automatic configuration formed the basis of the first tool for ornamentation to operate over a non-embedded dependently typed language, initially called DEVOID—but later generalized to arbitrary equivalences and renamed to PUMPKIN Pi. This stands in contrast to the many existing embedded implementations of ornaments [44, 90, 45, 91, 43] that had arisen since their discovery [108]. DEVOID essentially moved the automation-heavy approach of Ornamentation in ML [167], which operates on non-embedded ML

code, into **CIC$_\omega$**. It also introduced the first differencing algorithm to identify ornaments, which in the past had been identified as a "gap" in the literature [91]. Other kinds of ornaments may prove useful for future PUMPKIN Pi differencing algorithms and configurations. A recent thesis [166] on ornaments may prove especially useful.

TRANSFER & TRANSPORT The need to automatically transfer functions and proofs across equivalences and other relations is a long-standing challenge for proof engineers [105, 20, 104]. Particular successful is the widely used Transfer [79] package, which supports proof reuse in Isabelle/HOL. Transfer works by combining a set of extensible transfer rules with a type inference algorithm. Transfer is not yet suitable for repair, as it necessitates maintaining references to both datatypes. Section 6.2 describes one possible path building on Transfer to implement a proof repair tool for Isabelle/HOL.

The PUMPKIN Pi transformation implements a particular kind of transfer called automatic **transport**. The name is potentially confusing: **transport** itself refers to a particular form of rewriting along a **propositional equality**, or inhabitance of the identity type. It often (but not always) refers to **univalent transport**, an application of an **eliminator** derivable in **homotopy type theory** from **univalence** [154] that rewrites along the identity type corresponding to an **equivalence** [56]. *Automatic transport* refers to any automated tooling for rewriting along propositional equalities that behaves like transport at the level of either **internally** to the theory or (as in PUMPKIN Pi) **externally** in the metatheory. Automatic transport is a kind of transfer, but by virtue of it being automatic, not by virtue of it applying transport.[2] Transfer may apply more broadly than across equalities or equivalences.

PUMPKIN Pi implements automatic univalent transport **externally**, without relying on any additional axioms at the level of the theory itself. The UP black-box transformation [150] approximates automatic univalent transport **internally** in Coq, only sometimes relying on additional axioms. The UP black-box transformation does not remove references to the old type, making it poorly suited for repair. However, unlike PUMPKIN Pi, it supports type-directed search—analogous functionality may help improve PUMPKIN Pi substantially.

Recent work [151] extends UP with a white-box transformation that may work for repair. However, the white-box transformation imposes proof obligations on the proof engineer beyond those imposed by PUMPKIN Pi. In addition, it comes with neither **differencing** algorithms for equivalences nor **proof script** generation. It also does not support changes in inductive structure, instead relying on its original black-box functionality; `Iota` solves this in PUMPKIN Pi, and is based on lessons learned from reading that article. The most fruitful progress may come from combining these tools.

2 I got this wrong in **QED at Large**.

5.1.5 *Other Proof Automation*

Proof repair implements a kind of proof automation. New proof automation continues to make proof repair more feasible.

ONTOLOGY REPAIR GALILEO [29] is a tool built on **Isabelle/HOL** for identifying and repairing faulty ontologies in response to contradictory evidence. It uses repair plans to determine when to trigger a repair, and how to repair the ontology. It has been applied to repair faulty physics ontologies, and may have applications for proof repair.

KNOWLEDGE SHARING METHODS Knowledge sharing methods [60] match concepts across different proof assistants with similar logics and identify isomorphic types, and may have implications for proof repair. Later work uses these methods in combination with HOL(y)Hammer to reprove parts of the standard library of HOL4 and HOL Light using combined knowledge from the two proof assistants [61]. More recently, this approach has been used to identify similar concepts across libraries in proof assistants with different logics [62]. These methods may have applications when repairing proofs even within the same logic, using information from different libraries, different commits, or different representations of similar types.

E-GRAPHS The PUMPKIN Pi proof term transformation can in some sense be viewed as a rewrite system across equivalences. A number of modern rewrite systems use data structures called *e-graphs* [118] for managing equivalences. E-graphs have been implemented in **Lean** [145] (assuming **UIP**), and in **Cubical Agda** [65] (implying **univalence**). Similar implementations of e-graphs could help improve PUMPKIN Pi and similar tools to support type-directed search and more (see Section 6.2).

5.2 PROGRAM REPAIR

Proof repair can be viewed as a form of *program repair* [114, 63] for proof assistants. Proof assistants like Coq are an especially good fit for program repair (Section 5.2.1). While it is not straightforward to apply existing program repair techniques to proof assistants, looking to them for inspiration may help improve proof repair tools more in the future (Section 5.2.2).

5.2.1 *A Good Fit*

A recent survey of program repair distinguishes between repair tools based on the *oracle* they use to judge whether a patch is correct. For example, proof repair is a kind of *specification-based* repair, since it uses

a specification (a goal type derived from differencing) as an oracle. Program repair tools sometimes use other oracles—commonly, test suites (*test-based* repair).

A recent review [134] of a popular test-based program repair tool [96] and its variants shows that most of the reported patches generated by the tool are not correct. These observations are later reaffirmed in a different setting [106]. In response, the paper recommends that program repair tools draw on extra information, like specifications or example patches. In **Coq**, specifications and examples are rich and widely available: specifications thanks to dependent types, and examples thanks to **constructivism**. This shows why proof repair in Coq is an especially good fit for program repair.

SPECIFICATIONS One limitation of test-based program repair tools is that tests in evaluation suites are often underspecified, so it can be hard to know when a patch to a program is correct. For example, the review notes that some tests in the evaluation suite for the tool check whether a program terminates with an exit code of 0, but do not check the program output. In addition, patches are often overfit to the tests in the test suite; additional tests expose problems with those patches. In fact, some patches are outright harmful, as they introduce new problems which the test suite does not check for.

In contrast, in the world of proof repair, there is always a specification to work with—the theorem being proven—so a proof repair tool does not need to rely on tests. Furthermore, the scope of properties that can be specified in proof assistants like Coq is especially large thanks to its expressive type system **CIC**$_\omega$, with polymorphism and dependent types. The richness of the type theory further makes it possible for PUMPKIN PATCH to check itself along the way and make sure it is on the right track.

Still, there is always a chance that the specification itself must change in order for proof repair to work, as I showed with PUMPKIN Pi. In those cases, it is helpful to have some assurance that the specifications the tool produces are meaningful—in the case of PUMPKIN Pi, that the old and new specifications are equal up to **transport** along the change in the datatype. It is also useful to have a human in the loop to check specifications in the end, as all of the plugins in PUMPKIN PATCH do. Section 5.2.2 discusses other specification-based repair tools, as well as other repair tools that bring a human into the loop.

EXAMPLES Another challenge for test-based program repair tools is defining the correct search space and searching efficiently within it. For example, the review found that running the same tool on strengthened versions of the test suites produced no patches at all in the time allotted. One possible reason for this is that the tools could

not search for the correct patches efficiently enough. Example-based techniques can help navigate a large search space quickly.

PUMPKIN PATCH uses examples—in the form of changes to datatypes or proofs—to derive patches. The **constructive** foundations of Coq make this especially appealing and powerful. For example, existence proofs in Coq must be accompanied by a witness. Each of these witnesses is in effect an example that PUMPKIN PATCH can extract and generalize, narrowing down the search space of possible repairs.

Thanks to the richness of the type theory, PUMPKIN PATCH can in practical use cases repair proofs by generalizing a very small number of examples, like a single example patched proof, or a single example change to a datatype. Section 5.2.2 describes other example-based repair tools.

5.2.2 *Techniques for Inspiration*

Proof repair can in the future draw on many of the techniques that program repair tools use, even though the tools themselves do not carry over in a straightforward way (recall Section 2.3.1). This section discusses techniques from existing program repair tools that are relevant to proof repair. It focuses in particular on what a recent survey [114] of program repair calls behavioral repair, or patching the code, rather than state repair, or patching the dynamic behavior. Among behavioral repair tools, it focuses on regression repair, specification-based repair, repair by example, and other techniques that bring a human into the loop.

REGRESSION REPAIR Regression repair tools target regression bugs, like changes that cause a set of tests (the *regressed tests*) that used to pass to no longer pass. Test-based regression repair tools repair code such that regressed tests pass on the repaired code. In some sense, proof repair is a kind of regression repair, as it repairs proofs that used to succeed, but after some change, no longer do (the *regressed proofs*). A section by **Karl** in **QED at Large** describes the correspondence between regressed proofs and regressed tests in more detail, and details existing techniques [126, 159, 18, 27, 158, 19, 47, 160, 161, 162] for rechecking regressed proofs.

One test-based regression program repair tool is ReAssert [46] for Java, which focuses on regressions caused by refactoring. ReAssert uses a program analysis to identify broken code, chooses a strategy for repair, and suggests repairs to the programmer using that strategy that cause regressed tests to pass. It loops through strategies until one works or none remain. Another such tool is Relifix [152] for C. Relifix uses a manual inspection to find code transformations based on regressions, then searches those transformations for patches that make the regressed tests pass without making other tests fail.

Both ReAssert and Relifix are configurable like PUMPKIN Pi, and may provide interesting examples of configurations or ways of inferring new configurations.

While not quite a regression repair tool, GRAFTER [171] is a related tool that adapts the tests themselves, rather than the code under test. Its focus is on testing software clones for errors introduced during the cloning process. It uses a static analysis to identify variables and methods that correspond between the clones, then ensures that the flow is preserved using that mapping. The user can then run the new tests to compare behavior. It provides a guarantee about type safety, and it performs reasonably well on some real-world software. GRAFTER, like PUMPKIN PATCH, takes an approach to repair that uses a form of **differencing**. Looking to this to help inform new differencing algorithms for PUMPKIN PATCH could be fruitful, in spite of foundational differences of the target domains.

SPECIFICATION-BASED REPAIR Some tools use specifications as an oracle, like PUMPKIN PATCH. For example, AutoFix-E [156, 130] uses contracts to repair Eiffel programs. Specification-Based Program Repair Using SAT [68] encodes pre and post conditions in combination with other constraints from the code into SAT and then uses Alloy to generate patches. Other tools combine test-based program repair with logical specifications and automated solving [74, 119, 119, 169, 109, 86]. For future proof reuse tools, making better use of existing proof automation in proof assistants to generate patches that satisfy specifications may prove fruitful.

Proof-directed repair [52] presents a methodology for repairing programs based on information from incomplete proofs in **Isabelle/HOL**. Essentially, the programmer writes a proof, and then uses feedback from the attempted proof to debug and fix the code. In a sense, since if the repair succeeds the proof should go through, it uses proofs as an oracle. The paper presents a few techniques for fixing the broken code, then shows some examples using those techniques with existing tools. It does not yet automate it in a tool. Still, perhaps using partial proofs as in proof-directed repair can help a proof repair tool like PUMPKIN PATCH better repair functions. It also matches the workflow of proof engineers seen in the **REPLICA** user study.

REPAIR BY EXAMPLE Some program repair tools work **by example**, like PUMPKIN. Prophet [103], a test-based repair tool for C, uses human-generated patches from software repositories as examples. These examples can come from different applications from the one that is being repaired. Prophet uses differencing over ASTs to extract features that describe the behavior of the example patch abstracted from its particular application. From these patches, it learns a model of correct code. Then, it localizes faults and generates candidates,

which it ranks according to the learned model. In this way, it produces patches that not only cause the tests to succeed, but also are likely according to the learned model to be correct to humans.

The repair tool QACrashFix [59] uses pairs of buggy and fixed code from Q & A sites like StackOverflow to derive patches for crashing input bugs. These patches are in the form of edit scripts, so that they can apply in different contexts. It uses a preprocessing step to find the right query for the Q & A site, then they look at answers for buggy and fixed code examples, then from those they derive edit scripts to try to fix the bug. It then uses a combination of tests and human validation to determine whether the patches are correct.

SearchRepair [86] turns code from repositories into a searchable database. To form this database, it uses a static analysis to encode the input-output behavior of the code as constraints for an SMT solver. It then localizes the fault in the buggy program, encodes the buggy program similarly, performs a semantic code search over that database to identify candidate patches, and finally uses the test suite as an oracle to determine whether candidates succeed.

Systematic editing [111] is a technique that could help repair by example tools. This technique generalizes an edit to a program into a program transformation that can apply in similar program contexts. It works by syntactic differencing over the AST of the example edit, abstracting the difference, and applying it elsewhere. It can handle insertions, deletions, updates, and moves. Lase [112] implements and improves on this, making use of multiple examples instead of just one, and also automatically identifying locations to apply transformations. Similarly, `spdiff` [8] generalizes patches into semantic patches for Cocinelle [125], which can then apply those patches automatically in different contexts. This way, the library designer can write a semantic patch himself, or `spdiff` can infer one.

A number of program repair tools above—much like the proof repair tools described in this thesis—build on **differencing** algorithms. Existing work in differencing and incremental computation may help improve semantic differencing algorithms for both program and proof repair. Type-directed diffing [113] finds differences in algebraic data types. Semantics-based change impact analysis [12] models semantic differences between documents. Differential assertion checking [94] analyzes different versions of a program for relative correctness with respect to a specification. Incremental λ-calculus [26] introduces a general model for program changes. All of these may be useful for improving semantic differencing.

Some of these program repair tools use machine learning to generalize examples—I discuss some ideas combining proof repair with machine learning in Section 6.2. Several of the tools identify examples from code repositories and libraries. These tools may offer some insights for how to break down the large composite changes typically

found in static artifacts or in code repositories into smaller incremental changes like those made during development in the REPLica user study. Isolating changes may help with extracting repair benchmarks from artifacts, supporting library and version updates, and integrating with *Continuous Integration* (CI) systems.

HUMAN IN THE LOOP Some tools avoid using test suites to judge correctness of candidate patches, and instead bring the programmer into the loop. For example, both ReAssert [46] and QACrashFix [59] suggest repairs directly to the programmer—a workflow that partially inspired the tactic suggestion interface in Pumpkin Pi. In general, an approach that suggests repairs to proof engineers in the end and allows them to vet the specifications and tactics used seems to fit naturally into proof engineering workflows.

A natural integration point for a repair tool like Pumpkin Patch is at the IDE level. CatchUp! [75] is an IDE plugin (implemented for Eclipse in Java) that automatically adapts library clients to API refactorings. It records refactorings that the library developer makes inside of the IDE, then replays the refactorings in in client code, reconstructing everything from the recorded trace. Future proof repair tools may benefit from IDE integration of this kind. For example, it may be useful to record changes within a project inside of an IDE so that Pumpkin Patch can find patches corresponding to incremental changes without the proof engineer needing to deconstruct them manually. It may also help to have something like the trace file in CatchUp! so that library developers can easily provide patches for client proof developments.

6

CONCLUSIONS & FUTURE WORK

Through a combination of **semantic differencing** and **proof term transformations**, my **proof repair** tool suite can extract, generalize, and apply the information that a change carries to fix proofs broken by the same change. Proof repair can save and in fact **already has saved work for proof engineers** relative to reference manual repairs in practical use cases. And so proof repair 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, even when the proof engineer does not follow good development processes, or when the change occurs outside of the proof engineer's control.

This sentiment was echoed recently in an article by an industrial proof engineer [54] (emphasis mine):

> We have *reason to think* such proof repair is tractable. Rather than trying to synthesize a complete proof from nothing—a problem known to be immensely difficult—we start from a correct proof of fairly similar software. We will be attempting proof reconstruction *within a known neighborhood*.

The proof engineer credited my proof repair work on social media, but noted that there ought to be much more work in this space.

I agree (Secton 6.1), and I want to take that a step further: I believe that we can build on proof repair to build the next era of proof engineering. I believe that era will be one in which programmers of all skill levels across all domains can develop and maintain verified systems—an era of *proof engineering for all* (Section 6.2).

6.1 FUTURE WORK: PATCHING THE GAPS OF REPAIR

In this thesis, I have shown you two kinds of proof repair: **by example** and **across equivalences**. The corresponding tools together support just a small chunk of the proof repair scenarios that proof engineers encounter—albeit a practical chunk. Solving proof repair more generally hinges on broadening the scope of changes and terms that proof repair tools can support.

SCOPE OF CHANGES The PUMPKIN Pi transformation supports equivalences; ideally, it should support arbitrary relations. The refinement framework CoqEAL [36] already supports relations that are not equivalences, though only for functions and not yet for proofs; extending CoqEAL and integrating it with the PUMPKIN Pi transformation is a natural first step toward supporting arbitrary relations. Integrating CoqEAL may help with supporting changes in datatypes that—without quotient types—cannot be expressed as equivalences [9]. It may also help with supporting changes that can be expressed as equivalences only with some resistance, like the change adding a constructor in Figure 38 on page 99, or any of the many similar changes in the **REPLICA** user study data. And it may help with supporting changes in algorithms, like replacing slow unary addition with fast binary addition—a step still left to the proof engineer.

Differencing for an arbitrary change is of course undecidable, but that does not mean each proof engineer should resign herself to only the differencing algorithms that ship with the proof repair tools she uses. Proof repair tools should make it easy for each proof engineer to implement new differencing algorithms to support the classes of changes that matter to her. PUMPKIN PATCH lets only those proof engineers who are OCaml and Coq experts implement these differencing algorithms; future proof repair tools ought to expose frameworks that help even non-experts do the same.

Proof repair tools should ideally help proof engineers fix broken proofs in response to changes in the implementations of tactics, and should even help proof engineers repair the implementations of tactics themselves. The same holds for changes in notation. Supporting either of these will require innovations beyond those seen in this thesis.

SCOPE OF TERMS Both PUMPKIN and PUMPKIN Pi place some restrictions on **Gallina** terms. For example, the preprocessing tool by **Nate** translates only *some* fixpoints to **eliminators**, even though in principle it ought to be possible to translate *all* fixpoints to eliminators (Section 4.5.1.3). Furthermore, there is not yet a corresponding *postprocessing* tool to get back from transformed eliminators to fixpoints. That is, even though eliminators in Gallina *do* reduce to fixpoints, preprocessing a fixpoint and then reducing the result does not necessarily produce a fixpoint that is **definitionally equal** to the original. Right now, the step of getting from a proof about the preprocessed term back to a proof about the original term—while easy in my experience—is left to the proof engineer. A postprocessing tool ought to automate this step with strong guarantees and a smooth user experience.

An alternative way to support pattern matching and fixpoints is to support them in the proof term transformations natively. I have found reasoning about pattern matching and fixpoints more difficult than reasoning about eliminators, but of course not everyone is me.

This may prove not to be so difficult, and if so, it may be an approach worth taking in a future repair tool.

Neither PUMPKIN nor PUMPKIN Pi supports the two features that make it possible to construct infinite streams of data in Gallina: cofixpoints and coinduction. Differencing in PUMPKIN struggles to reason about nested induction. The PUMPKIN Pi transformation sometimes fails to reason properly about terms with existential variables, which may show up in terms when the **unification** step is unable to fully resolve parameters and indices. And PUMPKIN Pi does not yet make it possible to write custom unification heuristics, and so sometimes forces proof engineers to write manual annotations instead. Supporting all of these ought to help build an even more useful proof repair tool—so that we may look to the next era.

6.2 THE NEXT ERA: PROOF ENGINEERING FOR ALL

So, what *would* it take to empower programmers of all skill levels across all domains to formally **verify** software systems? Since I first asked this question at the beginning of this thesis, I have introduced proof repair tools that bring us closer to this dream. But these tools still target expert proof engineers. There is a lot more that we as a community can do to make proof engineering accessible more broadly.

I conclude with a discussion of twelve future project ideas building up to the next era of proof engineering for all. I hope that these ideas inspire you!

Proof Engineering for Experts

In the future, I want maintaining proofs to be *seamless* for expert proof engineers. I want experts to have easy access to proof repair tools that automate all but the creative parts of maintenance. But for that to happen, we need to make proof repair tools more widely available, powerful, and natural to use.

AVAILABILITY The biggest barrier to widely available proof repair for experts is that the implementation is for just one proof assistant, **Coq**. The techniques from this thesis should handle proof assistants with similar foundations, like **Agda**, and possibly **Lean**.[1] With a bit of adjustment, my hope is that the techniques should handle even proof assistants with radically different foundations, like **Isabelle/HOL**, which is **classical** and has **ephemeral proof objects**. One idea for adapting PUMPKIN Pi to Isabelle/HOL is to first reify proof terms using Isabelle/HOL-Proofs, then apply a transformation based on the

1 Lean assumes **UIP**, which is incompatible with **univalence**. It is not yet clear to me what that assumption would mean for implementing the PUMPKIN Pi transformation in Lean. Everything else should carry over.

Transfer [79] package, and finally decompile the transformed terms to updated automation in the end.

POWER Proof repair is powerful, but not as powerful as it could be. In the future, proof repair tools should run fully automatically in response to proof assistant version updates. They should break down large changes into smaller pieces—perhaps by drawing on work in change and dependency management [80, 11, 27] to identify changes, then use the factoring transformation to break those changes into smaller parts. And they should support an even broader and more practical class of changes than they do now, like all of the changes identified in Section 6.1.

NATURAL USE The decompiler is a step toward a natural proof repair tool, but it still produces proof scripts heuristically, with no regard for style. Proof repair tools should ideally produce proof scripts that are natural for experts, regardless of style. Toward this end, I have just begun a promising project with **RanDair Porter**, *Emily First*, and *Yuriy Brun* on integrating the decompiler with the machine learning proof synthesis tool TacTok [57]. By ranking hints with TacTok, it should be straightforward to produce more natural proof scripts, even using fixed training data. More difficult—but highly valuable—will be to train the decompiler to match the style of the expert using the tool.

Proof Engineering for Practitioners

In the future, I want developing and maintaining proofs to be much easier for practitioners. But for that to happen, we need to work much more on usability. We need to create tools with scalable automation and smooth workflow integration, and continually improve them in response to feedback from user studies.

SCALABLE AUTOMATION Proof repair still struggles with repair over large libraries when many changes occur at once. The tools of the future should feature scalable automation that supports this elegantly, all while imposing little effort on the proof engineer. They should also be simple to extend with new optimizations, all while preserving correctness. One promising path toward this is integrating the PUMPKIN Pi transformation with e-graphs (Section 5.1.5), as e-graphs were built with these kinds of problems in mind. E-graphs were recently adapted to express path equality in **cubical type theory** [65]—a perfect fit for the PUMPKIN Pi transformation. E-graphs in other proof assistants, like those in **Lean** [145], may help with similar automation for repair tools for other proof assistants.

SMOOTH INTEGRATION A natural place for proof repair integration is at the level of an **IDE** or **CI** system. The tools of the future should integrate smoothly with IDEs like Proof General [2], and with CI systems like Travis [3]. CI support hinges on the ability to break large changes into smaller pieces—an outstanding challenge. At the level of the IDE, perhaps recording changes during development using the infrastructure from **REPLICA** will help circumvent this problem. Program repair tools with IDE integration like CatchUp! [75] can serve as inspiration for both infrastructure and user experience.

USER FEEDBACK Proof repair tools should continually adapt to feedback from the proof engineers who use them. This means user studies not just of proof engineers using proof assistants (as with **REPLICA**), but also of proof engineers using the proof repair tools themselves. The same principle applies to other proof engineering tools. Of particular use would be a large user study, enough to run statistically significant quantitative analyses and gather useful data for machine learning tools. Reaching enough users for this was one of the major challenges of **REPLICA**. Setting fewer barriers to registration and improving user study incentives may help address this.

Proof Engineering for Software Engineers

In the future, I want *any* software engineer to be able to develop and maintain verified software systems. But I do not believe that it will always be economically feasible or even desirable for software engineers to formally verify the *entire* system this way. Instead, I believe that the future of proof engineering lies in *mixed methods verification*: verification using multiple techniques while guaranteeing that their composition preserves correctness. I advocated for this **QED at Large**, and I implemented one case of this at Galois: using PUMPKIN PATCH to help a proof engineer interoperate between a constraint solver and Coq. The proof engineering tools of the future should integrate with tools familiar to software engineers, assist software engineers in redesigning software systems for verification, and help software engineers ensure those systems are robust to change.

FAMILIAR TOOLS Software engineers famously resist new tools. Proof engineering tools should integrate naturally with tools already familiar to software engineers. They should, for example, lift programs from familiar languages to proof assistants in a verified manner. They should help software engineers interactively generalize tests to specifications for writing proofs, or infer specifications from analyses of programs. They should check those specifications for correctness—perhaps using property-based testing tools like QuickChick [127, 95]—and integrate with debuggers to help software engineers fix incorrect

programs or specifications. They should prove as much as possible automatically, then prompt the software engineer with only the relevant questions needed to finish off the proofs. And they should integrate with proof repair tools to automatically adapt those proofs in response to changes. The experience of the future ought to exist along a continuum from testing to formal verification.

TOOL-ASSISTED REDESIGN Proof engineering often hinges on redesigning a system to be more amenable to verification.[2] The tools of the future should help software engineers with this. They should, for example, automatically identify relevant proof design principles (Section 5.1.2). They should help guide proof engineers through the process of redesigning software systems to use those principles, automating the manual effort by way of proof refactoring and repair. They should do all of this in a way that is trustworthy and transparent, perhaps even teaching the software engineer about proof design principles in the process.

TOOL-ASSISTED ROBUSTNESS Proof engineering tools should help software engineers build verified systems that are robust to change. They should, for example, infer more general specifications from changes to programs and specifications over time—perhaps leveraging some of the **differencing** algorithms and **proof term transformations** from this thesis. They should record breaking changes, and use those to suggest improvements to programs and specifications to prevent future breaking changes. They should integrate with refactoring and repair tools to automate those improvements to the extent possible, preserving guarantees and maintaining trust.

Proof Engineering for New Domains

In the future, I want domain experts from a broad spectrum of critical domains to be able to prove the properties about their software systems that matter to them—without any proof engineering expertise. I believe the best path to this future will build on **mixed methods verification**, but with a catch: the tools that are familiar to domain experts will vary by domain, as will the desired user experience. Accounting for this in proof engineering tools will mean partnering with domain experts directly, building new abstractions for critical domains like machine learning, cryptography, and medicine.

MACHINE LEARNING There has been only very preliminary verification of machine learning tools using proof assistants so far. This is a loss—proof engineering tools for machine learning experts could perhaps bring strong safety, fairness, and robustness guarantees to

2 A fun conversation with **James Wilcox** inspired this whole paragraph.

complex systems like autonomous vehicles or robots, or even to the social media, search, and advertising algorithms that many of us interact with on a daily basis. These tools could perhaps also help us build more reliable, understandable, and explainable neural networks. Current methods for verifying neural networks are not sufficient for this: automated checking of guarantees is slow for some properties, and fails on large neural networks.[3] One possible path to formally verifiable neural networks is to interactively factor neural networks into symbolic and neural parts. The former can hopefully be verified, with the latter capturing functionality that cannot easily be specified.

CRYPTOGRAPHY Proof engineering tools for cryptography already help ensure that cryptographic systems are not just designed, but also implemented correctly. But cryptography is constantly evolving, and proof engineering for cryptography is not keeping up. The tools of the future should keep up with the a variety of cryptographic technologies as they evolve—everything from quantum and post-quantum cryptography to emerging cryptographic proof systems to lattice-based cryptography.[4]

MEDICINE Medical devices are a natural domain for proof engineering, since strong guarantees about medical devices can save lives. Proof engineering tools for medical experts should bring strong guarantees to the medical devices of tomorrow. This should empower medical experts to build safer and more reliable medical devices, like pacemakers, insulin pumps, hearing aides, surgical robots, medication pumps, artificial organs, genetic arrays, neuromodulation implants, and genetic sequencing hardware and software.[5]

Proof Engineering for All

All of these new proof engineering technologies can drive the world of the future—the world of proof engineering for all. I believe that this will be a world of much more secure, reliable, and robust systems. Please work with me to make this world a reality!

3 This whole paragraph—but especially this sentence—is based on a really fun conversation with **Matthew Dwyer** at Virginia this past spring.

4 My Twitter followers helped me learn about some of the ongoing trends in cryptography and—unfortunately—also cryptocurrencies.

5 Most of these suggestions came from many of my wonderful Twitter followers. One also came from **Matthew Dwyer** at Virginia, and one came from **Matt Might**. All who contributed are acknowledged.

INDEX

BIBLIOGRAPHY

[1] Coq reference manual, programmable proof search, 1999-2020. URL: https://coq.inria.fr/refman/proofs/automatic-tactics/auto.html.

[2] Proof General, 2021. URL: http://proofgeneral.github.io/.

[3] Travis CI, 2021. URL: http://travis-ci.org/.

[4] User A. Software Foundations solution, 2017. URL: http://github.com/blindFS/Software-Foundations-Solutions.

[5] Mark Adams. Refactoring proofs with Tactician. In Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, pages 53–67, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-49224-6_6.

[6] Agda Development Team. Cubical type theory in Agda, 2005-2021. URL: http://agda.readthedocs.io/en/latest/language/cubical.html.

[7] Agda Development Team. The Agda wiki, 2007-2021. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

[8] Jesper Andersen and Julia L Lawall. Generic patch inference. *Automated software engineering*, 17(2):119–148, 2010.

[9] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. Internalizing representation independence with univalence. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. URL: https://doi.org/10.1145/3434293, doi:10.1145/3434293.

[10] Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher-dimensional type theory. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 680–693, New York, NY, USA, 2017. Association for Computing Machinery. URL: https://doi.org/10.1145/3009837.3009861, doi:10.1145/3009837.3009861.

[11] Serge Autexier, Dieter Hutter, and Till Mossakowski. Verification, induction termination analysis. chapter Change Management for Heterogeneous Development Graphs, pages 54–80. Springer-Verlag, Berlin, Heidelberg, 2010. URL: http://dl.acm.org/citation.cfm?id=1986659.1986663.

[12] Serge Autexier and Normen Müller. Semantics-based change impact analysis for heterogeneous collections of documents. In *Proceedings of the 10th ACM Symposium on Document Engineering*, DocEng '10, pages 97–106, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1860559.1860580, doi:10.1145/1860559.1860580.

[13] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328443.

[14] User B. Software Foundations solution, 2017. URL: http://github.com/marshall-lee/software_foundations.

[15] Henk Barendregt. *Foundations of Mathematics from the Perspective of Computer Verification*, pages 1–49. Springer International Publishing, Cham, 2013. doi:10.1007/978-3-319-00966-7_1.

[16] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002. doi:10.1023/A:1015761529444.

[17] Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 363(1835):2351–2375, 2005. doi:10.1098/rsta.2005.1650.

[18] Bruno Barras, Lourdes del Carmen González Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel, and Burkhart Wolff. Pervasive parallelism in highly-trustable interactive theorem proving systems. In *Intelligent Computer Mathematics: MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, pages 359–363, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39320-4_29.

[19] Bruno Barras, Carst Tankink, and Enrico Tassi. Asynchronous processing of Coq documents: From the kernel up to the user interface. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 51–66, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-22102-1_4.

[20] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '01, pages 57–71, London, UK, UK, 2001.

Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646793.704711.

[21] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of Automated Reasoning*, 61(1):333–365, Jun 2018. doi:10.1007/s10817-018-9455-7.

[22] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *Journal of Automated Reasoning*, 57(3):219–244, Oct 2016. doi:10.1007/s10817-016-9362-8.

[23] 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.

[24] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In *Intelligent Computer Mathematics*, pages 32–48, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-31374-5_3.

[25] Pierre Boutillier. *New tool to compute with inductive in Coq*. Theses, Université Paris-Diderot - Paris VII, February 2014. URL: https://tel.archives-ouvertes.fr/tel-01054723.

[26] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2594291.2594304, doi:10.1145/2594291.2594304.

[27] Ahmet Celik, Karl Palmskog, and Milos Gligoric. iCoq: Regression proof selection for large-scale verification projects. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 171–182, Piscataway, NJ, USA, 2017. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=3155562.3155588.

[28] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*,

PLDI 2019, pages 1054–1068, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314585.

[29] Michael Chan, Jos Lehmann, and Alan Bundy. GALILEO: A system for automating ontology evolution. *ARCOE-11*, page 46, 2011.

[30] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 270–286, New York, NY, USA, 2017. ACM. doi:10.1145/3132747.3132776.

[31] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM. doi:10.1145/2815400.2815402.

[32] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

[33] Jacek Chrząszcz. Modules in Coq are and will be correct. In *Types for Proofs and Programs*, pages 130–146, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-24849-1_9.

[34] Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification*. PhD thesis, KU Leuven, 2017.

[35] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the Univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2015.5.

[36] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs*, pages 147 – 162, Melbourne, Australia, December 2013. URL: https://hal.inria.fr/hal-01113453, doi:10.1007/978-3-319-03545-1_10.

[37] CompCert Development Team. Merge of the newmem and newextcalls branches, 2010. URL: http://github.com/AbsInt/CompCert/commit/a74f6b45d72834b5b8417297017bd81424123d98.

[38] Coq Development Team. The Coq proof assistant, 1989-2021. URL: http://coq.inria.fr.

[39] Coq Development Team. Tactics, 1999-2021. URL: http://coq.inria.fr/refman/proof-engine/tactics.html.

[40] Thierry Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986. URL: https://hal.inria.fr/inria-00076024.

[41] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[42] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1):423–453, June 2018. doi:10.1007/s10817-018-9458-4.

[43] Pierre-Évariste Dagand. The essence of ornaments. *Journal of Functional Programming*, 27:e9, 2017. doi:10.1017/S0956796816000356.

[44] Pierre-Evariste Dagand and Conor McBride. A categorical treatment of ornaments. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 530–539, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/LICS.2013.60.

[45] Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of functional programming*, 24(2-3):316–383, 2014.

[46] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. ReAssert: Suggesting repairs for broken unit tests. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 433–444. IEEE, 2009.

[47] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21401-6_26.

[48] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429094.

[49] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330, New York, NY, USA, 2013. ACM. doi:10.1145/2500365.2500587.

[50] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[51] 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. doi:10.1145/512950.512970.

[52] Louise A Dennis, Raul Monroy, and Pablo Nogueira. Proof-directed debugging and repair. In *Seventh Symposium on Trends in Functional Programming*, volume 2006, pages 131–140. Citeseer, 2006.

[53] Dominik Dietrich, Iain Whiteside, and David Aspinall. Polar: A framework for proof refactoring. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 776–791, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-45221-5_52.

[54] Mike Dodds. Proofs should repair themselves, 2020. URL: https://galois.com/blog/2020/12/proofs-should-repair-themselves/.

[55] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM. doi:10.1145/2517349.2522720.

[56] Martin Hötzel Escardó. A self-contained, brief and complete formulation of Voevodsky's univalence axiom. *CoRR*, abs/1803.02294, 2018. arXiv:1803.02294.

[57] Emily First, Yuriy Brun, and Arjun Guha. TacTok: Semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. URL: https://doi.org/10.1145/3428299, doi:10.1145/3428299.

[58] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards Formally Verified Optimizing Compilation in Flight Control Software. In *Bringing Theory to*

Practice: Predictability and Performance in Embedded Systems, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 59–68, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.PPES.2011.59.

[59] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing Q&A sites (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 307–318. IEEE, 2015.

[60] Thibault Gauthier and Cezary Kaliszyk. Matching concepts across HOL libraries. In *Intelligent Computer Mathematics*, pages 267–281, Cham, 2014. Springer. doi:10.1007/978-3-319-08434-3_20.

[61] Thibault Gauthier and Cezary Kaliszyk. Sharing HOL4 and HOL Light proof knowledge. In *Logic for Programming, Artificial Intelligence, and Reasoning: 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 372–386, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-48899-7_26.

[62] Thibault Gauthier and Cezary Kaliszyk. Aligning concepts across proof assistant libraries. *Journal of Symbolic Computation*, 90:89–123, 2019. Symbolic Computation in Software Science. doi:10.1016/j.jsc.2018.04.005.

[63] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1219–1219, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3180155.3182526, doi:10.1145/3180155.3182526.

[64] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[65] Emil Holm Gjørup and Bas Spitters. Congruence closure in cubical type theory. In *Workshop on Homotopy Type Theory / Univalent Foundations*, 2020. URL: https://www.cs.au.dk/~spitters/Emil.pdf.

[66] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008. URL: http://www.ams.org/notices/200811/tx081101382p.pdf.

[67] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010. doi:10.6092/issn.1972-5787/1979.

[68] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188. Springer, 2011.

[69] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 119–130, New York, NY, USA, 1978. ACM. doi:10.1145/512760.512773.

[70] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, GA, 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[71] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. URL: https://doi.org/10.1561/2500000010, doi:10.1561/2500000010.

[72] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. URL: http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz.

[73] Robert W Hasker and Uday S Reddy. Generalization at higher types. In *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, 1992.

[74] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In Michel Wermelinger and Tiziana Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*, pages 267–280, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[75] Johannes Henkel and Amer Diwan. CatchUp!: Capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 274–283, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1062455.1062512, doi:10.1145/1062455.1062512.

[76] Arend Heyting. Intuitionism. an introduction. 1956.

[77] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[78] HOL Light Development Team. HOL Light, 1996-2021. URL: `http://www.cl.cam.ac.uk/~jrh13/hol-light`.

[79] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs: Third International Conference*, CPP 2013, pages 131–146, Cham, 2013. Springer International Publishing. `doi:10.1007/978-3-319-03545-1_9`.

[80] D. Hutter. Management of change in structured verification. In *ASE 2000*, pages 23–31, Sept 2000. `doi:10.1109/ASE.2000.873647`.

[81] Isabelle Development Team. Isabelle, 1994-2021. URL: `http://isabelle.in.tum.de`.

[82] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 8–8, Berkeley, CA, USA, 2012. USENIX Association.

[83] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*, pages 152–167. Springer, Berlin, Heidelberg, 2004. `doi:10.1007/978-3-540-30142-4_12`.

[84] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, Aug 2014. `doi:10.1007/s10817-014-9303-3`.

[85] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap – the formally verified optimizing compiler CompCert. In *SSS'17: Safety-critical Systems Symposium 2017*, Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium, pages 163–180, Bristol, United Kingdom, February 2017. CreateSpace. URL: `https://hal.inria.fr/hal-01399482`.

[86] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 295–306, Washington,

DC, USA, 2015. IEEE Computer Society. URL: https://doi.org/10.1109/ASE.2015.60, doi:10.1109/ASE.2015.60.

[87] Gerwin Klein. Proof engineering considered essential. In *FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 16–21, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-06410-9_2.

[88] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014. doi:10.1145/2560537.

[89] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. doi:10.1145/1629575.1629596.

[90] Hsiang-Shang Ko and Jeremy Gibbons. Relational algebraic ornaments. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming*, pages 37–48. ACM, 2013.

[91] Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27, 2016.

[92] Thomas Kolbe and Christoph Walther. *Proof Analysis, Generalization and Reuse*, pages 189–219. Springer, Dordrecht, 1998. doi:10.1007/978-94-017-0435-9_8.

[93] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535841.

[94] Shuvendu Lahiri, Kenneth McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Foundations of Software Engineering (FSE'13)*. ACM, August 2013. URL: https://www.microsoft.com/en-us/research/publication/differential-assertion-checking-2/.

[95] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, December 2017. doi:10.1145/3158133.

[96] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program

repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2337223.2337225.

[97] Lean Development Team. Theorem proving in Lean, 2014-2021. URL: http://leanprover.github.io/tutorial/.

[98] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM. doi:10.1145/1111037.1111042.

[99] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. doi:10.1145/1538788.1538814.

[100] Xavier Leroy. Commit to CompCert: lib/integers.v, 2013. URL: http://github.com/AbsInt/CompCert/commit/6f3225b0623b9c97eed7d40ddc320b08c79c6518.

[101] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012. URL: https://hal.inria.fr/hal-00703441.

[102] letouzey. Commit to Coq: change definition of divide (compat with Znumtheory), 2011. URL: http://github.com/coq/coq/commit/81c4c8bc418cdf42cc88249952dbba465068202c.

[103] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2837614.2837617, doi:10.1145/2837614.2837617.

[104] Nicolas Magaud. Changing data representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.

[105] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *International Workshop on Types for Proofs and Programs*, pages 181–196. Springer, 2000.

[106] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset. *Empirical Software Engineering*,

22(4):1936–1964, 2017. URL: https://hal.archives-ouvertes.fr/hal-01387556, doi:10.1007/s10664-016-9470-4.

[107] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar 2016. doi:10.1007/s10817-015-9360-2.

[108] Conor McBride. Ornamental algebras, algebraic ornaments, 2011. URL: http://plv.mpi-sws.org/plerg/papers/mcbride-ornaments-2up.pdf.

[109] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2818754.2818811.

[110] Guillaume Melquiond. Commit to Coq: Make IZR use a compact representation of integers, 2017. URL: http://github.com/coq/coq/commit/a4a76c253474ac4ce523b70d0150ea5dcf546385.

[111] Na Meng, Miryung Kim, and Kathryn S McKinley. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

[112] Na Meng, Miryung Kim, and Kathryn S McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. IEEE Press, 2013.

[113] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. Type-directed diffing of structured data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 2–15, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3122975.3122976, doi:10.1145/3122975.3122976.

[114] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, January 2018. URL: http://doi.acm.org/10.1145/3105906, doi:10.1145/3105906.

[115] Anne Mulhern. Proof weaving. In *In Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2006.

[116] Toby Murray and P. C. van Oorschot. BP: Formal proofs, the fine print and side effects. In *IEEE Cybersecurity Development (SecDev)*, pages 1–10, Sep. 2018. doi:10.1109/SecDev.2018.00009.

[117] Magnus O. Myreen. Guide to HOL4 interaction and basic proofs, 2008-2021. URL: http://hol-theorem-prover.org/HOL-interaction.pdf.

[118] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[119] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 772–781. IEEE, 2013.

[120] nLab authors. beta-reduction. http://ncatlab.org/nlab/show/beta-reduction, July 2020. Revision 6.

[121] nLab authors. eta-conversion. http://ncatlab.org/nlab/show/eta-conversion, July 2020. Revision 12.

[122] nLab authors. initial algebra of an endofunctor. http://ncatlab.org/nlab/show/initial%20algebra%20of%20an%20endofunctor, May 2021. Revision 28.

[123] NuPRL Development Team. Nuprl, 1986-2021. URL: http://www.nuprl.org/.

[124] William F Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, 1992.

[125] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *ACM SIGOPS operating systems review*, volume 42, pages 247–260. ACM, 2008.

[126] Karl Palmskog, Ahmet Celik, and Milos Gligoric. piCoq: Parallel regression proving for large-scale verification projects. In *ISSTA*, pages 344–355, New York, NY, USA, 2018. ACM. doi:10.1145/3213846.3213877.

[127] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 325–343, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-22102-1_22.

[128] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *International Workshop on the Implementation of Logics (IWIL 2010)*, volume 2 of *EPiC Series*, pages 1–11. EasyChair, 2012.

[129] Pierre-Marie Pédrot. Ltac2: Tactical warfare. In *CoqPL 2019*, 2019.

[130] Yu Pei, Carlo A Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *arXiv preprint arXiv:1403.1117*, 2014.

[131] Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1987.

[132] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. `http://www.cis.upenn.edu/ bcpierce/sf`.

[133] Olivier Pons. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs*. PhD thesis, 1999.

[134] 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. ACM. URL: `http://doi.acm.org/10.1145/2771783.2771791`, `doi:10.1145/2771783.2771791`.

[135] RedPRL Development Team. The RedPRL proof assistant, 2015-2021. URL: `http://www.redprl.org`.

[136] 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. URL: `http://dx.doi.org/10.1561/2500000045`, `doi:10.1561/2500000045`.

[137] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2021, 2021.

[138] 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, page 99–113, New York, NY, USA, 2020. Association for Computing Machinery. URL: `https://doi.org/10.1145/3372885.3373823`, `doi:10.1145/3372885.3373823`.

[139] Talia Ringer and Nathaniel Yazdani. Pumpkin-git, 2018. URL: http://github.com/uwplse/PUMPKIN-git.

[140] 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, 2018.

[141] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2019/11081, doi:10.4230/LIPIcs.ITP.2019.26.

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

[143] Kenneth Roe and Scott Smith. CoqPIE: An IDE aimed at improving proof development productivity. In *Interactive Theorem Proving: 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, pages 491–499, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-43144-4_32.

[144] Amokrane Saibi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types: application à la Théorie des Catégories*. PhD thesis, Université Paris VI, Paris, France, 1999.

[145] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning: 8th International Joint Conference*, IJCAR 2016, pages 99–115, Cham, 2016. Springer International Publishing. URL: http://dx.doi.org/10.1007/978-3-319-40229-1_8, doi:10.1007/978-3-319-40229-1_8.

[146] Daniel Selsam, Percy Liang, and David L. Dill. Developing bug-free machine learning systems with formal mathematics. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3047–3056. PMLR, 06–11 Aug 2017. URL: http://proceedings.mlr.press/v70/selsam17a.html.

[147] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In Matt Kaufmann and Lawrence C. Paulson, editors,

Interactive Theorem Proving, pages 419–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[148] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 278–293, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-71067-7_23.

[149] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. hs-to-coq. https://github.com/antalsz/hs-to-coq, 2018-2021. Accessed: 2019-03-12.

[150] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: Univalent parametricity for effective transport. *Proc. ACM Program. Lang.*, 2(ICFP):92:1–92:29, July 2018. URL: http://doi.acm.org/10.1145/3236787, doi:10.1145/3236787.

[151] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. The marriage of univalence and parametricity. *Journal of the ACM*, 68(1):5:1–5:44, January 2021. doi:https://doi.org/10.1145/3429979.

[152] Shin Hwei Tan and Abhik Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 471–482, Piscataway, NJ, USA, 2015. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2818754.2818813.

[153] Amin Timany and Bart Jacobs. First steps towards cumulative inductive types in CIC. In *ICTAC*, 2015.

[154] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: https://homotopytypetheory.org/book.

[155] Paul Van Der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, pages 157–173. Springer, 2012. doi:10.1007/978-3-642-41582-1_10.

[156] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1831708.1831716, doi:10.1145/1831708.1831716.

[157] Makarius Wenzel. Isabelle/Isar–a generic framework for human-readable proof documents. *From Insight to Proof–Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.

[158] Makarius Wenzel. PIDE as front-end technology for Coq. *CoRR*, abs/1304.6626, 2013. arXiv:1304.6626.

[159] Makarius Wenzel. Shared-memory multiprocessing for interactive theorem proving. In *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 418–434, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39634-2_30.

[160] Makarius Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 515–530, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_33.

[161] Makarius Wenzel. Scaling Isabelle proof document processing, December 2017. URL: http://sketis.net/wp-content/uploads/2017/12/Isabelle_Scaling_Dec-2017.pdf.

[162] Makarius Wenzel. Further scaling of Isabelle technology, April 2018. URL: https://files.sketis.net/Isabelle_Workshop_2018/Isabelle_2018_paper_1.pdf.

[163] Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–183, Berlin, Heidelberg, 1999. Springer. doi:10.1007/3-540-48256-3_12.

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

[165] Karin Wibergh. Automatic refactoring for Agda. Master's thesis, Chalmers University of Technology and University of Gothenburg, 2019.

[166] Ambre Williams. *Refactoring functional programs with ornaments*. PhD thesis, 2020.

[167] Thomas Williams and Didier Rémy. A principled approach to ornamentation in ML. *Proc. ACM Program. Lang.*, 2(POPL):21:1–21:30, December 2017. doi:10.1145/3158109.

[168] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings*

of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016, pages 154–165, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2854065.2854081, doi:10.1145/2854065.2854081.

[169] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Trans. Softw. Eng.*, 43(1):34–55, January 2017. URL: https://doi.org/10.1109/TSE.2016.2560811, doi:10.1109/TSE.2016.2560811.

[170] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993532.

[171] Tianyi Zhang and Miryung Kim. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 665–676, Piscataway, NJ, USA, 2017. IEEE Press. URL: https://doi.org/10.1109/ICSE.2017.67, doi:10.1109/ICSE.2017.67.