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

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*. Still, while proof engineering—like software engineering—is about both development and maintenance, most proof engineering technologies so far have focused solely 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 foundational 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 have always believed the acknowledgments section to be one of the most important parts of a paper. But there is never enough room to thank everyone I want to thank. Now that I have the chance, though, I am having trouble figuring out where I should begin. In the past, when I had trouble figuring out where to begin, I just followed my sister.

MY SISTER   I followed my sister *Lymor* to Maryland for college. I followed her and when I got there, she told me to do research. I had no idea what that meant—neither of our parents are researchers—but she told me that if I wanted to go to graduate school, I needed to do research first. 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 forget about in acknowledgments sections. I do not think I could have even had the opportunity to write this if I had not listened to 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 and find my passion? Draw me to the world of programs and proofs? Instill in me the dream to become a professor—and the means to achieve that dream? I start in October, and I owe a lot of that to Lymor.

MARYLAND   At Maryland, I was really fortunate to have two math professors ask me to do research with them: *Kasso Okoudjou* and *Larry Washington*. This was a wonderful foray into the worlds of linear algebra and cryptology, and helped me build the skills I needed to become a researcher.

But I had also never written a program until college at Maryland. I am thankful that I was afraid enough of physics and statistics to take introductory computer science as my supporting sequence for my math degree. I am thankful that *Ida Chan* talked me into taking the sequence for computer science majors. I am thankful that I had a chance to attend the second ever *Google* Computer Science Summer Institute, where for the first time I felt empowered—so much that when I returned to Maryland, I decided to try to minor in computer science. And I am thankful that when I tried to minor in computer science, *Brandi K. Adams* talked me into picking it up as a second major instead.

As a mathematics and computer science double major, I had really wonderful opportunities to work at both *Carr Astronautics* and *Amazon*,

and I continued at Amazon for three years after I graduated. Both of these work experiences gave me really wonderful software engineering experience that helped me build good tools and learn to collaborate with other people—skills that were essential to this thesis work. My experiences as a software engineer also inspired a lot of the work that I did throughout graduate school. So I am grateful to everyone I worked with at both of those companies, and especially to *Musachy* and *Ernesto* for making my experience at Amazon so worthwhile and, above all, fun.

But I think the pivotal moment at Maryland came when I took an advanced undergraduate programming languages class taught by *Jeff Foster*. Jeff spent maybe one day during that class talking about the Curry-Howard Correspondence, which relates programs and proofs to one another. I thought it was the coolest thing ever, but I did not really understand it or its implications well at the time. It must have planted a seed or something, though, because I kept finding myself coming back to again and again years later. And this thesis really gets to the heart of Curry-Howard, treats proofs just like the programs they really are, differences them and transforms them to evolve them over time.

Jeff also introduced me to my advisor and helped me get into graduate school at the University of Washington. I spoke to him once in a while all the way until the very end of this graduate school journey, even talking to him about the faculty job search. I really cannot thank him enough.

MY ADVISOR   I could not have chosen a more beautiful and supportive place to spend my graduate school years than the University of Washington. And I could not have been advised by a more patient and wise advisor than *Dan Grossman*. Dan always gave me great advice, whether or not I wanted to hear it. Sometimes I *really* did not want to hear it, and even argued back a lot. And he just kind of patiently talked me through those moments and then often, months later, I would look back and be like, "oh yeah, he was right."

Dan also gave me the freedom that I needed to explore the problems that I love. He did push me a few times—just only when I really, really needed to be pushed. On the rare occasions when he did, I resisted a lot, and he just kind of accepted that as part of the job. Maybe he knew that I would appreciate it later. That I would come to understand that those were pivotal moments in my career. I do and I have.

MY LAB   The most amazing thing about my graduate school experience was being a part of the *PLSE* lab. I am about to cry as I write this part because I have not seen them in over a year. One day the pandemic hit and from then on, on less than a day's notice, the lab was closed from then on. If I could see everyone in my lab right now,

I would hug all of them. What a wonderful and amazing group of people, always there to support each other, to read papers and give feedback, to inspire new ideas, to chat, to get lunch, to send donuts, to spend time with—just everything about that lab was amazing. There is nothing like PLSE.

I love and miss all of you so much, and I really appreciate everything you all have done for me. Here is a (probably non-exhaustive) list of people who overlapped with me as students in PLSE and made my graduate school experience so amazing: *Chandrakana Nandi, Max Willsey, Martin Kellog, Jared Roesch, Gus Smith, Bill Zorn, Ben Kushigian, Steven Lyubomirsky, Jacob Van Geffen, Marisa Kirisame, Audrey Seo, James Wilcox, Doug Woos, Eric Mullen, Remy Wang, Konstantin Weitz, Mike He, Alex Polozov, Alex Sanchez-Stern, Nate Yazdani, Sarah Chasins, John Toman, Ryan Doenges, Melissa Hovik, Julie Newcomb, James Bornholt, Daryl Zuniga, Joe Redmond, Amanda Swearngin, Ali Köskal, Calvin Loncaric, Chenglong Wang, Mangpo Phitchaya Phothilimthana, Rohin Shah, Stuart Pernsteiner, Spencer Pearson, Sam Elliott, Maaz Ahmad, Dan Cascaval, Krzysztof Drewniak, Michael Flanders, Eunice Jun, Sam Kaufman, Rashmi Mudduluru, Sorawee Porncharoenwase, Thomas Schweizer, Luis Vega, Sahil Verma, Zhen Zhang, Amy Zhu, RanDair Porter, Jasper Hugunin, Logan Weber, Max Ruttenberg, Taylor Blau, Josh Pollock, Luke Nelson*, and *Shumo Chu*. I wish all of you nothing but joy and success for the rest of your lives.

MY FRIENDS (WIP)   I went through a really difficult time during graduate school, and there is a small group of friends whose help was life-saving. Those friends are *Zach Tatlock*, Alex Polozov, Chandrakana Nandi, *Roy Or-El, Vikram Iyer, Esther Jang, Anna Kornfeld Simpson, Mark Velednitsky, Sarah Chasins*, the entire *UCSD ProgSys* lab, *Marcela Mendoza, Anne Spencer Ross, Laura Chick*, and any fuzzy animals[1] I interacted with that year. You are the best.

WIP: Tony Chick. David Lasky. Ezgi. Erica. Danielle.

And I cannot begin to imagine my life over these years had I not met *Qi Cheng, Ellie Berry, Mer Joyce, Wade Gordon, Jasper Tran O'Leary*, and *Misha Kolmogorov*. Nothing but love to all of you.

MY STUDENTS (WIP)   Jasper Hugunin. Taylor Blau.

MY COAUTHORS (WIP)

MY COMMUNITY (WIP)   Cyril Cohen, Tej Chajed, Ben Delaware, Janno, James Decker, Bas Spitters, Derek Dreyer, Alexandra Silva, the Coq community (Emilio J. Gallego Arias, Enrico Tassi, Gaëtan Gilbert, Maxime Dénès, Matthieu Sozeau, Vincent Laporte, Théo Zimmermann, Jason Gross, Nicolas Tabareau, Cyril Cohen, Pierre-

---

1 May Boris' memory be a blessing.

Marie Pédrot, Yves Bertot, Tej Chajed, Ben Delaware, Janno), Michael Shulman (feels like univalence, like categorical coherence, like an endofunctor). Valentin Robert. Anders Mörtberg, Conor McBride, Carlo Angiuli, Bas Spitters, Jon Sterling, PL Twitter, other Twitter friends, Edward Z. Yang, Robert Rand. Matthew Dwyer at Virginia, Nathanael from Twitter, Matt Might, UVA folks, Quinn Wilton (all for medical device ideas). Jonathan Aldrich. David van Horn. Michael Hicks. Rebecca Turner from Twitter.

MY WRITING BUDDY (WIP)   Michelle Lee.

MY RUNNING BUDDIES (WIP)   Club Northwest.

MY PUPPY (WIP)   Belle.

MY GRANDPARENTS (WIP)

MY PARENTS (WIP)

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 [79] verified operating system (OS) microkernel, for example, is the effort of a team of proof engineers spanning more than a million lines of proof, costing over 20 person-years. Given a famous 1977 critique of verification [45] (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 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. Proof repair tools understand how software systems evolve, and use the crucial information that evolution carries to automatically evolve proofs about those systems. This gives us reason to believe.

1.1   THESIS

Proof repair falls under the umbrella of *proof engineering*: the technologies that make it easier to develop and maintain verified systems. 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 [79, 78], machine learning systems [135], distributed systems [**?**], constraint solvers [**?**], web browser kernels [**?**], compilers [87, 88], file systems [**?**], and even a quantum optimizer [**?**]. Practitioners have found these 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 [32], Isabelle/HOL [72], HOL Light [69], and Lean [86]. 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 Chapter 6.

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

1. implements a program using a functional programming language,

2. specifies what it means for the program to be correct, and

3. proves that the program satisfies the specification.

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

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

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

> **Thesis**: Changes in programs, specifications, and proofs carry information that a tool can extract, generalize, and
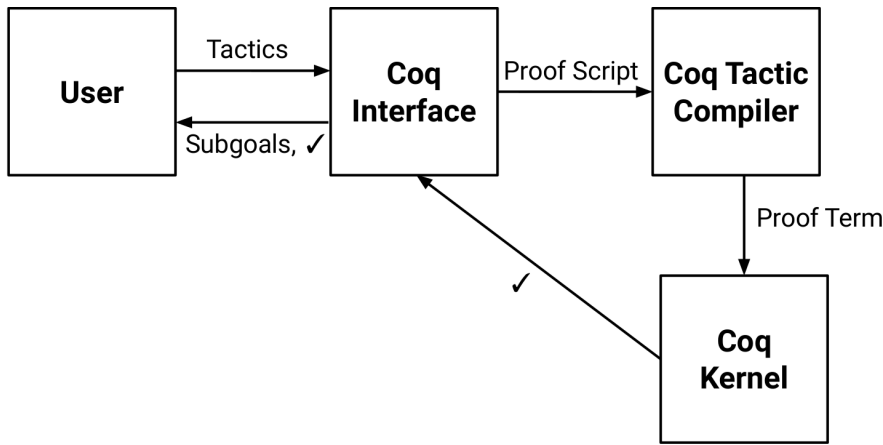
Figure 1: An illustration of the typical interactive workflow of using Coq to write a proof. The checkmark at the end represents correctness of the proof, which is communicated back to the proof engineer in the end.

apply to fix other proofs broken by the same change in practical use cases. A tool that automates this can save work for proof engineers relative to reference manual repairs.

1.2 APPROACH

The way that proof engineers typically write proofs can obfuscate the information that changes in programs, specifications, and proofs carry. The typical proof engineering workflow in Coq is interactive: 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 proof obligation for the base case). This loop of tactics and goals continues until no goals remain, at which point the proof engineer has constructed a high-level sequence of tactics called a *proof script*. To check the proof, the proof assistant compiles it down to a low-level representation called a *proof term*, then checks that the proof term has the expected type. Figure **??** illustrates this workflow.

The high-level language of tactics can abstract away important details that a proof repair tool needs, but the low-level language of proof terms can be brittle and challenging to work with. Crucially, though, the low-level language comes with lots of structure and strong guarantees. My approach to proof repair works in the low-level language to take advantage of that. It then builds back up to the high-level language in the end.

By working at the low-level language, it 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, it can in the end produce proofs that integrate more naturally with proof engineering workflows.

This works using a combination of semantic differencing and program transformations in this low-level language. In particular, it uses a semantic differencing algorithm to extract information from a breaking change in a program, specification, or proof. It then combines that with program 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.

1.3 RESULTS

The technical results of this thesis are threefold:

1. the **design** of differencing algorithms & program transformations for proof repair,

2. an **implementation** of a proof repair tool suite, and

3. **case studies** to evaluate the tool suite on real proof repair scenarios.

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

> **Thesis Proof**: Changes in programs, specifications, and proofs carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change (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, in some cases, 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 that information corresponding to that 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. These design is guided heavily by foundational developments in depenedent type theory; the theory is sprinkle throughout as appropriate. More details including limitations are in the corresponding chapters.

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 implements two kinds of proof repair: proof repair by example (Chapter 3) and proof repair across type equivalences (Chapter 4). Notably, since all repairs that PUMPKIN PATCH produces are checked Coq in the end, PUMPKIN PATCH does not extend *Trusted Computing Base* (TCB): the set of unverified components that the correctnes of the proof development depends on [**?**]. In total, PUMPKIN PATCH is about 15000 lines of code implemented in OCaml. These 15000 lines of code consist of three plugins and a library, which together bridge the gap between the theory supported by design and the practical proof repair needed for the case studies. Toward that end, five notable features include:

1. a preprocessing tool to support features in the implementation language missing from the theory,

2. a prototype decompiler from proof terms to proof scripts for better workflow integration,

3. optimizations for efficiency,

4. meaningful error messages for usability, and

5. additional automation for applying patches.

More details and other features are in the corresponding chapters.

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.

1.4   READING GUIDE

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 (Chapters 3 and 4). It concludes with a discussion of related work (Chapter 5), followed by a reflection on the thesis and how it can inform the next era of proof engineering (Chapter 6).

Expected Background

This thesis assumes some background in type theory and proof engineering. 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 [125], which includes a detailed list of resources and is available for free on my website.[1] I recommend that readers with less background on proof engineering, dependent type theory, or the Coq proof assistant 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, though you may always contact me with questions.

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 [129]. CPP 2018.

- **Talia Ringer**, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof Repair Across Type Equivalences [126]. PLDI 2021.

It also includes material from three other papers:

- **Talia Ringer**, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq [130]. ITP 2019.

- **Talia Ringer**, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. REPLica: REPL Instrumentation for Coq Analysis [127]. CPP 2020.

- **Talia Ringer**, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at Large: A Survey of Engineering of Formally Verified Software [125]. Foundations and Trends® in Programming Languages: Vol. 5: No. 2-3, pp 102-281. 2019.

Below is a map from each of these papers to corresponding parts of the thesis, along with an explanation of what is new in this thesis and what is omitted. All of these papers can be found for free on my website.

ADAPTING PROOF AUTOMATION TO ADAPT PROOFS   The bulk of Chapter 3 comes from this paper, though it is significantly reorganized and reframed. In addition, the introducton and conclusion of Chapter 3 are fresh content. Sections 3.2, 3.3, 3.4, and 3.5 all include additions and elaborations not found in the original paper. Chapter 5 includes some related work from this paper, and Chapter 6 includes some future work from this paper.

---

1 https://dependenttyp.es

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 it is likewise reorganized and reframed. In addition, the conclusion of Chapter 4 is fresh content. Sections 4.2, 4.3, 4.4, and 4.5 all include additions and elaborations not found in 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 uses a simplified version of the search algorithm from this paper as an example. Section 4.6.2 includes the evaluation from this paper with some additional context. Chapter 5 includes a small amount of related work from this paper.

REPLICA   Section 2.2.2 includes a few samples of this paper, as does the abstract.

QED AT LARGE   Chapter 2 includes a few samples of this 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. Below is a summary of the contributions of each of those coauthors, indexed for later reference. The contributions of my faculty and professional coauthors were of course also extremely valuable.

NATHANIEL YAZDANI   I worked with Nate starting from when he was an undergraduate student. Nate 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. Three of Nate's contributions stand out to me:

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 for the final paper that appears in this thesis. RanDair 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 that paper. This decompiler was necessary for integrating proof repair tools with real proof engineering workflows, and it continues to inspire exciting work to this day.

ALEX SANCHEZ-STERN   Alex worked with me as a PhD student on a user study of proof engineers during my visit to another university. Alex designed, implemented, deployed, 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 Alex.

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

Pronouns

In this thesis, I use "I" to refer to work that I did as part of my thesis work, even though of course no work happens in a vacuum. I use the names of my coauthors like "Nate" or "RanDair" when referring to work that my coauthors did, when I was operating 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 pronouns that the person prefers.

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. 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 [105]. 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 [87, 88] verified optimizing C compiler. Both the back-end and front-end compilation passes of CompCert have been verified, ensuring the correctness of their composition [75]. 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 [51]. It has also stood up to rigorous testing: while the test generation tool Csmith [159] 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 of approximately 35000 lines of Coq 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 an OS microkernel, for example, consisted of 480000 lines of specifications and proofs [78]. Proof engineering technologies make it possible to develop verified systems at this scale. See Ringer and friends 2019 [125] for a comprehensive overview of proof engineering.

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

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 for which PUMPKIN PATCH is implemented. I motivate this 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 in Coq to begin with. In the introduction, I briefly explained the workflow for using Coq to develop a verified system:

> The proof engineer does three things:
>
> 1. implements a program using a functional programming language,
>
> 2. specifies what it means for the program to be correct, and
>
> 3. proves that the program satisfies the specification.

That functional programming language is a rich functional programming language called *Gallina*. It is possile 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 in the introduction as well, when I explained the typical proof development workflow in Coq:

> 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 proof obligation for the base case). This loop of tactics and goals continues until no goals remain, at which point the proof engineer has constructed a high-level sequence of tactics called a *proof script*. To check the proof,

```
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 tool [?].

the proof assistant compiles it down to a low-level representation called a *proof term*, then checks that the proof term has the expected type.

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 we will soon see.

In this thesis, I will not teach you all of Coq. Good sources for learning more about Coq include the books Certified Programming with Dependent Types [28] and Software Foundations [121], and the survey paper by Ringer and friends 2019 [125]. 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 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. Coq checks this proof and lets us now that our proof is correct, so our zip function is verified.

PROGRAM   The list zip function is in Figure 2. It takes as arguments two lists l₁ and l₂ of possibly different types T₁ and T₂, and zips them together into a list of pairs (T₁ * T₂). For example, if the input lists are:

```
(* [1; 2; 3; 4] *)
l₁ := cons 1 (cons 2 (cons 3 (cons 4 nil))).

(* ["x"; "y"; "z"] *)
l₂ := cons "x" (cons "y" (cons "z" nil)).
```

then zip applied to those two lists returns:

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

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

```
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.

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

Otherwise, the implementation is fairly standard. If `l₁` is `nil` (first case), or if `l₂` is `nil` (second case), then `zip` returns `nil`. Otherwise, `zip` combines the first two elements of each list into a pair ((`t₁`, `t₂`)), then sticks that in front of (using `cons`) the result of recursively calling `zip` on the tails of each list (`zip tl₁ tl₂`).

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 `l₁` and `l₂`. We do this by defining a type `zip_preserves_length` (Figure 3, left), which in Coq we state as a `Theorem`. This theorem takes advantage of Gallina's rich type system to quantify over all possible input lists `l₁` and `l₂`. 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, right) 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 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:

```
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 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 type and step past, Coq responds by refining the goal into some subgoal and updating the context. We continue this until no goals remain. The survey paper [125] has a good overview of the tactic language in Coq and in other proof assistants, plus different interfaces and IDEs for writing proofs interactively and screenshots of them 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)
∀ {T₁} {T₂} (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 T₁ T₂ l₁.
```

This is essentially the equivalent of the natural language proof strategy "assume arbitrary T_1, T_2, and l_1." That is, it moves the universally quantified arguments from our goal into our context:

```
T₁ : Type
T₂ : Type
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:

```
T₁ : Type
T₂ : Type
_____(1/2)
∀ l₂ : list T₂,
  length nil = length l₃ →
  length (zip nil l₂) = length nil.
```

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

In the inductive case:

```
T₁ : Type
T₂ : Type
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 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`. What happens when we write `Defined` is that Coq compiles that proof script down to a proof term. It then checks the type of that term and ensures that it is exactly 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-leve proof term do not matter to us, 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 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.

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 write proofs, it is thanks to the proof terms these compile down to that Coq is able to check our proof in the end. These proof terms are in a rich functional programming language called 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,

```
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 l₁¹ as [|t₁ tl₁ IHtl₁].
  - auto.²
  - intros l2. induction l2³ 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_rect¹ (fun (l₁ : list T₁) => ...)
    (fun (l₂ :  list T₂) _=> eq_refl)²
    (fun (t₁ : T₁) (tl₁ : list T₁) (IHtl₁ : ...) (l₂ : list T₂) =>
      list_rect³ (fun (l₂ : list T₂) => ...)
        (fun (H : ...) => eq_sym H)⁴
        (fun (t₂ : T₂) (tl₂ : list T₂) (IHtl₂ : ...) =>
          fun (H : ...) => eq_ind_r ... eq_refl (IHtl₁ ...)⁵)
        l₂³)
    l₁¹
    l₂.
```

Figure 4: A proof script (top) and corresponding proof term (bottom) in Coq that shows that the list zip function preserves its length. Some details of the proof term are omitted for simplicity. Corresponding parts of the proof are highlighted in the same color and annotated with the same number; the rest is boilerplate.

```
                          nat_rect :
                           ∀ (P : nat → Type),
Inductive nat :=             P 0 → (* base case *)
| O : nat                   (∀ (n : nat),
| S : nat → nat.              (* inductive case *)
                              P n → P (S n)) →
                           ∀ (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).

specifications, and proofs in Coq, and have a small part of Coq check those proofs in the end.

2.1.2.1 *Gallina*

Gallina is a rich functional programming language with support for writing proofs. To see the power of Gallina, let us dissect our proof that `zip` preserves its length. I mentioned earlier that the `nat` and `list` datatypes, and the `length` function itself, can all be found inside of the Coq standard library. But looking into these datatypes and this function already points to many useful features of Gallina.

Each of `nat` and `list` in Gallina is an *inductive type*: it is defined by its *constructors* (ways of constructing a term with that type). A `nat` (Figure 5, left), for example, is either `0` or the successor `S` of another `nat`; these are the two constructors of `nat`.

Every inductive type in Gallina comes equipped with an *eliminator* or 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 for natural numbers, which Coq calls `nat_rect`. This eliminator states that a statement `P` (called the inductive *motive*) about the natural numbers holds for every number if it holds for `0` in the base case and, in the inductive case, assuming 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.

```
                              list_rect :
                               ∀ {T : Type} (P : list T → Type
Inductive list {T : Type} :=      ),
| nil : list T                   P nil → (* base case *)
| cons :                         (∀ (t : T) (tl : list T),
   T → list T → list T.            (* inductive case *)
                                   P tl → P (cons t tl)) →
                                 ∀ (l : list T), P l.
```

Figure 6: The type of polymorphic lists `list` in Coq defined induc-
tively by its two constructors (left), and the type of the
corresponding eliminator or induction principle `list_rect`
that Coq generates (right). The curly brace notation means
that the type parameter `T` is implicit in applications.

```
Fixpoint length {T} l :=      Definition length {T} l :=
  match l with                  list_rect T (fun _ => nat)
  | nil => O                      O
  | cons t tl =>                  (fun t tl (length_tl : nat) =>
     S (length tl)                  S length_tl)
  end.                            l.
```

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

One interesting thing about the types of these eliminators `list_rect`
and `nat_rect` 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 functions and proofs, like the
`length` function (Figure 7, right). More standard is to write functions
using pattern matching and guarded recursion, like the `length` func-
tion from the Coq standard library (Figure 7, left). Both of these two
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 `0`, and the length of any other list
is just 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 just a constant that refers to a function
itself defined using pattern matching and guarded recursion. In fact,
eliminators are equally expressive to pattern matching and guarded
recursion [**?**].

For the sake of this thesis, however, I will assume *primitive elimina-
tors*: eliminators that are a part of the core syntax and theory itself, and
that do not reduce to terms that use pattern matching and guarded
recursion. Likewise, when I show Gallina code, from this point for-

```
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, taken from an existing tool [**?**] and translated to use eliminators.

ward, 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 guarded recursion, I preprocesss the code using a tool by my coauthor Nate Yazdani (more about this later). Figure 8, for example, shows the `zip` function after running this tool. In the rest of the paper, I skip this preprocessing step in examples, but I describe it more in the implementation section later.

With that in mind, we can now look back at the proof script and correpsondence proof term in Figure 4. There is a correpsondence 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 an eliminator over the inductive equality type called `eq_ind_r`.

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 from proof terms back up to proof scripts that makes it possible for PUMPKIN PATCH to work over highly structured Gallina terms, but produce 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 Constructions (CoC), a variant of the lambda calculus with polymorphism (types that dependent on types) and dependent types (types that depend on terms) [34]. CoC with an infinite universe hierarchy is called CoC_ω.

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

$\langle t \rangle ::= \langle v \rangle \ | \ \langle s \rangle \ | \ \Pi \left(\langle v \rangle : \langle t \rangle \right) . \ \langle t \rangle \ | \ \lambda \left(\langle v \rangle : \langle t \rangle \right) . \ \langle t \rangle \ | \ \langle t \rangle \ \langle t \rangle$

Figure 9: Syntax for CoC_ω with (from left to right) variables, sorts, dependent types, functions, and application.

$\langle t \rangle ::= \ ... \ | \ \text{Ind} \left(\langle v \rangle : \langle t \rangle \right) \{ \langle t \rangle , ... , \langle t \rangle \} \ | \ \text{Constr} \left(\langle i \rangle , \langle t \rangle \right) \ | \ \text{Elim}(\langle t \rangle ,$
$\langle t \rangle) \{ \langle t \rangle , ... , \langle t \rangle \}$

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

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.

The syntax for CIC_ω is in Figure 10), building on syntax from an existing paper [141]; the typing rules are standard and omitted. CIC_ω extends CoC_ω with inductive types [35]. As in Gallina, inductive types are defined by their constructors and eliminators. Consider the inductive type nat of unary natural numbers that we saw in Figure 5, this time in CIC_ω:

```
Ind (nat : Set) {
  nat,
  nat → nat
}
```

where the O constructor type is the zeroth element in the list, and the S constructor type is the first element. Accordingly, the terms:

```
  Constr (0, nat)
```

and:

```
  Constr (1, nat)
```

refer to the constructors O and S, respectively.

As in Gallina, nat comes associated with an eliminator. 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 natural number n with motive P, we write:

```
Elim (n, P) {
  f_O,
  f_S
}
```

where functions:

$\text{f}_O \ : \ \text{P} \ \text{O}$

and:

$\text{f}_S \ : \ \Pi \ (\text{n} : \text{nat}) \ . \ \Pi \ (\text{IHn} : \text{P} \ \text{n}) \ . \ \text{P} \ (\text{S} \ \text{n})$

prove the base and inductive cases, respectively. When n, P, f_O, and f_S are arbitrary, this statement has the same type as nat_rect in Gallina.

CONVENTIONS   Throughout this thesis, I assume an inductive type Σ with constructor $\exists$ and projections π_l and π_r, and an equality type = with constructor eq_refl. I use $\vec{t}$ and $\{t_1, \ldots, t_n\}$ to denote lists of terms. I often present algorithms 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. The default rule 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 a few important differences. More information is on the website, but two differences are relevant to repair: The first is that Gallina lacks primitive eliminators, as I mentioned earlier. The second notable difference is that Gallina has constants that define terms—later on, this will help with building optimizations for repair tools.

Otherwise, a proof repair tool for Gallina can harness the power of CIC_ω. This type theory is fairly simple, but 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 COQ   But this structure can be difficult for proof engineers to work with, which is why proof engineers typically rely on the tactics we saw in Section 2.1.1. Tactics more generally are a form of *proof automation*, and 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 hap-

pen 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*

At its core, a verified system has three parts, corresponding to the workflow from Section 2.1:

1. programs,

2. specifications, and

3. proofs.

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 **??**. We had two choices for the specification of `zip_preserves_length`. We chose the weaker specification on the left of Figure 3. This gives us some freedom in how we implement our `zip` function. At some point, 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 right of Figure 3 We may even wish to use a datatype more expressive than `list`, as we will see in Section **??**. Any of these changes can break proofs in our proof development.

CHANGING OUR PROGRAM   Our specification of `zip_preserves_length` gives us some freedom 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) =>
      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:

```coq
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.

CHANGING OUR SPECIFICATION   Suppose we had instead chosen the stronger specficiation on the right of Figure 3, and kept our `zip` function the same. We can update our proof accordingly, but after changing this specification, other proofs may break. For example, if we had written a lemma for the `cons` case:

```coq
Lemma zip_preserves_length_cons :
  ∀ {T₁ : Type} {T₂ : Type} (l₁ : list T₁) (l₂ : list T₂) (t₁ : T
      ₁) (t₂ : T₂),
    length l₁ = length l₂ →
    length (zip (cons t₁ l₁) (cons t₂ l₂)) = S (length l₁).
Proof.
  intros T₁ T₂ l₁ l₂ t₁ t₂ H.
  simpl. f_equal.
  rewrite zip_preserves_length; auto.
Defined.
```

that followed by `zip_preserves_length`, then after the change, this proof would break.

We would have two choices to fix it. Either we could leave our specification alone, and fix our proof. In that case, the proof would look like this instead (with the difference highlighted):

```coq
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, we could strengthen the specification of that lemma as well, and leave the proof alone:

```coq
Lemma zip_preserves_length_cons :
  ∀ {T₁ : Type} {T₂ : Type} (l₁ : list T₁) (l₂ : list T₂) (t₁ : T
      ₁) (t₂ : T₂),
    length (zip (cons t₁ l₁) (cons t₂ l₂)) = S (min (length l₁) (
        length l₂)).
```

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

But this could continue to break other downstream proofs that depend on `zip_preserves_length_cons`, causing a cascading effect of change.

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 5.1.4). For example, they might use information hiding techniques [157, **?**] 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. To find evidence of this in the real world, Alex and I built a Coq plugin called REPLICA (REPL Instrumentation for Coq Analysis) that 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. We used REPLICA to collect a month's worth of granular data on the proof developments of 8 intermediate to expert Coq users. 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.[1]

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 wraps 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.

---

[1] http://github.com/uwplse/analytics-data

```
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.

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. And this is an big problem for proof developments at the scale of CompCert or seL4 (Section 5.1.2).

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 enginers 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 respresenting changes in programs, specifications, and proofs over time. So when traditional proof automation breaks in respose 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 changes. In contrast with traditional proof automation, proof repair views programs, specifications, and proofs as fluid entities. When a program or specification changes and this breaks many proofs, proof repair extracts informatiton from those changes, generalizes it, and applies it to fix other proofs broken by the same change.

The name of proof repair is inspired by program repair [103, 56], or automatically fixing bugs in programs. But my proof repair tools work differently from program repair tools, using a combination of semantic differencing and program 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:

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

My proof repair tools are the tools that automate this. These tools accomplish this using a combination of differencing and program transformations. *Differencing* compares the old and new version of the program, specification, or proof that has changed, and from that extracts 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 show you 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 [103]. 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 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*

My proof repair tools operate over proof terms in Gallina. In terms of the approach, what this means is that differencing operates over Gallina terms, and is guided by their semantics to narrow down the search space—in other words, it is a kind of *semantic differencing*. The program transformations then use that diff to transform some proof term to a more general patch (in Chapter 3) or the patched proof itself (in Chapter 4)—in other words, 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 different tool that supports a different class of changes, and 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.

3

PROOF REPAIR BY EXAMPLE

The first tool in the PUMPKIN PATCH proof repair plugin suite is the original namesake PUMPKIN PATCH plugin, which I implemented in 2018 as a prototype to show that proof repair was possible. To prevent confusion, when I refer to the PUMPKIN PATCH prototype and not to the plugin suite as a whole, I will abbreviate it as PUMPKIN. As PUMPKIN is a prototype, it includes only preliminary automation for *applying* the information that it finds, and supports changes that are limited in scope in a way that is fundamental to the approach. The results for PUMPKIN are also preliminary, and the implementation does not yet integrate smoothly into proof engineering workflows. The PUMPKIN Pi extension that I will show you in Chapter 4 will address these limitations.

PUMPKIN implements *proof repair by example*—so called because of its resemblence to programming by example (Section **??**). 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 [99]. This commit redefined injection from integers

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

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 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, the 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, and saw preliminary success (Section 3.6). The later PUMPKIN Pi extension to PUMPKIN PATCH brought this vision to life (Chapter 4).

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). PUMPKIN implements a prototype of this.

To use 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 term in Gallina.

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. PUMPKIN has only preliminary support for automatically *applying* the extracted and generalized information (see Section 3.5), though the later PUMPKIN Pi extension addresses this limitation (Chapter 4).

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 define new terms. In this case, PUMPKIN extends Coq with a new command called `Patch Proof`, with the syntax:

```
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 are patches then return patches
    return  failure
```

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

```
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 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`:

```
Proof.
  ...
  apply old.✓
  ...
Defined.
```

so that they fail after migrating to `new`:

```
Proof.
  ...
  apply new.✗
  ...
Defined.
```

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

---

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

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

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 tranforms 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:

```
Proof.
  ...
  apply patch. apply new.✓
  ...
Defined.
```

or by using the automation in Section 3.5.

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 does not support any changes that add, remove, or rearrange any hypotheses. The Pumpkin Pi extension to Pumpkin Patch in Chapter 4 supports a broad class of changes in datatypes that may change in these ways.

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 arbitrary `x`, `H`, `P`, and `Q`, Pumpkin searches for a patch with this 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.

---

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 when developing Pumpkin Pi.

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 compiled 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.
```

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 the argument `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**: `old_proof, new_proof`, an instance `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 itself, 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 in the end. All terms that PUMPKIN produces type check in the end.

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 are what take each candidate and tries to modify it to produce a term that *does* generalize to other contexts. 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.
```

The reusable patch that PUMPKIN is looking for, however, should have this type:

$$\forall\ n\ m\ p,\ n <= m \rightarrow m <= p \rightarrow n <= p \rightarrow n <= p + 1$$

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 patch candidates into reusable proof patches:

1. *patch specialization* to arguments,

2. *patch generalization*[3] of arguments or functions,

3. *patch inversion* to reverse a patch, and

---

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.

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* in the instance 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 11), 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.
```

PUMPKIN then *specializes* this generalized candidate to `p` (line 16), 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 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 of a given instance leads to a final `goal` type is not useful to the proof engineer, but this cannot sacrifice soundness: every `patch` PUMPKIN produces type checks in the end 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, it is 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_ω. 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 induction principle, with slightly different motives. Accordingly, differencing knows that there are two places to look for candidates, namely the base case (line 13) and the inductive case (line 14). 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 desired type for the instance 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 (line 14), the inductive hypothesis `IHle` 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 (Section 3.3.1) and limitations (Section 3.3.2) of these algorithms. Section 3.5.1.2 describes the implementation in PUMPKIN.

3.3.1 *Design*

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.

For historical reasons, as PUMPKIN was a prototype, these algorithms are not formalized. Instead, I describe them at a high level, as heuristics in CIC_ω. I still refer to the judgment $\Gamma \vdash (t_a, t_b, G) \Downarrow_d \vec{t}$ as the interface, even though I do not provide the derivations. In Chapter 4, I will formalize a more elegant differencing algorithm for the PUMPKIN

Pi extension, building on some of the insighst from differencing in the PUMPKIN prototype.

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 sub-term 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 13). 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:

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

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 hypothesis that need not show up in the goal type.

ELIMINATORS   Recall that inductive types in CIC_ω come equipped with primitive eliminators. The semantic differencing algorithms views inductive types as *trees* that represent these eliminators. In these
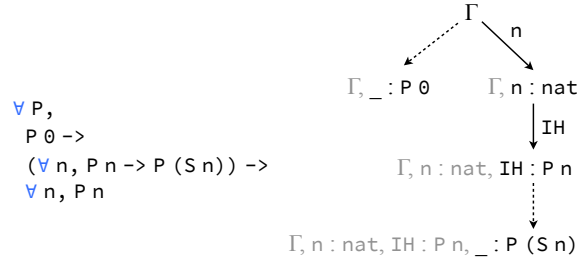
Figure 15: The type of (left) and tree for (right) the eliminator of `nat`. The solid edges represent hypotheses, and the dotted edges represent the proof obligations for each case in an inductive proof.

trees, every node is a type context, and every edge is an extension to that type context with a new term.[4] 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 good 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:

$Q \rightarrow P$

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:

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

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 $Q \rightarrow P$.

---

4 These trees are inspired by categorical models of dependent type theory [68].

3.3.2 *Limitations*

This section describes a few fundamental limitations of differencing in PUMPKIN, and whether they are addressed in the later PUMPKIN Pi extension. Limitations that are due to the choice of implementation strategy are in Section 3.5.1.2.

HEURISTICS   Differencing in PUMPKIN is heuristic-based. This is to some degree inevitable, as the space of possible changes is infinite. In particular, it is the cartesian product of the space of every possible old term and type, combined with every possible new term and type. Still, this is partly historical: for the PUMPKIN prototype, the heuristics were designed bottom-up, and were in many cases too narrow or redundant. The extension to differencing in PUMPKIN Pi not only supports a large class of changes not supported by PUMPKIN, but also abstracts many more details of heuristics.

ISOLATING CHANGES   Differencing supports only incremental changes. The REPLICA user study that I ran with Alex showed that, during proof development, proof engineers often make incremental changes— so this should be helpful for development already. But changes in existing artifacts like Git repositories may capture multiple changes at the same time. An outstanding challenge for differencing that neither PUMPKIN nor PUMPKIN Pi yet support is to break down large, composite changes into small, isolated changes. Isolating changes would help with extracting repair benchmarks from artifacts, supporting library and version updates, and integrating with CI systems. One idea for this is to draw on work in change and dependency management [71, 11, 26] to identify changes, then use the factoring transformation to break those changes into smaller parts.

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_ω. This section describes the design (Section 3.4.1) and limitations (Section 3.4.2) of these transformations. Section 3.5.1.3 describes the implementation in PUMPKIN.

3.4.1 *Design*

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-direct 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.

For historical reasons, as Pumpkin was a prototype, these tranformations are not formalized. Instead, I describe at a high level the design of each of the four transformations in CIC_ω. I still refer to the corresponding judgments as the interfaces, even though I do not provide the derivations. In Chapter 4, I will formalize a more elegant proof term transformation for the Pumpkin Pi extension, building on some of the insights from the transformations in the Pumpkin prototype.

specialization   Sometimes, patch candidates are too general. Specialization takes a candidate that is too general, and specializes it to specific arguments as determined by the difference in terms. To find a patch for Figure 14, for example, Pumpkin specialized the patch candidate to p to produce the final patch.

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 [?]. Section 3.5.1.3 decribes 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 the argument m (before applying specialization).

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

$Q\ t\ \to P\ t$

by the common argument:

$\Pi(\texttt{t'} : T),\ Q\ \texttt{t'} \to P\ \texttt{t'}$

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

$P \ t' \ \to P \ t$

by the common function:

$\Pi(\text{Q} : T), \ \text{Q}t' \to \text{Q}t$

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 this candidate by `m`:

$\lambda(H : n <= m).\text{le\_plus\_trans} \ n \ m \ (S \ O) \ H$
$: n <= m \to n <= \text{plus} \ m \ 1$

where `<=` 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`:

$\lambda(\text{n0} : \text{nat}).\lambda(H : n <= m).\text{le\_plus\_trans} \ n \ m \ (S \ O) \ H)$
$: \Pi(\text{n0} : \text{nat}), n <= m \to n <= \text{plus} \ m \ 1$

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

$\lambda(\text{n0} : \text{nat}).\lambda(H : n <= \text{n0}).\text{le\_plus\_trans} \ n \ \text{n0} \ (S \ O) \ H)$
$: \Pi(\text{n0} : \text{nat}), n <= \text{n0} \to n <= \text{plus} \ \text{n0} \ 1$

In general, generalization is a kind of anti-unification problem, as the terms and types may be reduced, so that the common function or argument does not appear explicitly. This is of course undecidable. 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 genearlize 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.

---

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

INVERSION   Sometimes, when two types are propositionally equal, candidate patches may appear in the wrong direction. For example, consider two list lemmas (this example is in Coq rather than CIC_ω for simplicity):

```
old : ∀ l' l, length (l' ++ l) = length l' + length l
new : ∀ 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 l' l (H : old l' l) :=
  eq_rect_r ... (rev_length l)
 : ∀ l' l, old l' l → new l' l
```

The component can invert this to get the patch:

```
patch l' l (H : new l' l) :=
  eq_rect_r ... (eq_sym (rev_length l))
 : ∀ l' l, 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 : ∀ l l',
  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 : ∀ l l',
  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 component, 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, equality is symmetric, so the component can invert any application of the equality eliminators. I will explain this more in Section 3.5.1.2.

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 3.5.1.2. 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 $\to$ Y for some Y. If it finds such a term, then it recursively searches for a term with type Y $\to$ Z. It maintains all possible paths of factors along the way, and it discards any paths that cannot reach Z. It does not yet support paths where Y depends on X.

3.4.2 *Limitations*

This section describes a few fundamental limitations of the transformations in Pumpkin, and whether they are addressed in the later Pumpkin Pi extension. Limitations that are due to the choice of implementation strategy are in Section 3.5.1.3.

UNDECIDABILITY   Several of the transformations are undecidable. For example, generalization will necessarily sometimes fail, since fundamentally it relies on a kind of unification. The general problem of finding all possible nontrivial factors of a function is likewise 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.[6] Inversion will also sometimes fail, in part because it depends on factoring, and in part because not every term is invertible to begin with. The later Pumpkin Pi transformation does not fully circumvent undecidability, but it does more cleanly separate the transformation itself from the undecidable pieces.

---

6 Edward Z. Yang wrote a very 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.

MODELING DIVERSE PROOF STYLES   Some of the transformations make assumptions about the proof style. For example, proofs about decidable domains that work by provable double negation elimination (`dec_not_not`) pose difficulties for generalization and inversion. The later PUMPKIN Pi transformation, in contrast, is generic over the proof style.

3.5   IMPLEMENTATION

The source code of the PUMPKIN PATCH proof repair plugin suite is on Github.[7] It includes the source code of the PUMPKIN prototype plugin, extended with a number of new features, including the more advanced PUMPKIN Pi plugin that I will present in Chapter 4. The latest version supports Coq 8.8, with Coq 8.9.1 support in a branch.

This section decribes 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 4.6.2). 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).[8] The implementation always maintains pointers to easily switch between the tree and AST 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 trusted computing base (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. For example, to find a patch for the case study we will see in Section 3.6.1, I used this command:

---

7 `http://github.com/uwplse/PUMPKIN-PATCH`
8 Representing proof terms directly as trees in the implementation rather than just in the theory is one of my biggest regrets three years later, and I am actively working on removing this representation from the implementation. It is very useful for understanding how differencing works over inductive types, but it adds clutter to the implementation that make it difficult to maintain. PUMPKIN Pi represents proof terms as terms, and avoids this representation.

```
Patch Proof Old.unsigned_range unsigned_range as patch.
```

PUMPKIN analyzed both versions of `unsigned_range` and determined that a constructor of the `int` type changed (Figure 17), so it initialized the search procedure to the instance for changes in constructors.

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 in power; more information on limitations in scope can be found in the repository. Extending PUMPKIN with new instances 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 reducers a Coq term. It applies the candidate function to the arguments, then reduces the result using the supplied reducer.

The default specializer $\beta\iota\zeta$-reduces [**?**] the result using Coq's `Reduction.nf_betaiotazeta` function. Other reducers include one that does not reduce at all, one that removes unecessary applications of the identity function, one that does weak head reduction, one that completely normalizes terms, one that removes unused hypotheses, and one that applies δ reduction. 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, and further ends up being a useful programming paradigm for developing the PUMPKIN Pi extension in Chapter 4. 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 (I will show an example of this in 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, 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 a single application of `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 in the factor.

FACTORING   Factoring (`factoring.ml`) takes as input a proof term in Gallina, 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 trusted computing base (TCB). PUMPKIN is implemented as a Coq plugin. 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 PUMPKIN with many features for better integration into proof engineering workflows. This section summarizes three: Git integration, support for applying patches, and proof optmization. I detail tactic generation and more when I discuss the PUMPKIN Pi extension to PUMPKIN PATCH in Chapter 4.

GIT INTEGRATION   PUMPKIN PATCH exposes a Git interface to PUMPKIN called PUMPKIN-git [128]. 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:

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

This will search for a patch that corresponds to the change in `example_proof` in `file.v` compared to the 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 form example patched proofs in the form of reusable patches. The prototype does not yet support applying the patches that it finds automatically. Since implementing the prototype, I have extended

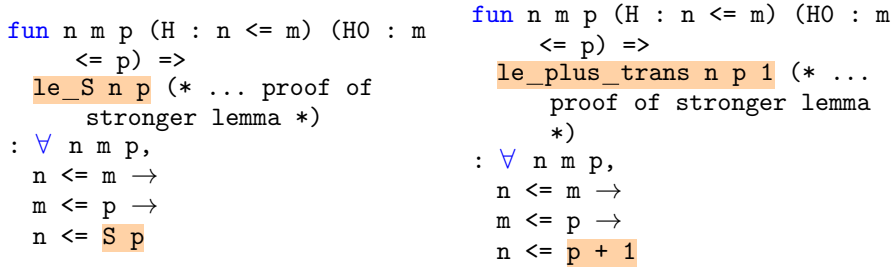PUMPKIN-git with preliminary support for patch application via hint generation. The interface is:

```
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.

Since the PUMPKIN prototype, I have also extended PUMPKIN PATCH with the PUMPKIN Pi plugin. The PUMPKIN Pi plugin does applies the changes that it finds in all cases. Furthermore, it exposes a decompiler from proof terms back up to tactic scripts—this decompiler is exposed to the PUMPKIN plugin as well, so that proof engineers can decompile patches back to tactics by calling the `Decompile` command. I will discuss this and more in Chapter 4.

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

3.5.3 *Testing Boundaries*

```
fun n m p (H : n <= m) (HO : 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) (HO : 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` (left) and `new` (right) that contain the same proof of a stronger lemma.

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 tell inform how to improve differencing and transformations in the future. (Section 3.5.3.2)

2. **The pairs unearth generalization strategies.**
   PUMPKIN produced an exponential number of candidates in 5 of 50 pairs. New generalization strategies can dramatically reduce the number of candidates. (Section 3.5.3.2)

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

3.5.3.1   *Patch Generation Suite*

I wrote a suite[9] of 50 pairs of proofs. 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.

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

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

In total, I chose 11 pairs of theorems `old` and `new`, and I wrote 50 pairs of proofs of those theorems.

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.

---

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

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. Future tools can use these as challenge problems to improve upon PUMPKIN.

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 (the PUMPKIN Pi extension in Chapter 4 is deliberately strategic about δ reduction, building on this insight). 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 (again, something the PUMPKIN Pi extension in Chapter 4 is deliberately strategic about). 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_ind_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 substitution strategies that are semantics-aware. 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 conclusion types):

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

3.5.3.3 *Performance*

PUMPKIN performed well for all pairs. The slowest success took 48 ms.[10] When PUMPKIN failed, it failed fast. The slowest failure took 7 ms. While I found this promising, proof terms were small (≤ 67 LOC); the evaluation for the PUMPKIN Pi extension in Chapter 4 will show how PUMPKIN PATCH can support large proof developments in an order of seconds.

3.6 RESULTS

To show how PUMPKIN could save work for proof engineers on real proof developments, I used the PUMPKIN prototype to emulate three motivating scenarios from real-world code:

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

---

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

```
Record int : Type :=
  mkint { intval: Z; intrange
    : 0 <= intval < modulus
      }.
```

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

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

```
Fixpoint bin_to_nat (b : bin) :
    nat :=
 match b with
 | B0 => 0
 | B2 b' => 2 * (bin_to_nat b
    ')
 | B21 b' => 1 + 2 * (
    bin_to_nat b')
 end.
```

```
Fixpoint bin_to_nat (b : bin) :
    nat :=
 match b with
 | B0 => 0
 | 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).

useful and that both differencing and the transformations are effective
and flexible.

IDENTIFYING CHANGES   I identified Git commits from popular Coq
projects that demonstrated each scenario. These commits updated
proofs in response to breaking changes. 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. 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.

   PUMPKIN does not yet handle structural changes like adding con-
structors or parameters, so I focused on changes that preserve struc-
ture, like modifying constructors. Chapter 4 describes an extension to
PUMPKIN PATCH that supports changes in structure.

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 [89] 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 PUMP-
KIN. 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.✓
```

3.6.1.1 *Instance*

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

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

PUMPKIN searches in the difference between two inductive proofs of
theorems:

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

for an isomorphism[11] 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:

---

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

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.

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_0. rewrite → plus_comm.
```

When I ported User A's solution to use User B's definition of `bin_to_nat`, the application of this inline lemma failed. I changed the conclusion of the inline 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 in the difference between the old and new versions of the example patched proof 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, 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_0`—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,[12] PUMPKIN found a patch that was not the most natural patch. The natural patch would be to remove the rewrite, just as I removed a different rewrite from the example proof. This did not occur when I

---

12 User A uses `*`; User B uses `+`. For arbitrary n, the term 2 `*` n reduces to n `+` (n `+` 0), which does not reduce any further.

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

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

ported User B's definition, which suggests that in the future, a patch finding tool may help inform novice users which definition is simpler: It can factor the proof, then inform the user if two factors are inverses. Tactic-level changes do not provide enough information to determine this; the tool must have a semantic understanding of the terms. My Magic tutorial plugin [13] implements a prototype of the functionality needed for this, based on lessons from this case study.

3.6.2.1 *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'
```

PUMPKIN searches in the difference between 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.

---

13 https://github.com/uwplse/magic

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 [91]. 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.✓
```

3.6.3.1   *Instance*

This scenario used the search procedure instance for changes in de-
pendent arguments to constructors. PUMPKIN searches within the
difference between two versions of a proof that apply the same con-
structor 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 (in this case study, to fix broken proofs that instantiate `divide` with some specific `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:

$\forall$x, `P` x $\to$ `P'` x

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

($\exists$x, `P` x) $\to$ ($\exists$ 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

Recall that this thesis set out to prove that:

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

Or, in other words, there *is* reason to believe that verifying a modified system should often, in practical use cases, be easier than verifying the original the first time around, even when proof engineers do not follow good development processes, or when change occurs outside of proof engineers' control.

With PUMPKIN, it is fair to say:

> **Thesis So Far**: Some changes in programs and specifications, and the corresponding changes in proofs broken

by those changes 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).

This is progress, but it is not so satisfying. 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. And as I mentioned earlier, without considering the extension from the next chapter, the PUMPKIN prototype does not include any support for tactics beyond the use of hints.

The next chapter will address these limitations. 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 Pi extension to the PUMPKIN PATCH proof repair plugin suite for Coq 8.8 that is available on Github.[1] PUMPKIN Pi is a plugin that adds support for 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 the PUMPKIN prototype 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 [131, 153]. As my user study showed, none of these were 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 [125, 131], and in some cases imposed additional proof obligations like proving relations corresponding to changes [139].

ADDRESSING CHALLENGE 1: FLEXIBLE TYPE SUPPORT   The case studies in Section 4.6—summarized in Table 1 on page 101—show that PUMPKIN Pi is flexible enough to support a wide range of proof engineering use cases. In general, PUMPKIN Pi can support any change described by an equivalence, though it takes the equivalence in a deconstructed form—the *configuration* for PUMPKIN Pi. The configuration expresses to the proof term transformation how to translate functions and proofs defined over the old version of a type to refer only to the new version, and how to do so in a way that does not break 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.

ADDRESSING CHALLENGE 2: WORKFLOW INTEGRATION   Research on workflow integration for proof repair tools is in its infancy. PUMP-

---

1 I annotate each claim to which code is relevant with a circled number like ①. These circled numbers are links to code, and are detailed in `GUIDE.md`.

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

Figure 20: A change from the old version of `list` (left) to the new version of `list` (right). 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 two constructors (orange).

KIN Pi is built with workflow integration in mind. For example, PUMPKIN Pi is the only proof repair tool I am aware of that produces suggested proof scripts (rather than proof terms) for repaired proofs, a challenge highlighted in the previous chapter, in other existing proof repair work [131], and in my survey of proof engineering [125]. 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 this proof from the Coq standard library (comments ours for clarity):[2]

```
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.
```

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 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

---

[2] We use induction instead of pattern matching.

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 `induction x as [| a l IHl
]` becomes `induction x as [a l IHl|]`. Valentin Robert's thesis [131]
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 automat-
ically patch broken proofs in response to a broad class of changes
in datatypes. PUMPKIN Pi in particular repairs proofs in response to
changes in types that correspond to *type equivalences* [142], or pairs of
functions that map between two types (possibly with some additional
information) and are mutual inverses (Section 4.2.2).[3] In other words,
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 human extracts and generalizes the infor-
mation corresponding to the change, but PUMPKIN Pi can still apply it
to fix other broken proofs.

Like the original PUMPKIN prototype, it also does this using a com-
bination of differencing and proof term transformations. The corre-
sponding 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 dataype directly to a proof term
defined over the new version of the datatype. PUMPKIN Pi further

---

3 The adjoint follows, and PUMPKIN Pi includes machinery to prove it ⑩ ㉓.

Figure 21: The workflow for Pumpkin Pi.

supports proof script integration and other features for better work-flow integration (Section 4.5), with real payoffs for proof engineers (Section 4.6).

4.2.1 *Workflow: Configure, Transform, Decompile*

As with Pumpkin, the interface to Pumpkin Pi is exposed to the proof engineer as a command. 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 dataype, 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 21 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.

There are currently four search procedures for automatic configuration implemented in Pumpkin Pi (see Table 1 on page 101). 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 **??**, for example, used automatic configuration. When we invoked Pumpkin Pi:

```
swap T (l : Old.list T) : New.      swap⁻¹ T (l : New.list T) :
   list T :=                            Old.list T :=
 Old.list_rect T (fun (l :          New.list_rect T (fun (l :
    Old.list T) => New.list T          New.list T) => Old.list T
   )                                   )
  New.nil                           (fun t _ (IHl : Old.list T)
  (fun t _ (IHl : New.list T)           => Old.cons T t IHl)
      => New.cons T t IHl)          Old.nil
  l.                                l.
Lemma section: ∀ T (l : Old.       Lemma retraction: ∀ T (l : New.
   list T),                           list T),
 swap⁻¹ T (swap T l) = l.           swap T (swap⁻¹ T l) = l.
Proof.                             Proof.
  intros T l. symmetry.             intros T l. symmetry.
    induction l as [ |a l0 H           induction l as [t l0 H|
    ].                                 ].
 - auto.                            - simpl. rewrite ← H. auto.
 - simpl. rewrite ← H. auto.        - auto.
Qed.                               Qed.
```

Figure 22: Two functions between `Old.list` and `New.list` (top) that form an equivalence (bottom).

```
Inductive I :=                      Inductive J :=
| A : I                             | makeJ : bool → J.
| B : I.
```

Figure 23: 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).

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

it invoked a search procedure that differenced `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 to us 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, those types are *equivalent* (denoted $A \simeq B$). Figure 22 shows a type equivalence between the two versions of `list` from Figure 20 that PUMPKIN Pi discovered and proved automatically ①.

To give some intuition for what kinds of changes can be described by equivalences, I preview two changes below. See Table 1 on page 101 for more examples.

FACTORING OUT CONSTRUCTORS   Consider changing the type `I` to the type `J` in Figure 23. `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 constructor. With PUMPKIN Pi, the proof engineer can repair functions

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

Figure 24: A vector (bottom) is a list (top) indexed by its length (orange). Vectors effectively make it possible to enforce length invariants about lists at compile time.

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 ≃ 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 24). PUMPKIN Pi can repair functions and proofs about lists to functions and proofs about vectors of particular lengths ③, since Σ(l:list T) .length l = n ≃ vector T n. From the proof engineer's perspective, after updating specifications from `list` to `vector`, to fix her functions and proofs, she must additionally prove invariants about the lengths of her lists. PUMPKIN Pi makes it easy to separate out that proof obligation, then automates the rest.

More generally, in homotopy type theory, with the help of quotient types, it is possible to form an equivalence from a relation, even when the relation is not an equivalence [9]. While Coq lacks quotient types, it is possible to achieve a similar outcome and use PUMPKIN Pi for changes that add or remove information when those changes can be expressed as equivalences between Σ types or sum types.

4.2.3 *Differencing: Equivalences from Changes*

Differencing in PUMPKIN Pi is what configures the proof term transformation with 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 **??**, differencing discovered and proved the equivalence in Figure 22. 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 correspond to type equivalences. In the end, it returns a configuration that corresponds to the equivalence (see Section 4.3 for details), 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. 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 correspond to the equivalence to PUMPKIN Pi's `Configure` `Repair` command before invoking `Repair`. This is what I did for the change in Figure 23 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 24, 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 a kind of proof reuse known as *transport* [142], but in a way that is

suitable for repair. Informally, transport 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.

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 $t \equiv_{A \sim B} t'$).[4] 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 22, 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.

Transport typically works by applying the functions that make up the equivalence to convert inputs and outputs between types. This approach would not be suitable for repair, since it does not make it possible to remove the old type A. PUMPKIN Pi implements transport 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 corresponds to 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

---

4 This notation should be interpreted in a metatheory with *univalence*—a property that Coq lacks—or it should be approximated in Coq. The details of transport with univalence are in the Homotopy Type Theory book [142], and an approximation in Coq is in the univalent parametricity framework paper [138]. 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.

```
DepConstr(0, list T) : list T        DepConstr(0, list T) : list T
   := Constr(0, list T).                := Constr(1, list T).
DepConstr(1, list T) t l :           DepConstr(1, list T) t l :
   list T :=                            list T :=
  Constr (1, list T) t l.             Constr(0, list T) t l.

DepElim(l, P) { pnil, pcons } :      DepElim(l, P) { pnil, pcons } :
   P l :=                               P l :=
  Elim(l, P) { pnil, pcons }.         Elim(l, P) { pcons, pnil }.
```

Figure 25: The dependent constructors and eliminators for old (left) and new (right) list, with the difference in orange.

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 differening 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 corresponds to an equivalence $A \simeq B$. 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 induction principle 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 of terms:
  ((DepConstr, DepElim), (Eta, Iota))
each of which corresponds to one of the two goals: DepConstr and DepElim define how to transform constructors and eliminators, thereby preserving the equivalence, and Eta and Iota define how to transform η-expansion and ι-reduction of constructors and eliminators, thereby producing well-typed terms.

Intuitively, every correct configuration can be defined in terms of what in category theory are known as initial algebras, and correctness—that every configurations and equivalences are isomorphic—follows

by Lambek's theorem (Section 4.3.3).[5] 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 4.3.5 revisits the limitations of differencing from PUMPKIN, and 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 *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 are 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 25. The dependent constructors for `Old.list` are the normal constructors with the order unchanged, while the dependent constructors for `New.list` swap constructors back to the original order. 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. Consider the type of vectors of some length:

```
Σ(n : nat).vector T n
```

PUMPKIN Pi can port proofs across the equivalence between this type and `list T` ③. The dependent constructors PUMPKIN Pi discovers pack the index into an existential, like:

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

and the eliminator it discovers eliminates the projections:

```
DepElim(s, P) { f₀ f₁ } : 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))
}.
```

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` to constructors and cases over `packed_vect`. 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. CIC_ω distinguishes between two important kinds of equality: those 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 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.

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. Otherwise, if the transformation transforms a term `t : T` to some `t' : T'`, it does not necessarily transform `T` to `T'` [139].

`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 [110]. ι-reduction (β-reduction for inductive types) describes how to reduce an elimination of a constructor [109].

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

```
Eta(Σ (n : nat) . vector T n) := λ(s : Σ (n : nat) . vector T
    n).∃ (π_l s) (π_r s).
```

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).

```
                    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 26: A unary natural number `nat` (left) is either zero (`0`) 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`). Unary and binary natural numbers are equivalent, but have different inductive structures. Consequentially, definitional equalities over `nat` may become propositional over `N`.

Each `Iota`—one per constructor—describes and proves the ι-reduction behavior of `DepElim` on the corresponding case. This is needed, for example, to port proofs about unary numbers `nat` to proofs about binary numbers `N` (Figure 26). 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:

```
∀ P p0 ps n,
  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:

```
∀ P p0 ps n (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:

```
∀ P p0 ps 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:

```
∀ P p0 ps n (Q: P (DepConstr(1, N) n) → s),
  Iota(1, N, Q) :
    Q (ps n (DepElim(n, P) { p0, ps })) →
```
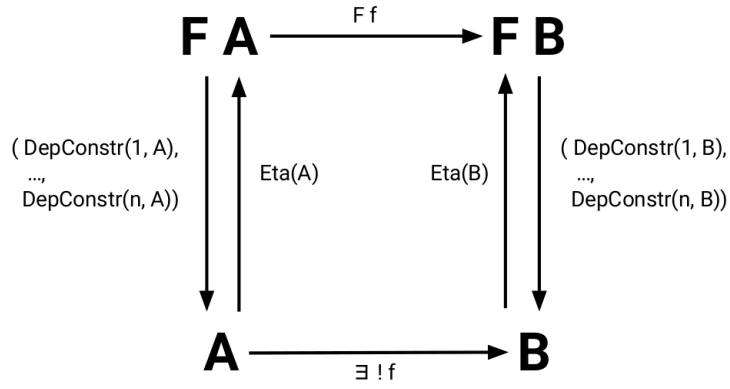
Figure 27: The categorical representation of a configuration for equivalent types *A* and *B* in terms of initial algebras.

$$\texttt{Q (DepElim(DepConstr(1, N) n, P) \{ } p_0, \ p_S \texttt{)).}$$

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* [97]), 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*

The analogy tying the configuration to contructors and eliminators by way of the names `DepConstr` and `DepElim` is not a coincidence. In fact, it has meaning in terms of what in category theory are known as *initial algebras* (Section 4.3.3.1). A more syntactic version of this claim can be used to specify and prove correctness of configurations in CIC_ω (Section 4.3.3.2).

4.3.3.1   *Categorical Intuition for Correctness*

Configurations have meaning in terms of *initial algebras* [111], which in homotopy type theory represent inductive types [142]. 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 27 shows this for an arbitrary configuration for an equivalence between *A* and *B*. Here, *FA* is the shape of *A*. `DepConstr` maps

from *FA* to *A*, and `Eta` maps from *A* back to *FA*. `DepElim` (not pictured explicitly) is the arrow corresponding to `DepConstr` defined over the analogous diagram lifted to $A \rightarrow$ `Type` (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 square commutes. The configuration parts for *B* are similar.

For example, fixing *A* as `nat`, *FA* is `1 + nat`: the sum of the unit type and `nat`. Going from *FA* → *A*, the left injection maps to the `O` constructor, and the right injection maps to the `S` constructor. The inverse is essentially (up to differences in the type theory) 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` → `Type`. 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 really does later on is follow the arrows in the diagram to get from *A* directly to *B*. But thanks to the type theory of CIC_ω, in order to do that, it needs to handle the nuances of definitional equality and dependent types—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 in CIC_ω*

The categorical definition may help with some of the intuition, but it does not help with validating correct configurations inside of the type theory. And this is necessary, since 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).

Fortunately, in CIC_ω, it is also possible to specify 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 28 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 28, left), `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`

```
section: ∀ (a : A), g (f a)
    = a.
retraction: ∀ (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] ≡_{ζ(A,P_A,j)≃ζ(B,P_B,j)}
       f⃗_B[j]→
  DepElim(a, P_A) f⃗_A
     ≡_{(Pa)≃(Pb)} DepElim(b, P
    _B) f⃗_A.
```

```
elim_eta(A): ∀ a P f⃗, DepElim(a, P)
    f⃗ : P (Eta(A) a).
eta_ok(A): ∀ (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 28: Correctness criteria for a configuration to ensure that the transformation preserves equivalence (left) coherently with equality (right, shown for A; B is similar). f and g are defined in text. s, $\vec{f}$, $\vec{x}$, and $\vec{IH}$ represent sorts, eliminator cases, constructor arguments, and inductive hypotheses. ζ (A, P, j) is the type of DepElim(A, P) at DepConstr(j, A) (similarly for B).

(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 28, right), 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* [24]) behavior that corresponds to a case 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 *i*, the proof obligation is to show that `g (`
`f a)` is equal to `a`, where `a` is `DepConstr(A, i)` applied to the non-
inductive arguments (by `elim_eta(A)`). Expand the right-hand side
using `Iota(A, i)`, then expand it again using `Iota(B, i)` (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 gen-
erate 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 [138] or a
univalent type theory [142]. 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 configu-
ration 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.

   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 unique fold over *A*. 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 24 on page 71). The
equivalence that we already saw in Section 4.2.2 follows from this:

   `Σ(l : list T).length l = n` $\equiv$ `vector T n`

Alternatively, we can say that a list is equivalent to a vector of *some*
length:

   `list T` $\equiv$ `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:

$\forall$ `(l : list T)`, `length l` $= \pi_l$ `(f l)`

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

In Section 4.6, I will show you a case study of porting functions and proofs from lists to length-indexed vectors. Nominally this works by porting the functions and proofs along the equivalence from Section 4.2.2, but in practice this works by chaining two different 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 this in detail. Assume the existence of 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 types.[6] Then there is a type equivalence:

$A \simeq \Sigma(i : I).A_I\ i$

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

`indexer` $:\ A\ \to\ I$.

which projects the promoted index:

`coherence` $:\ \forall(a : A)$, `indexer` $a = \pi_l$ `(f a)`.

Following existing work, I call this equivalence the *ornamental promotion isomorphism* [81]; when it holds and the 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.

---

6 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 those who are *sufficiently fanatical*.

$$A \; := \; \text{Ind}(Ty_A : s_A)\{C_{A_1}, \ldots, C_{A_n}\}$$
$$A_I \; := \; \text{Ind}(Ty_{A_I} : (\Pi(i : I).s_{A_I}))\{C_{A_{I_1}}, \ldots, C_{A_{I_n}}\}$$
$$B \; := \; \Sigma(i : I).A_I i$$

$$P_A \; := \; \Pi(a : A).s_A$$
$$P_{A_I} \; := \; \Pi(i : I)(a_i : A_I \; i).s_{A_I}$$

$$\forall 1 \le j \le n,$$
$$E_{A_j} \; (p_A : P_A) \; := \; \xi(A, \; p_A, \; \text{Constr}(j, \; A), \; C_{A_j})$$
$$E_{B_j} \; (p_{A_I} : P_{A_I}) \; := \; \xi(A_I, \; p_{A_I}, \; \text{Constr}(j, \; A_I), \; C_{A_{I_j}})$$

Figure 29: Common definitions.

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 29 contains other common definitions, the names for which are reserved: Input type A expands to an inductive type with constructors $\{C_{A_1}, \ldots, C_{A_n}\}$. 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 30 shows the algorithm for generating `indexer`. The algorithms for generating `f` and `g` are similar; Figure 31 shows only the derivations for generating `f` that are different from those for generating `indexer`, and the derivations for generating `g` are omitted.

4.3.4.1 *Differencing for the Indexer*

Then differencing algorithm generates the `indexer` by traversing the types of the eliminators for A and A_I in parallel using the algorithm from Figure 30, 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 29). It does this by constructing a function from A to I. Consider `list` and `vector`:

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

For these types, INDEX-MOTIVE computes the motive:

```
λ (l :  list T) . nat
```

INDEX-MOTIVE

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

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

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

INDEX-CONCLUSION

INDEX-HYPOTHESIS
$$\frac{\text{new } n_{A_I}\ b_{A_I} \qquad \Gamma,\ n_{A_I} : t_{A_I} \vdash (\Pi(n_A : t_A).b_A,\ b_{A_I}) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_{A_I} : t_{A_I}).b_{A_I}) \Downarrow_{i_c} t}$$

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

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
$$\frac{\begin{array}{l} \Gamma \vdash (A,\ A_I) \Downarrow_{i_m} p \\ \Gamma,\ p_A : P_A,\ p_{A_I} : P_{A_I} \vdash \{(E_{A_1}\ p_A,\ E_{A_{I_1}}\ p_{A_I}), \ldots, (E_{A_n}\ p_A,\ E_{A_{I_n}}\ p_{A_I})\} \Downarrow_{i_c} \vec{f} \end{array}}{\Gamma \vdash (A,\ A_I) \Downarrow_i \lambda(a : A).\text{Elim}(a, p)\vec{f}}$$

Figure 30: Differencing for the indexer function.

GENERATING EACH CASE   The $\Gamma \vdash (T_A, T_{A_I}) \Downarrow_{i_c} t$ judgment gener-
ates 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 (t_l:T) (l : list T) (IH_l : p_A l), ...      Π (t_v : T) (v :
    vector T n) (IH_v : p_{A_I} n v), ...
```

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) (consV n t_l l)
```

INDEX-IH then takes over. It substitutes the new motive in the induc-
tive 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) (consV 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 that computes the length of `cons t l`:

```
λ (t_l :  T) (l :  list T) (IH_l :  (λ (l :  list T) . nat) l) .  S IH_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 that computes the length of a list:

```
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}
```

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

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

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

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

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

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

F-IND
$$\frac{\Gamma \vdash (A,\ A_I) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ A_I) \Downarrow_{f_m} p \qquad \Gamma,\ p_A : P_A,\ p_{A_I} : P_{A_I} \vdash \{(E_{A_1}\ p_A,\ E_{A_{I_1}}\ p_{A_I}),\dots,(E_{A_n}\ p_A,\ E_{A_{I_n}}\ p_{A_I})\} \Downarrow_{p_c} \vec{f}}{\Gamma \vdash (A,\ A_I) \Downarrow_p \lambda(a : A).\exists\ (\pi\ a)\ (\mathrm{Elim}(a,p)\vec{f})}$$

Figure 31: Differencing for f.

to discover the configuration. It also proves that these functions form an equivalence, and that the indexer is coherent with the equivalence.

DISCOVERING THE EQUIVALENCE PARTS   Figure 31 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.

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, Σ(n : nat).vector T n) : Σ(n : nat).vector T n
   :=
  ∃ (Constr(0, nat)) (Constr(0, vector T)).
```

This is the same as applying the function f that PUMPKIN Pi derives to the empty list constructor `Constr(0, list T)`, 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, Σ(n : nat).vector T n) : T → Σ(n : nat).vector
    T n → Σ(n : nat).vector T n :=
  λ (t : T) . λ (l : Σ(n : nat).vector T n) .
    ∃ ((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) . λ (l : list T) .
  ∃ ((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 then 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 I presented 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 `Eta` 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 the coherence theorem `coherence` for algebraic ornaments. This proves the correctness property that the configuration induces an equivalence, thereby increasing confidence in the output of the search procedure. 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.

4.3.5   *Revisiting Limitations*

Differencing in PUMPKIN had a two fundamental limitations: its heuristic-based nature and the difficulty of isolating changes. With PUMPKIN Pi in the PUMPKIN PATCH repair suite, these limitations are partially addressed, but remain as challenges. As in the previous chapter, limitations that are due to the choice of implementation strategy are in Section 4.5.1.2.

HEURISTICS   The PUMPKIN differencing algorithms were heuristic-based. The differencing algorithms here are still heuristic-based, as is fundamental. But some progress has been made, as the domain of each search procedure is defined as a large class of type equivalences, drawing on inspiration from type theory and from the needs of real proof engineers. For example, algebraic ornaments are a flexible class of changes drawing on about a decade of type theory literature. In addition, a number of the algorithms presented in this section, like the algorithm the prove an equivalence from a configuration, are generic over any possible type equivalence. Most notably, that the configuration can describe any possible type equivalence means that the heuristics in PUMPKIN Pi are now abstracted away from the transformation, whereas for PUMPKIN the heuristics leaked into the transformations as well.

ISOLATING CHANGES   While PUMPKIN Pi adds support for a large class of changes, the challenge of isolating changes from PUMPKIN's differencing algorithms remain. Most search procedures allow for only a single change at once, and further impose some syntactic restrictions on input types and how they change. However, the presence of manual configuration in PUMPKIN Pi now makes it possible for proof engineers to describe the changes themselves, circumventing this difficulty. Some search procedures for automatic configuration, like the one for swapping constructors, further prompt the human for input to choose among multiple choices and resolve ambiguities. This fits with the typical interactive workflows of proof engineers, but makes it possible to dodge many of the challenges of isolating changes seen earlier.

4.4   TRANSFORMATION

At the heart of PUMPKIN Pi is a configurable proof term transformation for transporting proofs across equivalences ④. Figure 32 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

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

DEP-ELIM
$$\frac{\Gamma \vdash a \Uparrow b \qquad \Gamma \vdash p_a \Uparrow p_b \qquad \Gamma \vdash \vec{f_a} \Uparrow \vec{f_b}}{\Gamma \vdash \text{DepElim}(a, \ p_a)\vec{f_a} \Uparrow \text{DepElim}(b, \ p_b)\vec{f_b}}$$

DEP-CONSTR
$$\frac{\Gamma \vdash \vec{t_a} \Uparrow \vec{t_b}}{\Gamma \vdash \text{DepConstr}(j, \ A) \ \vec{t_a} \Uparrow \text{DepConstr}(j, \ B) \ \vec{t_b}}$$

ETA
$$\frac{}{\Gamma \vdash \text{Eta}(A) \Uparrow \text{Eta}(B)}$$

IOTA
$$\frac{\Gamma \vdash q_A \Uparrow q_B \qquad \Gamma \vdash \vec{t_A} \Uparrow \vec{t_B}}{\Gamma \vdash \text{Iota}(j, \ A, \ q_A) \ \vec{t_A} \Uparrow \text{Iota}(j, \ B, \ q_B) \ \vec{t_B}}$$

EQUIVALENCE
$$\frac{}{\Gamma \vdash A \Uparrow B}$$

CONSTR
$$\frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{t} \Uparrow \vec{t'}}{\Gamma \vdash \text{Constr}(j, \ T) \ \vec{t} \Uparrow \text{Constr}(j, \ T') \ \vec{t'}}$$

IND
$$\frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{C} \Uparrow \vec{C'}}{\Gamma \vdash \text{Ind}(Ty : T)\vec{C} \Uparrow \text{Ind}(Ty : T')\vec{C'}}$$

APP
$$\frac{\Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash ft \Uparrow f't'}$$

ELIM
$$\frac{\Gamma \vdash c \Uparrow c' \qquad \Gamma \vdash Q \Uparrow Q' \qquad \Gamma \vdash \vec{f} \Uparrow \vec{f'}}{\Gamma \vdash \text{Elim}(c, Q)\vec{f} \Uparrow \text{Elim}(c', Q')\vec{f'}}$$

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
$$\frac{v \in \text{Vars}}{\Gamma \vdash v \Uparrow v}$$

Figure 32: Transformation for transporting terms across $A \simeq B$ with configuration ((DepConstr, DepElim), (Eta, Iota)).

```
(* 1: original term *)
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: after unifying with
    configuration *)
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.
```

```
(* 4: reduced to final term *)
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.

(* 3: after transforming *)
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.
```

Figure 33: Swapping cases of the append function, counterclockwise, the input term: 1) unmodified, 2) unified with the configuration, 3) ported to the updated type, and 4) reduced to the output.

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 previous 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). In doing so, it addresses several of the limitations from the PUMPKIN transformations (Section 4.4.3).

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 33 shows this with the list append function ++ from Section 4.1. To update ++ (top left), Pumpkin Pi unifies `Old.list T` with *A*, and `Constr` and `Elim` with `DepConstr` and `DepElim` (bottom left). After unification, the transformation recursively substitutes *B* for *A*, which moves `DepConstr` and `DepElim` to construct and eliminate over the updated type (bottom right). This reduces to a term with swapped constructors and cases over `New.list T` (top right).

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.

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 [138]. It would be possible to generate per-transformation proofs of correctness, but this does not serve an engineering need.

4.4.3 *Revisiting Limitations*

The transformations in Pumpkin struggled with undecidability and modeling diverse proof styles. The Pumpkin Pi transformation partially addresses this. As in the previous chapter, limitations that are due to the choice of implementation strategy are in Section 4.5.1.3.

UNDECIDABILITY   Pumpkin Pi cannot totally get rid of undecidability, but what it can do is move the burden of undecidability outside of the details of the algorithm. This is what the unification heuristics

accomplish. 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. Pumpkin Pi implements this. Still, the Pumpkin Pi transformation does struggle sometimes with termination. I decribe this more in Section 4.5.

MODELING DIVERSE PROOF STYLES   Unlike the procedures from Pumpkin, the Pumpkin Pi transformation is generic over the proof term structure. This benefit again comes from decoupling unification from the transformation, and from the fact that the transformation is defined over any possible Gallina term. The only difficulties show up at the level of implementation details, which I will describe in Section 4.5.

4.5 IMPLEMENTATION

The source code of the Pumpkin Pi plugin is on Github. It also ships with 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 decribes 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 21 on page 69. 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 next 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 possile to ask PUMPKIN Pi to automatically generate a name. The final step—DECOMPILE—runs in the end to produce a suggested proof script.

There are a few other useful commands in PUMPKIN Pi that are detailed in the repository. Notably, the `Repair Module` command operates over entire modules at once and defines new modules. The `Preprocess` command converts functions to use eliminators, so that the transformation can assume primitive eliminators. The `Lift` and `Lift Module` commands 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, and what it considers in its 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 of this form:

$A \equiv \Sigma(i : I).A_I i$

to equivalences of this form:

$\Sigma(a : A).\pi a = i \equiv A_I i$

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 24, 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 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 developer effort, 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 32 on page 89, but with a few differences highlighted below.

FROM CIC_ω TO GALLINA   The implementation ④ of the transformation handles language differences to scale from CIC_ω to Gallina. For example, recall that the transformation assumes primitive eliminators, while Gallina implements eliminators in terms of pattern

matching and fixpoints. To handle terms that use these features, Pumpkin Pi includes a `Preprocess` command that translates these terms into corresponding eliminator applications.[7] This command can preprocess a definition (like `length` from Figure 7 on page 19) 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.[8] 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   In addition to cases for the derivations, the implementation of the transformation includes a number of cases that correspond to optimizations. The code denotes these explicitly ④ and explains them in detail in comments ⑫. 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. 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 intent. That is, Pumpkin Pi automates the case of porting *every A* to *B*, but proof engineers sometimes wish to port only *some A*s to *B*s. Pumpkin Pi has some support for this using an interactive workflow ⑬, with plans for automatic support in the future.

---

7 My collaborator Nate Yazdani wrote this command, and it is available as a standalone plugin on Github. I helped him design and test the command, but did not write any of the code.

8 This is enough to preprocess many practical terms, including the entire `List` module. But it is not as general as it could be. A more general translation may help Pumpkin Pi support more terms, and discussions with Coq developers a couple of years ago suggest that the implementation of such a translation building on work from the equations [136] plugin was in progress. I do not know the current status of that project.

$\langle v \rangle \in$ 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 34: Qtac syntax.

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 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

$$\boxed{\Gamma \vdash t \Rightarrow p}$$

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 35: Qtac decompiler semantics.

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 Coq terms and produces tactics in Ltac, the proof script language for Coq. 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, 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_ω terms and produces tactics in a mini version of Ltac which I call Qtac. The syntax for Qtac is in Figure 34. 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 35 (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 are 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 36 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 that the proof engineer can step through to understand, modify, and maintain. There are few differences from the mini decompiler needed

```
fun (y0 :  list A)¹ =>
  list_rect²  _ _  (fun a l H² =>
    eq_ind_r³ _ eq_refl⁴ (app_nil_r (rev l) (a::[]))³)
    eq_refl⁵
    y0²

- intro y0.¹ induction y0 as [a l H|].²
  + simpl.  rewrite app_nil_r.³ auto.⁴
  + auto.⁵
```

Figure 36: 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.

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 similar to 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 inductive 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 COQ   Scaling the decompiler to Coq introduces let bindings, which are generated by tactics like `rewrite in`, `apply in`, and `pose`. **Decompile** implements ⑪ support for `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 the `induction` tactic, but leave the step of motive inference to the proof engineer. I have found these suggested tactics easier to work with (Section 4.6). Note that in the case the suggested proof script is not quite correct, it is still possible to use the generated proof term directly. Still, in the event that soundness is desirable, RanDair has since implemented a simplified sound version of the decompiler in a branch.

PRETTY PRINTING   After decompiling proof terms, **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*

In the end, 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

Class	Config.	Examples	Sav.	Repair Tools
Algebraic Ornaments	Auto	List to Packed Vector, hs-to-coq ③	☺	Pumpkin Pi, Devoid, UP
		List to Packed Vector, Std. Library ⑯	☺	Pumpkin Pi, Devoid, UP
Unpack Sigma Types	Auto	Vector of Particular Length, hs-to-coq ③	☺	Pumpkin Pi, UP
Tuples & Records	Auto	Simple Records ⑬	☺	Pumpkin Pi, UP
		Parameterized Records ⑰	☺	Pumpkin Pi, UP
		Industrial Use ⑱	☺	Pumpkin Pi, UP
Permute Constructors	Auto	List, Standard Library ①	☺	Pumpkin Pi, UP
		Modifying a PL, REPLica Benchmark ①	☺	Pumpkin Pi, UP
		Large Ambiguous Enum ①	☺	Pumpkin Pi, UP
Add new Constructors	Mixed	PL Extension, REPLica Benchmark ⑲	☹	Pumpkin Pi
Factor out Constructors	Manual	External Example ②	☺	Pumpkin Pi, UP
Permute Hypotheses	Manual	External Example ⑳	☹	Pumpkin Pi, UP
Change Ind. Structure	Manual	Unary to Binary, Classic Benchmark ⑤	☺	Pumpkin Pi, Magaud
		Vector to Finite Set, External Example ㉑	☺	Pumpkin Pi

Table 1: Some changes using Pumpkin Pi (left to right): class of changes, kind of configuration, examples, whether using Pumpkin Pi saved development time relative to reference manual repairs (☺ if yes, ☺ if comparable, ☹ if no), and Coq tools we know of that support repair along (Repair) or automatic proof of (Search) the equivalence corresponding to each example. Tools considered are Devoid [130], the Univalent Parametricity (UP) white-box transformation [139], and the tool from Magaud & Bertot 2000 [94]. Pumpkin Pi is the only one that suggests tactics. More nuanced comparisons to these and more are in Chapter 5.

user errors and bugs. Nate implemented machinery for whole module processing to handle entire libraries at once. These features helped with workflow integration.

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 Pi *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 prelim-

```
Inductive Term : Set :=          Inductive Term : Set :=
| Var : Identifier → Term         | Var : Identifier → Term
| Int : Z → Term                  | Bool : Identifier → Term
| Eq : Term → Term → Term          | Eq : Term → Term → Term
| Plus : Term → Term → Term        | Int : Z → Term
| Times : Term → Term → Term       | Plus : Term → Term → Term
| Minus : Term → Term → Term       | Times : Term → Term → Term
| Choose : Identifier → Term       | Minus : Term → Term → Term
   → Term.                         | Choose : Identifier → Term
                                      → Term.
```

Figure 37: A simple language (left) and the same language with two swapped constructors and an added constructor (right).

inary 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 $\simeq \Sigma(n : \text{nat}).\text{vector T n}$. 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 sigma types. By composition, this transported proofs across the equivalence from Section **??**.

Two tricks helped with workflow integration for this change: 1) have the search procedure view vector T n as $\Sigma(v : \text{vector T m}).\text{n} = \text{m}$ 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.

PERMUTE CONSTRUCTORS: MODIFYING A LANGUAGE   The swapping example from Section 4.1 was inspired by benchmarks from the REPLICA user study of proof engineers seen earlier. A change from one of the benchmarks is in Figure 37. 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 37, 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 [30] to get between proofs of the inefficient and efficient terms.

FACTOR OUT CONSTRUCTORS: EXTERNAL EXAMPLE   The change from Figure 23 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 Mörtberg, a cubical type theory expert. It shows how to use PUMPKIN Pi to swap two hypotheses of a type, since `T1` $\rightarrow$ `T2` $\rightarrow$ `T3` $\simeq$ `T2` $\rightarrow$ `T1` $\rightarrow$ `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 26. Binary numbers allow for a fast addition function, found in the Coq standard library. In the style of Magaud & Bertot 2000 [94], 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 p0 pS n,
  N.peano_rect P p0 pS (N.succ n) =
  pS n (N.peano_rect P p0 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 Co-qEAL. Repaired tactics were partially useful, but failed to understand custom eliminators like `N.peano_rect`, and to generate useful tactics for applications of `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 in this use case.

4.6.2 *Evaluation:* PUMPKIN P*i for Transport*

PUMPKIN Pi implements transport across equivalences in a way that is suitable for repair. Nate and I compared PUMPKIN Pi to another tool for 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 solely on proof terms. For comparison, we also used the ornaments that PUMPKIN Pi discovered to transport functions and proofs using the UP black-box transformation [138]. PUMPKIN Pi produced faster functions and smaller terms, especially when composing multiple iterations of repair. In addition, PUMPKIN Pi imposed little burden on the user, and the equivalences PUMPKIN Pi discovered proved useful to UP.

We chose UP for comparison because it also implements transport across equivalences in Coq, and because doing so demonstrates the benefits of specialized automation for classes of equivalences like ornaments. At the time of evaluation, the UP white-box transformation did not exist, so we used the black-box transformation. Using the white-box transformation would likely improve the performance of functions and proofs generated by UP when applicable. Our experi-

ences suggest that it is possible to use both tools in concert. 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 an ornament, and then transported functions and proofs along that ornament 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.[9]

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 for PUMPKIN Pi, we decided to use 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, Nate 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, Nate transported along each intermediate type, then unpacked the result. For example, the ornament from `bst` to `avl` passed through an intermediate type; Nate lifted `search` to this type

---

9 i5-5300U, at 2.30GHz, 16 GB RAM

		10	100	1000	10000		100000	
	Unlifted	0.0	0.0	0.0	3.0	(1.00x)	37.0	(1.00x)
preorder	Pumpkin 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 38: 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	
	Unlifted	0.0	0.0	0.0	3.0	(1.00x)	37.0	(1.00x)
preorder	Pumpkin 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)
	Unlifted	0.0	0.0	2.0 (1.00x)	3.0	(1.00x)	29.0	(1.00x)
search	Pumpkin 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 39: 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.

first, unpacked the result, and 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;[10] PUMPKIN Pi with automatic configuration is best fit where an incremental workflow is desirable.

4.6.2.2 *Performance*

Relative to UP, PUMPKIN Pi produced faster functions. Figure 38 summarizes runtime in the first scenario for `preorder`, and Figure 39 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 of lifting.

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.

---

10 The performances of the terms that UP produces are sensitive to the equivalence used; for a 100 node tree, this alternate workflow in UP produced a search function which is hundreds of times slower and traversal functions which are thousands of times slower than the functions that PUMPKIN Pi produced. In addition, 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 induction principles, 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.[11]

4.7   CONCLUSION

The PUMPKIN Pi plugin addresses the main limitations of the PUMPKIN prototype: PUMPKIN had limited support for patch application, supported a narrow class of changes, did not support typical proof engineering workflows like tactics, and only retroactively *could have* helped proof engineers in a few practical use cases. PUMPKIN Pi, in contrast, supports patch application in a principled manner, broadens the scope of PUMPKIN PATCH to include a large and flexible class of changes, 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:

> **Thesis**: Changes in programs, specifications, and proofs carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change (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 for inspiration.

---

11 After this evaluation and several discussions with the authors, UP later introduced the white-box transformation, which is similar to the PUMPKIN Pi transformation [139], though it does not yet support arbitrary equivalences (the black-box transformation of course does). At the time of the evaluation, PUMPKIN Pi also did not yet support arbitrary equivalences, though the UP black-box transformation did. 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 on one another. It was wonderful, and involved multiple trips to France. I will discuss UP more in Chapter 5.

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 engineering refers to the technologies that make it easier to develop and maintain systems verified using proof assistants. Proof repair falls into the domain of proof engineering. This section briefly describes work in proof engineering relevant to my work on proof repair: proof assistants other than Coq (Section 5.1.1), other technologies for proof evolution (Section 5.1.2), proof reuse (Section 5.1.3), proof design (Section 5.1.4), and other kinds of proof automation (Section 5.1.5. See my and my coauthors' survey paper on proof engineering for a detailed overview of the field [125].

5.1.1 *Proof Assistants*

Proof engineering as defined in our survey paper 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 verify. This includes Coq [32]—the proof assistant that this thesis focuses on—but also other proof assistants like Isabelle/HOL [72], HOL Light [69], HOL4 [106], Agda [7], Lean [86], and NuPRL [112]. These proof assistants have different foundations (in the case of Coq, CIC_ω), 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 in Ltac). 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 type theory CIC_ω. CIC_ω is *intuitionistic* [67] (or *constructive*) in that proofs in CIC_ω must be constructed from basic principles. For example, CIC_ω does 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 [72], HOL Light [69], and HOL4 [106] 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 equality at a given type are equal. As I detail in the survey paper, this axiom is incompatible with the *univalence* axiom from Homotopy Type Theory [142], which states that equivalence is equivalent to equality. Cubical type theory [29, 10] gives univalence a constructive interpretation, so that it is no longer an axiom.[2] Implementations of cubical type theory include RedPRL [124] 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 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, proof objects are the certificates that the kernel can check to make sure 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 our survey paper, explicitly producing proof objects and checking them is one of two dominant approaches to satisfying the de Bruijn criterion—an approach followed by Coq, Agda, and Lean. The other 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

---

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.

2 There are two flavors of cubical type theory. Bob Harper noted that I neglected to mention one of these in the proof engineering survey paper; I deeply regret this. So I would like to draw extra attention to the paper by Carlo Angiuli, Bob Harper, and Todd Wilson that I have cited here [10].

assistants with ephemeral proof objects is to follow the following workflow:

1. Reify ephemeral proof objects to be explicit.

2. Apply differencing and transformations over those objects.

3. Decompile the transformed proof objects to automation.

Chapter 6 describes an example of this for Isabelle/HOL.

AUTOMATION   Directly constructing proof objects may be too low-level to provide for a positive user experience for the proof engineer, and in some cases the proof object may not be exposed to the proof engineer at all. Typically, automation acts 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 [151, 145], which is a high-level *proof language*: a language for structuring and composing propositions, facts, and proof goals. Proof engineers can also mix these styles. For example, Coq proof engineers can write proofs using the high-level proof language SSReflect [59, 58], 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 [61]—or with the Ltac-inspired tactic language Eisbach [96]. Agda proof engineers typical rely only on reflection [44, 63, 143] to write proof automation.

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 especially difficult to difference and transform, or may make decompilation especially difficult.

Better languages for automation can also help circumvent some of the difficulties of repairing tactics directly, rather than relying a decompiler. Or, they may help simplify the implementation of the decompiler. The Ltac successor Ltac2 [118], for example, is much more structured than Ltac, and may help ease proof repair in Coq in the future.

5.1.2 *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, in my favorite critique of program verification [45]. This barrier has been realized in real developments; a review [48] of the evolution of the seL4 verified OS microkernel [79], 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." Leroy and friends 2012 [90] motivates and describes updates to the initial CompCert memory model that include changes in specifications, automation, and proofs [31].

As I have shown in this thesis, breaking changes are not always in the proof engineer's control. *Proof refactoring* [152] 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 [113]. While proof repair is a concept that I introduced with the work in this thesis, some proof refactoring tools developed in parallel with my 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 Pi is the only tool for proof evolution I am aware of that supports both proof scripts and proof terms.

PROOF SCRIPTS   A few proof refactoring tools operate directly over tactics: POLAR [47] refactors proof scripts in languages based on Isabelle/Isar [145], CoqPIE [132] is an 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 [131].

Some proof refactoring tools focus on specific refactoring tasks that are common in proof development. For example, Levity [23] 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 [131], 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 (in Chick and PUMPKIN Pi, but only sometimes in PUMPKIN) 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 [153] 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 indicies to and from types. Both Chick and RefactorAgda support primarily syntactic changes and do not have tactic support.

5.1.3 *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 Pi is yet to save proof engineers work when it comes to extending inductive types. In the future, it may help to draw on early work in proof reuse for extending inductive types. For example, a 2004 paper [22] describes a tactic to adapt proof obligations to changes in inductive types. Soon after, a 2006 paper [104] 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 combiness differencing with proof term transformations, or program transformations over proof terms. The idea of applying program transformations over proof terms dates back to at least 1987 [120]. 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.

One common proof term transformation is *generalization*, which arose in the context of proof assistants in the 1990s [64, 82, 122]. Coq's `generalize` tactic does basic syntactic generalization [33]. Both Pumpkin Patch and a tool [73] for generalizing theorems in Isabelle/HOL implement more complex transformations for generalization.

Magaud & Bertot 2000 [94] implement a proof term transformation between unary and binary numbers. This fits into a Pumpkin Pi configuration. Still, the expansion algorithm from the paper describing this transformation may help guide the design of better unification heuristics for Pumpkin Pi.

The refinement framework CoqEAL [30] 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 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 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 all equivalences and renamed Pumpkin Pi. This stands in contrast to the many wonderful embedded implementations of ornaments [38, 80, 39, 81, 37] that had arose since their discovery [97]. It essentially moved the automation-heavy approach of Ornamentation in ML [155], which operates on non-embedded ML code, into CIC_ω. It also introduced the first differencing algorithm to identify ornaments, which in the past had been identified as a "gap" in the literature [81]. Other kinds of ornaments may prove useful for future Pumpkin Pi differencing algorithms and configurations. A recent thesis [154] on ornaments may prove especially useful.

TRANSFER & TRANSPORT   Pumpkin Pi automatically transports proofs across equivalences. The need to automatically *transfer* functions and proofs across equivalences and other relations is a long-standing challenge for proof engineers [94, 20, 93]. Particular successful is the widely used Transfer [70] 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. Chapter 6 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 here is a bit overloaded and (in my opinion) 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 induction principle derivable in homotopy type theory from univalence [142] that rewrites along the identity type corresponding to an equivalence [49]. *Automatic transport* refers to any automated tooling for rewriting along propositional equalities that behaves like transport at the level of either the theory or (as in PUMPKIN Pi) the metatheory. Automatic transport is a kind of transfer, though transport itself is not, and transfer may apply more broadly than across equalities or equivalences.[3]

PUMPKIN Pi implements automatic univalent transport without relying on any additional axioms at the level of the theory itself. UP [138] approximates automatic univalent transport in Coq, only sometimes relying on functional extensionality. While powerful, it 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 [139] extends UP with a white-box transformation that may work for repair, but that imposes proof obligations on the proof engineer beyond those imposed by PUMPKIN Pi, and that includes 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.

5.1.4   *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 design principles to make proofs less likely to break to beign with. Examples of this include using information hiding techniques [157, 77] or any of the structures [?, 137, 133] for encoding interfaces in Coq. CertiKOS [62] introduces the idea of a deep specification to ease verification of large systems. Design principles for specific domains (like formal metatheory [13, 42, 43]) 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.

---

3 I got this wrong in the survey paper.

ROBUST AUTOMATION   Another approach to robust design is to use heavy proof automation, for example through program-specific proof automation [28] or general-purpose hammers [21, 117, 74, 36]. 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 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.

5.1.5   *Proof Automation*

Proof repair implements a kind of proof automation. New proof automation continues to make proof repair more feasible.

ONTOLOGY REPAIR   GALILEO [27] is a tool build on Isabelle for identifying and repairing faulty ontologies in response to contradictory evidence; it has been applied to repair faulty physics ontologies, and may have applications more generally for mathematical proofs. GALILEO uses repair plans to determine when to trigger a repair, as well as how to repair the ontology.

KNOWLEDGE SHARING METHODS   Knowledge sharing methods [53] 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 [54]. More recently, this approach has been used to identify similar concepts across libraries in proof assistants with different logics [55]. These methods combined with automation like hammers may help the proof engineer adapt proofs between isomorphic types, and 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* [107] for managing equivalences [156]. E-graphs have been implemented in Lean [134] (assuming UIP), and in Cubical Agda [57] (implying univalence). Similar implementations of e-graphs could help improve PUMPKIN Pi and similar tools to support type-directed search and more (Chapter 6).

5.2 PROGRAM REPAIR

Proof repair can be viewed as a form of *program repair* [103, 56] 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 [123] of a popular test-based program repair tool [85] 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 [95]. 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 experessive type system CIC_{ω}, 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 Pumpkin Pi does. 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 alloted. 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 former of changes to datatypes or proofs—to derive patches. The constructive foundations of Coq makes this especially appealing and powerful, since it means that existence proofs, for example, 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 learn a lot from program repair tools, even though the tools do not carry over in a straightforward way (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 [103] 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, or bugs introduced by changes in code. Test-based regression repair tools repair code that causes a set of tests (the *regressed tests*) that used to pass to no longer pass, 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 that Karl wrote in our survey paper [125] describes the correspondence between regressed proofs and regressed tests in more detail, and details existing techniques [115, 147, 18, 26, 146, 19, 41, 148, 149, 150] for rechecking regressed proofs.

One test-based regression program repair tool is ReAssert [40] 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 [140] 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 PATCH, and may provide interesting examples of configurations or ways of inferring new configurations.

While not quite a regression repair tool, GRAFTER [160] 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 Pi, 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 [144, 119] uses contracts to repair Eiffel programs. Specification-Based Program Repair Using SAT [60] 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 [65, 108, 108, 158, 98, 76]. 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 [46] presents a methodology for repairing programs based on information from incomplete proofs in Isabelle. 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 PATCH. Prophet [92], 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 [52] 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 [76] 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 [100] 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 [101] 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 [114], 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 decribed 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 [102] finds differences in algebraic data types. Semantics-based change impact analysis [12] models semantic differences between documents. Differential assertion checking [83] analyzes different versions of a program for relative correctness with respect to a specification. Incremental λ-calculus [25] introduces a general model for program changes. All of these may be useful for improving semantic differencing.

Several of the repair by example tools make use of code repositories and libraries to identify examples. One challenge that I faced in my proof repair work was making use of examples changes in code repositories, since Github commits are typically large. These tools may offer some insights for how to approach this problem. Some of these tools also use machine learning—I discuss some ideas combining proof repair with machine learning in Section **??**.

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 [40] and QACrashFix [52] 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! [66] 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 engineering 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 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 proof engineers do not follow good development processes, or when change occurs outside of proof engineers' control.

Do not just take my word for it. Consider a recent article by a proof engineer saying just this (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 Twitter, but noted that there ought to be much more work in this space.

I agree. Actually, I want to take that a step further: proof repair is just a small piece of what will carry us to the next era of proof engineering. And that era will be one in which programmers of all skill levels across all domains can develop and maintain verified systems—the era of *proof engineering for all*.

THE NEXT ERA: PROOF ENGINEERING FOR ALL

Proof engineering has come a long way, but it is still accessible mostly to experts, and perhaps the occasional practitioner. Proof repair has made proof engineering easier for experts, and a bit easier for practitioners. But there is a lot more that we as a community can do to bring proof engineering to all: not just experts and practitioners, but also software engineers and potential users from in other domains.

I conclude with a discussion of 12 future projects building up to the next era of proof engineering for all. I write about these projects in

the future tense to work your imagination, but of course every project I write about here is but an idea for now, so this section is more of an informed opinion piece. If any of these ideas inspire you, please work with me to bring them to life!

Proof Engineering for Experts

In the future, maintaining proofs will be seamless for expert proof engineers. Expert proof engineers will have easy access to technologies—like proof repair—that automate everything but the creative parts of maintenance tasks, leaving the creativity to the experts. But compared to the maintenance technologies of today, the technologies of the future will reach all proof assistants, be much more powerful, and produce natural proof scripts in the end.

ALL PROOF ASSISTANTS   Proof repair will accessible across all proof assistants. The techniques from this thesis should be simple to apply to proof assistants with similar foundations to Coq, like Agda, and potentially Lean.[1] But Isabelle/HOL, for example, lacks proof terms and is based on a classical logic. A future proof repair tool for Isabelle/HOL will perhaps reify proof terms using Isabelle/HOL-Proofs, apply a transformation based on the Transfer [70] package for proof reuse, and then (as in PUMPKIN Pi) decompile those terms to automation that does not apply Transfer or refer to the old datatype.

MORE POWER   Proof repair will be much more powerful than it is right now. The repair tools of the future will run automatically in response to proof assistant version updates. They will break down large changes into smaller pieces—perhaps by drawing on work in change and dependency management [71, 11, 26] to identify changes, then use the factoring transformation to break those changes into smaller parts. And they will support an even broader and more practical class of changes than they do now, like arbitrary relations.

NATURAL PROOFS   Proof repair tools will produce repaired proof scripts that are natural for expert proof engineers, regardless of style. Toward this end, I have just begun a promising project with RanDair and some collaborators at UMass Amherst on integrating the prototype decompiler with the machine learning proof synthesis tool TacTok [50] to rank tactic hints. Producing more natural proofs than the prototype decompiler using fixed training data seems feasible. More difficult—but desirable—is to train the decompiler to produce scripts that match the style of the proof engineer using the tool.

---

[1] Lean assumes an axiom that is incompatible with univalence. It is not quite clear to me what that assumption would mean for implementing the PUMPKIN Pi transformation in Lean. Everything else should carry over.

Proof Engineering for Practitioners

In the future, developing and maintaining proofs will be much easier for practitioners to use. Proof engineering is just starting to reach practitioners, and my proof repair tools have been a small part of that. Workflow integration has been central to that effort, but there is a long way to go. The repair tools of the future will include much more automation, integrate smoothly into IDEs and CI tools, and continually improve in response to feeeedback from user studies.

MORE AUTOMATION   Proof repair tools will include much more automation. They will elegantly support repair over large libraries when many changes occur at once, while imposing little additional effort on the proof engineer. They will be simple to extend with new optimizations, all while preserving correctness. One promising path to these ends is integrating the PUMPKIN Pi transformation with e-graphs (Section 5.1.5), which were build with these kinds of problems in mind. E-graphs were recently adapted to express path equality in cubical [57]—a perfect fit for the PUMPKIN Pi transformation. E-graphs in other proof assistants, like those in Lean [134], could help with similar automation for repair tools for other proof assistants.

SMOOTH INTEGRATION   Proof repair tools will integrate smoothly into IDEs like Proof General [2], or into continuous integration (CI) systems like Travis [3]. CI support hinges on the ability to break large changes into smaller pieces. At the level of the IDE, perhaps recording changes during development using the infrastructure from the REPLICA user study will help circumvent this problem. Program repair tools like CatchUp! [66] with existing IDE integration can serve as inspiration for both infrastructure and user experience.

USER FEEDBACK   Proof repair tools will continually adapt to feedback from the proof engineers who use them. This means user studies not just of proof engineers using proof assistants (like the REPLICA user study), but also of proof engineers using proof repair tools. The same principle applies to other proof engineering tools. The REPLICA user study paper discusses many important challenges of and potential solutions to conducting user studies of proof engineers, and can serve as inspiration for the design.

Proof Engineering for Software Engineers

In the future, any software engineer will be able to develop and maintain verified software systems. But even with good proof engineering technologies, it will not always economically feasible or even desirable for software engineers to formally verify the entire system using a

proof assistant. 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 in the survey paper [125], 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 will similarly integrate with tools familiar to software engineers, assist software engineers in redesiging software systems for verification, and help software engineers ensure those systems are robust to change.

FAMILIAR TOOLS   Proof engineering tools will integrate naturally with tools already familiar to software engineers. They will, for example, compile programs from familiar languages to proof assistants in a verified manner. They will help software engineers interactively generalize tests to specifications for writing proofs, and infer specifications from analysis of the program itself. They will check those specifications for correctness—perhaps using property-based testing tools like QuickChick [116, 84]—and integrate with debuggers to help software engineers fix incorrect programs or specifications. They will prove as much as possible for the proof engineer, then prompt the proof engineer with only the relevant questions needed to finish off the proof. They will integrate with proof repair tools to automatically adapt those proofs in response to changes. The experience of the future will exist along a continuum from testing to formal verification.

TOOL-ASSISTED REDESIGN   Proof engineering tools will help software engineers redesign their systems to be more amenable to verification. They will, for example, automatically identify proof engineering design principles (Section 5.1.4) that are a good fit for a software system. They will help guide proof engineers through the process of redesigning software systems to use those principles, automating the manual effort through proof refactoring and repair tools. They will all of this in a way that preserves trust and is transparent to the software engineer, perhaps even teaching the software engineer about proof engineering design principles in the process.

TOOL-ASSISTED ROBUSTNESS   Proof engineering tools will help software engineers build verified systems that are robust to change. They will, for example, use differencing algorithms to analyze changes to programs and specifications over time, and from that infer more general specifications. They will record breaking changes, and use those to suggest improvements to programs and specifications to prevent future breaking changes. They will 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, domain experts from a broad spectrum of critical domains will be able to prove the properties of their software systems that matter to them—without needing any proof engineering expertise. This will build on mixed-methods verification, but with one 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 means partnering with domain experts directly. It means building new abstractions for critical domains like machine learning, cryptography, and medicine.

MACHINE LEARNING   Proof engineering tools for machine learning experts will bring strong safety, fairness, and robustness guarantees to complex systems like autonomous vehicles or robots, or even to the social media and advertising algorithms that many of us interact with on a daily basis. These tools will also make it possible to 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.[2] One possible approach is to interactively factor neural networks into two parts: a program with strong guarantees, and some "fuzz" that captures the counterexamples that cannot easily be explained. The former can hopefully be verified, with the latter capturing some margin of error.

CRYPTOGRAPHY   Proof engineering tools for cryptography will help ensure that cryptographic systems are not just designed, but also implemented correctly. The tools of the future will keep up with the a variety of cryptographic technologies as they evolve—everything from quantum cryptography to emerging cryptographic proof systems to distributed cryptosystems to lattice-based cryptography.

MEDICINE   Proof engineering tools for medical experts will bring strong guarantees to the medical devices of tomorrow. This will empower medical experts to build safer and more reliable medical devices, like pacemakers, insulin pumps, hearing aides, surgical robots, medication pumps, artifical organs, genetic arrays, neuromodulation implants, and genetic sequencing hardware and software.[3]

---

2 This whole paragraph—but especially this sentence—is based on a really fun conversation with Matthew Dwyer at Virginia this past spring.

3 Most of these suggestions came from many of my wonderful Twitter followers. One also came from Matthew Dwyer at Virginia. All who contributed are acknowledged.

Proof Engineering for All

All of these new proof engineering technologies technologies will drive the world of the future—the world of proof engineering for all. This will be a world of much more secure, reliable, and robust systems. Please work with me to make this world a reality!

INDEX

BIBLIOGRAPHY

[1] Coq reference manual, section 8.9: Controlling automation, 2017.

[2] Proof general, 2017.

[3] Travis ci, 2017.

[4] User A. Software foundations solution, 2017.

[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.

[6] Agda Development Team. Cubical type theory in agda – agda 2.6.0 documentation, 2005-2018.

[7] Agda Development Team. The Agda wiki, 2007-2021.

[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, 2020.

[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.

[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.

[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.

[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.

[14] User B. Software foundations solution, 2017.

[15] Henk Barendregt. *Foundations of Mathematics from the Perspective of Computer Verification*, pages 1–49. Springer International Publishing, Cham, 2013.

[16] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.

[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.

[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.

[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.

[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.

[21] 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.

[22] 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.

[23] 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.

[24] Pierre Boutillier. *New tool to compute with inductive in Coq*. Theses, Université Paris-Diderot - Paris VII, February 2014.

[25] 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.

[26] 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.

[27] Michael Chan, Jos Lehmann, and Alan Bundy. GALILEO: A system for automating ontology evolution. *ARCOE-11*, page 46, 2011.

[28] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[29] 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.

[30] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs*, pages 147 – 162, Melbourne, Australia, December 2013.

[31] CompCert Development Team. Merge of the newmem and newextcalls branches, 2010.

[32] Coq Development Team. The Coq proof assistant, 1989-2021.

[33] Coq Development Team. Tactics, 1999-2021.

[34] Thierry Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[35] 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.

[36] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1):423–453, June 2018.

[37] Pierre-Évariste Dagand. The essence of ornaments. *Journal of Functional Programming*, 27:e9, 2017.

[38] 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.

[39] Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of functional programming*, 24(2-3):316–383, 2014.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Francois-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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] Martin Hötzel Escardó. A self-contained, brief and complete formulation of Voevodsky's univalence axiom. *CoRR*, abs/1803.02294, 2018.

[50] Emily First, Yuriy Brun, and Arjun Guha. Tactok: Semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[51] 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.

[52] 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.

[53] Thibault Gauthier and Cezary Kaliszyk. Matching concepts across HOL libraries. In *Intelligent Computer Mathematics*, pages 267–281, Cham, 2014. Springer.

[54] 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.

[55] Thibault Gauthier and Cezary Kaliszyk. Aligning concepts across proof assistant libraries. *Journal of Symbolic Computation*, 90:89–123, 2019. Symbolic Computation in Software Science.

[56] 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.

[57] Emil Holm Gjørup and Bas Spitters. Congruence closure in cubical type theory. In *Workshop on Homotopy Type Theory / Univalent Foundations*, 2020.

[58] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[59] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[60] 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.

[61] 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.

[62] 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.

[63] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[64] 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.

[65] 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.

[66] 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.

[67] Arend Heyting. Intuitionism. an introduction. 1956.

[68] Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[69] HOL Light Development Team. HOL Light, 1996-2021.

[70] 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.

[71] D. Hutter. Management of change in structured verification. In *ASE 2000*, pages 23–31, Sept 2000.

[72] Isabelle Development Team. Isabelle, 1994-2021.

[73] 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.

[74] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, Aug 2014.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27, 2016.

[82] Thomas Kolbe and Christoph Walther. *Proof Analysis, Generalization and Reuse*, pages 189–219. Springer, Dordrecht, 1998.

[83] Shuvendu Lahiri, Kenneth McMillan, , and Chris Hawblitzel. Differential assertion checking. In *Foundations of Software Engineering (FSE'13)*. ACM, August 2013.

[84] 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.

[85] 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.

[86] Lean Development Team. Theorem proving in lean, 2014-2021.

[87] 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.

[88] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[89] Xavier Leroy. Commit to compcert: lib/integers.v, 2013.

[90] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012.

[91] letouzey. Commit to coq: change definition of divide (compat with znumtheory), 2011.

[92] 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.

[93] Nicolas Magaud. Changing data representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.

[94] 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.

[95] 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. *CoRR*, abs/1811.02429, 2018.

[96] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar 2016.

[97] Conor McBride. Ornamental algebras, algebraic ornaments, 2011.

[98] 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.

[99] Guillaume Melquiond. Commit to coq: Make izr use a compact representation of integers, 2017.

[100] Na Meng, Miryung Kim, and Kathryn S McKinley. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

[101] 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.

[102] 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.

[103] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, January 2018.

[104] Anne Mulhern. Proof weaving. In *In Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2006.

[105] 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.

[106] Magnus O. Myreen. Guide to hol4 interaction and basic proofs, 2008-2021.

[107] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[108] 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.

[109] nLab authors. beta-reduction. `http://ncatlab.org/nlab/show/beta-reduction`, July 2020. Revision 6.

[110] nLab authors. eta-conversion. `http://ncatlab.org/nlab/show/eta-conversion`, July 2020. Revision 12.

[111] nLab authors. initial algebra of an endofunctor. `http://ncatlab.org/nlab/show/initial%20algebra%20of%20an%20endofunctor`, May 2021. Revision 28.

[112] NuPRL Development Team. Nuprl, 1986-2021.

[113] William F Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, 1992.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] Pierre-Marie Pédrot. Ltac2: Tactical warfare. In *CoqPL 2019*, 2019.

[119] 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.

[120] Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1987.

[121] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cațălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. http://www.cis.upenn.edu/ bcpierce/sf.

[122] Olivier Pons. *Conception et rÉalisation d'outils d'aide au dÉveloppement de grosses thÉories dans les systèmes de preuves interactifs*. PhD thesis, 1999.

[123] 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.

[124] RedPRL Development Team. The redprl proof assistant, 2015-2021.

[125] 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.

[126] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. In *Pro-*

ceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2021, 2021.

[127] 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.

[128] Talia Ringer and Nathaniel Yazdani. Pumpkin-git, 2018.

[129] 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.

[130] 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.

[131] Valentin Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. PhD thesis, UC San Diego, 2018.

[132] 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.

[133] 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.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. The marriage of univalence and parametricity, 2019.

[140] 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.

[141] Amin Timany and Bart Jacobs. First steps towards cumulative inductive types in CIC. In *ICTAC*, 2015.

[142] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[143] 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.

[144] 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.

[145] 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.

[146] Makarius Wenzel. PIDE as front-end technology for Coq. *CoRR*, abs/1304.6626, 2013.

[147] 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.

[148] 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.

[149] Makarius Wenzel. Scaling Isabelle proof document processing, December 2017.

[150] Makarius Wenzel. Further scaling of isabelle technology, April 2018.

[151] 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.

[152] Iain Johnston Whiteside. *Refactoring proofs*. PhD thesis, University of Edinburgh, November 2013.

[153] Karin Wibergh. Automatic refactoring for agda. Master's thesis, Chalmers University of Technology and University of Gothenburg, 2019.

[154] Ambre Williams. *Refactoring functional programs with ornaments*. PhD thesis, 2020.

[155] Thomas Williams and Didier Rémy. A principled approach to ornamentation in ML. *Proc. ACM Program. Lang.*, 2(POPL):21:1–21:30, December 2017.

[156] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[157] 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.

[158] 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.

[159] 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.

[160] 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.