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

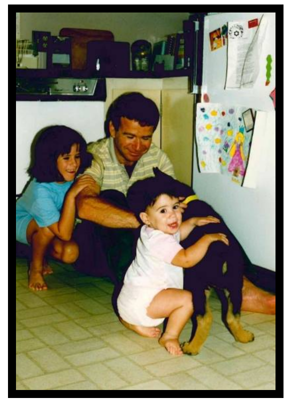
PROOF REPAIR

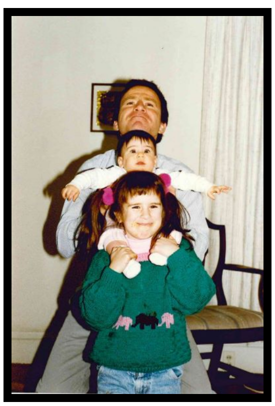
Talia Ringer

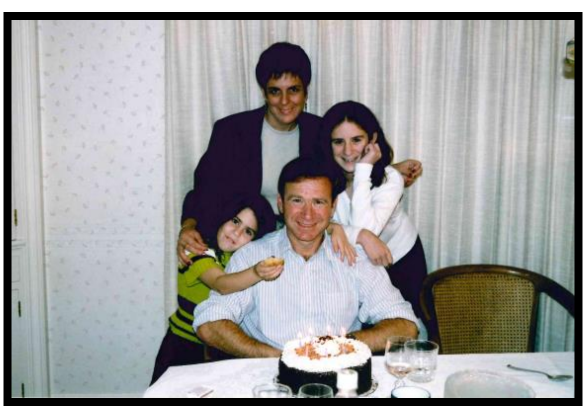
Chair of the Supervisory Committee:
Dan Grossman
Computer Science & Engineering

Abstract will go here.

To my family.







I love all of you.

CONTENTS

1 introduction 3
  1.1 Thesis 4
  1.2 Approach 5
  1.3 Results 6
  1.4 Reading Guide 7
2 motivating proof repair 9
  2.1 Proof Development 10
  2.2 Proof Maintenance 19
  2.3 Proof Repair 23
3 proof repair by example 25
  3.1 Motivating Example 26
  3.2 Approach 27
  3.3 Differencing 34
  3.4 Transformation 38
  3.5 Implementation 42
  3.6 Results 47
  3.7 Conclusion 52
4 proof repair across type equivalences 55
  4.1 Motivating Example 55
  4.2 Approach 57
  4.3 Differencing 61
  4.4 Transformation 74
  4.5 Implementation 80
  4.6 Results 84
  4.7 Conclusion 89
5 related work 91
  5.1 Programs 91
  5.2 Proofs 92
6 conclusions & future work 97

ACKNOWLEDGMENTS

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

We got other wonderful feedback on the paper from Cyril Cohen, Tej Chajed, Ben Delaware, Jacob Van Geffen, Janno, James Wilcox, Chandrakana Nandi, Martin Kellogg, Audrey Seo, James Decker, and Ben Kushigian. And we got wonderful feedback on e-graph integration for future work from Max Willsey, Chandrakana Nandi, Remy Wang, Zach Tatlock, Bas Spitters, Steven Lyubomirsky, Andrew Liu, Mike He, Ben Kushigian, Gus Smith, and Bill Zorn. The Coq developers have for years given us frequent and efficient feedback on plugin APIs for tool implementation. Merge in more PUMPKIN Pi thank yous. Merge in survey paper thank yous. Michael Shulman (feels like univalence, like categorical coherence, like an endofunctor).

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

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, especially when it comes to the scale of systems that can be verified. The seL4 [66] 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 [42] (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.

As we will soon see, this remains so difficult that sometimes, even experts give up in the face of change.

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 [66, 64], machine learning systems [**?**], distributed systems [**?**], constraint solvers [**?**], web browser kernels [**?**], compilers [72, 73], file systems [**?**], and even a quantum optimizer [**?**]. As we will soon see, practitioners have found these verified systems to be more robust and secure in deployment.

Proof engineering focuses in particular on verified systems that have been developed using special tools called *proof assistants* or interactive theorem provers (ITPs). Examples of proof assistants include Coq [30], Isabelle/HOL [59], HOL Light [55], and Lean [2]. 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 Section 5.

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 [10, 11]. 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: TODO

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

1.2 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 1 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. (This paragraph is not done yet, but not really needed to understand flow.)

1.4 READING GUIDE

(This subsection is not done yet, but not really needed to understand what I'm going for, since it's pretty low-level.)

How to read this thesis

Mapping of papers to chapters (conclusion paves path to the next era of verification)

Authorship statements for included paper materials, to credit coauthors. Discussion of "we" versus "I" or something.

Expected reader background & where to find more info

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 [91]. 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 [72, 73] verified optimizing C compiler. Both the back-end and front-end compilation passes of CompCert have been verified, ensuring the correctness of their composition [63]. 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 [48]. It has also stood up to rigorous testing: while the test generation tool Csmith [115] 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 [64]. Proof engineering technologies make it possible to develop verified systems at this scale. See Ringer and friends 2019 [99] 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,

```
Inductive nat :=
| O : nat
| S : nat → nat.
```

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

Figure 2: 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).

> 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 [24] and Software Foundations [96], and the survey paper by Ringer and friends 2019 [99]. 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*

Before we can write our small verified program, we need some datatypes and functions that we can find in the Coq standard library (Section 2.1.1.1). We can then use these datatypes and functions to write the `zip` function and show that it preserves its length (Section 2.1.1.2).

2.1.1.1 *Preliminaries*

To prove that the list zip function preserves its length, we need the `list` datatype and the `length` function. To write the `length` function, we need the `nat` datatype of unary natural numbers. All of these can be found in the Coq standard library.

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 2, left), for example, is either `O` or the successor `S` of another `nat`; these are the two constructors of `nat`.

Every inductive type in Gallina comes equipped with an *eliminator* or induction principle that the proof engineer can use to write func-

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

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

Figure 3: The type of polymorphic lists `list` in Coq defined inductively by its two constructors (left), and the type of the corresponding eliminator or induction principle `list_rect` that Coq generates (right). The curly brace notation means that the type parameter `T` is implicit in applications.

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

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

Figure 4: The list `length` function, defined both by pattern matching and recursion (left) and using the eliminator `list_rect` (right).

tions and proofs about the datatype. For example, the eliminator for `nat` (Figure 2, 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 `O` 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 3, 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 3, right) looks like the eliminator for `nat` but with an argument corresponding to the parameter `T` over which `list` is polymorphic, and with an additional argument corresponding to the new element in the inductive case.

One interesting thing about the types of these eliminators `list_rect` and `nat_rect` include universal quantification over all inputs, written ∀. Gallina's type system is expressive enough to include universal quantification over inputs—I will explain how in Section 2.1.2.

We can use these eliminators to write functions and proofs, like the `length` function we will need for our proof development (Figure 4,

```
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 5: The list `zip` function, taken from an existing tool [**?**] and translated to use eliminators.

right). More standard is to write functions using pattern matching and guarded recursion, like the `length` function from the Coq standard library (Figure 4, 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 eliminators*: 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, I will always use functions that apply eliminators rather than pattern matching, like the `length` function from Figure 4 on the right. I remove the `Definition` and `Fixpoint` keywords, since everything from here on out is a `Definition`. 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). In the rest of the paper, I skip this preprocessing step in examples, but I describe it more in the implementation section later.

2.1.1.2 *List Zip Preserves Length*

With `nat`, `list`, and `length` defined, we can now write our small verified program. We start by writing the `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.

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

PROGRAM The list `zip` function is in Figure 5. It takes as arguments two lists l_1 and l_2 of possibly different types T_1 and T_2, and zips them together into a list of pairs $(T_1 * T_2)$. For example, if the 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)).
```

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 l_1 is `nil` (base case of the outer induction), `zip` returns `nil`. Otherwise (inductive case of the outer induction), if l_2 is `nil` (base case of the inner induction), `zip` returns `nil`. If l_2 is anything else (inductive case of the inner induction), `zip` combines the first two elements of each list into a pair ((t_1, t_2)), then sticks that in front of (using `cons`) the result of recursively calling `zip` on the tails of each list (`zip 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 l_1 and l_2. We do this by defining a type `zip_preserves_length` (Figure 6, 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 l_1 and l_2. It says that if the lengths of the inputs are the same, then the length of the output is the same as the lengths of the inputs. Our proof will soon show that this type is inhabited, and so this statement is true.

It is worth noting that this step of choosing a specification is a bit of an art—we have some freedom when we choose our specification. We could just as well have chosen a different version of `zip_preserves_length` (Figure 6, 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 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 [99] 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.

Figure 7 shows a proof script for this theorem (top), along with the corresponding proof term (bottom). As we can see, the proof term is quite complicated. Thankfully, the details 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.

To write the proof script on the top of Figure 7, we start by stepping past `Proof` in our IDE. After this, 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 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:

```
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 7: 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.

```
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 produces the proof term on the bottom of Figure 7. 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. Thankfully, though, we never have to write the low-level proof term ourselves; we see proofs as these high-level proof csripts.

Some correspondence between the proof script and proof term may already be clear. For example, every call to `induction` in the proof script shows up as an application of the eliminator `list_rect` in the proof term. In Section **??**, I will explain this connection in more detail

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

$\langle t \rangle ::= \langle v \rangle \mid \langle s \rangle \mid \Pi\,(\langle v \rangle : \langle t \rangle)\,.\,\langle t \rangle \mid \lambda\,(\langle v \rangle : \langle t \rangle)\,.\,\langle t \rangle \mid \langle t \rangle\ \langle t \rangle$

Figure 8: Syntax for CoC_ω with (from left to right) variables, sorts, dependent types, functions, and application.

$\langle t \rangle ::= \ldots \mid \text{Ind}\ (\langle v \rangle : \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\} \mid \text{Constr}\ (\langle i \rangle, \langle t \rangle) \mid \text{Elim}(\langle t \rangle, \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\}$

Figure 9: CIC_ω is CoC_ω with inductive types, inductive constructors, and primitive eliminators.

by introducing a prototype decompiler from proof terms back up to proof scripts. This decompiler makes it possible for PUMPKIN to work over highly structured Gallina terms, but produce proof scripts that the proof engineer can use going forward.

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*

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) [31]. CoC with an infinite universe hierarchy is called CoC_ω. The syntax for CoC_ω is in Figure 8. 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 9), building on syntax from an existing paper [107]; the type theory is standard and omitted. CIC_ω extends CoC_ω with inductive types [32]. 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 2, 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:

$\texttt{f}_O$: P O

and:

$\texttt{f}_S$: Π (n : nat) . Π (IHn : P n) . P (S n)

prove the base and inductive cases, respectively. When `n`, `P`, $\texttt{f}_O$, and $\texttt{f}_S$ are arbitrary, this statement has the same type as `nat_rect` in 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 we 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.

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 (Section 2.2.2). Still, even with these development processes, breaking changes happen all the time, even for experts (Section 2.2.3). 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 2.1.1.2. We had two choices for the specification of `zip_preserves_length`. We chose the weaker specification on the left of Figure 6. 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 6 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 5 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:

```
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 6, 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:

```
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):

```
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:

```
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 *Building Robust Proofs*

(This subsection is not done yet, but not really needed to understand flow of paper.)

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

Two examples I like: First, affinity lemmas and reference to `zip` example. Second, better tactics like from CPDT and how could help is in `zip` example.

Quick summary of some other ideas from survey paper. For more, see the survey paper.

2.2.3 *Even Experts are Human*

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, 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. I used REPLica to collect a month's worth of granular data on the proof developments of 8 intermediate to expert Coq users. I 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.

Changes to programs and specifications were often formulaic and repetitive. For example, Figure 10 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

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 10: Patches to a lemma by an expert proof engineer.

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`.

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

But sometimes, proof engineers did not successfully fix proofs broken by changes in programs and specifications. For example, for the change in Figure 10, 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.

(The rest of this subsection is not done yet, but not really needed to understand flow.) And it's an extra big problem when you have a large development and the changes are outside of your control.

Hence Social Processes.

Why automation breaks, even with good development processes. In other words, even experts are human. And automation doesn't understand how things change, so can't help the human out. But proof repair—smarter proof automation—can.

2.3 PROOF REPAIR

(Still outline text, since I'm getting too antsy and feel like I want to move to the technical chapters again for a bit, but I'm going to explain the parallel structure of the technical chapters at the bottom of this section, so that you can understand what I'll be attempting and how this will flow in.)

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

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

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

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

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

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

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

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

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

So next two chapters will show two tools in my tool suite that work this way, how they handle these challenges, and how they save work. They will have parallel organization (informal for now but hopefully gives you a sense of what I'm thinking):

1. **Motivating Example**: an example change that motivates the class of repairs the tool supports.

2. **Approach**: a specification for the repair tool, including what kind of changes it supports, where it looks to extract & generalize changes, what the extracted & generalized change actually is, and (for the second tool) how it applies those changes. expected inputs and outputs, plus fit to thesis frame.

3. **Differencing**: design of differencing algorithms.

4. **Transformation**: design of proof term transformations.

5. **Implementation**: how this is actually implemented for Coq.

6. **Results**: proof that this can save work relative to reference manual repairs.

7. **Conclusion**: conclusion, limitations, lessons.

3

PROOF REPAIR BY EXAMPLE

The first tool in the Pumpkin Patch 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* patches, 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 all of these limitations.

Pumpkin implements an *example-based* approach to proof repair in response to breaking changes in the content of programs and specifications, so called because of its resemblence to programming by example in the domain of program synthesis [53]. In this approach, the proof engineer provides an *example* of how to adapt a proof to a breaking change. A tool then generalizes the example adaptation into a *reusable patch* that the proof engineer can use to fix other proofs broken by that change. In this way, example-based proof repair is a new form of proof automation that accounts for how breaking changes in programs and specifications are reflected in repairs to the proofs they break.

In other words, in the frame of the thesis, example-based proof repair 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 a novel semantic differencing algorithm over proof terms (Section 3.3) and a suite of 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).

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

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

Figure 11: 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.

3.1 MOTIVATING EXAMPLE

Traditional proof automation considers only the current state of programs, specifications, and proofs. This is a missed opportunity: verified software systems are rarely static. Like unverified systems, verified systems evolve over time.

With traditional proof automation, the burden of change largely falls on proof engineers. Proof repair by example shows that this does not have to be true. It is a form of proof automation that views programs, specifications, and proofs as fluid entities. When a program or specification changes and this breaks many proofs, it extracts information found in the difference between the old and new versions of a single patched proof, and generalizes that to a *reusable patch* that can fix other proofs broken by the same change.

WITHOUT PROOF REPAIR Experienced proof engineers use design principles and custom tactics to make proofs resilient to change. These techniques are useful for large proof developments, but they place the burden of change on the proof engineer. This can be problematic when change occurs outside of the proof engineers's control.

Consider a commit from the Coq 8.7 release [83]. This commit redefined injection from integers to reals (Figure 11). This change broke 18 proofs in the standard library.

The Coq standard library developer who committed the change fixed most of the broken proofs, but failed to fix some of them. 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 proof of the lemma `plus_negative_positive` broke on this line:

```
rewrite Pos2Nat.inj_sub by trivial.X
```

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 their 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 use. Furthermore, while the library developer responsible for the changes knows about the lemma that describes the change, the proof engineer does not. The proof engineer must determine how the definition has changed and how to address the change, perhaps by reading documentation or by talking to the Coq standard library developers.

WITH PROOF REPAIR When a proof engineer updates the Coq standard library, a proof repair tool can determine that the definition has changed, 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 can extract a reusable patch from those changes, which it can automatically apply within broken user proofs. The proof engineer never has to consider how the definition has changed.

3.2 APPROACH

In the example from Section 3.1, the change in `IZR` broke many proofs. The example patch to a single proof (the proof of `plus_negative_positive`) carried enough information (the rewrite by `INR_IPR`) to fix the other broken proofs. More generally, example-based proof repair takes advantage of the fact that an example patch to a broken proof can carry enough information to fix other proofs broken by the same change.

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* of how to adapt a proof to a change. PUMPKIN extracts that information 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 Coq term. In other words, looking back to the thesis statement, the information shows up in the difference between versions of the example patched proof. PUMPKIN can extract, generalize, and in some cases apply that information.

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

Figure 12: 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.

(Section 3.2.3 looks at the difference between versions of the example patched proof for this information, and finds the candidate. Then, proof term transformations (Section **??**) modify that candidate to produce the reusable proof patch. All of this happens over proof terms in Gallina, since tactics may hide necessary information as I will soon show. Pumpkin has only preliminary support for proof script integration and patch application (see Section 3.5), though I will address this limitation with the Pumpkin Pi extension in 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:

```
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. 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 candidate proof patches, which are themselves proof terms. Finally, it transforms those proof patches directly into a reusable patch. If it finds a patch with the goal type, it succeeds and defines it.

Consider, for example, the change from the theorem `old` to the slightly stronger theorem `new` in Figure 14. Changing `old` to `new` can

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

```
Theorem old: ∀ (n m p : nat), n <= m -> m <= p ->
  n <= p + 1.                        (* P p *)
Proof.
  intros. induction H0.
  - auto with arith.
  - constructor. auto.
Qed.

fun (n m p : nat) (H : n <= m) (H0 : m <= p) =>
  le_ind
    m                                (* m *)
    (fun p0 => n <= p0 + 1)          (* P *)
    (le_plus_trans n m 1 H)          (* : P m *)
    (fun (m0 : nat) (_ : m <= m0) (IHle : n <= m0 + 1) =>
      le_S n (m0 + 1) IHle)
    p                                (* p *)
    H0
```

```
Theorem new: ∀ (n m p : nat), n <= m -> m <= p ->
  n <= p.
  (* P' p *)
Proof.
  intros. induction H0.
  - auto with arith.
  - constructor. auto.
Qed.

fun (n m p : nat) (H : n <= m) (H0 : m <= p) =>
  le_ind
    m
    (* m *)
    (fun p0 => n <= p0)
    (* P' *)
    H
    (* : P' m *)
    (fun (m0 : nat) (_ : m <= m0) (IHle : n <= m0) =>
      le_S n m0 IHle)
    p
    (* p *)
    H0
```

Figure 14: Two proofs with different conclusions (top) and the corresponding proof terms (bottom) with relevant type information. We highlight the change in theorem conclusion and the difference in terms that corresponds to a patch.

break proofs that used to successfully apply `old`, so that a proof like this:

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

fails 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. In this case, it infers the following goal:

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

which takes us from the conclusion of `new` back to the conclusion of `old`. 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 constant `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 content (the conclusion of the theorem) changes, but all else remains identical. In general, the Pumpkin prototype does not support any changes that add, remove, or rearrange any hypotheses. Chapter 4 introduces an extension to Pumpkin that supports a broad class of changes in datatypes that may change in those ways.

The search procedure can be configured to different classes of change in the content of programs and specifications. Thus, before running the search procedure, Pumpkin infers a *configuration* from the example change. This configuration 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 configuration for a change in the conclusion of a theorem. Given two such proofs:

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

PUMPKIN searches for a patch with this goal type:

```
∀ x, H x → P' x → P x
```

Section 3.6 describes real-world examples that demonstrate more configurations. In total, the PUMPKIN prototype currently implements five configurations, corresponding 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 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 configurations amounts to extending key functions in the implementation with a case corresponding to the new configuration.

3.2.3 *Differencing: Candidates from Examples*

Differencing operates over terms and types. Differencing tactics would be insufficient, since tactics and hints may mask patches. For example, for the change in Figure 14, the tactics are identical, even though the proof term changes. Differencing instead 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. The semantic difference is the difference between two terms that corresponds to the difference between their types. I will explain this more in Section 3.3.

The details of the semantic difference and where differencing looks to find it vary by configuration. Consider a simplified version of the example in Figure 14, using the configuration for changes in conclusions:

Rather than look at the entire example, let us look for now at just the base case (line 13):

2 Since this is a simple example, replaying an existing tactic happens to work. There are additional examples in the repository (`Cex.v`).

diff theorem conclusions of old_proof and new_proof for goals
diff function bodies of old_proof and new_proof for candidates
if there are candidates **then**
    transform candidates

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

The semantic differencing component first identifies the difference in their types (lines 11 and 12), here:

`n <= m` $\rightarrow$ `n <= m + 1`

This is the *candidate* goal type. It then finds a difference in terms that has that type (line 13):

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

This is the *candidate* for a reusable patch. 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`, a configuration `config`, and a final goal type `goal`, assuming:
  - the change from `old_proof` to `new_proof` is in the class of changes supported by `config`.
- **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 configuration type and the final goal from the change itself, so the proof engineer does not have to provide this information. Pumpkin could in theory infer the wrong configuration or the wrong 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 are well-typed.

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 does 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 it succeeds, it has found a *reusable patch*.

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

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

has this type:

```
candidate : n <= m → n <= m + 1
```

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

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

as this is the goal that Pumpkin inferred for this configuration. 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 configuration. There are four transformations that turn patch candidates into reusable proof patches:

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

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

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

```
diff conclusions of the theorems of old_proof and new_proof for goals
diff bodies of the proof terms for candidates
if there are candidates then
    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.

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, and present real-world examples that demonstrate more configurations in Section 3.6.

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 configuration leads to a final `goal` type is not useful to the proof engineer, but this cannot soundness: every `patch` REPLICA produces is well-typed.

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 configuration, 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 configuration 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 3.5.1.1 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 the structure of the term and the configuration, with different heuristics corresponding to different subterm-configuration combinations. For historical reasons, as PUMPKIN was a prototype and differencing of proof terms was novel at the time, these heuristics are not formalized. Here I describe the design of some of the heuristics in CIC_ω. Section 3.5.1.1 describes additional features needed for implementation in Gallina, and the PUMPKIN Pi extension in Chapter 4 formalizes a more elegant differencing algorithm building on some of the insights from the PUMPKIN differencing 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 subterm of $f\ t_a$ that is definitionally equal to t_b. This is how differencing can identify the candidate for the base case of Figure 14 (line 13). It is also a core building block that other differencing heuristics rely on.

When both proof terms are function applications, and the above heuristic fails:

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

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 configuration.

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:

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

```
∀ P,
 P 0 ->
 (∀ n, P n -> P (S n)) ->
 ∀ n, P n
```

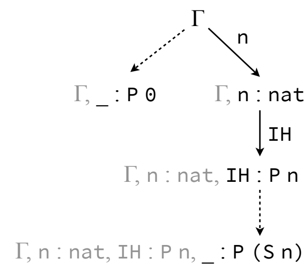


Figure 15: The type of (left) and tree for (right) the eliminator of `nat`. The solid edges represent hypotheses, and the dotted edges represent the proof obligations for each case in an inductive proof.

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 The semantic differencing algorithms views inductive types as *trees* that represent their eliminators. In these trees, every node is a type context, and every edge is an extension to that type context with a new term. Correspondingly, type differencing (to identify goal types) compares nodes, and term differencing (to find candidates) compares edges.

Differencing uses these nodes and edges 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 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

3 These trees are inspired by categorical models of dependent type theory [54].

conclusions of two inductive proofs of theorems about the natural numbers:

$Elim(nat,\ P)\ \{f_O,\ f_S\}$
$Elim(nat,\ Q)\ \{g_O,\ g_S\}$

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:

$\Gamma \vdash (f_O,\ g_O,\ Q\ \ O \rightarrow P\ \ O) \Downarrow_d \vec{t_O}$
$\Gamma \vdash (f_S,\ g_S,\ \Pi(n : nat).Q\ \ (S\ \ n) \rightarrow P\ \ (S\ \ n)) \Downarrow_d \vec{t_S}$

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 hypotheis 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 O or $S\ n$. This makes it simple for Pumpkin to later query the transformations for the final patch, with type $Q \rightarrow P$.

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 3.5.1.1.

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. Proof engineers, in contrast, often make multiple changes to a verification project in the same commit. The challenge for differencing is to break down large, composite changes into small, isolated changes, then use the appropriate heuristics. Differencing cannot do this yet, and the later Pumpkin Pi cannot do this either. However, the user study made it clear that in real life, this would be very useful dur-

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

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

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

ing development, since proof engineers make much more granular changes in development time, so this is less of an issue than I had originally expected it to be. Still, change isolation 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 [58, 6, 22] to identify changes, then use the factoring component 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$_\omega$. This section describes the design 3.4.1 and limitations 3.4.2 of these transformations. Section 3.5.1.2 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 configuration. For historical reasons, as Pumpkin was a prototype, these tranformations are not formalized. Here I describe at a high level the design of each of the four transformations in CIC$_\omega$. Section 3.5.1.2 describes additional features needed for implementation in Gallina, and the Pumpkin Pi extension in Chapter 4 formalizes a more elegant proof term transformation building on some of the insights from the Pumpkin transformation prototypes.

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 [26]. Section 3.5.1.2 decribes other reducers. The only requirement for a reducer is that the end result should be definitionally equal to the original.

Depending on the configuration 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:

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

to abstract the common argument:

$$\Pi(\texttt{t0} : T),\ \ Q\ \ \texttt{t0} \rightarrow P\ \ \texttt{t0}$$

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

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

to abstract the common function:

$$\Pi(\texttt{P0} : T),\ \ \texttt{P0}t' \rightarrow \texttt{P0}t$$

where T is the type of $P0$.

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

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

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`:

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

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

$$\lambda(\texttt{n0} : nat).\lambda(H : n <= \texttt{n0}).\text{le\_plus\_trans}\ \ n\ \ \texttt{n0}\ \ (SO)\ \ H)$$
$$: \Pi(\texttt{n0} : nat), n <= \texttt{n0} \rightarrow n <= \text{plus}\ \ \texttt{n0}\ \ 1$$

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 abstraction in the second step—substitution.

To handle this challenge, generalization uses a list of *abstraction strategies* 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 abstract 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 abstract the term. Section 3.5.1.2 describes a sample of other strategies.

It is the job of the patch finding procedure to provide the candidate and the terms to abstract. In addition, each configuration includes a list of strategies. The configuration 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 abstraction simple to extend with new strategies and simple to call with different strategies for different configurations, or even as an optimization for the proof engineer.

INVERSION Sometimes, when two types are propositionally equal, candidate patches may appear in the wrong direction. For example, consider two list lemmas (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 3.5.1.1.

FACTORING The other transformations sometimes need help breaking a large function into smaller subterms. This can break other problems, like abstraction, into smaller subproblems. It is also necessary to invert certain terms. Consider inverting an arbitrary sequence of two rewrites (again in Coq rather than CIC_ω):

```
t : X → Z := eq_ind_r G ... (eq_ind_r F ...).
```

We can view `t` as a term that composes two functions:

```
g : Y → Z := eq_ind_r G ...
f : X → Y := eq_ind_r F ...
t : X → Z := g ∘ f.
```

The inverse of `t` is the following:

```
t⁻¹ : Z → X := f⁻¹ ∘ g⁻¹.
```

To invert `t`, 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 lemma factoring component looks within a term for its factors. For the term above, it returns both factors: `f` and `g`. In this case, factoring takes the composite term and `X` as arguments. It first searches as deep as possible for a term of type `X` → `Y` for some `Y`. If it finds such a term, then it recursively searches for a term with type `Y` → `Z`. It maintains all possible paths of factors along the way, and it discards any paths that cannot reach `Z`.

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 3.5.1.2.

UNDECIDABILITY (Needs rewording but draft text, whatever.) Not reduced and so on. Abstracting candidates is not always possible; abstraction will necessarily be a collection of heuristics. Inversion will necessarily sometimes fail, since not all terms are invertible. (Needs a sentence about SOTA.)

DEPENDENT FACTORING The current factoring algorithm can handle paths with more than two factors, but it fails when `Y` depends on `X`. Other components may benefit from dependent factoring; we leave this to future work. (Needs a sentence about SOTA.)

MODELING DIVERSE PROOF STYLES (Needs adjustment to current frame.) Coq programmers use diverse proof styles; the ideal tool should support many different styles. Proofs about decidable domains that apply the term `dec_not_not` pose difficulties for abstraction and inversion; the ideal tool should support these. PUMPKIN has limited support for changes in hypotheses, fixpoints, constructors, pattern matching, and nested induction; the ideal tool should implement these features. (Needs a sentence about SOTA.)

3.5 IMPLEMENTATION

parts of PUMPKIN PATCH Inside the Core, plus more.

add something about Git, add something about hint application, add something about regrets (compiling to trees) and limitations (consider the challenges & patch generation suite if not used at all in the previous section), add something about optimization via the identity class of change (changes in nothing, sixth configuration).

you said efficiently the approach chapter for search, so probably include the numbers from that experiment here

debruijn, constants (reducing), evar maps, whatever

3.5.1 *Tool Details*

While our system is a very early prototype under active development, we have made the source code available on Github. The interested reader can follow along in the repository. Our prototype has no impact on the trusted computing base (Section 3.5.3).

4 http://github.com/uwplse/PUMPKIN-PATCH/tree/cpp18

3.5.1.1 *Semantic Differencing*

We implement semantic differencing over *trees*: PUMPKIN compiles each proof term into a tree (`evaluation.ml`). In these trees, every node is a type context, and every edge is an extension to that type context with a new term. Correspondingly, type differencing (to identify goal types) compares nodes, and term differencing (to find candidates) compares edges.

The component (`differencing.ml`) uses these nodes and edges 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 key benefit to this model is that it gives us 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:

```
nat_ind P ... (fun (IH : P n) => ...) : ∀ n, P n
nat_ind P' ... (fun (IH : P' n) => ...) : ∀ n, P' n
```

In each case, the component diffs the terms in the dotted edges of the tree for `nat_ind` (Figure 15) to try to find a term that maps between conclusions of that case:

```
P' 0 -> P 0           (* base case candidate *)
P' (S n) -> P (S n) (* inductive case candidate *)
```

The component also knows that the change in the type of `IH` is inconsequential (it occurs for any change in conclusion). Furthermore, it knows that `IH` cannot show up as a hypothesis in the patch, so it attempts to remove any occurrences of `IH` in any candidate.

When the component finds a candidate, it knows `P'` and `P` as well as the arguments `0` or `(S n)`. This makes it simple to query abstraction for the final patch:

```
∀ n, P' n -> P n
```

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, consider searching 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.

The tool always maintains pointers to easily switch between the tree and AST representations of the terms. This representation enables extensibility.

5 These trees are inspired by categorical models of dependent type theory [54].

3.5.1.2 *Transformations*

PATCH SPECIALIZATION Specialization (`specialize.ml`) takes a patch candidate and some arguments, all of which are Coq terms. It applies the candidate to the arguments, then it $\beta\iota$-reduces [26] the result using Coq's `Reduction.nf_betaiota` function. It is the job of the patch finding procedure to provide both the candidate and the arguments.

(Explain: other reducers do not reduce at all, or remove unecessary applications of the identity function.)

PATCH ABSTRACTION Abstraction (`abstraction.ml`) takes a patch candidate, the goal type, and the function arguments or function to abstract. It first generalizes the candidate, wrapping it inside of a lambda from the type of the term to abstract. Then, it substitutes terms inside the body with the abstract term. It continues to do this until there is nothing left to abstract, then filters results by the goal type. Consider, for example, abstracting this candidate by `m`:

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

The generalization step wraps this in a lambda from some `nat`, the type of `m`:

```
fun (n0 : nat) =>
  (fun (H : n <= m) => le_plus_trans n m 1 H)
: ∀ n0, n <= m -> n <= m + 1
```

The substitution step replaces `m` with `n0`:

```
fun (n0 : nat) =>
  (fun (H : n <= n0) => le_plus_trans n n0 1 H)
: ∀ n0, n <= n0 -> n <= n0 + 1
```

Abstraction uses a list of *abstraction strategies* 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 abstract 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 abstract the term. It may be possible to produce a patch only by abstracting *some* of the subterms convertible to the argument or function (we show an example of this in Section **??**), or the term may not contain any subterms convertible to the argument or function at all. We 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 abstract argument. The pattern-based strategy substitutes subterms that match a certain pattern with a term that corresponds to that pattern.

It is the job of the patch finding procedure to provide the candidate and the terms to abstract. In addition, each configuration includes a list of strategies. The configuration 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 abstraction simple to extend with new strategies and simple to call with different strategies for different classes of changes.

PATCH INVERSION Patch inversion (`inverting.ml`) 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 `eq_ind` or `eq_ind_r` (any rewrite). Indeed, `eq_ind` and `eq_ind_r` are inverses, and are related by symmetry:

```
eq_ind_r A x P (H : P x) y (H0 : y = x) :=
  eq_ind x (fun y0 : A => P y0) H y (eq_sym H0)
```

If inversion does not recognize that the type is symmetric, it swaps subterms and type-checks the result to see if it is an inverse.

LEMMA FACTORING The lemma factoring component (`factoring.ml`) searches within a term for its factors. For example, if the term composes two functions, it returns both factors:

```
 t : X -> Z               (* term *)
[f : X -> Y; g : Y -> Z] (* factors *)
```

In this case, the component takes the composite term and `X` as arguments. It first searches as deep as possible for a term of type `X -> Y` for some `Y`. If it finds such a term, then it recursively searches for a term with type `Y -> Z`. It maintains all possible paths of factors along the way, and it discards any paths that cannot reach `Z`.

The current implementation can handle paths with more than two factors, but it fails when `Y` depends on `X`. Other components may benefit from dependent factoring; we leave this to future work.

3.5.1.3 *Inside the Procedure*

The implementation (`patcher.ml4`) of the procedure from Section **??** starts with a preprocessing step which compiles the proof terms to trees (like the tree in Figure 15). It then searches for candidates one step at a time, expanding the trees when necessary.

The PUMPKIN prototype exposes the patch finding procedure to users through the Coq command `Patch Proof`. PUMPKIN automatically infers which configuration to use for the procedure from the example change. For example, to find a patch for the case study in Section 3.6.1, we used this command:

```
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 configuration for changes in constructors.

Internally, PUMPKIN represents configurations 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 abstract candidates) and how to customize components (for example, whether semantic differencing should look for an intermediate lemma). To implement new configurations for different classes of changes, we simply tweak the options.

3.5.2 *Workflow Integration*

Needed: hints and so on, any work done since, the Git interface, whatever.

3.5.3 *Trusted Computing Base*

A common concern for Coq plugins is an increase in the trusted computing base. The Coq developers provide a safe plugin API in Coq 8.7 to address this [43]. Our prototype takes this into consideration: While Pumpkin does not yet support Coq 8.7, it only calls the internal Coq functions that the developers plan to expose in the safe API [69]. Furthermore, Coq type-checks terms that plugins produce. Since Pumpkin does not modify the type checker, it cannot produce an ill-typed term.

3.5.4 *Performance*

(From the evaluation section.)

A CHALLENGE FOR DIFFERENCING (orphaned) For one pair of proofs of theorems with propositionally equal conclusions (Figure 16), the differencing component 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 [26]) `le_plus_trans`, this is not obvious:

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

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

3.6 results

Needed: key technical results

We 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 we chose for these scenarios demonstrated different classes of changes. For each case, we describe how Pumpkin configures the procedure to use the core components for that class of changes. Our experiences with these scenarios suggest that patches are useful and that the components are effective and flexible.

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

Our goal was to simulate incremental use of a patch finding tool, at the level of a small change or a commit that follows best practices. We favored commits with changes that we could isolate. When isolating examples for Pumpkin, we replayed changes from the bottom up, as if we were making the changes ourselves. This means that we did not always make the same change as the user. For example, the real change from Section 3.6.1 updated multiple definitions; we updated only one.

Pumpkin is a proof-of-concept and does not yet handle some kinds of proofs. In each scenario, we made minor modifications to proofs so that we could use Pumpkin (for example, using induction instead of destruction). Pumpkin does not yet handle structural changes like adding constructors or parameters, so we focused on changes that preserve structure, like modifying constructors. Chapter 4 describes an extension to Pumpkin that supports changes in structure.

```
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 => O
  | 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 => O
  | B2 b' => (bin_to_nat b') +
       (bin_to_nat b')
  | B21 b' => S ((bin_to_nat b
      ') + (bin_to_nat b'))
  end.
```

Figure 18: Definitions of `bin_to_nat` for Users A (left) and B (right).

3.6.1 *Updating Definitions*

Coq programmers sometimes make changes to definitions that break proofs within the same project. To emulate this use case, we identified a CompCert commit [74] with a breaking change to `int` (Figure 17). We used Pumpkin to find a patch that corresponds to the change in `int`. The patch Pumpkin found fixed broken inductive proofs.

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

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

We replayed the change to `unsigned_range`:

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

SEARCH We 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. We 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 *Configuration*

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

```
Inductive T := ... | C : ... -> H -> T
Inductive T' := ... | C : ... -> H' -> T'
```

Pumpkin searches two inductive proofs of theorems:

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

for an isomorphism between the constructors:

```
... -> H -> H'
... -> H' -> H
```

The user 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 configuration for changes in constructors:

1: *diff* inductive constructors for goals
2: use *all components* 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*

Coq programmers sometimes port theorems and proofs to use definitions from different libraries. To simulate this, we used Pumpkin to port two solutions [3, 9] to an exercise in Software Foundations to each use the other solution's definition of the fixpoint `bin_to_nat` (Figure 18). We demonstrate one direction; the opposite was similar.

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

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

When we ported User A's solution to use User B's definition of `bin_to_nat`, the application of this inline lemma failed. We 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 We used Pumpkin to search the example 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)
```

6 If Pumpkin finds just one implication, it returns that.

```
∀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.X
```

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

```
 ... apply patch_inv. rewrite -> plus_0_r.✓
```

In this case, since we ported User A's definition to a simpler definition, PUMPKIN found a patch that was not the most natural patch. The natural patch would be to remove the rewrite, just as we removed a different rewrite from the example proof. This did not occur when we 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.

3.6.2.1 *Configuration*

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

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

PUMPKIN searches two proofs of theorems:

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

for an isomorphism that corresponds to the change:

```
∀P, P x -> P x'
∀P, P x' -> P x
```

The user 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 abstracts candidates by `P`, not by its arguments. PUMPKIN uses this configuration for changes in fixpoint cases:

For the prototype, we 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

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

diff fixpoint cases for goals
use *all components* to recursively search an intermediate lemma for a change in conclusions
if there are candidates **then**
    specialize and *factor* the candidate
    abstract the factors by functions
    try to *invert* the patch to find an isomorphism

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

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

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

intermediate lemma and the arguments: It can search within the proof for some proof of a function that is applied to the fixpoint.

3.6.3 *Updating Proof Assistant Versions*

Coq sometimes makes changes to its standard library that break backwards-compatibility. To test the plausibility of using a patch finding tool for proof assistant version updates, we identified a breaking change in the Coq standard library [75]. 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. We used Pumpkin to find an isomorphism that corresponds to the change in `divide`. The isomorphism Pumpkin found fixed broken proofs.

REPLAY We 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.✗
```

We replayed changes to `mod_divide`:

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

SEARCH We used Pumpkin to search the example 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 *Configuration*

This scenario used the configuration for changes in dependent arguments to constructors. Pumpkin searches two proofs that apply the same constructor to different dependent arguments:

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

for an isomorphism between the arguments:

```
∀ x, P x -> P' x
∀ x, P' x -> P x
```

The user 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:

```
∀x, P x -> P' x
```

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

```
(∃x, P x) -> (∃ x, P' x)
```

This changes the goal type that semantic differencing determines. Pumpkin uses this configuration for changes in constructor arguments:

1: *diff* constructor arguments for goals
2: use *all components* to recursively search those arguments for changes in conclusions

3: **if** there are candidates **then**
4: *abstract* 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 we ask the user to provide the type of the change in argument (to guide line 2). We can extend semantic differencing to remove this restriction.

3.7 CONCLUSION

Rehashing thesis and how we do it

What we haven't accomplished yet at this point (parts of PUMPKIN PATCH future work).

The PUMPKIN PATCH prototype did not apply the patches that it finds, handle changes in structure, or include support for tactics beyond the use of hints. The next chapter addresses these limitations.

4

PROOF REPAIR ACROSS TYPE EQUIVALENCES

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

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

PUMPKIN Pi generalizes that information and applies it automatically.

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

4.1 MOTIVATING EXAMPLE

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):

```
Lemma rev_app_distr {A} :
  ∀ (x y : list A), rev (x ++ y) = rev y ++ rev x.
```

1 We use induction instead of pattern matching.

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

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

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

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

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

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

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

Figure 21: Two functions between `Old.list` and `New.list` (top) that form an equivalence (bottom).

```
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.
Qed.
```

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.
Qed.
```

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.
Qed.
```

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: Coq compiles every proof script to a proof term in a rich functional programming language called 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|]`. A recent thesis [100] 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 4.2.1) to automatically 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* [108], or pairs of functions that map between two types (possibly with some additional information) and are mutual inverses (Section **??**). In other words, looking back to the thesis statement, the information shows up in the

2 The adjoint follows, and Pumpkin Pi includes machinery to prove it ⑩ ㉓.

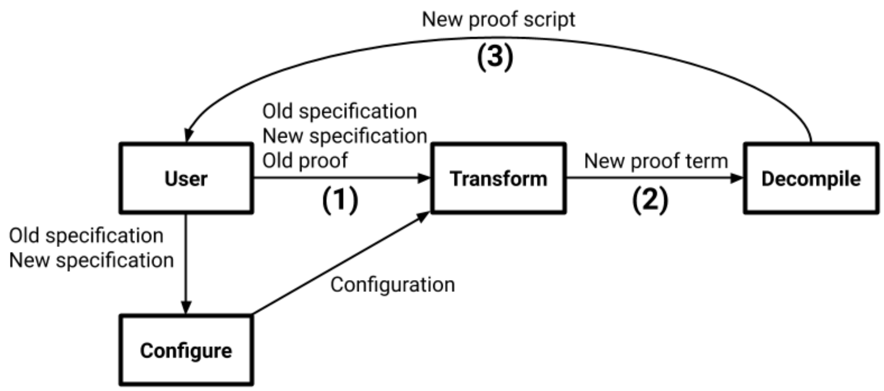


Figure 22: The workflow for Pumpkin Pi.

difference between versions of the changed datatype. Pumpkin Pi can extract and generalize that information, then apply it to fix other broken proofs.

Like the original Pumpkin prototype, it also does this using a combination of differencing and proof term transformations. The corresponding 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. (Mention decompiler briefly, tease implementation section and case studies, say something cute.)

4.2.1 *Workflow: Configure, Transform, Decompile*

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

1. The proof engineer **Configure**s Pumpkin Pi, either manually or automatically.

2. The configured **Transform** transforms the old proof term into the new proof term.

3. **Decompile** suggests a new proof script.

There are currently four search procedures for automatic configuration implemented in Pumpkin Pi (see Table 1 on page 85). Manual configuration makes it possible for the proof engineer to configure the transformation to any equivalence, even without a search procedure. Section 4.6 shows examples of both workflows applied to real scenarios.

```
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).

```
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.

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 21 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, we preview two changes below. See Table 1 on page 85 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 and proofs about `I` to instead use `J`, as long as she configures Pumpkin Pi to describe which constructor of `I` maps to `true` and which maps to `false`. This information about constructor mappings induces an equivalence `I` $\simeq$ `J` across which Pumpkin Pi repairs functions and proofs. File ② shows an example of this, mapping `A` to `true` and `B` to false, and repairing proofs of De Morgan's laws.

Adding a Dependent Index. At first glance, the word *equivalence* may seem to imply that Pumpkin Pi can support only changes in which the proof engineer does not add or remove information. But equivalences are more powerful than they may seem. Consider, for example, changing a list to a length-indexed vector (Figure 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` $\simeq$ `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 [5]. 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 takes as inputs . . . and returns . . .:

- **Inputs**: . . ., assuming:
  - . . .
- **Outputs**: . . ., guaranteeing:
  - . . .

other details like what is proven or whatever (here, maybe the list of differencing procedures actually supported?)

4.2.4 *Transformation: Transport with a Twist*

The goal of PUMPKIN Pi is to implement a kind of proof reuse known as *transport* [108], 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', we say that t and t' are *equal up to transport* across that equivalence (denoted $t \equiv_{A \simeq B} t'$). 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 21, 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`.

The transformation takes as input . . . and returns . . .:

3 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 [108], and an approximation in Coq is in the univalent parametricity framework paper [105]. For equivalent A and B, there can be many equivalences $A \simeq B$. Equality up to transport is across a *particular* equivalence, but we 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.

- **Inputs**: . . ., assuming:
  - . . .
- **Outputs**: . . ., guaranteeing:
  - . . .

other details like what is proven or whatever (here, maybe the list of differencing procedures actually supported?)

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.

4.3 Differencing

At the heart of Pumpkin Pi is a configurable proof term transformation for transporting proofs across equivalences (4). It is a generalization of the transformation from an earlier version of Pumpkin Pi called Devoid [**?**], which solved this problem a particular class of equivalences.

The transformation takes as input a deconstructed equivalence that we call a *configuration*. This section introduces the configuration (Section 4.3.1), defines the transformation that builds on that (Section **??**), then specifies correctness criteria for the configuration (Section 4.3.2). Section **??** describes the additional work needed to implement this transformation.

Conventions. All terms that we introduce in this section are in the Calculus of Inductive Constructions (CIC_ω), the type theory that Coq's proof term language Gallina implements. CIC_ω is based on the Calculus of Constructions (CoC), a variant of the lambda calculus with polymorphism (types that depend on types) and dependent types (types that depend on terms) [31]. CIC_ω extends CoC with inductive types [32]. Inductive types are defined solely by their constructors (like `nil` and `cons` for `list`) and eliminators (like the induction principle for `list`); this section assumes that these eliminators are primitive.

The syntax for CIC_ω with primitive eliminators is in Figure 28. The typing rules are standard. We assume inductive types Σ with constructor $\exists$ and projections π_l and π_r, and an equality type = with constructor `eq_refl`. We use $\vec{t}$ and $\{t_1, \ldots, t_n\}$ to denote lists of terms.

4.3.1 *The 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: preserve equality up to transport across the equivalence between A and B, and produce well-typed terms. This configuration is a pair of pairs:

```
((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. Each of these is defined in CIC_ω for each equivalence.

Carlo theory will go here: basically the names aren't coincidences, it's because this corresponds to an initial algebra, so it's more natural when you have inductive types but more general than that. Draw diagram, explain what each part corresponds to.

Preserving the Equivalence. To preserve the equivalence, the configuration ports terms over A to terms over B by viewing each term of type B as if it were an A. This way, the rest of the transformation 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 4.3.2 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:

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

and the eliminator it discovers eliminates the projections:

```
DepElim(s, P) { f0 f1 } : P (∃ (πl s) (πr s)) :=
  Elim(πr s, λ(n : nat)(v : vector T n).P (∃ n v)) {
    f0,
    (λ(t : T)(n : nat)(v : vector T n).f1 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. Furthermore, both examples use automatic configuration, so Pumpkin Pi's **Configure** component discovers `DepConstr` and `DepElim` from just the types A and B, taking care of even the difficult work.

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'` [106].

`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 [93].

```
Inductive nat :=
| O : nat
| S : nat → nat.
```

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

Inductive N :=
| N0 : N
| Npos : positive → N.
```

Figure 26: A unary natural number `nat` (left) is either zero (`O`) or the successor of some other natural number (`S`). A binary natural number `N` (right) is either zero (`N0`) or a positive binary number (`Npos`), where a positive binary number is either 1 (`xH`), or the result of shifting left and adding 1 (`xI`) or 0 (`xO`). Unary and binary natural numbers are equivalent, but have different inductive structures. Consequentially, definitional equalities over `nat` may become propositional over `N`.

ι-reduction (β-reduction for inductive types) describes how to reduce an elimination of a constructor [92].

The configuration for the change from `list` to `packed_vect` has propositional `Eta`. It uses η-expansion for Σ:

```
Eta(packed_vect) := λ(s:packed_vect).∃ (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).

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 pO pS n,
  DepElim(DepConstr(1, nat) n, P) { pO, pS } =
  pS n (DepElim(n, P) { pO, pS }).
```

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

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

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

Taken together over both A and B, `Iota` describes how the inductive structures of A and B differ. The transformation requires that `DepElim` over A and over B have the same structure as each other, so if A and B *themselves* have the same inductive structure (if they are *ornaments* [80]), 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.

4.3.2 *Specifying Correct Configurations*

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

While we cannot control for intentions, we *can* 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 27 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 27, 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` (`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.

```
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] ≡_{ξ(A,P_A,j)≃ξ(B,P_B,j)}
       f⃗_B[j]→
  DepElim(a, P_A) f⃗_A
      ≡_{(Pa)≃(Pb)} DepElim(b, P
      _B) f⃗_A.
```

```
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 27: 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 $\vec{\mathrm{IH}}$ represent sorts, eliminator cases, constructor arguments, and inductive hypotheses. ξ (A, P, j) is the type of DepElim(A, P) at DepConstr(j, A) (similarly for B).

Equality. To ensure coherence with equality (Figure 27, 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* [17]) 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.

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.

4.3.3 *Search Procedures*

PUMPKIN Pi implements four search procedures for automatic configuration ⑥. Three of the four procedures are based on the search procedure from DEVOID [**?**], while the remaining procedure instantiates the types A and B of a generic configuration that can be defined inside of Coq directly.

The algorithm above is essentially what **Configure** uses to generate functions `f` and `g` for the automatic configurations ⑨, and also generate proofs `section` and `retraction` that these functions form an equivalence ⑩. To minimize dependencies, PUMPKIN Pi does not produce proofs of `constr_ok` and `elim_ok` directly, as stating these theorems cleanly would require either a special framework [105] or a univalent type theory [108]. 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.

4.3.3.1 *Algebraic Ornaments*

Differencing in DEVOID discovers equivalences that correspond to *algebraic ornaments*. An algebraic ornament relates an inductive type A to an indexed version of that type B with a new index of type I_B, 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. Consequentially, there are two functions:

```
ltv : list T                  → Σ(n : nat).vector T n.
vtl : Σ(n : nat).vector T n → list T.
```

that are mutual inverses:

```
∀ (l : list T),                vtl (ltv l) = l.
∀ (v : Σ(n : nat).vector T n), ltv (vtl v) = v.
```

and therefore form the type equivalence from Section **??**. Moreover, since the new index is fully determined by `length`, we can relate `length` to `ltv`:

`∀ (l : list T), length l =` π_l `(ltv l).`

In general, we can view an algebraic ornament as a type equivalence:

$$A\ \vec{i}\ \simeq\ \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i})$$

where $\vec{i}$ are the indices of A, I_B is a function over those indices, and the `index` operation inserts the new index n at the right offset. Such a type equivalence consists of two functions [108]:

$$\begin{aligned}&\texttt{promote} : A\ \vec{i} && \rightarrow\ \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i}).\\ &\texttt{forget}\ : \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i}) && \rightarrow\ A\ \vec{i}.\end{aligned}$$

that are mutual inverses:

4 The adjunction condition follows from section and retraction.

```
section    : ∀ (a : A i⃗),                         forget (promote
     a)  = a.
retraction : ∀ (b_Σ : Σ(n : I_B i⃗).B (index n i⃗)), promote (forget
     b_Σ) = b_Σ.
```

An algebraic ornament is additionally equipped with an indexer, which is a unique fold:

$$\texttt{indexer} : A\ \vec{i} \rightarrow I_B\ \vec{i}.$$

which projects the promoted index:

$$\texttt{coherence} : \forall (a : A\ \vec{i}),\ \texttt{indexer}\ a = \pi_l\ (\texttt{promote}\ a).$$

Following existing work [68], we call this equivalence the *ornamental promotion isomorphism*; when it holds and the indexer exists, we say that B is an algebraic ornament of A.

`Find ornament` searches for algebraic ornaments between types and is, to the best of our knowledge, the first search algorithm for ornaments.

In their original form, ornaments are a programming mechanism: Given a type A, an ornament determines some new type B. We invert this process for algebraic ornaments: Given types A and B, DEVOID searches for an ornament 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.

`Find ornament` takes two inductive types and searches for the components of the ornamental promotion isomorphism between them:

- **Inputs**: Inductive types A and B, assuming:
  - B is an algebraic ornament of A,
  - B has the same number of constructors in the same order as A,
  - A and B do not contain recursive references to themselves under products, and
  - for every recursive reference to A in A, there is exactly one new hypothesis in B, which is exactly the new index of the corresponding recursive reference in B.
- **Outputs**: Functions `promote`, `forget`, and `indexer`, guaranteeing:
  - the outputs form the ornamental promotion isomorphism between the inputs.

`Find ornament` includes an option to generate a proof that the outputs form the ornamental promotion isomorphism; by default, this option is false, since `Lift` does not need this proof.

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

$\langle t \rangle ::= \langle v \rangle \mid \langle s \rangle \mid \Pi\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \lambda\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \langle t \rangle\ \langle t \rangle \mid \text{Ind}\ (\langle v \rangle : \langle t \rangle)\{\langle t \rangle,\ldots,\langle t \rangle\} \mid \text{Constr}\ (\langle i \rangle, \langle t \rangle) \mid \text{Elim}(\langle t \rangle, \langle t \rangle)\{\langle t \rangle,\ldots,\langle t \rangle\}$

$\Gamma \vdash t : T$ `// type checking`

$\Gamma \vdash t_1 \equiv_{\beta\delta\iota} t_2$ `// definitional equality`

t_β `// beta-reduction`

$t_{\beta\delta\iota}$ `// normalization`

$t\ [y\ /\ x]$ `// substitution`

$\xi\ (I,\ Q,\ c,\ C)$ `// type of eliminator`

Figure 28: CIC_ω syntax (left, from existing work [107]) and judgments and operations (right).

Presentation. We present both algorithms 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.

Notes on Syntax. The language the algorithms operate over is CIC_ω with primitive eliminators; this is a simplified version of the type theory underlying Coq. Figure 28 contains the syntax (which includes variables, sorts, product types, functions, inductive types, constructors, and eliminators), as well as the syntax for some judgments and operations, the rules for which are standard and thus omitted. For simplicity of presentation, we assume variables are names; we assume that all names are fresh. As in Coq, we assume the existence of an inductive type Σ for sigma types with projections π_l and π_r; for simplicity, we assume projections are primitive. Throughout, we use $\vec{i}$ and $\{t_1, \ldots, t_n\}$ to denote lists of terms, and we use $\vec{i}[j]$ to denote accessing the element of the list $\vec{i}$ at offset j.

Common Definitions. The algorithms assume list insertion and removal functions `insert` and `remove`, plus two functions Devoid implements: `off` computes the offset of the new index of type I_B in B's indices, and `new` determines whether a hypothesis in a case of the eliminator type of B is new. Figure 29 contains other common definitions, the names for which are reserved: The `index` and `deindex` functions insert an index into and remove an index from a list at the index computed by `off`. Input type A expands to an inductive type with indices of types $\vec{X_A}$, sort s_A, and constructors $\{\mathrm{C}_{A_1}, \ldots, \mathrm{C}_{A_n}\}$. P_A denotes the type of the motive of the eliminator of A, and each E_{A_i} denotes the type of the eliminator for the ith constructor of A. Analogous names are also reserved for input type B.

The `Find ornament` algorithm implements the specification. It builds on three intermediate steps: one to generate each of `indexer`, `promote`, and `forget`. Figure 30 shows the algorithm for generating `indexer`. The algorithms for generating `promote` and `forget` are similar; Figure 31 shows only the derivations for generating `promote` that are different

$$
\begin{aligned}
&A \ := \\
&\quad \text{Ind}(Ty_A : \Pi(\vec{i_A} : \vec{X_A}).s_A)\{C_{A_1}, \ldots, C_{A_n}\} \\
&B \ := \ \text{Ind}(Ty_B : \Pi(\vec{i_B} : \vec{X_B}).s_B)\{C_{B_1}, \ldots, C_{B_n}\} \\
&\forall 1 \leq i \leq n, \\
&\ \ E_{A_i} \ (p_A : P_A) \ := \\
&\qquad \xi(A, \ p_A, \ \text{Constr}(i, \ A), \ C_{A_i}) \\
&\ \ E_{B_i} \ (p_B : P_B) \ := \\
&\qquad \xi(B, \ p_B, \ \text{Constr}(i, \ B), \ C_{B_i})
\end{aligned}
$$

$$
\begin{aligned}
&P_A \ := \\
&\quad \Pi(\vec{i_A} : \vec{X_A})(a : A \ \vec{i_A}).s_A \\
&P_B \ := \ \Pi(\vec{i_B} : \vec{X_B})(b : B \ \vec{i_B}).s_B
\end{aligned}
$$

$$
\begin{aligned}
&\text{index} \ := \ \text{insert} \ (\text{off} \ A \ B) \\
&\text{deindex} \ := \\
&\quad \text{remove} \ (\text{off} \ A \ B)
\end{aligned}
$$

Figure 29: Common definitions for both algorithms.

from those for generating `indexer`, and the derivations for generating `forget` are omitted.

SEARCHING FOR THE INDEXER Search generates the `indexer` by traversing the types of the eliminators for A and B 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 $(T_A, T_B) \Downarrow_{i_m} t$ judgment consists of only the derivation INDEX-MOTIVE, which computes the indexer motive from the types A and B (expanded in Figure 29). It does this by constructing a function with A and its indices as premises, and the type I_B in the conclusion with the appropriate indices. Consider `list` and `vector`:

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

For these types, INDEX-MOTIVE computes the motive:

```
λ (l:list T) . nat
```

Generating Each Case. The $\Gamma \vdash (T_A, T_B) \Downarrow_{i_c} t$ judgment generates each case of the indexer by traversing in parallel the corresponding cases of the eliminator types for A and B. 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 B. It adds the new index to the environment, then recurses into the body of only the type for which the index already

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

$$\textsc{Index-Motive}\ \frac{}{\Gamma \vdash (A, B) \Downarrow_{i_m} \lambda(\vec{i_A} : \vec{X_A})(a : A\ \vec{i_A}).(I_B\ \vec{i_A})_\beta}$$

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

$$\textsc{Index-Conclusion}\ \frac{}{\Gamma \vdash (p_A\ \vec{i_A}\ a,\ p_B\ \vec{i_B}\ b) \Downarrow_{i_c} \vec{i_B}[\text{off}\ A\ B]}$$

$$\textsc{Index-Hypothesis}\ \frac{\text{new}\ n_B\ b_B \qquad \Gamma,\ n_B : t_B \vdash (\Pi(n_A : t_A).b_A,\ b_B) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_B : t_B).b_B) \Downarrow_{i_c} t}$$

$$\textsc{Index-IH}\ \frac{\begin{array}{l}\Gamma \vdash (A, B) \Downarrow_{i_m} p \\ \Gamma,\ n_A : p\ \vec{i_A}\ a \vdash (b_A,\ b_B[n_A/\vec{i_B}[\text{off}\ A\ B]]) \Downarrow_{i_c} t\end{array}}{\begin{array}{l}\Gamma \vdash (\Pi(n_A : p_A\ \vec{i_A}\ a).b_A,\ \Pi(n_B : p_B\ \vec{i_B}\ b).b_B) \\ \quad \Downarrow_{i_c} \lambda(n_A : p\ \vec{i_A}\ a).t\end{array}}$$

$$\textsc{Index-Prod}\ \frac{\Gamma,\ n_A : t_A \vdash (b_A,\ b_B[n_A/n_B]) \Downarrow_{i_c} t}{\begin{array}{l}\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_B : t_B).b_B) \\ \quad \Downarrow_{i_c} \lambda(n_A : t_A).t\end{array}}$$

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

$$\textsc{Index-Ind}\ \frac{\begin{array}{l}\Gamma \vdash (A,\ B) \Downarrow_{i_m} p \\ \Gamma,\ p_A : \mathrm{P}_A,\ p_B : \mathrm{P}_B \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{B_1}\ p_B), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{B_n}\ p_B)\} \Downarrow_{i_c} \vec{f}\end{array}}{\Gamma \vdash (A,\ B) \Downarrow_i \lambda(\vec{i_a} : \vec{X_A})(a : A\ \vec{i_a}).\text{Elim}(a, p)\vec{f}}$$

Figure 30: Identifying the indexer function.

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 ($\mathtt{t}_l$:T) (l:list T) ($\mathtt{IH}_l$:$\mathtt{p}_A$ l), ... Π ($\mathtt{t}_v$:T) (v:vector T n) ($\mathtt{IH}_v$:$\mathtt{p}_B$ 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 ($\mathtt{IH}_l$:$\mathtt{p}_A$ l), $\mathtt{p}_A$ (cons $\mathtt{t}_l$ l) Π ($\mathtt{IH}_v$:$\mathtt{p}_B$ n l), $\mathtt{p}_B$ (S n) (consV n $\mathtt{t}_l$ l)

Index-IH then takes over. It substitutes the new motive in the inductive hypothesis, then recurses into both bodies, substituting the new inductive hypothesis for the index in the eliminator type for B. Here, it substitutes the new motive for $\mathtt{p}_A$ in the type of $\mathtt{IH}_l$, extends the environment with $\mathtt{IH}_l$, then substitutes $\mathtt{IH}_l$ for `n`, so that it recurses on these types:

$\mathtt{p}_A$ (cons $\mathtt{t}_l$ l) $\mathtt{p}_B$ (S $\mathtt{IH}_l$) (consV $\mathtt{IH}_l$ $\mathtt{t}_l$ l)

Finally, Index-Conclusion computes the conclusion by taking the index of motive p_B at off A B, here S $\mathtt{IH}_l$. In total, this produces a function that computes the length of `cons t l`:

λ ($\mathtt{t}_l$:T) (l:list T) ($\mathtt{IH}_l$:(λ (l:list T).nat) l).S $\mathtt{IH}_l$

Composing the Result. The $\Gamma \vdash (T_A,\ T_B) \Downarrow_i t$ 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 ($\mathtt{t}_l$:T) (l:list T) ($\mathtt{IH}_l$:(λ (l:list T).nat) l).S $\mathtt{IH}_l$}

4.3.3.2 *Searching for Promote and Forget*

Figure 31 shows the interesting derivations for the judgment $(T_A,\ T_B) \Downarrow_p t$ that searches for `promote`: Promote-Motive identifies the motive as B with a new index (which it computes using `indexer`, denoted by metavariable π). When Promote-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. Promote-Conclusion returns the entire term, rather than its index. Finally, Promote-Ind not only recurses into each case, but also packs the result.

The omitted derivations to search for `forget` are similar, except that the domain and range are switched. Consequentially, `indexer` is never needed; Forget-Motive removes the index rather than inserting it, and Forget-IH no longer substitutes the index. Additionally, Forget-Hypothesis adds the hypothesis for the new index rather than

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

PROMOTE-MOTIVE
$$\frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi}{\Gamma \vdash (A,\ B) \Downarrow_{p_m} \lambda(\vec{i_a} : \vec{X_A})(a : A\ \vec{i_a}).B\ (\text{index}\ (\pi\ \vec{i_a}\ a)\ \vec{i_a})}$$

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

PROMOTE-CONCLUSION
$$\frac{}{\Gamma \vdash (p_A\ \vec{i_A}\ a,\ p_B\ \vec{i_B}\ b) \Downarrow_{p_c} b}$$

PROMOTE-IH
$$\frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ B) \Downarrow_{p_m} p \qquad \Gamma,\ n_A : p\ \vec{i_A}\ a \vdash (b_A,\ b_B[n_A/b][\pi\ \vec{i_A}\ a/\vec{i_B}[\text{off}\ A\ B]]) \Downarrow_{p_c} t}{\Gamma \vdash (\Pi(n_A : p_A\ \vec{i_A}\ a).b_A,\ \Pi(n_B : p_B\ \vec{i_B}\ b).b_B) \Downarrow_{p_c} \lambda(n_A : p\ \vec{i_A}\ a).t}$$

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

PROMOTE-IND
$$\frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ B) \Downarrow_{p_m} p \qquad \Gamma,\ p_A : \mathrm{P}_A,\ p_B : \mathrm{P}_B \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{B_1}\ p_B), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{B_n}\ p_B)\} \Downarrow_{p_c} \vec{f}}{\Gamma \vdash (A,\ B) \Downarrow_p \lambda(\vec{i_A} : \vec{X_A})(a : A\ \vec{i_A}).\exists\ (\pi\ \vec{i_A}\ a)\ (\text{Elim}(a, p)\vec{f})}$$

Figure 31: Identifying the promotion function.

skipping it, and FORGET-IND eliminates over the projection rather than packing the result.

CORE SEARCH ALGORITHM The core search algorithm produces `indexer`, `promote`, and `forget`, then composes them into a tuple. This tuple is how DEVOID represents ornaments internally. DEVOID has options (used in `Example.v`) that tell search to generate proofs that its outputs are correct, thereby increasing confidence in and usefulness of those outputs. The proof of `coherence` is reflexivity. The intuition behind the automation to prove `section` and `retraction` (`equivalence.ml`) is that `promote` and `forget` map along corresponding constructors, so inductive cases preserve equalities. Thus, each inductive case of these proofs is generated by a fold that rewrites each recursive reference, with reflexivity as identity.

4.3.3.3 *Other Search Procedures*

Brief explanation of these and how differencing works for them in detail based on algebraic ornaments

DESIGNING NEW SEARCH PROCEDURES How hard, how useful

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

$$\textsc{Dep-Elim}\quad \frac{\Gamma \vdash a \Uparrow b \qquad \Gamma \vdash p_a \Uparrow p_b \qquad \Gamma \vdash \vec{f_a} \Uparrow \vec{f_b}}{\Gamma \vdash \text{DepElim}(a,\ p_a)\vec{f_a} \Uparrow \text{DepElim}(b,\ p_b)\vec{f_b}}$$

$$\textsc{Dep-Constr}\quad \frac{\Gamma \vdash \vec{t_a} \Uparrow \vec{t_b}}{\Gamma \vdash \text{DepConstr}(j,\ A)\ \vec{t_a} \Uparrow \text{DepConstr}(j,\ B)\ \vec{t_b}} \qquad \textsc{Eta}\quad \frac{}{\Gamma \vdash \text{Eta}(A) \Uparrow \text{Eta}(B)}$$

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

$$\textsc{Constr}\quad \frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{t} \Uparrow \vec{t'}}{\Gamma \vdash \text{Constr}(j,\ T)\ \vec{t} \Uparrow \text{Constr}(j,\ T')\ \vec{t'}} \qquad \textsc{Ind}\quad \frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{C} \Uparrow \vec{C'}}{\Gamma \vdash \text{Ind}(Ty : T)\vec{C} \Uparrow \text{Ind}(Ty : T')\vec{C'}}$$

$$\textsc{App}\quad \frac{\Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash f t \Uparrow f' t'} \qquad \textsc{Elim}\quad \frac{\Gamma \vdash c \Uparrow c' \qquad \Gamma \vdash Q \Uparrow Q' \qquad \Gamma \vdash \vec{f} \Uparrow \vec{f'}}{\Gamma \vdash \text{Elim}(c, Q)\vec{f} \Uparrow \text{Elim}(c', Q')\vec{f'}}$$

$$\textsc{Lam}\quad \frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \lambda(t : T).b \Uparrow \lambda(t' : T').b'}$$

$$\textsc{Prod}\quad \frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \Pi(t : T).b \Uparrow \Pi(t' : T').b'} \qquad \textsc{Var}\quad \frac{v \in \text{Vars}}{\Gamma \vdash v \Uparrow v}$$

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

4.3.4 *Limitations*

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

4.4 TRANSFORMATION

Figure 37 shows the proof term transformation $\Gamma \vdash t \Uparrow t'$ that forms the core of Pumpkin Pi. The transformation is parameterized over equivalent types A and B (Equivalence) as well as the configuration. 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. Of course, typical proof terms

```
(* 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.
```

```
(* 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.
```

```
(* 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.
```

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.

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. It then runs the proof term transformation, which transports proofs across the equivalence that corresponds to the configuration.

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 transfor-

$$
\begin{array}{ll}
\uparrow \quad \{\vec{i_a} : \vec{X_A}\} := \texttt{promote}\ \vec{i_a}. & \downarrow \\
\quad\ \{\vec{i_b} : \vec{X_B}\} := \texttt{forget}\ \vec{i_b}. & \\
\pi_{I_B}\ \{\vec{i_a} : \vec{X_A}\} := \texttt{indexer}\ \vec{i_a}. & \exists_{I_B}\ \{\vec{i_b} : \vec{X_B}\}\ (b : B\ \vec{i_b}) \\
\quad := \exists\ \vec{i_b}\texttt{[off] b}. & \\
\uparrow_B\ := \pi_r \circ \uparrow. & \downarrow_A\ := \downarrow \circ \exists_{I_B}. \\
\uparrow_{I_B}\ := \pi_l \circ \uparrow. & \downarrow_{I_B}\ := \pi_{I_B} \circ \downarrow_A.
\end{array}
$$

Figure 34: Common definitions for the core lifting algorithm.

mation 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).

Algebraic Ornaments. Consider instantiating the transformation to algebraic ornaments. We show only one direction of the algorithm, promoting from A to packed B; the forgetful direction is similar. The core algorithm (Figure 37) builds on a set of common definitions (Figure 34) and two intermediate judgments: one to lift eliminators (Figure 35) and one to lift constructors (Figure 36).

Common Definitions. The common definitions (Figure 34) define some useful syntax: $\uparrow$ applies `promote`, $\downarrow$ applies `forget`, and π_{I_B} applies `indexer`. $\exists_{I_B}$ packs a term of type B into an existential with the index at the appropriate offset. $\uparrow_B$ and $\uparrow_{I_B}$ promote and then project; $\downarrow_A$ packs and forgets, and $\downarrow_{I_B}$ packs, forgets, and then applies `indexer` to project the index.

LIFTING ELIMINATORS The $\Gamma \vdash t \Uparrow_E t'$ judgment (Figure 35) defines rules for lifting the motive and case of an eliminator, changing the *domain of induction* from A to B. The intuition is that any term of type A is the result of forgetting some term of type packed B. Then, since A and B have the same inductive structure, we can lift the eliminator of A to the eliminator of B, and move that forgetfulness *inside of each case*. For example, the following terms are propositionally equal:

```
Elim((↓A b), pA){
  fnil,
  (λ(tl:T)(l:list T)(IHl:
      pA l).
    fcons tl l IHl)
}
```

```
Elim(b, λ(n:nat)(v:vector T n).pA
    (↓A v)){
  fnil,
  (λ(n:nat)(tv:T)(v:vector T n)(IHv:pA
      (↓A v)).
    fcons tv (↓A v) IHv)
}
```

The induction rules implement this transformation. Case lifts a case of the eliminator by first recursively lifting the motive, then using the lifted motive to compute the type of the new case, and then using that type to compute the body of the new case. In the example above, when lifting the inductive case, it first recursively lifts the motive p_A

$$\boxed{\Gamma \vdash (t,\ T) \Uparrow_{E_x} t'}$$

$$\textsc{Drop-Index}\ \frac{\text{new } n\ b \qquad \Gamma,\ n : t \vdash (f,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : t).b) \Uparrow_{E_x} \lambda(n : t).b'}$$

$$\textsc{Forget-Arg}\ \frac{\Gamma \vdash \vec{i} : \vec{X}_B \qquad \Gamma,\ n : B\ \vec{i} \vdash ((f\ (\downarrow_A n))_\beta,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : B\ \vec{i}).b) \Uparrow_{E_x} \lambda(n : B\ \vec{i}).b'}$$

$$\textsc{Arg}\ \frac{\Gamma,\ n : t \vdash ((f\ n)_\beta,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : t).b) \Uparrow_{E_x} \lambda(n : t).b'} \qquad \textsc{Concl}\ \frac{}{\Gamma \vdash (t,\ p_B\ \vec{y}) \Uparrow_{E_x} t}$$

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

$$\textsc{Motive}\ \frac{\Gamma \vdash p_A : \mathrm{P}_A}{\Gamma \vdash p_A \Uparrow_E \lambda(\vec{i} : \vec{X}_B)(b : B\ \vec{i}).(p_A\ (\text{deindex}\ \vec{i})\ (\downarrow_A b))_\beta}$$

$$\textsc{Case}\ \frac{\Gamma \vdash p_A : \mathrm{P}_A \qquad \Gamma \vdash f_i : \mathrm{E}_{A_i}\ p_A \qquad \Gamma \vdash p_A \Uparrow_E p_B \qquad \Gamma \vdash (f_i,\ \mathrm{E}_{B_i}\ p_B) \Uparrow_{\mathrm{E_x}} f_i'}{\Gamma \vdash f_i \Uparrow_E f_i'}$$

Figure 35: Lifting eliminators.

using Motive, which drops the index, packs and forgets the argument of type B, and then β-reduces the result, eliminating references to B. This produces the new motive:

```
λ(n:nat)(v:vector T n).p_A (↓_A v)
```

which Case then uses to compute the type of the inductive case of the eliminator for B:

```
Π(t_v:T)(n:nat)(v:vector T n)(IH_v:p_A (↓_A v)).p_A (↓_A (consV t_v (
    S n) v))
```

The $\Gamma \vdash (t,\ T) \Uparrow_{E_x} t'$ judgment then uses that type to compute the lifted function body. It computes this in a similar way to Motive, except that there are as many indices to drop and arguments to pack and forget as there are inductive hypotheses, and these do not occur in predictable places, so more rules are involved. This computes the new function:

```
λ(n:nat)(t_v:T)(v:vector T n)(IH_v:p_A (↓_A v)).f_cons t_v (↓_A v) IH_v
```

LIFTING CONSTRUCTORS The $\Gamma \vdash t \Uparrow_C t'$ judgment (Figure 36) lifts applications of constructors of A to applications of constructors of B. This judgment computes one step of the promotion, leaving the recursive lifting of the arguments to the final algorithm. Using the same types, in the base case:

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

NORMALIZE

$$\frac{}{\Gamma \vdash \text{Constr}(j,\ A)\ \vec{x} \Uparrow_C (\uparrow (\text{Constr}(j,\ A)\ \vec{x}))_{\beta\delta\iota}}$$

Figure 36: Lifting constructors.

$\uparrow$ `nil` $\equiv_{\beta\delta\iota}$ $\exists$ `0 nilV`

and in the inductive case:

$\uparrow$ `(cons t l)` $\equiv_{\beta\delta\iota}$ $\exists$ `(S (`$\uparrow_{I_B}$` l)) (consV (`$\uparrow_{I_B}$` l) t (`$\uparrow_B$` l))`

This derivation consists of only one rule: NORMALIZE, which normalizes the promotion of the constructor. This is guaranteed to succeed because the application of the constructor is fully η-expanded. The core algorithm later internalizes the promotion functions in the result.

CORE LIFTING ALGORITHM The core algorithm (Figure 37) builds on these intermediate judgments. The interesting derivations for correctness are the first six: LIFT-ELIM and LIFT-CONSTR use the judgments for lifting eliminators and constructors of A. INTERNALIZE internalizes the explicit `promote` functions from the lifted constructors to recursive applications of the algorithm. RETRACTION and COHERENCE use the respective properties of the ornamental promotion isomorphism metatheoretically: the first to drop the explicit `forget` functions from the lifted eliminators, and the second to lift the `indexer` to a projection (in the forgetful direction, SECTION replaces RETRACTION). Finally, EQUIVALENCE lifts A along the equivalence to packed B. The remaining derivations recurse predictably.

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 trusted computing base. 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). For metatheoretical reasons, without additional axioms, a proof of this theorem in Coq can only be approximated [105]. It would be possible to generate per-transformation proofs of correctness, but this does not serve an engineering need.

4.4.1 *Limitations*

Limitations and whether they're addressed in other tools yet

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

$$\textsc{Lift-Elim}\ \dfrac{\begin{array}{l}\Gamma \vdash \vec{i} : \vec{X_A} \qquad \Gamma \vdash a : A\ \vec{i} \\ \Gamma \vdash p_a \Uparrow_E p' \qquad \Gamma \vdash \vec{f_a} \ \Uparrow_E \vec{f'} \\ \Gamma \vdash p' \Uparrow p_b \qquad \Gamma \vdash \vec{f'} \Uparrow \vec{f_b} \qquad \Gamma \vdash a \Uparrow b_\Sigma\end{array}}{\Gamma \vdash \text{Elim}(a,\ p_a)\vec{f_a} \Uparrow \text{Elim}(\pi_r\ b_\Sigma,\ p_b)\vec{f_b}}$$

$$\textsc{Lift-Constr}\ \dfrac{\begin{array}{l}\Gamma \vdash \vec{i} : \vec{X_A} \qquad \Gamma \vdash \text{Constr}(j,\ A)\ \vec{t_a} : A\ \vec{i} \\ \Gamma \vdash \text{Constr}(j,\ A)\ \vec{t_a} \Uparrow_C t' \\ \Gamma \vdash t' \Uparrow t''\end{array}}{\Gamma \vdash \text{Constr}(j,\ A)\ \vec{t_a} \Uparrow t''}$$

$$\textsc{Internalize}\ \dfrac{\Gamma \vdash a \Uparrow b_\Sigma}{\Gamma \vdash \uparrow\ a \Uparrow b_\Sigma}$$

$$\textsc{Retraction}\ \dfrac{\Gamma \vdash\ b_\Sigma \Uparrow b'_\Sigma}{\Gamma \vdash \downarrow\ b_\Sigma \Uparrow b'_\Sigma}$$

$$\textsc{Coherence}\ \dfrac{\Gamma \vdash \vec{i} : \vec{X_A} \qquad \Gamma \vdash a : A\ \vec{i} \qquad \Gamma \vdash a \Uparrow b_\Sigma}{\Gamma \vdash \pi_{I_B}\ a\ \Uparrow (\pi_l\ b_\Sigma)_\beta}$$

$$\textsc{Equivalence}\ \dfrac{\Gamma \vdash \vec{i} : \vec{X_A}}{\Gamma \vdash A\ \vec{i} \Uparrow \Sigma(n : (I_B\ \vec{i})_\beta).B\ (\text{index}\ n\ \vec{i})}$$

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

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

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

$$\textsc{App}\ \dfrac{\Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash f t \Uparrow f' t'}$$

$$\textsc{Lam}\ \dfrac{\Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \lambda(t : T).b \Uparrow \lambda(t : T').b'}$$

$$\textsc{Prod}\ \dfrac{\Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \Pi(t : T).b \Uparrow \Pi(t : T').b'}$$

Figure 37: Core lifting algorithm.

4.5 IMPLEMENTATION

Parts of PUMPKIN Pi and DEVOID implementation, plus more (still need to arrange, fill in, and so on)

4.5.1 *Tool Details*

Implemented in blah blah blah, and so on.

4.5.2 *Workflow Integration*

4.5.2.1 *Configure*

4.5.2.2 *Transform*

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 we can always run EQUIVALENCE over and over again, forever. We thus include some simple termination checks in our 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* As to Bs. PUMPKIN Pi has some support for this using an interactive workflow (13), with plans for automatic support in the future.

From CIC$_\omega$ to Coq. The implementation (4) of the transformation handles language differences to scale from CIC$_\omega$ to Coq. We use the existing `Preprocess` [?] command to turn pattern matching and fixpoints into eliminators. We handle refolding of constants in constructors using `DepConstr`.

Reaching Real Proof Engineers. Many of our design decisions in implementing PUMPKIN Pi were informed by our 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 we implemented optional aggressive caching, even caching intermediate subterms encountered while running the transformation (14). We 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, we implemented a search procedure to generate custom eliminators to help reason about types like Σ`(l : list T).length l`

$\langle v \rangle \in$ 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 38: Qtac syntax.

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

$$\textsc{Intro}\ \frac{\Gamma,\ n : T \vdash b \Rightarrow p}{\Gamma \vdash \lambda(n : T).b \Rightarrow \text{intro } n.\ p} \qquad \textsc{Symmetry}\ \frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \texttt{eq\_sym}\ H \Rightarrow \text{symmetry}.\ p}$$

$$\textsc{Split}\ \frac{\Gamma \vdash l \Rightarrow p \qquad \Gamma \vdash r \Rightarrow q}{\Gamma \vdash \text{Constr}(0,\ \wedge)\ l\ r \Rightarrow \text{split}\{p, q\}.}$$

$$\textsc{Left}\ \frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \text{Constr}(0,\ \vee)\ H \Rightarrow \text{left}.\ p} \qquad \textsc{Right}\ \frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \text{Constr}(1,\ \vee)\ H \Rightarrow \text{right}.\ p}$$

$$\textsc{Rewrite}\ \frac{\Gamma \vdash H_1 : x = y \qquad \Gamma \vdash H_2 \Rightarrow p}{\Gamma \vdash \text{Elim}(H_1,\ P)\{x,\ H_2,\ y\} \Rightarrow \text{symmetry. rewrite } P\ H_1.\ p}$$

$$\textsc{Induction}\ \frac{\Gamma \vdash \vec{f} \Rightarrow \vec{p}}{\Gamma \vdash \text{Elim}(t,\ P)\ \vec{f} \Rightarrow \text{induction } P\ t\ \vec{p}} \qquad \textsc{Apply}\ \frac{\Gamma \vdash t \Rightarrow p}{\Gamma \vdash f t \Rightarrow \text{apply } f.\ p}$$

$$\textsc{Base}\ \frac{}{\Gamma \vdash t \Rightarrow \text{apply } t}$$

Figure 39: Qtac decompiler semantics.

`= n` by reasoning separately about the projections (15). We added informative error messages (22) to help the proof engineer distinguish between user errors and bugs. These features helped with workflow integration.

4.5.2.3 *Decompile*

Transform produces a proof term, while the proof engineer typically writes and maintains proof scripts made up of tactics. We improve usability thanks to the realization that, since Coq's proof term language Gallina is very structured, we can decompile these Gallina terms to suggested Ltac proof scripts for the proof engineer to maintain.

Decompile implements a prototype of this translation (11): it translates a proof term to a suggested proof script that attempts to prove the same theorem the same way. Note that this problem is not well defined: while there is always a proof script that works (applying the proof term with the `apply` tactic), the result is often qualitatively unreadable. This is the baseline behavior to which the decompiler defaults. The goal of the decompiler is to improve on that baseline as much as possible, or else suggest a proof script that is close enough to correct that the proof engineer can manually massage it into something that works and is maintainable.

Decompile achieves this in two passes: The first pass decompiles proof terms to proof scripts that use a predefined set of tactics. The second pass improves on suggested tactics by simplifying arguments, substituting tacticals, and using hints like custom tactics and decision procedures.

First Pass: Basic Proof Scripts. The first pass takes 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, we start by defining a mini decompiler that implements a simplified version of the first pass. Section **??** explains how we scale this to the implementation.

The mini decompiler takes CIC_ω terms and produces tactics in a mini version of Ltac which we call Qtac. The syntax for Qtac is in Figure 38. 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 39 (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 to move this to Coq. The generated proof term of `rev_app_distr`

```
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 40: 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.

from Section 4.1, for example, consists only of induction, rewriting, simplification, and reflexivity (solved by `auto`). Figure 40 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 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 refold 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 we pass 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, **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. We 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.

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.

4.6 RESULTS

This section summarizes eight case studies using Pumpkin Pi, corresponding to the eight rows in Table 1. These case studies highlight Pumpkin Pi's flexibility in handling diverse scenarios, the success of automatic configuration for better workflow integration, the preliminary success of the prototype decompiler, and clear paths to better serving proof engineers. Detailed walkthroughs are in the code.

Algebraic Ornaments: Lists to Packed Vectors. The transformation in Pumpkin Pi is a generalization of the transformation from Devoid. Devoid supported proof reuse across *algebraic ornaments*, which describe relations between two inductive types, where one type is the other indexed by a fold [80]. A standard example is the relation between a list and a length-indexed vector (Figure 24).

Pumpkin Pi implements a search procedure for automatic configuration of algebraic ornaments. The result is all functionality from Devoid, plus tactic suggestions. In file (3), we used this to port functions and a proof from lists to vectors of *some* length, since `list T` $\simeq$

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 [**?**], the Univalent Parametricity (UP) white-box transformation [106], and the tool from Magaud & Bertot 2000 [78]. Pumpkin Pi is the only one that suggests tactics. More nuanced comparisons to these and more are in Section 5.

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

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

Figure 41: A simple language (left) and the same language with two swapped constructors and an added constructor (right).

`packed_vect T`. The decompiler helped us write proofs in the order of hours that we 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), we then ported functions and proofs to vectors of a *particular* length, like `vector T n`. DEVOID had left this step to the proof engineer. We 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 Σ`(v : vector T m).n = m` for some `m`, then let PUMPKIN Pi instantiate those equalities via unification heuristics, and 2) generate a custom eliminator for combining list terms with length invariants. The resulting workflow works not just for lists and vectors, but for any algebraic ornament, automating manual effort from DEVOID. The suggested tactics were helpful for writing proofs in the order of hours that we 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 [**?**]. A change from one of the benchmarks is in Figure 41. 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, we 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. We then used PUMPKIN Pi to repair functions after adding the new constructor in Figure 41, separating out the proof obligations for the new constructor from the old terms (19). This change combined manual and automatic configuration. We 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, we hope to automate the discovery of the more efficient terms, or use the refinement framework CoqEAL [28] 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 a non-author. We 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 us 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, we expect time savings would increase for a larger development.

Permute Hypotheses: External Example. The change in (20) came at the request of a different non-author (a cubical type theory expert), and 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), we 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 [78], we 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. We 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 we used for `DepElim`:

```
N.peano_rect_succ : ∀ P pO pS n,
  N.peano_rect P pO pS (N.succ n) =
  pS n (N.peano_rect P pO pS n).
```

The need for nontrivial `Iota` comes from the fact that `N` and `nat` have different inductive structures. By writing a manual configuration with this `Iota`, it was possible for us to implement this 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, we 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, we obtained automatically repaired proofs about slow binary addition, which we found simple to port to fast binary addition. We hope to automate this last step in the future using CoqEAL. Repaired tactics were partially useful, but failed to understand custom eliminators like `N.peano_rect`, and to generate useful tactics for 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.

4.7 CONCLUSION

Rehashing thesis and how we do it

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

5

RELATED WORK

TODO somewhere here or elsewhere (if elsewhere, fix references): talk about what lessons carry over to automated theorem provers, and which lessons carry over to other ITPs, and what work is needed to reach those tools.

5.1 PROGRAMS

Program Refactoring

Refactoring [84].

Program Repair

Proof repair can be viewed as a form of *program repair* [89, 50] for proof assistants. Proof assistants like Coq are a good fit for program repair: A recent paper [98] recommends that program repair tools draw on extra information such as specifications or example patches. In Coq, specifications and examples are rich and widely available: specifications thanks to dependent types, and examples thanks to constructivism.

Program repair tools for languages with non-dependent type systems [95, 76, 71, 82, 88] may have applications in the context of a dependently typed language. Similarly, our work may have applications within program repair in these languages: Future applications of our approach may repurpose it to repair programs for functional languages.

Ornaments

Devoid automates discovery of and lifting across algebraic ornaments in a higher-order dependently typed language. In the decade since the discovery of ornaments [80], there have been a number of formalizations and embedded implementations of ornaments [36, 67, 37, 68, 35]. Devoid is the first tool for ornamentation to operate over a non-embedded dependently typed language. It essentially moves the automation-heavy approach of Ornamentation in ML [113], which operates on non-embedded ML code, into the type theory that forms the basis of theorem provers like Coq. In doing so, it takes advantage of the properties of algebraic ornaments [80]. It also introduces the first search algorithm to identify ornaments, which in the past was identified as a "gap" in the literature [68].

Programming by Example

Our approach generalizes an example that the programmer provides. This is similar to programming by example, a subfield of program synthesis [53]. This field addresses different challenges in different logics, but may drive solutions to similar problems in a dependently typed language.

Differencing & Incremental Computation

Existing work in differencing and incremental computation may help improve our semantic differencing component. Type-directed diffing [87] finds differences in algebraic data types. Semantics-based change impact analysis [7] models semantic differences between documents. Differential assertion checking [70] analyzes different versions of a program for relative correctness with respect to a specification. Incremental λ-calculus [20] introduces a general model for program changes. All of these may be useful for improving semantic differencing.

5.2 PROOFS

Proof Reuse

Our approach reimagines the problem of proof reuse in the context of proof automation. While we focus on changes that occur over time, traditional proof reuse techniques can help improve our approach.

Proof reuse for extended inductive types [15] adapts proof obligations to structural changes in inductive types. Later work [90] proposes a method to generate proofs for new constructors. These approaches may be useful when extending the differencing component to handle structural changes. Existing work in theorem reuse and proof generalization [46, 97, 60] abstracts existing proofs for reusability, and may be useful for improving the abstraction component. Our work focuses on the components critical to searching for patches; these complementary approaches can drive improvements to the components.

A few proof reuse tools work by proof term transformation and so can be used for repair. Existing work [60] describes a transformation that generalizes theorems in Isabelle/HOL. PUMPKIN Pi generalizes the transformation from DEVOID [?], which transformed proofs along algebraic ornaments [80]. Magaud & Bertot 2000 [78] implement a proof term transformation between unary and binary numbers. Both of these fit into PUMPKIN Pi configurations, and none suggests tactics in Coq like PUMPKIN Pi does. The expansion algorithm from Magaud & Bertot 2000 [78] may help guide the design of unification heuristics in PUMPKIN Pi.

Existing work in proof reuse focuses on transferring proofs between isomorphisms, either through extending the type system [12] or through an automatic method [79]. This is later generalized and implemented in Isabelle [56] and Coq [?, ?]; later methods can also handle implications.

The widely used Transfer [56] package 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. One possible path toward implementing proof repair in Isabelle/HOL may be to reify proof terms using something like Isabelle/HOL-Proofs, apply a transformation based on Transfer, and then (as in REPLICA) decompile those terms to automation that does not apply Transfer or refer to the old datatype in any way.

CoqEAL [28] 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. Both CoqEAL and recent ornaments work [112] may help with better workflow support for changes that do not correspond to equivalences.

The PUMPKIN Pi transformation implements transport. Transport is realizable as a function given univalence [108]. UP [105] approximates it in Coq, only sometimes relying on functional extensionality. While powerful, neither approach removes references to the old type.

Recent work [106] extends UP with a white-box transformation that may work for repair. This imposes proof obligations on the proof engineer beyond those imposed by REPLICA, and it includes neither search procedures for equivalences nor tactic script generation. It also does not support changes in inductive structure, instead relying on its original black-box functionality; `Iota` solves this in REPLICA. The most fruitful progress may come from combining these tools.

DEVOID identifies and lifts proofs along a specific equivalence similar to that from existing ornaments work [68]. The need to automatically lift functions and proofs across equivalences and other relations is a long-standing challenge for proof engineers [78, 13, 77, 57, 116, 29]. The univalence axiom from Homotopy Type Theory [108] enables transparent transport of proofs; cubical type theory [27] gives univalence a constructive interpretation.

The problem that we solve is fundamentally about proof reuse, which applies software reuse principles to ITPs. There is a wealth of work in proof reuse, from tactic languages [47] and logical frameworks [21], to tools for proof abstraction and generalization [?, 61], to domain-specific methodologies [38] and frameworks [39].

DEVOID focuses on the specific problem of reuse when adding fully-determined indices to types. Other approaches to this problem include combinators which definitionally reduce to desirable terms [44] in the language Cedille, and automatic generation of conversion functions in Ghostbuster [81] for GADTs in Haskell. Our work focuses on a type theory different from both of these, in which the properties that allow for such combinators in Cedille are not present, and in which dependent types introduce challenges not present in Haskell.

DEVOID is not the first tool to combine search with reuse. Optician [86] synthesizes bidirectional string transformations; a similar approach may help extend tooling to handle transformations for low-level data. PUMPKIN PATCH [?] searches the difference in proofs for patches that can be used to repair proofs broken by changes; DEVOID uses a similar approach to identify functions that form an equivalence. The resulting tools are complementary: DEVOID supports the addition of indices and hypotheses, which PUMPKIN PATCH does not support; PUMPKIN PATCH supports changes in values, which DEVOID does not support.

Proof Evolution

There is a small body of work on change and dependency management for verification, both to evaluate impact of potential changes and maximize reuse [58, 6] and to optimize build performance [22]. These approaches may help isolate changes, which is necessary to identify future benchmarks, integrate with CI systems, and fully support version updates.

Proof Refactoring

Proof repair is related to proof refactoring [110]. The proof refactoring tool Levity [16] for Isabelle/HOL has seen large-scale industrial use. Levity focuses on a different task: moving lemmas. Chick [100] and RefactorAgda [111] are proof refactoring tools in a Gallina-like language and in Agda, respectively. These tools support primarily syntactic changes and do not have tactic support.

A few proof refactoring tools operate directly over tactics: POLAR [45] refactors proof scripts in languages based on Isabelle/Isar [109], CoqPIE [101] is an IDE with support for simple refactorings of Ltac scripts, and Tactician [4] is a refactoring tool for switching between tactics and tacticals. This approach is not tractable for more complex changes [100].

Proof Design

Much work focuses on designing proofs to be robust to change, rather than fixing broken proofs. This can take the form of design principles, like using information hiding techniques [114, 65] or any of the structures [**?**, 104, 102] for encoding interfaces in Coq. CertiKOS [52] introduces the idea of a deep specification to ease verification of large systems. Design principles for specific domains (like formal metatheory [8, 40, 41]) 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.

Another approach to this is to use heavy proof automation, for example through program-specific proof automation [25] or general-purpose hammers [14, 94, 62, 33]. The degree to which proof engineers rely on automation varies, as seen in the data from a 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 REPLICA to better support developments in this style.

Proof Automation

We address a missed opportunity in proof automation for ITP: searching for patches that can fix broken proofs. This is complementary to existing automation techniques. Nonetheless, there is a wealth of work in proof automation that makes proofs more resilient to change. Powerful tactics like `crush` [24] can make proofs more resilient to changes. Hammers like Isabelle's sledgehammer [94] can make proofs agnostic to some low-level changes. Recent work [34] paves the way for a hammer in Coq. Even the most powerful tactics cannot address all changes; our hope is to open more possibilities for automation.

Powerful project-specific tactics [24, 23] can help prevent low-level maintenance tasks. Writing these tactics requires good engineering [51] and domain-specific knowledge, and these tactics still sometimes break in the face of change. A future patching tool may be able to repair tactics; the debugging process for adapting a tactic is not too dissimilar to providing an example to a tool.

Rippling [19] is a technique for automating inductive proofs that uses restricted rewrite rules to guide the inductive hypothesis toward the conclusion; this may guide improvements to the differencing, abstraction, and specialization components. The abstraction and factoring components address specific classes of unification problems; recent developments to higher-order unification [85] may help improve these components. Lean [103] introduces the first congruence closure algorithm for dependent type theory that relies only on the Uniqueness of Identity Proofs (UIP) axiom. While UIP is not fundamental to Coq, it is frequently assumed as an axiom; when it is, it may be tractable to use a similar algorithm to improve the tool.

GALILEO [18] repairs faulty physics theories in the context of a classical higher-order logic (HOL); there is preliminary work extending this style of repair to mathematical proofs. Knowledge-sharing methods [49] can adapt some proofs across different representations of HOL. These complementary approaches may guide extensions to support decidable domains and classical logics.

6

CONCLUSIONS & FUTURE WORK

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

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

THE NEXT ERA: PROOF ENGINEERING FOR ALL

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

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

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

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

Proof Engineering for Experts

Unifying theme: lateral reach. Some examples:

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

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

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

Proof Engineering for Practitioners

Unifying theme: usability. Some examples:

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

INTEGRATION IDE & CI integration, HCI for repair.

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

Proof Engineering for Software Engineers

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

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

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

SPECIFICATION INFERENCE Analysis to infer specs (TA1).

Proof Engineering for New Domains

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

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

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

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

INDEX

Conventions
Primitive Eliminators, 13, 18, 19

Languages
Gallina, 10, 11

BIBLIOGRAPHY

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

[2] Lean theorem prover, 2017.

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

[4] 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.

[5] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. Internalizing representation independence with univalence, 2020.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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 Compu-*

tation Structures, FoSSaCS '01, pages 57–71, London, UK, UK, 2001. Springer-Verlag.

[13] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *International Conference on Foundations of Software Science and Computation Structures*, pages 57–71. Springer, 2001.

[14] 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.

[15] 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.

[16] 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.

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

[18] Alan Bundy. The interaction of representation and reasoning. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 469(2157), 2013.

[19] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press, New York, NY, USA, 2005.

[20] 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.

[21] Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 106–113. ACM, 1995.

[22] 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.

[23] Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 391–402, New York, NY, USA, 2013. ACM.

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

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

[26] Adam Chlipala. Library equality, 2017.

[27] 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.

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

[29] Cyril Cohen and Damien Rouhling. A refinement-based approach to large scale reflection for algebra. In *JFLA 2017 - Vingt-huitième Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

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

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

[32] 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.

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

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

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

[36] 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.

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

[38] Benjamin Delaware, William Cook, and Don Batory. Product lines of theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 595–608, New York, NY, USA, 2011. ACM.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Maxime Dénes. Coq 8.7 beta 1 is out, 2017.

[44] Larry Diehl, Denis Firsov, and Aaron Stump. Generic zero-cost reuse for dependent types. *CoRR*, abs/1803.08150, 2018.

[45] 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.

[46] Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In Frank Pfenning, editor, *Logic Programming and Automated Reasoning: 5th International Conference*, LPAR '94, pages 1–15, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[47] Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 1–15. Springer, 1994.

[48] 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.

[49] Thibault Gauthier and Cezary Kaliszyk. Matching concepts across HOL libraries. In Stephen Watt, James Davenport, Alan Sexton, Petr Sojka, and Josef Urban, editors, *CICM '14*, volume 8543 of *LNCS*, pages 267–281. Springer Verlag, 2014.

[50] 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.

[51] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 163–175, New York, NY, USA, 2011. ACM.

[52] 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.

[53] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

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

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

[56] 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.

[57] Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pages 131–146. Springer, 2013.

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

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

[60] 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.

[61] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *International Conference on Theorem Proving in Higher Order Logics*, pages 152–167. Springer, 2004.

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

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

[69] Matej Kosik. Coq pull request # 652: Put all plugins behind an "api", 2017.

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

[71] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 593–604, New York, NY, USA, 2017. ACM.

[72] 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.

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

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

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

[76] 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.

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

[78] 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.

[79] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *Types for Proofs and Programs: International Workshop*, TYPES 2000, pages 181–196, Berlin, Heidelberg, 2002. Springer.

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

[81] Trevor L. McDonell, Timothy A. K. Zakian, Matteo Cimini, and Ryan R. Newton. Ghostbuster: A tool for simplifying and converting GADTs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 338–350, New York, NY, USA, 2016. ACM.

[82] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

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

[84] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.

[85] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

[86] Anders Miltner, Kathleen Fisher, Benjamin C Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proceedings of the ACM on Programming Languages*, 2(POPL):1, 2017.

[87] 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.

[88] Martin Monperrus. Automatic Software Repair: a Bibliography. *ACM Computing Surveys*, 2017.

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

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

[91] 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.

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

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

[94] 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.

[95] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automatic program repair by fixing contracts. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 246–260, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

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

[97] Olivier Pons. Generalization in type theory based proof assistants. In *Types for Proofs and Programs*, pages 217–232, Berlin, Heidelberg, 2002. Springer.

[98] 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.

[99] 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.

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

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

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

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

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

[109] 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.

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

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

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

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

[114] 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.

[115] 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.

[116] Theo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the Coq proof assistant. *arXiv preprint arXiv:1505.05028*, 2015.