Proof Engineering Tools for a New Era

Talia Ringer

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

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Name of Chairperson, Chair

First committee member

Next committee member

etc

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Proof Engineering Tools for a New Era

Talia Ringer

Chair of the Supervisory Committee:
Title of Chair Name of Chairperson
Department of Chair

This is my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Number Page

ACKNOWLEDGMENTS

Dan, Zach, Derek, the Coq community, Valentin, my family, PLSE lab, Qi.

DEDICATION

To my parents. Hope this fits on your fridge.

Chapter 1

INTRODUCTION

Chapter 2

PROOF DEVELOPMENT

Chapter 3

PROOF MAINTENANCE

Chapter 4
PROOF REPAIR

4.1 Introduction

Proof automation makes verification more accessible to programmers, but it is often intractable without programmer guidance. In *interactive theorem proving* (ITP), the programmer guides the proof search process. The guidance reduces the search space to make proof automation more tractable. This in turns helps the programmer, who does not have to manually verify the entire system.

Despite automation, programming in these proof assistants is brittle: Even a minor change to a definition or theorem can break many dependent proofs. This is a major source of development inefficiency in proof assistants based on dependent type theory [90, 14, 36].

Traditional proof automation does not consider how proofs, definitions, and theorems change over time. Instead, it is driven by the state of the current proof, sometimes with supplementary information from other proofs, definitions, and theorems (as in hint databases [3] and rippling [24]). This puts the burden of dealing with brittleness on the programmer.

We present a new approach to proof automation that accounts for breaking changes. In our approach, the programmer guides proof search by providing an *example* of how to adapt proofs to changes in definitions or theorems. A tool then generalizes the example adaptation into a *reusable patch* that the programmer can use to fix other broken proofs.

In doing so, we chart a path for a future that moves the burden of brittleness away from the programmer and into proof automation. Programmers typically address brittleness through design principles that make proofs resilient to change [90, 14, 36], or through program-specific proof automation [28]. These techniques, while useful, have limitations: Planning a verification effort around future change is challenging, and program-specific automation requires specialized knowledge. The programmer's ability to anticipate likely changes determines the robustness of both techniques in the face of change. Even then, many breaking changes are outside of the programmer's control. Updating proof assistant versions, for example, can break proofs regardless of planning or automation [74].

```
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 4.1: 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.

We identify a set of core components that are critical to searching an example for patches in Coq. We use the components to build a procedure for finding patches in a Coq plugin as a proof-of-concept. Case studies on real projects like CompCert [55] and the Coq standard library suggest that patches are useful for realistic scenarios, and that it is simple to compose the components to handle different classes of changes. We test the boundaries of the components on a suite of tests and show how our findings can help build the future we envision.

In summary, we contribute the following:

1. We identify a set of core components that are key to searching for patches.

2. We demonstrate that these core components are useful on real changes in Coq code.

3. We explore how to drive a future that moves the burden of change in ITP away from the programmer and into automated tooling.

4.2 Automation, Reimagined

Traditional proof automation considers only the current state of theorems, proofs, and definitions. This is a missed opportunity: Verification projects are rarely static. Like other software, these projects evolve over time.

Currently, the burden of change largely falls on programmers. This does not have to be true. Proof automation can view theorems, proofs, and definitions as fluid entities: When a proof or specification changes, a tool can search the difference between the old and new versions for a *reusable patch* that can fix broken proofs.

Status Quo Experienced Coq programmers 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 programmer. This can be problematic when change occurs outside of the programmer's control.

Consider a commit from the upcoming Coq 8.7 release [65]. This commit redefined injection from integers to reals (Figure 4.1). This change broke 18 proofs in the standard library.

The Coq developer who committed the change fixed the broken proofs, then made an additional 12 commits to address the change in `coq-contribs`, a regression suite of projects that the Coq developers maintain as 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, one proof broke on this line:

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

It succeeded with the lemma:

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

These changes are outside-facing: Coq users have to make similar changes to their own proofs when they update to Coq 8.7. The Coq developer can update some tactics to account for this, but it is impossible to account for every tactic that users could use. Furthermore, while the developer responsible for the changes knows about the lemma that describes the change, the Coq user does not. The Coq user must determine how the definition has changed and how to address the change, perhaps by reading documentation or by talking to the developers.

Our Vision When a user updates Coq, a 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 user never has to consider how the definition has changed.

4.3 Generating Reusable Patches

We identify five components that are key to finding reusable patches. We implement these components in a prototype Coq plugin, which we call Pumpkin Patch (Proof Updater Mechanically Passing Knowledge Into New Proofs, Assisting The Coq Hacker), or Pumpkin.[1]

We focus the Pumpkin prototype on the proof assistant Coq [2]. In Coq, each theorem is a type, and a proof of that theorem is a term that inhabits that type. Rather than write proof terms directly, users write proof scripts in a high-level tactic language; Coq then uses these scripts to guide search for a proof term.

The Pumpkin repository contains a detailed user guide. To use Pumpkin, the programmer modifies a single proof script to provide an *example* of how to adapt a proof to a change. Pumpkin generalizes the example adaptation into a *reusable patch*: a function that can be used to fix other broken proofs, which Pumpkin defines as a Coq term. We focus on the problem of *finding* these patches; we discuss how to extend Pumpkin to automatically *apply* patches it finds in Section 4.7.

[1] `http://github.com/uwplse/PUMPKIN-PATCH/tree/cpp18`

We motivate the core components (Section 4.3.1) and describe how they compose to find patches (Section 4.3.2). We find patches for real code in Section 4.4, and we describe our implementation in Section 4.5.

```
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 4.2: 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.

4.3.1 Motivating the Core

We identify and isolate five core components critical to finding reusable patches:

1. *Semantic differencing* between terms

2. *Patch specialization* to arguments
3. *Patch abstraction* of arguments or functions
4. *Patch inversion* to reverse a patch
5. *Lemma factoring* to break a term into parts

The semantic differencing component finds the difference between two terms, which produces a *candidate* for a reusable patch. The other components modify the candidate to try to produce a patch. To motivate this workflow, consider using Pumpkin to search the proofs in Figure 4.2 for a patch between conclusions. We invoke the plugin using `old` and `new` as the example change:

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

Pumpkin first determines the type that a patch from `new` to `old` should have. To determine this, it semantically *diffs* the types and finds this goal type (line 2):

```
∀ n m p, n <= m -> m <= p -> n <= p -> n <= p + 1
```

It then breaks each inductive proof into cases and determines an intermediate goal type for the candidate. In the base case, for example, it *diffs* the types and determines that a candidate between the base cases of `new` and `old` should have this type (lines 11 and 12):

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

It then *diffs* the terms (line 13) for such a candidate:

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

This candidate is close, but it is not yet a patch. 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`). Pumpkin *abstracts* this 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 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
```

The user can then use `patch` to fix other broken proofs. For example, given a proof that applies `old`, the user can use `patch` to prove the same conclusion by applying `new`:

```
apply old.✓
apply patch. apply new.✓
```

This simple example uses only three components. The other components help turn candidates into patches in similar ways. We discuss all five components in more detail in the rest of this section.

Semantic Differencing The tool should be able to identify the semantic difference between terms. The semantic difference is the difference between two terms that corresponds to the difference between their types. Consider the base case terms in Figure 4.2 (line 13):

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

The semantic differencing component first identifies the difference in their types, or the *goal type*:

```
n <= m -> n <= m + 1
```

It then finds a difference in terms that has that type:

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

This is the *candidate* for a reusable patch that the other components modify to find a patch.

Differencing operates over terms and types. Differencing tactics is insufficient, since tactics and hints may mask patches (line 5).[2] Furthermore, differencing is aware of the semantics of terms and types. Simply exploring the syntactic difference makes it hard to identify which changes are meaningful. For example, in the inductive case (line 14), the inductive hypothesis changes:

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

However, 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. Our semantic differencing component knows that it can ignore this change.

Patch Specialization The tool should be able to specialize a patch candidate to specific arguments as determined by the differences in terms. To find a patch for Figure 4.2, for example, Pumpkin must specialize the patch candidate to `p` to produce the final patch.

Patch Abstraction A tool should be able to abstract patch candidates of this form by the common argument:

```
candidate : P’ t -> P t
candidate_abs : ∀ t0, P’ t0 -> P t0
```

and it should be able to abstract patch candidates of this form by the common function:

```
candidate : P t’ -> P t
candidate_abs : ∀ P0, P0 t’ -> P0 t
```

This is necessary because the tool may find candidates in an applied form. For example, when searching for a patch between the proofs in Figure 4.2, Pumpkin finds a candidate in the difference of base cases. To produce a patch, Pumpkin must abstract the candidate by the argument `m`. Abstracting candidates is not always possible; abstraction will necessarily be a collection of heuristics.

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

Patch Inversion The tool should be able to invert a patch candidate. This is necessary to search for isomorphisms. It is also necessary to search for implications between propositionally equal types, since candidates may appear in the wrong direction. For example, consider two list lemmas (we write `length` as `len`):

```
old : ∀ l' l, len (l' ++ l) = len l' + len l
new : ∀ l' l, len (l' ++ l) = len l' + len (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_ind_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_ind_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 [3], we can port this proof:

```
Theorem app_rev_len : ∀ l l',
  len (rev (l' ++ l)) = len (rev l) + len (rev l').
Proof.
  intros. rewrite rev_app_distr. apply old.✓
```

to this proof:

```
  intros. rewrite rev_app_distr. apply new.✓
```

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 4.2. Inversion will necessarily sometimes fail, since not all terms are invertible.

Lemma Factoring The tool should be able to factor a term into a sequence of lemmas. This can help break other problems, like abstraction, into smaller subproblems. It is also necessary to invert certain terms. Consider inverting an arbitrary sequence of two rewrites:

```
t := eq_ind_r G ... (eq_ind_r F ...)
```

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

```
g := eq_ind_r G ...
f := eq_ind_r F ...
t := g ∘ f
```

The inverse of t is the following:

$t^{-1} := f^{-1} \circ g^{-1}$

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.

4.3.2 Composing Components

The components come together to form a proof patch finding procedure:

Pseudocode: find_patch(term, term', direction)

```
1:  diff types of term and term' for goals
2:  diff term and term' for candidates
3:  if there are candidates then
4:      factor, abstract, specialize, and/or invert candidates
5:      if there are patches then return patches
6:  if direction is forwards then
7:      find_patch(term', term, backwards) for candidates
8:      factor and invert candidates
9:      if there are patches then return patches
10: return failure
```

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

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

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

Figure 4.3: Old (left) and new (right) definitions of `int` in CompCert.

2, and how to use the components on line 4.

For example, to find a patch for Figure 4.2, Pumpkin used the configuration for changes in conclusions of two proofs that induct over the same hypothesis. Given two such proofs:

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

Pumpkin searches for a patch with this type:

```
∀ x, H x -> P' x -> P x
```

using this configuration:

```
1: diff conclusion types for goals
2: diff conclusion terms for candidates
3: if there are candidates then
4:     abstract and then specialize candidates
```

Section 4.4 describes real-world examples that demonstrate more configurations.

4.4 Case Studies: Changes in the Wild

We used the Pumpkin prototype to emulate three motivating scenarios from real-world code:

1. **Updating definitions** within a project

   (CompCert, Section 4.4.1)

2. **Porting definitions** between libraries

   (Software Foundations, Section 4.4.2)

3. **Updating proof assistant versions**
   (Coq Standard Library, Section 4.4.3)

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 4.4.1 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 shape, like modifying constructors. We discuss supporting these features and whether they may necessitate other components in Section 4.7.

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

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

Figure 4.4: Definitions of `bin_to_nat` for Users A (left) and B (right).

4.4.1 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 [57] with a breaking change to `int` (Figure 4.3). 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.✓
```

Core Components

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[3] 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:

[3] If Pumpkin finds just one implication, it returns that.

```
diff inductive constructors for goals
use all components to recursively search for changes in conclusions of the corresponding case of the proof
if there are candidates then
    try to invert the patch to find an isomorphism
```

4.4.2 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 [10, 15] to an exercise in Software Foundations to each use the other solution's definition of the fixpoint `bin_to_nat` (Figure 4.4). 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)
∀P b, P (bin_to_nat b + 0) -> P (bin_to_nat b)
```

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

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

Attempting the obvious patch from the difference in tactics—rewriting by plus_n_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,[4] 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.

Core Components

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

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

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:

```
*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
```

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

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

Figure 4.5: Old (left) and new (right) definitions of `divide` in Coq.

4.4.3 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 [58]. The commit changed the definition of `divide` prior to the Coq 8.4 release (Figure 4.5). 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.✓
```

Core Components

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.

4.5 Inside the Core

We have shown that the core components are useful for finding proof patches. This section describes our implementation of the components (Section 4.5.1) and the procedure (Section 4.5.2) so that it is possible to implement them in other systems. While our system is a very early prototype under active development, we have made the source code available on Github.[5] Our prototype has no impact on the trusted computing base (Section 4.5.3).

4.5.1 Inside the Components

The interested reader can follow along in the repository.

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.[6] 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 4.6) 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 *)
```

[5] `http://github.com/uwplse/PUMPKIN-PATCH/tree/cpp18`

[6] These trees are inspired by categorical models of dependent type theory [45].

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

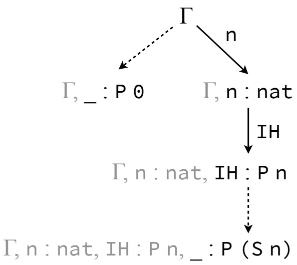

Figure 4.6: The type of (left) and tree for (right) the induction principle `nat_ind`. The solid edges represent hypotheses, and the dotted edges represent the proof obligations for each case in an inductive proof.

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 only 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; we discuss a broad range of extensions in Sections 4.6.2 and 4.7.

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 [29] 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.

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 4.6.2), 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; we identify new strategies in Section 4.6.2.

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. In the future, it may be possible to use this swapping functionality in a reflexive application of the induction principle to infer symmetry properties like `eq_sym` for other types. The component can then use these properties to generate reverse induction principles like `eq_ind_r`. We further detail inversion in Section 4.6.2.

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.

4.5.2 Inside the Procedure

The implementation (`patcher.ml4`) of the procedure from Section 4.3.2 starts with a preprocessing step which compiles the proof terms to trees (like the tree in Figure 4.6). 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 4.4.1, 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 4.3), 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.

4.5.3 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 [37]. 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 [52]. Furthermore, Coq type-checks terms that plugins produce. Since PUMPKIN does not modify the type checker, it cannot produce an ill-typed term.

4.6 Testing Boundaries

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

In the case studies in Section 4.4, we showed how the core components are useful for real scenarios. In this section, we explore the boundary between what the PUMPKIN prototype can and cannot handle. It is precisely this boundary that informs us how to improve the implementations of the core components.

To evaluate this boundary, we tested the core components of PUMPKIN on a suite of 50 pairs of proofs (Section 4.6.1). We designed 11 of these pairs to succeed, then modified their proofs to produce the remaining 39 pairs that try to stress the core functionality of the tool. We learned the following from the pairs that tested PUMPKIN's limitations:

1. **The failed pairs drive improvements.**

   PUMPKIN failed on 17 of 50 pairs. These pairs tell us how to improve the core. (Section 4.6.2)

2. **The pairs unearth abstraction strategies.**

   PUMPKIN produced an exponential number of candidates in 5 of 50 pairs. New abstraction strategies can dramatically reduce the number of candidates. (Section 4.6.2)

3. **Pumpkin was fast, and it can be faster.**

   The slowest successful patch took 48 ms. The slowest failure took 7 ms. Simple changes can make PUMPKIN more efficient. (Section 4.6.3)

4.6.1 Patch Generation Suite

We wrote a suite[7] of 50 pairs of proofs. We wrote these proofs ourselves since searching for proof patches is a new domain, so there was no existing benchmark suite to work with. We used the following methodology:

1. Choose theorems `old` and `new`

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

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

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

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

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

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

For example, one pair of theorems old and new was a simplification of the auxiliary lemmas that we encountered in the case study in Section 4.4.2. For the first proof of old, we added a rewrite, like in the case study:

```
rewrite <- plus_n_O. rewrite -> plus_comm.
```

For the second proof of old, we commuted the rewrites:

```
rewrite -> plus_comm. rewrite <- plus_n_O.
```

We then searched for patches in both directions, since the conclusions of old and new were propositionally equal.

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

4.6.2 Three Challenges

Pumpkin found patches for 33 of the 50 pairs. 28 of the 33 successes did not stress Pumpkin at all: Pumpkin found the correct candidate immediately and was able to abstract it in one try. The pairs that Pumpkin failed to patch and the successful pairs that stressed abstraction reveal key information about how to improve the core components. We walk through three examples. Future tools can use these as challenge problems to improve upon Pumpkin.

A Challenge for Differencing For one pair of proofs of theorems with propositionally equal conclusions (Figure 4.7), 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 [29]) `le_plus_trans`, this is not obvious:

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

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

A Challenge for Inversion For one pair of proofs with propositionally equal conclusions, Pumpkin found a patch in one direction, but failed to invert it:

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

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

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

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

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

```
  ...
  : ∀ n m, S n <= m -> n < m
```

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

A Challenge for Abstraction Abstraction produced an exponential number of candidates when abstracting a patch candidate with this type:

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

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

```
∀ m0 n n0,
  n <= max m0 n0 ->
  n <= max n0 m0
```

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

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

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

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

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

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

Then the candidate type is exactly this:

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

Abstraction should thus abstract the highlighted subterms and the terms that have types constrained by those subterms. This produces a patch in one candidate:

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

This same strategy would find a patch for one of the pairs that Pumpkin failed to abstract. This is a natural future direction for abstraction.

4.6.3 Performance

Pumpkin performed well for all pairs. The slowest success took 48 ms.[8] When Pumpkin failed, it failed fast. The slowest failure took 7 ms. While we find this promising, proof terms

[8] i7-4790K, at 4.00 GHz, 32 GB RAM

were small ($\leq$ 67 LOC); we leave evaluating performance on larger terms to future work.

PUMPKIN was slowest when the patch showed up inverted in the difference of proofs, since PUMPKIN had to search twice, once in each direction. A future procedure may determine which direction to search first ahead of time; proof term size may be a simple heuristic for this.

4.7 Conclusions & Future Work

Our vision is a future of proof automation in ITP that adapts proofs to breaking changes. This will unify the spirit of ITP—collaboration between the programmer and the tool—with the realities of modern proof engineering: Verification projects are large, specifications evolve over time, and dependencies change and break backward-compatibility. Too much of the burden of change rests on the programmer; not enough rests on the tool.

We conclude with a discussion of improvements that can help bring this vision to life. These improvements are driven by our experiences using the PUMPKIN prototype and by conversations within the ITP community.

Supporting structural changes. Coq programmers often make structural changes. It is common, for example, to add new hypotheses, constructors, or parameters to a type. The ideal tool should find patches for these changes. Existing work in proof reuse [20, 70] and type-directed diffing [68] may help guide these improvements.

Exploring new components. The core components of PUMPKIN are critical to searching for patches in Coq, but they may not be sufficient for the ideal tool. While we find the flexibility of these components promising thus far, in implementing new features, we may discover new components. For example, patches for certain structural changes may be program transformations as opposed to terms; supporting these patches may reveal new components.

Modeling diverse proof styles. 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.

Improving user experience. The ideal proof patching tool should be natural for programmers to use. A future patching tool can produce tactics from the patches it finds, that way programmers can remove references to old specifications. A future patching tool can integrate with an IDE (such as Proof General [8]) or continuous integration (CI) system (such as Travis [9]) to suggest patches at natural steps in the development process.

Handling version updates. Updating Coq versions is an ideal use case for finding proof patches: When the client updates Coq, a tool can automatically search commits to the standard library or to `coq-contribs` for patches. In this case, the example comes from the Coq developers, not from the library client—the client never has to look at the changes that the Coq developers make. The ideal tool should fully support updates in Coq versions. PUMPKIN can patch certain changes in the standard library, but it does not yet search for those changes automatically and determine which configuration to use depending on what has changed. Furthermore, as proof assistant versions change, so may the AST and the plugin interface. To fully support version updates, the tool should support different language versions.

Isolating changes. Programmers sometimes make multiple changes to a verification project in the same commit; the ideal tool should break down large changes into small, isolated changes to find patches. This will help in finding benchmarks, supporting library and version updates, and integrating with CI systems. We may draw on work in change and dependency management [47, 12, 26] to identify changes, then use the factoring component to break these changes into smaller parts.

Supporting other proof assistants. Coq is just one of many proof assistants; ideal tools should support different proof assistants. While Pumpkin focuses on Coq, the underlying concepts extend to other proof assistants with constructive logics (for example, Lean [6] and Agda [1]). Proof assistants with non-constructive logics (for example, Isabelle/HOL [5]) may benefit from a different approach; this is similar to the problem of finding patches for proofs of decidable domains in Coq, since classical properties provably hold for decidable propositions [7].

Applying patches. The ideal tool should not only find patches, but also apply the patches it finds automatically to fix broken proofs. In some cases, this may be as simple as adding the patches as hints to a hint database, so that proofs go through with no changes. However, hint databases in Coq cannot support certain terms [3], and adding too many hints may impede performance. More generally, we can integrate Pumpkin with a transfer tactic [46, 93], which is a perfect fit: Transfer tactics automatically adapt proofs between isomorphic types and implications, but they do not find these functions; Pumpkin finds these functions, but it does not apply them.

4.8 Related Work

Our work builds upon prior research in proof reuse, proof automation, proof engineering, refactoring, differencing & incremental computation, programming by example, and program repair.

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. Existing work in proof reuse focuses on transferring proofs between isomorphisms, either through extending the type system [16] or through an automatic method [61]. This is later generalized and implemented in Isabelle [46] and Coq [93, 86]; later methods can also handle implications. Integrating a transfer tactic with a proof patch finding tool will create an end-to-end tool that can both find patches and apply them automatically.

Proof reuse for extended inductive types [20] adapts proof obligations to structural changes in inductive types. Later work [70] 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 [38, 80, 49] 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.

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` [28] can make proofs more resilient to changes. Hammers like Isabelle's sledgehammer [75] can make proofs agnostic to some low-level changes. Recent work [31] 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 [28, 27] can help prevent low-level maintenance tasks. Writing these tactics requires good engineering [41] 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 [24] 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 [67] may help improve these components. Lean [83] 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 [23] 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 [40] can adapt some proofs across different representations of HOL. These complementary approaches may guide extensions to support decidable domains and classical logics.

Proof Engineering Existing proof engineering work addresses brittleness by planning for changes [90] and designing theorems and proofs that make maintenance less of an issue. Design principles for specific domains (such as formal metatheory [14, 35, 36]) can make verification more tractable. CertiKOS [42] introduces the idea of a deep specification to ease verification of large systems. Ornaments [32, 89] separate the computational and logical components of a datatype, and may make proofs more resilient to datatype changes. These design principles and frameworks are complementary to our approach. Even when programmers use informed design principles, changes outside of the programmer's control can break proofs; our approach addresses these changes.

There is a small body of work on change and dependency management for verification, both to evaluate impact of potential changes and maximize reuse [47, 12] and to optimize build performance [26]. These approaches may help isolate changes, which is necessary to identify future benchmarks, integrate with CI systems, and fully support version updates.

Refactoring Our approach is close in spirit to refactoring [66]. The Haskell refactoring tool HaRe [4] automatically lifts definitions, and may be useful for improving abstraction. There is a growing body of work on refactoring in the context of ITP [87, 21]. The IDE CoqPIE [81] and the verification platform Why3 [19] can both adapt Coq proofs to simple syntactic changes. It may be possible to use our lemma factoring component to improve proof refactoring tools. Proof refactoring tools are semantics-preserving; unlike these tools, our approach handles semantic changes.

Differencing & Incremental Computation Existing work in differencing and incremental computation may help improve our semantic differencing component. Type-directed diffing [68] finds differences in algebraic data types. Semantics-based change impact analysis [13] models semantic differences between documents. Differential assertion checking [53] analyzes different versions of a program for relative correctness with respect to a specification. Incremental λ-calculus [25] introduces a general model for program changes. All of these may be useful for improving semantic differencing.

Programming by Example Our approach generalizes an example that the programmer provides. This is similar to programming by example, a subfield of program synthesis [43]. This field addresses different challenges in different logics, but may drive solutions to similar problems in a dependently typed language.

Program Repair Adapting proofs to changes is essentially program repair for dependently typed languages. Program repair tools for languages with non-dependent type systems [76, 60, 54, 64, 69] 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.

Chapter 5

OFTEN, PRACTICALLY

Chapter 6

UNDER THE HOOD

Chapter 7

THE NEXT ERA

BIBLIOGRAPHY

[1] Agda, 2017.

[2] Coq, 2017.

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

[4] Hare: The haskell refactoring tool, 2017.

[5] Isabelle/hol: A proof assistant for higher-order logic, 2017.

[6] Lean theorem prover, 2017.

[7] Library coq.logic.decidable, 2017.

[8] Proof general, 2017.

[9] Travis ci, 2017.

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

[11] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.

[12] 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.

[13] 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.

[14] 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.

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

[16] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '01, pages 57–71, London, UK, UK, 2001. Springer-Verlag.

[17] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Mike Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: A formalization of homotopy type theory in coq. *CoRR*, abs/1610.04591, 2016.

[18] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 460–475, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving user proofs across specification changes. In Ernie Cohen and Andrey Rybalchenko, editors, *Fifth Working Conference on Verified Software: Theories, Tools and Experiments*, volume 8164, pages 191–201, Atherton, United States, May 2013. Springer.

[20] Olivier Boite. Proof reuse with extended inductive types. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *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 Berlin Heidelberg.

[21] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Makarius Wenzel, editor, *Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management*, pages 32–48, Bremen, Germany, July 2012. Springer.

[22] E. Brady. *Type-Driven Development with Idris*. Manning Publications Company, 2016.

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

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

[25] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation.

In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA, 2014. ACM.

[26] Ahmet Celik, Karl Palmskog, and Milos Gligoric. icoq: Regression proof selection for large-scale verification projects. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 171–182, Piscataway, NJ, USA, 2017. IEEE Press.

[27] 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.

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

[29] Adam Chlipala. Library equality, 2017.

[30] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In Maria Paola Bonacina, editor, *24th International Conference on Automated Deduction*, CADE-24, pages 392–406, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[31] Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory, 2017.

[32] Pierre-Évariste Dagand. The essence of ornaments. *J. Funct. Program.*, 27:e9, 2017.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] David Delahaye. A tactic language for the system coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, pages 85–95, 2000.

[35] 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.

[36] 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.

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

[38] 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.

[39] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 599–612, New York, NY, USA, 2017. ACM.

[40] 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.

[41] 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.

[42] 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.

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

[44] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996.

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

[46] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in isabelle/hol. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs: Third International Conference*, CPP 2013, pages 131–146, Cham, 2013. Springer International Publishing.

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

[48] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. *CoRR*, abs/1405.3426, 2014.

[49] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics: 17th International Conference*, TPHOLs 2004, pages 152–167, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[50] Roope Kaivola and Katherine Kohatsu. Proof engineering in the large: formal verification of pentium®4 floating-point divider. *International Journal on Software Tools for Technology Transfer*, 4(3):323–334, 2003.

[51] Gerwin Klein. Proof engineering considered essential. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *Formal Methods: 19th International Symposium*, FM 2014, pages 16–21, Cham, 2014. Springer International Publishing.

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

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

[54] 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.

[55] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

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

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

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

[59] Pierre Letouzey. Extraction in coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

[60] 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.

[61] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs: International Workshop*, TYPES 2000, pages 181–196, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[62] Gregory Malecha and Jesper Bengtson. Extensible and efficient automation through reflective tactics. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 532–559, 2016.

[63] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[64] 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.

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

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

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

[68] 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.

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

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

[71] Emerson Murphy-Hill and Dan Grossman. How programming languages will co-evolve with software engineering: A bright decade ahead. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 145–154, New York, NY, USA, 2014. ACM.

[72] Emerson Murphy-Hill and Dan Grossman. How programming languages will co-evolve with software engineering: A bright decade ahead. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 145–154, New York, NY, USA, 2014. ACM.

[73] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP*, 2003.

[74] Karl Palmskog. Commit to verdi-raft: Port to coq 8.6, 2017.

[75] L. C. Paulson and J. C. 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.

[76] 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.

[77] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. Investigating safety of a radiotherapy machine using system models with pluggable checkers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification: 28th International Conference*, CAV 2016, pages 23–41, Cham, 2016. Springer International Publishing.

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

[79] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, 2016. ACM.

[80] Olivier Pons. Generalization in type theory based proof assistants. TYPES '00, pages 217–232, 12 2000.

[81] Kenneth Roe and Scott Smith. CoqPIE: An IDE aimed at improving proof development productivity. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving: 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, pages 491–499, Cham, 2016. Springer International Publishing.

[82] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[83] 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.

[84] Matthieu Sozeau. Towards a better-behaved unification algorithm for coq beta ziliani, 2014.

[85] Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. Injective type families for haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 118–128, New York, NY, USA, 2015. ACM.

[86] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for Free! working paper or preprint, July 2017.

[87] Iain Whiteside, David Aspinall, Lucas Dixon, and Gudmund Grov. Towards formal proof script refactoring. In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Intelligent Computer Mathematics: 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings*, pages 260–275, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[88] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI 2015, Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, pages 357–368, Portland, OR, USA, June 15–17, 2015.

[89] Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. Ornaments in practice. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, pages 15–24, New York, NY, USA, 2014. ACM.

[90] 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.

[91] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. *J. Funct. Program.*, 25, 2015.

[92] Beta Ziliani and Matthieu Sozeau. A unification algorithm for coq featuring universe polymorphism and overloading. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 179–191, 2015.

[93] Théo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the coq proof assistant. *CoRR*, abs/1505.05028, 2015.