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:

TODO, Chair

TODO

TODO

Program Authorized to Offer Degree:

Computer Science & Engineering

ABSTRACT

PROOF ENGINEERING TOOLS
FOR A NEW ERA

Talia Ringer

Chairs of the Supervisory Committee:
TODO
Computer Science & Engineering

Abstract will go here.

To my parents. Hope this fits on your fridge.

CONTENTS

ACKNOWLEDGMENTS

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

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, Esther, Ellie, Mer, students, Qi, Saba.

1

INTRODUCTION

Motivation for verifying systems

Era of scale—enter proof engineering [8]

Looking back (Social Processes [3]), development has come a long way, but maintenance is still hard! And this is a problem in practice!

But missed opportunity: automation doesn't understand that proofs evolve

So we build automation that does, and we call this proof repair. Proof repair 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.

Or, in other words (thesis statement): 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.

Key technical bit: differencing and program transformations, taking advantage of the rich and structured language proofs are written in.

We implement this in a tool suite for Coq, get some sweet results.

Pave path to the next era of verification

READING GUIDE

How to read this thesis

Mapping of papers to chapters

Authorship statements for included paper materials, to credit coauthors

Expected reader background & where to find more info

2

MOTIVATING PROOF REPAIR

Before we talk more about proof repair, it helps to know what it's like to develop and maintain proofs to begin with, and what happens under the hood when you do that. This chapter gives you that context, then explains the high-level approach to proof repair that builds on that.

2.1 PROOF DEVELOPMENT

Cartoon version of development: program, spec, proof

Proof assistants: short overview of foundations & different options (survey paper), then say focus on Coq

Slightly less brief overview of Coq and its foundations and automation and so on (including proof terms), going through a running example of proof development in Coq

2.2 PROOF MAINTENANCE

Problem is when something changes—change something in running example

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

But still, even with these, the reality: This happens all the time (REPLICA)

And in fact not just after developing a proof, but during development too (REPLICA)

And breaks proofs even for experts (REPLICA)

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

Hence proof repair—smarter automation

2.3 PROOF REPAIR

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 our tool suite that work this way, how they handle these challenges, and how they save work.

3

PROOF REPAIR BY EXAMPLE

The first tool (PUMPKIN PATCH) focuses on changes in programs and specifications, though these changes are limited in scope as we'll see later.

What this tool does is, when programs and specifications change and this breaks a lot of proofs, it lets the proof engineer fix just one of those proofs. It then generalizes the example patch into something that can fix other proofs broken by the same change.

So in other words, the information from those changes is carried in the difference between the old and new version of the example patched proof. PUMPKIN PATCH generalizes that information.

Application can be automated in some cases at the end, or it can be manual.

The work saved is shown retroactively on case studies replaying changes from large proof devleopments in Git. Results for this tool are preliminary compared to what we'll see later, since this was the first prototype.

3.1 MOTIVATING EXAMPLE

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.

With traditional proof automation, the burden of change largely falls on proof engineers. 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.

WITHOUT PROOF REPAIR 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.

```
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 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.

Consider a commit from the Coq 8.7 release [7]. This commit redefined injection from integers to reals (Figure 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 from Coq 8.6 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.

WITH PROOF REPAIR When a user 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 user never has to consider how the definition has changed.

3.2 APPROACH

In the example from Section 3.1, we can see how the example change in one proof carries enough information to fix other proofs broken by the same change (namely the rewrite by `INR\_IPR`). So a tool can extract that, generalize it, and use it to fix other proofs broken by the same change.

The key insight behind PUMPKIN's approach is that this is true more generally. To use PUMPKIN, the programmer 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*—which is localized to the context of the example, but not enough to fix other proofs broken by the change. It 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 and generalize that information. Application works with hint databases or is manual. Here is the system diagram for PUMPKIN. The PUMPKIN repository contains a detailed user guide.

As mentioned earlier, PUMPKIN does this using a combination of semantic differencing and program transformations. Differencing looks at the difference between versions of the example patched proof for this information, and finds the candidate. Then, program transformations modify that candidate to produce the reusable proof patch.

And of course all of this happens over proof terms, since tactics might hide necessary in information. Of course this is hard to see on the example from Section 3.1, since we were lucky enough to see the difference in tactics here. Let's look at a toy example for which that isn't true.

To motivate this workflow, consider using PUMPKIN to search the proofs in Figure 2 for a patch between conclusions. Except we will show a place where the lemma is actually applied. Note that the tactics don't change even though the terms do—and even though the change could break other proofs.

So what do we do? 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,

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

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

Figure 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.

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`).

This is where the transformations come in. There are four:

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

Here, 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 can happen automatically through hint databases.

This simple example uses only two transformations. The other transformations help turn candidates into patches in similar ways. We discuss all of this in detail later.

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

For example, to find a patch for Figure 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

Later we will see real-world examples that demonstrate more configurations.

3.3 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 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).[1] 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) ...
```

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

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.

Plus parts of Inside the Core, Testing Boundaries, Future Work

How differencing works in detail

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

3.4 TRANSFORMATION

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 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 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.

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 [1], 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.✓
Qed.
```

to 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 new.✓
Qed.
```

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 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.

plus parts of PUMPKIN PATCH Inside the Core, Testing Boundaries, Future Work

How the four transformations work in detail

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

3.5 IMPLEMENTATION

parts of PUMPKIN PATCH Inside the Core, plus more

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.[2] The interested

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

reader can follow along in the repository. Our prototype has no impact on the trusted computing base (Section 3.5.2.1).

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.[3] 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 3) 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.

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

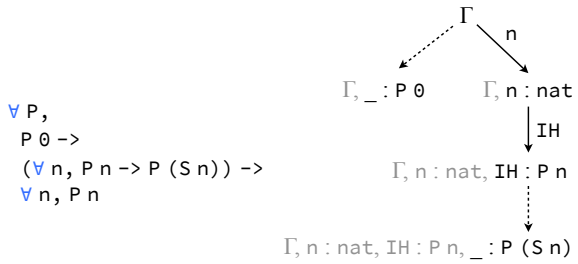

Figure 3: The type of (left) and tree for (right) the induction principle `nat_ind`. The solid edges represent hypotheses, and the dotted edges represent the proof obligations for each case in an inductive proof.

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

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 [2] 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 **??**), 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 3). 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 **??**, 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 **??**), 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.

3.5.2.1 *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 [4]. 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 [6]. Furthermore, Coq type-checks terms that plugins produce. Since PUMPKIN does not modify the type checker, it cannot produce an ill-typed term.

3.6 RESULTS

PUMPKIN PATCH Case Studies, key technical results

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), segue into next chapter

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

PUMPKIN Pi motivating example

4.2 APPROACH

Parts of PUMPKIN Pi intro, problem definition, plus more

Like I mentioned earlier, this also works using differencing and program transformations. And of course all of this happens over proof terms.

Here's the system diagram.

Here, differencing thus looks at the difference between versions of the changed datatype, and finds something called a type equivalence. I'll explain that with examples. Sometimes differencing is automatic, and sometimes it's manual.

Then, program transformation ports proofs across the equivalence directly. So they take care of application.

4.3 DIFFERENCING

DEVOID 3.1 and 4.1, with some more general things from PUMPKIN Pi and more.

How differencing works in detail

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

4.4 TRANSFORMATION

Parts of PUMPKIN Pi Transformation, with DEVOID 3.2 and 4.2 as examples, plus some of the beautiful Carlo theory to explain why we go from equivalences to configurations and what that really means

How the transformation works in detail

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

4.5 IMPLEMENTATION

Parts of PUMPKIN Pi and DEVOID implementation, plus more

4.5.1 *Tool Details*

4.5.2 *Workflow Integration*

PUMPKIN Pi Decompiler and Implementation

4.6 RESULTS

PUMPKIN Pi Case Studies, key technical results

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

5.1 PROGRAMS

Program Refactoring

Program Repair

Ornaments

Programming by Example

Differencing & Incremental Computation

5.2 PROOFS

Proof Reuse

Proof Refactoring

Proof Repair

Proof Design

Proof Automation

Transport

Parametricity

Refinement

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?

BIBLIOGRAPHY

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

[2] Adam Chlipala. Library equality, 2017.

[3] 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.

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

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

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

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

[8] 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.