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?

Dan, Jeff Foster, Zach, Derek, Alexandra, 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), coauthors, Valentin, Carlo, my family, PLSE lab (especially Chandra oh my gosh), James Wilcox, Jasper Hugunin, Marisa Kirisame, UCSD Programming Systems group, Misha, PL Twitter, students, Qi.

1

INTRODUCTION

2

PROOF DEVELOPMENT

Context, example from talk. (Maybe extra examples.)

3

PROOF MAINTENANCE

Context, user study. Examples. Related work.

3.1 INTRODUCTION

The days of verifying only toy programs in interactive theorem provers (ITPs) are long gone. The last two decades have marked a new era of verification at scale, reaching large and critical systems like operating system kernels, machine learning systems, compilers, web browser kernels, and file systems—an era of *proof engineering* [**?**].

In order to build useful proof engineering tools, it is crucial to understand the development processes of proof engineers. Data on the changes to programs, specifications, and proofs that proof engineers make, for example, can guide tools for proof maintenance to support features that matter.

Existing studies of proof development analyze coarse-grained data like version control history, archives, project logs, or artifacts, or build on personal experiences [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. These methods, while valuable, lack access to information on many of the processes that lead to the final artifact. For example, proof engineers may not plan for failing proofs or debugging statements to reach the final artifact, and may not even commit these to version control. In addition, data from version control may include many changes at once, which can be difficult to separate into smaller changes [**?**]. Personal experiences may not be enough to supplement this, as proof engineers may forget or omit details like *how* they discovered bugs in specifications.

This paper shows how to instrument the ITP Coq [**?**] to remotely collect fine-grained data on proof development processes. We have built a Coq plugin (Section 3.2) called REPLica (REPL Instrumentation for Coq Analysis) that listens to the Read Eval Print Loop (REPL)—a simple loop that all user interaction with Coq passes through—to collect data that the proof engineer sends to Coq during development. This includes data difficult to obtain from other sources, like failed proof attempts and incremental changes to definitions. REPLica collects this data regardless of the proof engineer's user interface (UI), and it does so in a way that generalizes to other REPL-based ITPs like HOL [**?**, **?**] and Idris [**?**].

We have used REPLica to collect a month's worth of data on the proof developments of 8 intermediate to expert Coq users (Section 3.3). The collected data is granular enough to allow us to reconstruct hundreds of interactive sessions. It is publicly available with the proof engineers' consent.[1]

We have visualized and analyzed this data to classify hundreds of fixes to broken proofs (Section 3.4) and changes to programs and specifications (Section 3.5). Our analysis suggests that our users most often fixed proofs by stepping up above those proofs in the UI and fixing something else, like a specification. It reveals four patterns of changes to programs and specifications among those users particularly amenable to automation: incremental development of inductive types, repetitive refactoring of identifiers, repetitive repair of specifications, and interactive discovery of programs and specifications. Both the changes we have classified and our findings about them suggest natural ways to improve existing proof engineering tools like machine learning tools for proofs [**?**, **?**, **?**, **?**, **?**, **?**], proof refactoring tools [**?**, **?**, **?**, **?**, **?**, **?**, **?**], and proof repair tools [**?**, **?**].

Our experiences collecting and analyzing this data have helped us to identify three wishes (Section 3.6), the fulfillment of which may facilitate future studies of proof development: better abstraction of user environments, more information about user interaction, and more users. We discuss important design considerations for both study designers and ITP designers who hope to grant each wish.

In summary, we contribute the following:

1. We build REPLica, a tool that instruments Coq to collect fine-grained proof development data.
2. We use REPLica to collect and publish a month's worth of data on 8 proof engineers' developments in Coq.
3. We visualize and analyze the data to classify hundreds of fixes to broken proofs and changes to programs and specifications.
4. We discuss the patterns behind these changes and how they suggest improvements to proof engineering tools.
5. We discuss design considerations both for the study designer and for the ITP designer that can facilitate future studies of proof development.

3.2 BUILDING REPLICA

REPLica is a tool that collects fine-grained proof development data. It is available on Github for Coq 8.10, with backported branches for Coq 8.8 and Coq 8.9.[2]

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

2 http://github.com/uwplse/coq-change-analytics

REPLica works by instrumenting Coq to listen to user interaction. Coq's interaction model (Section 3.2.1) implements a REPL, or Read Eval Print Loop: a loop that continually reads in messages from the user, evaluates the statements those messages contain, and prints responses to the user. All UIs for Coq, from the command line tool `coqtop` [**?**] to the IDEs CoqIDE [**?**] and Proof General [**?**], communicate with Coq through the REPL. REPLica instruments the REPL to collect fine-grained data while remaining decoupled from the UI.

Figure 1 summarizes the design of REPLica. REPLica is made of two parts: a client that the user installs, and a web server that stores the data from multiple users. The client instruments the REPL to record and log the data that the user's UI sends to Coq (Section 3.2.2), then sends this data to the server (Section 3.2.3), which stores it for analysis.

The implementation effort for REPLica was modest, with just 412 LOC of OCaml for the client and 178 LOC of Python for the server.[3] This modest effort was enough for us to collect (Section 3.3) and analyze (Sections 3.4 and 3.5) data from hundreds of interactive sessions.

3.2.1 *User-Coq Interaction*

The REPLica client listens for messages that the user's UI sends to Coq's REPL. To provide a common API for different UIs, the implementation of Coq's REPL is organized into an interactive state machine. Each state can include tactics in Coq's tactic language Ltac or commands in Coq's vernacular; both of these can reference terms in Coq's specification language Gallina. When a UI sends a message to Coq's REPL, the message contains a state machine statement (a labeled transition) that instructs Coq to do one of three things:

1. *Add*: produce a new state
2. *Exec*: execute an existing state to receive a response
3. *Cancel*: back up to an existing state

Coq reacts accordingly, then returns an ID for the new state that corresponds to the statement.

Taken together, these three statement types can be used to reconstruct the proof engineer's behavior, such as defining and redefining types, interactively constructing proofs, and stepping up when the user hits a dead-end.

FROM USER BEHAVIOR TO THE STATE MACHINE When the user runs a command or tactic in Coq, the UI first adds a new state corresponding to the command or tactic. Adding a state to the state

3 Counted using David M. Wheeler's SLOCCount.

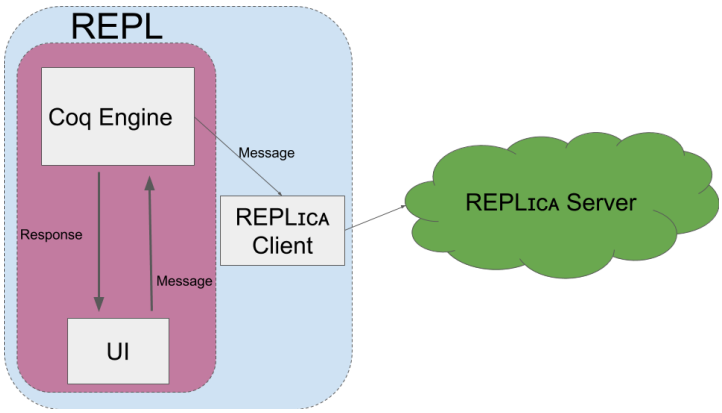

Figure 1: REPLica design. The REPLica client listens to the REPL and sends data to the REPLica server.

machine does not actually execute the command or tactic; to do that, the UI must execute the new state. In other words, in state machine terms, each addition follows a transition, but only returns the state number. To get the state information, the UI must call *Exec*.

Using this mechanism, the UI can group state machine statements and execute them all at once. For example, stepping down past 3 definitions at once in an IDE manifests as 3 additions followed by a constant number of executions, and successfully compiling a file manifests as many additions followed by a constant number of executions.[4]

When the user steps up in the UI, or attempts to run a tactic or command that fails, the UI backs up to an existing state in the state machine. Sending the state machine a cancellation statement is not the only way to back up to an existing state; cancellations can also take the form of state machine *additions* of `Cancel` or `BackTo` commands in Coq's vernacular. In either form, cancellations take a state ID as an argument to specify the state to back up to.

AN EXAMPLE: DEFINING AND EXTENDING AN INDUCTIVE TYPE To see how user behavior corresponds to state machine statements, consider an example from User 1, Session 41, simplified for readability. In this session, User 1 stepped past the definition of an inductive type `Alpha`. Later, User 1 stepped above `Alpha`, imported list notations, then stepped down to add a new constructor to `Alpha` using those notations.

When User 1 first stepped down below `Alpha`, Coq received a state machine addition (denoted `Add`, following SerAPI [**?**]) of this definition (again simplified for readability):

```
(Add () "Inductive Alpha : ... := ...")
```

4 Some UIs send Coq multiple executions for each group of additions.

at the state ID 1, producing state ID 2. Since User 1 stepped down a single step, Coq also received an execution (denoted `Exec`) of the new state:

```
(Exec 2)
```

which did not produce a state with a new state ID, since only additions produce states, and only states have state IDs. When User 1 stepped up to above the definition of `Alpha`, Coq received a cancellation (denoted `Cancel`):

```
(Cancel 1)
```

taking the state machine to before the definition of `Alpha`. When User 1 added the import, Coq received an addition at state ID 1, producing state ID 3, followed by an execution:

```
(Add () "Import Coq.Lists.List.ListNotations. ")
(Exec 3)
```

Finally, when User 1 added the new constructor to `Alpha` and stepped down past it, Coq received an addition of `Alpha` with the new constructor at state ID 3 producing state ID 4, followed by an execution:

```
(Add () "Inductive Alpha : ... := ...
         | alpha_rec_mt : ...")
(Exec 4)
```

The data that REPLica received provided us with enough information to visualize these changes as a sequence of diffs for later analysis (Section 3.5). As a consequence, we were able to find this pattern of development of incrementally extending inductive types for four of our users (Section 3.5.2.1).

3.2.2 *User-Client Interaction*

The REPLica client records all of the additions, executions, and cancellations that the proof engineer's UI or compiler sends. To use it, the proof engineer installs the client, then adds a line to the Coq requirements file [**?**] so that Coq always loads it. On the first build, REPLica asks the proof engineer for consent and for demographic information for the study. From then on, the proof engineer uses Coq normally.

The implementation of the client is a Coq plugin. Coq's plugin system makes it possible to extend Coq without compromising trust in the Coq core. REPLica does not include new commands or tactics; it simply attaches to Coq's REPL and listens for messages that the proof engineer sends to Coq.

The hooks in the plugin API that allow plugins to listen to the REPL were not initially present in Coq; we worked with one of the Coq developers to add this to Coq 8.10.[5] We also developed backported

5 http://github.com/coq/coq/pull/8768

User	Years	Expertise	Purpose	Frequency	UI	Version
0	2-4	○ ○ ○ ○ ○	Verification	Daily	Proof General	Master
1	2-4	○ ○ ○	Mathematics	Monthly	Proof General	8.10
2	¿ 4	○ ○ ○ ○	Verification	Daily	Proof General	8.10
3	¿ 4	○ ○ ○ ○ ○	Verification	Weekly	Proof General	Master
4	¿ 4	○ ○ ○ ○	Verification	Daily	coqtop	Master
5	2-4	○ ○ ○	Verification	Monthly	Custom	8.10
6	2-4	○ ○ ○ ○	Mathematics	Daily	Proof General	Master
7	2-4	○ ○ ○ ○	Mathematics	Weekly	Proof General	8.10
8	¿ 4	○ ○ ○ ○ ○	Verification	Daily	Proof General	8.8
9	¿ 4	○ ○ ○ ○ ○	Verification	Monthly	Proof General	8.9
10	¿ 4	○ ○ ○	Verification	Daily	Proof General	8.8
11	¿ 4	○ ○ ○	Mathematics	Daily	Proof General	8.9

Figure 2: User profiles of Coq development background: experience in years, self-assessed expertise (beginner, novice, intermediate, knowledgeable, or expert, represented by circles), purpose of use, frequency of use, UI, and current version.

User	Total	Interactive	Proof
0	16	0	0
1	42	10	4
2	7	3	0
3	1405	101	8
4	3	0	0
5	45	15	16
6	1	0	0
7	339	183	241
8	162	27	10
9	5	0	0
10	33	15	0
11	17	8	0

Figure 3: Number of total sessions, interactive sessions, and interactive proof subsessions per user.

branches of Coq 8.8 and 8.9 that include this API, so that users whose projects depended on those versions of Coq could participate in our study. This API is now available in all versions of Coq going forward.

3.2.3 *Client-Server Interaction*

To record a history of user behavior, the REPLica client sends a record of each state machine addition, execution, and cancellation to the REPLica server. The server then collects this data into a format suitable for analysis.

The message that the client sends to the server includes both the state machine statement and additional metadata.

METADATA FOR TWO CHALLENGES The metadata beyond the state ID and the state machine statement exists to handle two challenges: The first challenge is that state machine interactions alone are not enough to reconstruct a per-user history, nor to group the user's interaction into discrete sessions for a particular project or module. The second challenge is that the network can be slow or unreliable at times, causing messages to be received by the server out of order, and slowing down the UI if not handled properly.

To handle the first challenge, REPLica labels each message with a module name, user ID, and session ID. The module name comes from

the Coq plugin API. The user ID is generated by the server and stored on the user's machine. The session ID, in contrast, is generated by the client: When the user loads the client, upon opening any new file, the client records the start time of the session. This start time is then used to identify the session.

To handle the second challenge, REPLICA labels each message with two pieces of metadata: the time at which the state machine message was sent to Coq (to order messages) and the state ID (to detect missing states). TCP alone is not enough to address these issues since each invocation of the plugin creates a separate network stream (so the server may receive messages out of order), and since the user can disable the plugin (so the client may never send some messages).[6] In addition to this metadata, REPLICA sends messages in batches. If the network is not available, REPLICA logs messages locally, then sends those logs to the server the next time the network is available.

3.3 DEPLOYING REPLICA

We set out to answer the following questions:

- **Q1**: What kinds of mistakes do users make in interactive proofs, and how do they fix them? (Section 3.4)
- **Q2**: What kinds of changes to programs and specifications do users make often, and do those changes reveal patterns amenable to automation? (Section 3.5)

To answer these questions, we recruited 12 proof engineers to install REPLICA and use Coq normally for a month (Section 3.3.1). We received a month's worth of data containing granular detail on Coq development for 8 of 12 of those users (Section 3.3.2). After a month, we shut down the server, closed the study, and visualized and analyzed the data (Section 3.3.3).

3.3.1 *Recruiting*

We recruited proof engineers by distributing a promotional video and study description. All potential users went through a screening process, ensuring that they are at least 18 years old, fluent in English, and have at least a year of experience using Coq. Upon installation of the plugin, users filled out a consent form. Users then filled out a questionnaire about Coq background and usage, the results of which are in Figure 2.

All users reported more than 2 years of experience; 7 reported more than 4. Self-assessed expertise was evenly distributed between

6 The consent form allowed users to temporarily disable the plugin if necessary, as long as they informed us of this.

intermediate, knowledgeable, and expert; no beginners or novices participated. 4 users reported that they use Coq for writing mathematical proofs, while the other 8 reported that they use Coq for verifying software. 7 users said that they use Coq every day, 2 a few times per week, and 3 a few times per month. 10 users reported using Proof General, while 1 user reported using `coqtop`, and 1 user reported using a custom UI. 4 users installed the plugin with the master branch of Coq, while 4 users used Coq 8.10, 2 users used Coq 8.9, and 2 users used Coq 8.8.

3.3.2 *Collection*

The study period lasted one month, during which users agreed to develop Coq code with the plugin enabled whenever possible, or otherwise let us know why this was not possible. All of the data collected within this time period has been made publicly available with the consent of the users.[7]

Figure 3 shows the number of sessions by user. Every time the plugin was loaded, either inside of a compiled file or inside of a file open in a UI, this began a new session. For both Q1 and Q2, our analyses looked for changes only within sessions that involved some combination of failure and stepping up and then back down in a UI; we call these *interactive sessions*.

We marked a session as interactive if it contained at least one cancellation followed by other changes in state. This did not capture any compilation passes because Coq disallows cancellations outside of interactive mode. For example, User 3 logged 11495 total sessions, only 101 of which were interactive. The remainder of User 3's sessions were, for the most part, compilation passes over large sets of dependencies (one session per dependency). In total, across all users, there were 362 interactive sessions.

Within these interactive sessions, the analysis for Q1 looked for fixes to failing tactics only inside of spans of time spent *inside proofs* that involved some combination of failure and stepping up and then back down in a UI; we call these *interactive proof subsessions*. There could be zero, one, or multiple of these within an interactive session. We detected these similarly to how we detected interactive sessions. There were 279 interactive proof subsessions.

REPLica logged interactive sessions for only 8 of the 12 users. Section 3.6 discusses possible causes of, implications of, and remedies for this.

7 `http://github.com/uwplse/analytics-data`

3.3.3 *Analysis*

Once we had collected this data, we analyzed it to answer Q1 and Q2. To answer these questions, we developed a small Python codebase to analyze the data. The scripts used information about cancellations to build visualizations to aid in analysis. For Q1, we built this cancellation information into a search tree for each proof, and produced visualizations of each tree (see Figure 5). For Q2, we used this cancellation information to reconstruct a sequence of diffs, and used Git tooling to manually inspect these diffs (see Figure 7).

The scripts, visualizations, and results for Q1 and Q2 can be found alongside the data in the public data repository.

3.4 Q1: MISTAKES IN AND FIXES TO PROOFS

Q1 asked what mistakes proof engineers make in proofs, and how they fix those mistakes. This data may inform the development of tools that help users write proofs by suggesting next steps and changes to existing tactics.

A1 We found the following:

1. Most interactive proof subsessions ended in the user stepping up to an earlier definition. This shows that writing definitions and proofs about those definitions was usually a feedback loop. (Section 3.4.1)
2. Within proofs, application of lemmas was the main driver of proving. (Section 3.4.2)
3. Within proofs, over half of fixes to tactic mistakes fixed either an improper argument or wrong sequencing structure. (Section 3.4.3)

METHODOLOGY Our analysis for Q1 looked within the 279 interactive proof subsessions. We first counted the number of times each tactic was used, the number of times it was stepped above, and the number of times its invocation caused an error. Then, we constructed a search tree using the cancellation structure, and for any state which had multiple out edges (a proof state where multiple tactics were tried), we compared the cancelled attempts to the final tactic used.

3.4.1 *Fixing Proofs by Fixing Definitions*

Of the 279 interactive proof subsessions, 209 (over 75%) began a proof, only to step above the entire proof attempt and return to make changes to earlier definitions or commands, for example to change a

Tactic	Used	Failed	Stepped Above
apply	156	27	13
intros	108	16	5
destruct	94	24	12
rewrite	61	13	12
exists	47	9	1
unfold	45	2	8
simpl	32	6	4
reflexivity	27	1	1
simpl in	25	3	7
assumption	24	2	1
specialize	19	0	1
subst	14	1	0
split	12	0	0
eapply	11	0	5
dependent	10	0	8
inversion	10	0	0
constructor	10	3	0
eauto	9	0	4
assert	9	2	0
induction	8	0	1
validate	8	0	5
monoid	7	0	2
tauto	7	1	0
eassumption	7	0	5
repeat constructor	7	0	5

Figure 4: The top 25 most common tactics.

specification to make the proof possible. This suggests that tools that attempt to provide next steps to users writing proof may also benefit from suggesting possible fixes to definitions used in the proof.

> **Takeaway:** It may be beneficial to combine machine learning tools that automatically complete or suggest hints for proofs [**?**, **?**, **?**, **?**, **?**] with tools for repairing definitions [**?**, **?**].

We considered only the remaining 70 successful interactive proof subsessions (made up of 1085 tactic invocations) for the sake of analyzing behavior *within* proofs that ended in a successful proof. The analysis for Q2 (Section 3.5) reveals more about how specifications and programs changed.

3.4.2 *Tactics Used and Cancelled*

Figure 4 shows the counts for each tactic (regardless of arguments) in the 70 successful interactive proof subsessions. The distribution of tactics run was top-heavy; over 50% of the tactics invoked were either `apply`, `intros`, `destruct`, `rewrite`, `exists`, `unfold`, or `simpl`. The `apply` tactic had a significant lead over other tactics in invocations,

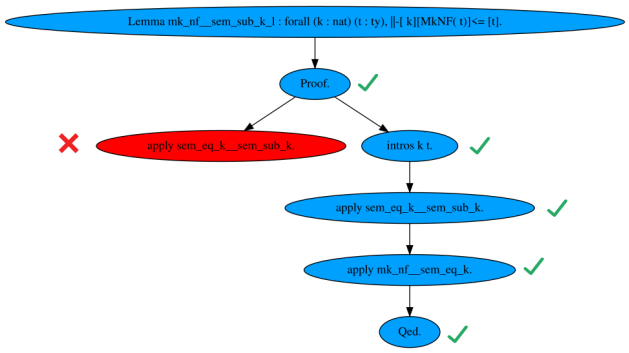

Figure 5: An example search tree, generated from the collected data by REPLica. It shows the user attempting to apply a lemma, which fails until they first run the `intros` tactic.

but invocations of `destruct` ended in failure or stepping above them nearly as many times as invocations of `apply` did.

This data indicates two takeaways for proof tooling: Firstly,

> **Takeaway:** Tools may be able to focus on understanding the behavior of and suggesting just a small number of tactics, and still benefit.

And second, since lemma and hypothesis application was the main driver of proofs,

> **Takeaway:** Assessing which lemmas and hypotheses are useful would be one of the main tasks of a tool which suggests tactics to the user.

The machine learning tool ML4PG [**?**] for Coq already offers promising developments in this direction by understanding and providing hints about similar lemmas, as does the proof automation tool CoqHammer [**?**]; similar functionality may help improve the performance of tools that suggest tactics.

3.4.3 *Fixing Proofs by Fixing Tactics*

The raw cancellation numbers do not give a broader context to each cancellation, namely what tactic it was cancelled in favor of. To address this, we built a search graph and analyzed the tactic attempts at each branching node (see Figure 5). Where there were more than 2 attempts, we compared all non-final attempts to the final one separately.

In our 71 successful interactive proof subsessions, 96 tactics were cancelled in favor of another tactic at the same state. Of the 96 cancelled-tactic and final-tactic pairs:

- 13 were semicolon clauses added after a tactic, like:

  `destruct w.` → `destruct w; reflexivity.`

- 4 were semicolon clauses removed from the end of a tactic, like:

```
intros; reflexivity. → intros.
```

- 31 were the same tactic with modified arguments, like:

```
intros k X t. → intros k X t Hfresh.
```

- 5 were similar, but a `Search` or `Check` command was first run before replacing the tactic, like:

```
apply IdSetFacts.remove_3. →
Check IdSetFacts.remove_3.
apply IdSetFacts.remove_3 with Y.
```

- 43 changes did not fall into this categorization.

The proofs shown were complex, and users often made nontrivial changes to attempted tactics, so not all changes could be easily categorized or analyzed. However, over half of the changes present could be categorized into simple changes, and potentially synthesized by automated tools.

This data also shows us that for a large proportion of tactics, users could correctly pick the tactic to invoke, even when they made mistakes in its arguments. In addition, fixing these arguments took both on average and in the worst case longer than fixing other kinds of mistakes. Accordingly,

> **Takeaway:** Automated tooling that suggests actions to take based on tactics that were recently stepped above or failed may focus on predicting new arguments for the attempted tactic.

3.5 Q2: CHANGES TO TERMS

Q2 asked what kinds of changes proof engineers make to programs and specifications, and whether those changes reveal patterns amenable to automation. This information may be useful to ensure tools for proof evolution support the features that help proof engineers.

A2 We found that while no single change was dominant across all users, users made related changes within and across sessions. Analysis of these changes revealed four patterns:

1. Incremental development of inductive types
2. Repetitive refactoring of identifiers
3. Repetitive repair of specifications
4. Interactive discovery of programs and specifications

METHODOLOGY To answer this question, we wrote a script to visualize changes over time as diffs on Github (see Figure 7). The script reconstructed the state of the file up to each cancellation within a session, then committed that to the public data repository. When possible (see Section 3.6.2), it augmented each commit with information on whether the cancellation was a failure or the user stepping up in an IDE.

We then manually analyzed the diffs that this visualization produced to build a classification of changes (Section 3.5.1), then classify changes that we found (Section 3.5.2). We did this for each of the 362 interactive sessions and, when relevant, across sessions as well. Finally, we looked at clusters of common changes for patterns. For each of the four patterns we found (Sections 3.5.2.1, 3.5.2.2, 3.5.2.3, and 3.5.2.4), we identified benchmarks (examples of the pattern in our data) and lessons for automation.

3.5.1 *Building a Classification*

After running the visualization script, we did a manual analysis of the diffs in order to build a classification of changes to Gallina terms. This analysis was thorough in that it involved inspecting each consecutive diff and, when relevant, diffs that spanned several commits (when a user stepped up, changed something, and then later stepped back down) or sessions (when a user modified the same file during two different sessions). However, it did not necessarily capture all changes to terms; Section 3.6.2 discusses some of the challenges.

CLASSIFICATION We designed a classification that groups changes along three dimensions:

1. **Command**: vernacular command used to define the term in which a subterm changed
2. **Operation**: how the subterm changed
3. **Location**: innermost subterm that changed

with the following categories for **Operation**:

1. **Structure**:
   a) **Add** or **Del**: add or delete information
   b) **Mov**: move information
2. **Content**:
   a) **Pch** or **Uch**: patch or unpatch
   b) **Cut** or **Uut**: cut or uncut
   c) **Rpl**: replace

3. **Syntax**:
    a) **Rnm**: rename
    b) **Qfy** or **Ufy**: qualify or unqualify

Changes listed together are inverse operations. The **Structure** changes are straightforward. Among the changes to **Content**, **Pch** is applying a function to the old term to get a new term, **Cut** is defining a new term or let-binding and then referring to that term inside of an existing term, and **Rpl** is replacing contents in any other way. Among the **Syntax** changes, **Qfy** is qualifying a constant after changing an import.

For **Structure** changes, there are five **Location**s:

1. **Hyp**: hypothesis of anything that can take arguments
2. **Arg**: argument in an application
3. **Ctr**: constructor of an inductive type
4. **Cas**: case of a match statement
5. **Bod**: body of anything that can take arguments

For **Content** changes, there are an additional two:

6. **Fun**: function in an application
7. **Typ**: type annotation

For **Syntax** changes, there are only three:

1. **Bnd**: binding in the local environment
2. **Idn**: identifier in the global environment
3. **Con**: constant

DESIGN CONSIDERATIONS That classification that we designed considers changes only within Gallina terms defined or stated using vernacular commands. Since it is focused solely on changes to defined terms, it does not consider other information like changes to vernacular commands, hints, tactics, notations, scope annotations, inference information, or imports, and it considers additions of new terms only in **Cut** changes.

In building this classification, we aimed to group changes at a level of granularity narrow enough to inform the design of proof engineering tools, but broad enough to capture patterns. We suspect that within these categories, there are more granular categories that can be useful for automation, like distinguishing among hypotheses to set apart indices of inductive types, or classifying a **Content** change as semantics-preserving. We did not design a more granular classification because we did not find it useful for describing our data.

User	Top Changes				# Changes	# Interactive	Expertise
1	**Add Ctr** (23)	**Add Cas** (22)	**Qfy Con** (6)		69	10	○ ○ ○
2	**Add Cas** (4)				10	3	○ ○ ○ ○
3	**Pch Arg** (13)	**Mov Arg** (8)	**Add Bod** (7)	**Cut Arg** (7)	55	101	○ ○ ○ ○ ○
5	**Add Cas** (20)	**Add Ctr** (13)	**Add Hyp** (12)		75	15	○ ○ ○
7	**Rnm Idn** (42)	**Mov Hyp** (18)	**Add Hyp** (18)		151	183	○ ○ ○ ○
8	**Rpl Fun** (29)	**Del Hyp** (4)	**Uch Arg** (4)		44	27	○ ○ ○ ○ ○
10	**Pch Arg** (4)	**Rpl Cas** (3)			15	15	○ ○ ○
11	**Pch Cas** (3)				7	8	○ ○ ○

Figure 6: Top changes, by user.

```
Inductive Term : Set :=
    | Var : Identifier -> Term
-   | Int : Z -> Term
+   | Bool : bool -> Term
    | Eq : Term -> Term -> Term
+   | And : Term -> Term -> Term
+   | Or : Term -> Term -> Term
+   | Not : Term -> Term
+   | If : Term -> Term -> Term -> Term
+   | Int : Z -> Term
    | Plus : Term -> Term -> Term
    | Times : Term -> Term -> Term
    | Minus : Term -> Term -> Term
    | Choose : Identifier -> Term -> Term.
```

```
Fixpoint identity (t : Term) : Term :=
  match t with
    | Var x => Var x
-   | Int i => Int i
+   | Bool b => Bool b
    | Eq a b => Eq (identity a) (identity b)
+   | And a b => And (identity a) (identity b)
+   | Or a b => Or (identity a) (identity b)
+   | Not a => Not (identity a)
+   | If a b c => If (identity a) (identity b) (identity c)
+   | Int i => Int i
    | Plus a b => Plus (identity a) (identity b)
    | Times a b => Times (identity a) (identity b)
    | Minus a b => Minus (identity a) (identity b)
    | Choose x P => Choose x (identity P)
  end.
```

Figure 7: A change to an inductive type (left) and corresponding change to a fixpoint (right) that User 5 made in Session 19.

3.5.2 *Classifying Changes*

Once we had designed this classification, we used it to classify the changes that we had found. We ignored intermediate changes that immediately failed to lex, parse, or type check.

Classifying changes revealed clusters of related changes. Further inspection of those clusters revealed common development patterns. Figure 6 lists, for each user, the top three changes by **Operation** and **Location** (four if there was a tie, and ignoring changes that we found fewer than three times for the user), the total number of changes that we found, and the total number of interactive sessions and self-rated expertise for reference. Changes for which more detailed inspection revealed patterns are highlighted in corresponding colors: blue for incremental development, orange for refactoring, pink for repair, and grey for discovery.

The remainder of this section discusses these patterns, complete with an example, benchmarks, and lessons for automation for each. The benchmarks and lessons for automation are mainly for tool designers: The benchmarks point to changes in specific sessions, the partial or complete automation of which would have helped our users. The lessons for automation are natural directions for improvements to proof engineering tools given the patterns we have observed.

The public data repository contains a complete list of the changes that we classified, as well as a detailed walkthrough of each benchmark.

3.5.2.1 *Incremental Development of Inductive Types*

The most common changes for Users 1 and 5 were adding cases to match statements (**Add Cas**) and adding constructors to inductive types (**Add Ctr**). These changes corresponded to incremental development of inductive types, followed by corresponding extensions to match statements of functions that destruct over them, or to inductive types that depend on or relate to them. This pattern sometimes spanned multiple sessions. While this pattern was most prevalent for Users 1 and 5, it was also present for Users 2 and 7.

The diffs in Figure 7 show an example change to an inductive type, along with a corresponding change to a fixpoint. The change on the left adds five constructors and moves one constructor down. The change on the right adds five corresponding cases and moves one corresponding case down.

BENCHMARK 1 The example change from Figure 7 came from User 5, Sessions 18, 19, 27, 33, and 35. There, over the course of three weeks, the user incrementally developed the inductive type `Term` along with a record `EpsilonLogic` and fixpoints `simplify` (later renamed to `identity`) and `free_vars`.

BENCHMARK 2 In Sessions 37 and 41, over two days, User 1 incrementally developed similar inductive types `ST` and `GT`, as well as fixpoints `Gamma`, `Alpha`, and `eq` that referred to them.

LESSONS FOR AUTOMATION Given that several users show this pattern, this is one use case for which better automation may help. Automation may help proof engineers adapt other inductive types, match statements, and proofs after extending inductive types with new constructors. We are not aware of any work on adapting related inductive types and match statements. There is some work on adapting proof obligations to new constructors [**?**], and a proposed algorithm for generating proofs that satisfy those obligations [**?**], but nothing that exists for a current version of Coq.

> **Takeaway**: Proof engineers could benefit from automation to help update proofs and definitions after adding constructors to inductive types.

3.5.2.2 *Repetitive Refactoring of Identifiers*

Users 1 and 7 showed a pattern of repetitive refactoring, through qualifying constants after changing imports (**Qfy Con**), and renaming

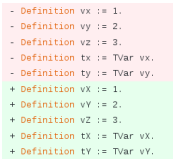

Figure 8: Renaming of definitions in User 7, Session 93.

identifiers (**Rnm Idn**) and the constants that referred to them (**Rnm Con**), respectively. This pattern sometimes spanned multiple sessions, and even simple refactorings sometimes resulted in failures.

Figure 8 shows an example renaming from the five definitions at the top to the five definitions at the bottom. The change renames the identifiers of these definitions to follow the same convention, then makes the corresponding changes to constants in the bodies of the last two definitions.

BENCHMARK 3 The example from Figure 8 came from User 7, Session 93. The definition of `ty` failed, since User 7 had already defined an inductive type with that name. In response, User 7 renamed all of these to follow the same convention. This took four attempts, but only a few minutes.

BENCHMARK 4 In Session 193, User 7 split the `TVar` constructor of the inductive type `ty` into two constructors: `TBVar` and `TFVar`. User 7 at the same time split the fixpoint `FV` into `FFV` and `FBV`. In Session 198, User 7 at the same time renamed the broken lemma `b_subst_var_eq` to `b_subst_bvar_eq`, and substituted in `TBVar` for `TVar` in its body.

BENCHMARK 5 User 1 imported the `List` module in Session 37, commit 10. After the import, `In` referred to the list membership predicate from the standard library, whereas previously it had referred to `Ensembles.In`. The 6 qualify constant changes that we found for User 1 were changing `In` to `Ensembles.In` inside of three existing definitions. This took multiple tries per definition, but only a few minutes in total.

LESSONS FOR AUTOMATION Refactoring terms (rather than proof scripts) as in RefactorAgda [**?**] and Chick [**?**] would have helped our users, but few refactoring tools for ITPs support this [**?**], and neither of these are implemented for Coq. Supporting making similar changes throughout a program, like Chick does, may be especially useful. Semantics-aware refactoring support may take this even further: A refactoring tool for Coq may, for example, determine that an import shadows an identifier, compute what the identifier used to refer to, and refactor appropriately. Or, it may guide the user to rename terms that refer to other recently renamed terms. Both of these would have helped our users.

```
Lemma proc_rspec_crash_refines_op T (p : proc C_Op T)
  (rec : proc C_Op unit) spec (op : A_Op T) :
  (forall sA sC,
-   absr sA sC tt -> proc_rspec c_sem p rec (refine_spec spec sA)) ->
-  (forall sA sC, absr sA sC tt -> (spec sA).(pre)) ->
+   absr sA (Val sC tt) -> proc_rspec c_sem p rec (refine_spec spec sA)) ->
+  (forall sA sC, absr sA (Val sC tt) -> (spec sA).(pre)) ->
   (forall sA sC sA' v,
-   absr sA' sC tt ->
+   absr sA' (Val sC tt) ->
    (spec sA).(post) sA' v -> (op_spec a_sem op sA).(post) sA' v) ->
  (forall sA sC sA' v,
-   absr sA sC tt ->
+   absr sA (Val sC tt) ->
    (spec sA).(alternate) sA' v -> (op_spec a_sem op sA).(alternate) sA' v) ->
  crash_refines absr c_sem p rec (a_sem.(step) op)
   (a_sem.(crash_step) + (a_sem.(step) op;; a_sem.(crash_step))).
```

Figure 9: Patches to a lemma in User 3, Session 73.

> **Takeaway**: Refactoring and renaming tools, similar to those available for programmers in languages like Java, could also help proof engineers, and could potentially be more powerful in ITPs.

3.5.2.3 *Repetitive Repair of Specifications*

The top changes that we found for Users 3 and 8 were patching arguments (**Pch Arg**) and replacing functions (**Rpl Fun**), respectively. These corresponded to a pattern of repetitive repair of specifications, often over several sessions. Sometimes these repairs were necessary in order for the specification to type check or for existing tactics to succeed. Sometimes, after repairing specifications, users also repaired their proofs.

Figure 9 shows an example change patching the arguments of a lemma. This change wraps two arguments into a single application in three different hypotheses of a lemma.

BENCHMARK 6 11 of the 13 patches to arguments that we found for User 3, including the example in Figure 9, came from Session 73. All of these changes similarly wrapped arguments into an application of `Val`. We suspect that this was due to a change in the definition of `absr`, but we were not able to confirm this since the change in question occurred before the beginning of the study. The user admitted or aborted the proofs of four of the five changed lemmas.

BENCHMARK 7 28 of the 29 replace function changes that we found for User 8 were changes from = to == over the course of about a week in Sessions 2, 14, 37, 40, 65, 79, 108, 125, and 160. The corresponding proof attempts suggest that while these terms were well-founded with =, the changes to use == may have been necessary to make progress in proofs using certain tactics. Sometimes, after making these changes, User 8 also fixed tactics that had worked before.

LESSONS FOR AUTOMATION Automation may help with these sorts of repairs to theorems and proofs. The proof repair tool PUMPKIN PATCH already handles some repairs to proofs after changes to both

```
- Theorem mult_n_Sm : forall m n, n * S m = m + n * m.
+ Theorem mult_n_Sm : forall m n, n * S m = n + n * m.
  Proof.
  (induction n as [| n' IHn']).
  -
  (simpl).
  (rewrite <- plus_n_0).
```

Figure 10: A partial attempt at proving and later correction to an incorrect theorem from User 3, Session 11377 (tactic formatting is not preserved in the data that REPLICA receives).

Content [**?**] and **Structure** [**?**], but has support for repairing the theorem statement itself only in the latter case, and only for a specific class of changes to inductive types. These changes provide examples where changing the theorem type is also desirable, and may make good benchmarks for further development to support this.

> **Takeaway**: Proof repair tools should repair programs and specifications, not just proofs.

3.5.2.4 *Interactive Discovery of Programs and Specifications*

In Q1 (Section 3.4), we found that users most often fixed proofs by stepping up outside of proofs and changing other things. Our observations from Q2 are consistent with this, and give some insight into the details. Changes from Users 3, 7, 8, and 10 all revealed a pattern of interactive discovery of programs and specifications: In some cases, these users discovered bugs in their programs during a proof attempt or test. In other cases, these users discovered that their specifications were incorrect, too weak, or difficult to work with. Users sometimes assigned temporary names to lemmas or theorems, then renamed them only after finalizing their types. Even experts made mistakes in programs and theorem statements (perhaps they were the ones catching them most effectively).

Figure 10 shows an example of catching a bug in a specification during an attempted proof attempt. The theorem before the change is impossible (let `m` be 3 and `n` be 1). After attempting to prove it and reaching this goal:

```
m : nat
--------
0 = m
```

the user steps up and fixes the theorem statement by replacing an argument (**Rpl Arg**), then later finishes the proof.

BENCHMARK 8 The change from Figure 10 can be found in User 3, Session 11377. The same session contains changes to the same theorem by moving arguments (**Mov Arg**). The user succeeded at the proof after about three minutes.

BENCHMARK 9 In Session 13, commit 11, User 10 patched a case (**Pch Cas**) of a fixpoint `fib'` after testing. This took the user about thirty seconds. The fixpoint `fib'` itself may have been used to test a different function.

BENCHMARK 10 User 7 mainly demonstrated this pattern through adding and moving hypotheses (**Add** and **Mov Hyp**). The latter most often corresponded to generalizing the inductive hypothesis after a partial proof attempt by swapping theorem hypotheses, in some cases in order to induct over a different hypothesis altogether. We found such changes in Sessions 19, 56, 93, 94, 104, 110, 153, 159, and 176. See, for example, `match_ty__value_type_l` in Session 94, commit 15.

BENCHMARK 11 In Session 2, User 7 temporarily named a lemma `weird_trans`, then renamed it to `sub_r_nf__trans` by Session 10 after finalizing its type after several partial proof attempts over the course of about a day and a half.

LESSONS FOR AUTOMATION The effect of discovering bugs by attempting a proof accounts for some of the benefits of verification [**?**]. It makes sense, then, to continue to build automation to support users in finding and fixing those bugs. One possible unexplored avenue for this is integrating repair tools with QuickChick [**?**] for testing specifications, or with the `induct` tactic from FRAP [**?**] or the hypothesis renaming functionality from CoqPIE [**?**] for simple generalization of inductive hypotheses. Integrating tools for discovering lemma and theorem names [**?**] during the development process may help users who use temporary names. Above all, repair is not just something that happens to stable proof developments—change is everywhere in developing those programs and proofs to begin with.

> **Takeaway**: Integrating tools for proof repair with tools for discovery and development of specifications is an unaddressed opportunity to support proof engineers.

3.6 CONCLUSIONS & FUTURE WORK

We built REPLICA, a Coq plugin that remotely collects fine-grained proof development data in a way that is decoupled from the UI. Using REPLICA, we collected data on changes to proofs, programs, and specifications at a level of granularity not previously seen for an ITP. Visualization and analysis of this data revealed evidence in our study population of development patterns, at times confirming folk knowledge—like that discovering the correct program or specification was often a conversation with the proof itself, even for experts—and providing useful insights and benchmarks for tools.

The infrastructure that we have built and the data that we have collected using it are publicly available. We hope to see it used as benchmarks for the improvement of proof engineering tools, and as data for future studies. We hope to see the infrastructure that we have built reused or adapted to other ITPs for future studies.

THREE WISHES We would like for our experiences building and using REPLICA to help the community conduct more studies of proof development processes. We thus conclude with three wishes, the fulfillment of which would help address the challenges that we encountered along the way:

1. Better abstraction of user environments (Section 3.6.1)
2. More information about user interaction (Section 3.6.2)
3. More users (Section 3.6.3)

We discuss how to grant each wish both at the level of study design and at the level of ITP design in order to facilitate future studies of this kind.

3.6.1 *Wish: Better Abstraction of User Environments*

In order to cast as broad of a net as possible for potential users, we designed REPLICA to be independent of the UI, the version of Coq, and the build system. Coq's plugin infrastructure and interaction model offered a promising avenue for this, and Coq's resource file infrastructure meant that users could load the plugin universally with just one LOC in one location. All of this gave us the impression of independence from the details of the UI, ITP version, and build system.

We later found that, while this infrastructure was useful, the impression of total independence was somewhat of an illusion; all of these details mattered. In particular, while REPLICA itself was UI-independent, the analyses that we ran did not achieve full independence. For example, we found that depending on the version of Coq and what event triggers a cancellation, different UIs use different mechanisms for cancellation (recall these mechanisms from Section 3.2). Our analyses had to deal with all of these mechanisms.

In addition, partway through the study, we received a bug report that noted that one common build system for Coq compiles files by default using a flag that disables loading the Coq resource file. This is one possible explanation for the lack of data that some users sent. To remedy this, we must ask future users of REPLICA to compile all of their Coq projects without using this flag; we have updated the REPLICA documentation to account for this.

THE STUDY DESIGNER Without help from the ITP designer, the study designer cannot achieve full abstraction from these details. The study designer may, however, work around the lack of full abstraction. We recommend testing the study infrastructure with many different environments for many different scenarios. It may help to identify potential users early and survey their development environments before even beginning to test, covering likely scenarios in advance. It may also help to build in a short trial period on the final users before the final data collection begins, thereby covering their particular development environments. The latter has the additional benefit of giving the study designer time to discover and discuss with users possible confounding variables for analysis, like development style or project phase.

We had the foresight to test REPLICA with `coqtop`, Proof General, and CoqIDE, but we did not anticipate User 5's custom UI, which treated failures differently from the others. We also did not anticipate the behavior of different build systems on Coq's resource file, since we tested only one build configuration. We could have avoided both of these issues with our recommendations. Instead, we had to work around both issues after deployment.

THE ITP DESIGNER Most of the power to grant this wish is in the hands of the ITP designer. Full abstraction over development environments may not be possible, but the ITP designer may design the interaction model with this as a goal. The REPL and state machine already strive for this—and come close to achieving it—but their current implementations in Coq fall short. Improvements to abstraction here may be minimal, like providing fewer mechanisms for UIs to accomplish the same thing, or providing a way to guarantee that a plugin is always loaded for all build systems.

For a more radical approach, the ITP designer may look to other interaction models. For example, Isabelle/HOL has recently moved away from its REPL, instead favoring the Prover IDE (PIDE) [**?**] framework as its interaction model. With PIDE, UIs and ITPs communicate using an asynchronous protocol to manage versions of a document [**?**], annotating the document with new information when it becomes available [**?**]. Personal communication suggests that there is an ongoing attempt to to centralize functionality like the build system into PIDE. While centralization may limit the potential for customization by different UIs, it may make studies of fine-grained proof development data easier, since the instrumentation may occur at a higher level, guaranteeing more uniform interaction.

3.6.2 *Wish: More Information about User Interaction*

By observing state machine messages through the plugin API, we were able to log data at high enough granularity to reconstruct hundreds of interactive sessions. This helped us identify incremental changes to tactics and terms that are difficult to gather from other sources.

The hooks that we designed together with the Coq developers, however, were not perfect. There was no way for us to use those hooks to listen to the messages that Coq sent back to the UI. Making this change would have required modifying Coq again, and by the time we realized this, it was too late. Listening to those responses would have given us more insight into user behavior. It would have also helped us distinguish between failures and stepping up. Instead, we had to distinguish between these using IDE-specific heuristics, which left us with no automatic way to distinguish failures from stepping up for User 5's custom UI.

We also found that once we had collected granular data, analyzing it was sometimes difficult. For example, sometimes users defined a term, stepped up and changed an earlier term, then stepped back down and changed the original term. For such a change, our visualization script constructed 3 consecutive commits; to determine that the original term had changed, we had to look at the difference between the 1st and the 3rd commits. Sometimes, changes occurred over tens of commits, or over several sessions. Thus, manual analysis of changes for Q2 was tedious, and involved not only inspecting thousands of diffs but also understanding each session well enough to track term information over time.

We considered automating the analysis for Q2, but we found automatically classifying changes in terms to be prohibitively difficult. While part of this was due to the inherent difficulty of the problem, part of this was also due to missing information: When the UI backed up to an earlier state, we did not know what was below the command or tactic corresponding to that state in the file. So, when a user copied and pasted a term and then modified both versions, or made several changes at once to a term before stepping down below it, even manually deciding whether it was the same term that had changed or a new term entirely proved to be difficult.

THE STUDY DESIGNER To grant this wish, we recommend that the study designer run a beta test, complete with analysis, as we did—and that they do so early, as we did not. That way, there is enough time to address needs discovered during testing, for example by communicating with ITP designers. It may also help to use multiple methods for data collection, for example by augmenting the collected data with information from the IDE to track the state of definitions in the file below the location to which the user has stepped.

The beta testing phase was extremely useful to us; as a result we reworked our server infrastructure to receive and easily analyze larger amounts of data, and we tested data backup infrastructure and network failure resilience code. We also discovered and reported a bug[8] in CoqIDE and Proof General that had made it impossible for us to tell which sessions corresponded to which files; the fix was marked as critical and backported to Coq 8.9, so only the analyses for Users 5 (custom UI), 8 (Coq 8.8), and 10 (Coq 8.8) were impacted. However, while we discovered the lack of response information in the plugin hooks during this period, we did not have enough time to coordinate with Coq developers on changes to the hooks. Leaving more time between testing and the final study would have helped us.

THE ITP DESIGNER By implementing hooks like the one that we helped implement for Coq, the ITP designer can make it easier for researchers to study proof development processes. To grant this wish, we recommend that these hooks expose not just request information, but also response information, especially error information.

Augmenting the interaction model with more information about the file may also help. For example, the interaction model could expose a simple and uniform way for UIs to track that a definition in a new state corresponds to a definition in a previous state (that is, that the user did not remove that definition from the file and replace it with something entirely new). This could make it much simpler for an analysis to track changes to a definition over time, and to choose the granularity with which to inspect changes. There is some tension between this and the wish for abstraction from Section 3.6.1, but it may be worth weighing the tradeoffs.

3.6.3 *Wish: More Users*

We attempted to recruit a large and representative sample of Coq proof engineers. In our attempts to recruit users, we contacted programming languages groups at several universities that were known to be working with Coq, and emailed `coq-club` with our project pitch. We included a promotional video in the hopes of attracting as many users as possible.

In the end, however, we were able to recruit only 12 users, all of whom were intermediate through expert users with at least two years of Coq experience. We received interactive sessions for just 8 of 12 of these users. While this data was rich and granular, with just 8 users, we were not able to reach broad conclusions about proof engineers more generally.

Part of this was likely due to the demographics of the community: Compared to other programming environments, Coq has only a small

8 `http://github.com/coq/coq/issues/8989`

number of users, many of whom have or are pursuing graduate degrees. However, part of this may have been due to deterrents in our screening process, or poor incentives for our users.

THE STUDY DESIGNER The study designer may fulfill this in part by casting as broad of a net as possible, carefully considering any deterrents in screening criteria. It may help to use welcoming language to encourage potential users who are unsure if they qualify. To reach beginner users, it may help to recruit students. To reach a more international audience, it may help to distribute promotional materials and consent forms in multiple languages. It may also help to consider incentives that appeal to proof engineers. However, until the ITP user community grows significantly, it will continue to be difficult to conduct large-scale studies of proof engineering.

We reached users from different institutions, but most of our users had similar levels of expertise. We suspect that this was in part because we asked for users to have at least a year of experience using Coq, so as to avoid mixing in data from users learning Coq for the first time. We also required fluency in English to ensure that users understood the consent forms. Both of these may have deterred users. One potential user who did not participate noted that we did not make it clear in our recruitment materials that data from occasional rather than frequent Coq users was still useful to us. The same potential user noted that we could have engaged more with community leaders, thereby giving others in the community more incentive to participate. We considered monetary incentives, but ultimately decided they were less tempting to the community than appeals to improving tools. Perhaps this was misguided or attracted a more advanced population, and perhaps we could have considered a different incentive.

THE ITP DESIGNER The ITP designer may help reduce the costs of participating in these studies. We suspect that the primary gains here will come from continuing to break down barriers to using plugins and other outside tooling. Some examples of this include reducing the brittleness of plugin APIs over time, improving build and distribution systems, making it possible to prove that a plugin does not interact with a kernel in a way that could compromise soundness of the system, and providing more support for users who port proof developments and tools between ITP versions.

Of course, continuing to improve the ITP itself may continue to expand the community to reach more users. This may cause a positive feedback loop, helping to collect more data in order to drive further improvements to the ITP, in turn continuing to help the community grow.

3.7 RELATED WORK

We consider work on analysis of development processes and on identifying, classifying, and visualizing changes, in proof engineering and in software engineering more generally.

ANALYSIS OF DEVELOPMENT PROCESSES REPLica instruments Coq's REPL to collect fine-grained data on proof engineering development processes. Outside of the context of ITPs, REPLica is not the first tool to collect data at this level of granularity. Mylar [**?**] monitors Java development activity and uses it to visualize codebases to make them easier to navigate. The Solstice [**?**] plugin instruments the IDE to make program analysis possible at this level of granularity. Several studies have used video recordings [**?**, **?**], sometimes combined with IDE instrumentation [**?**], to study how programmers develop and change code. REPLica brings fine-grained analysis to an ITP, taking advantage of an interaction model that is especially common for ITPs to build instrumentation that is minimally invasive, decoupled from the UI, and captures all information that the user sends to the ITP.

There have been a number of recent studies on the development processes and productivity of proof engineers, drawing on sources like version control history [**?**, **?**, **?**], archives and libraries [**?**, **?**, **?**, **?**], project logs [**?**, **?**], artifacts [**?**, **?**, **?**], meeting notes [**?**, **?**], and personal experiences [**?**, **?**]. Collecting proof development data at the level of user interaction was, until now, an unaddressed opportunity to understand proof development processes [**?**]. This has not gone unnoticed. For example, one study [**?**] of proof development processes cited lack of precision of version control data as a limitation in measuring the size and timing of changes. Another study [**?**] assumed that proof development is linear, but noted that this assumption did not reflect the interative nature of proof development (like that observed in Section 3.5.2.4). The study cited this assumption as a threat to validity, and noted that this threat is inherent to using an artifact with complete proofs. REPLica collects more precise data that may alleviate or eliminate these limitations and provide insights into proof development processes that other techniques lack access to.

IDENTIFYING, CLASSIFYING, & VISUALIZING CHANGES The REPLica analyses identify, classify, and visualize changes to proof scripts and terms over time in the context of an ITP. There is a wealth of work along these lines beyond the context of an ITP. Type-directed diffing [**?**] describes a general framework for representing changes in algebraic datatypes. Existing work [**?**] analyzes a large repository and collects data on changes to quantify code decay. TreeJuxtaposer [**?**] implements structural comparison of very large trees, which can be used to represent both terms and proofs scripts. Previous work has

also studied the types of mistakes that users make while writing their code [**?**]. Adapting these techniques to Coq and applying them to analyze the collected data may help improve the REPLica analyses to reveal new insights, even using the same data.

There is some work along these lines within the context of an ITP. The proof repair tool PUMPKIN PATCH [**?**] is evaluated on case studies identified using a manual change analysis of Git commits. The proof refactoring and repair tool Chick [**?**] includes a classification of changes to terms and types in a language similar to Gallina. This classification is similar to ours in that it includes adding, moving, or deleting information, though it does not include any breakdown of changes in content or syntax. The IDEs CoqIDE [**?**], CoqPIE [**?**], and PeaCoq [**?**] all have support for visualizing changes during development. The machine learning tool Proverbot9001 [**?**] uses similar search trees to the ones that we use to visualize proof state, though without information on intermediate failing steps.

The novelty in the REPLica analyses, classifications, and visualizations come from the fact that these are run on, derived from, and produced using remote logs of live ITP development data. The changes themselves and the patterns that they reveal suggest ways to continue to improve and to measure the improvement of proof engineering tools.

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 [**?**, **?**, **?**].

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 [**?**] and rippling [**?**]). 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 [**?**, **?**, **?**], or through program-specific proof automation [**?**]. 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 [**?**].

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

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

Figure 11: Old (left) and new (right) definitions of `IZR` in Coq. The old definition applies injection from naturals to reals and conversion of positives to naturals; the new definition applies injection from positives to reals.

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 [**?**] 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 [**?**]. This commit redefined injection from integers to reals (Figure 11). 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 [**?**]. 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.

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

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.

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.

```
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 12: 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 12 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 12 (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 12, 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 12, 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

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

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 [**?**], 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 12. 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
```

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

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

Figure 13: Old (left) and new (right) definitions of `int` in CompCert.

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 12, 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 => O
  | B2 b' => 2 * (bin_to_nat b
      ')
  | B21 b' => 1 + 2 * (
      bin_to_nat b')
  end.
```

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

Figure 14: 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 [**?**] with a breaking change to `int` (Figure 13). 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.✓
```

4.4.1.1 *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

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

over the changed type. PUMPKIN uses this configuration for changes in constructors:

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

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 [**?**, **?**] to an exercise in Software Foundations to each use the other solution's definition of the fixpoint `bin_to_nat` (Figure 14). 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.

4.4.2.1 *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 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:

1: *diff* fixpoint cases for goals
2: use *all components* to recursively search an intermediate lemma for a change in conclusions
3: **if** there are candidates **then**
4: *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.

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.

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

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

Figure 15: 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 [**?**]. The commit changed the definition of `divide` prior to the Coq 8.4 release (Figure 15). 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.✓
```

4.4.3.1 *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.

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

4.5.1.1 *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.[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 16) 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 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.

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

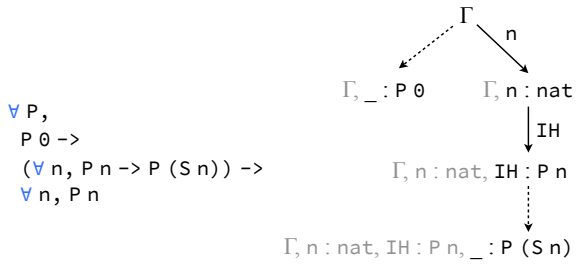

Figure 16: The type of (left) and tree for (right) the induction principle `nat_ind`. The solid edges represent hypotheses, and the dotted edges represent the proof obligations for each case in an inductive proof.

4.5.1.2 *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 [**?**] 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.

4.5.1.3 *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.

4.5.1.4 *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.

4.5.1.5 *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 16). 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 13), 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 [**?**]. 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 [**?**]. Furthermore, Coq type-checks terms that plugins produce. Since PUMPKIN does not modify the type checker, it cannot produce an ill-typed term.

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

4.6 TESTING BOUNDARIES

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`

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

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.

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 17), 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 [**?**]) `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 were small ($\leq$ 67 LOC); we leave evaluating performance on larger terms to future work.

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

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 [**?**, **?**] and type-directed diffing [**?**] 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 [**?**]) or continuous integration (CI) system (such as Travis [**?**]) 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 [**?**, **?**, **?**] 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 [**?**] and Agda [**?**]). Proof assistants with non-constructive logics (for example, Isabelle/HOL [**?**]) 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 [**?**].

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 [**?**], and adding too many hints may impede performance. More generally, we can integrate PUMPKIN with a transfer tactic [**?**, **?**], 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 [**?**] or through an automatic method [**?**]. This is later generalized and implemented in Isabelle [**?**] and Coq [**?**, **?**]; 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 [**?**] adapts proof obligations to structural changes in inductive types. Later work [**?**] 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 [**?**, **?**, **?**] 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` [**?**] can make proofs more resilient to changes. Hammers like Isabelle's sledgehammer [**?**] can make proofs agnostic to some low-level changes. Recent work [**?**] 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 [**?**, **?**] can help prevent low-level maintenance tasks. Writing these tactics requires good engineering [**?**] 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 [?] 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 [?] may help improve these components. Lean [?] 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 [?] 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 [?] 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 [?] and designing theorems and proofs that make maintenance less of an issue. Design principles for specific domains (such as formal metatheory [?, ?, ?]) can make verification more tractable. CertiKOS [?] introduces the idea of a deep specification to ease verification of large systems. Ornaments [?, ?] 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 [?, ?] and to optimize build performance [?]. 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 [?]. The Haskell refactoring tool HaRe [?] automatically lifts definitions, and may be useful for improving abstraction. There is a growing body of work on refactoring in the context of ITP [?, ?]. The IDE CoqPIE [?] and the verification platform Why3 [?] 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 [**?**] finds differences in algebraic data types. Semantics-based change impact analysis [**?**] models semantic differences between documents. Differential assertion checking [**?**] analyzes different versions of a program for relative correctness with respect to a specification. Incremental λ-calculus [**?**] 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 [**?**]. 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 [**?**, **?**, **?**, **?**, **?**] 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.

5

OFTEN, PRACTICALLY

5.1 INTRODUCTION

Indexed inductive types make it possible to internalize data into the type level, eliminating the need for certain functions and proofs. Consider, for example, a theorem from the Coq standard library [**?**] which states that mapping a function over lists preserves length:

```
map_length T1 T2 (f : T1 → T2) : ∀ (l : list T), length (List.
    map f l) = length l.
```

One way to eliminate the need for this theorem is to internalize the length of a list into its type, creating a dependently typed vector (Figure 18). The map function for vectors in Coq's standard library, for example, carries a proof that it preserves length:

```
Vector.map {T1} {T2} (f : T1 → T2) : ∀ (n : nat) (v : vector T1
    n), vector T2 n.
```

so that a theorem like `map_length` is no longer necessary.

Unfortunately, for all of the benefits they bring, indexed inductive types are notoriously difficult to use. Dependently typed vectors, for example, impose proof obligations about their lengths on the user; these can quickly spiral out of control. In recent coq-club threads asking for advice on how to use dependently typed vectors, experts called them "not suitable for extended use" [**?**] and noted that "almost no one should be using [them] for anything" [**?**].

We show how *proof reuse*—reusing existing proofs to derive new proofs—can tackle many of the challenges posed by indexed inductive types, allowing the user to move between unindexed and indexed versions of a type (for example, lists and vectors) and reap the benefits of indexed types without many of the costs. We focus in particular

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

```
vector (T : Type) : nat → Type :=
| nilV : vector T 0
| consV :
    ∀ (n :  nat), T → vector T n →
       vector T (S n).
```

Figure 18: A `vector` (right) is a `list` (left) indexed by its length (highlighted in orange).

on the benefits of this approach in deriving functions and proofs for fully-determined indexed types, when the index is a fold over the unindexed version (such as the length of a list). In our approach, the user writes functions and proofs over the unindexed version, and a tool then automatically *lifts* those functions and proofs to the indexed version. The user can then switch back to working with the unindexed version by running the tool in the opposite direction. In that way, the user can use lists when lists are convenient, and vectors when vectors are convenient.

Our approach uses *ornaments* [**?**], which express relations between types that preserve inductive structure, and which enable lifting of functions and proofs along those relations. Recent work introduced ornaments to a subset of ML and was heavily focused on automatically lifting functions [**?**]; until now, such an approach was not available in a dependently typed language. Existing implementations of ornaments in dependently typed languages work only in embedded languages, and have little to no automation [**?**, **?**, **?**].

Our main contribution is a Coq plugin for automatic function and proof reuse using ornaments. Our plugin DEVOID (Dependent Equivalences Via Ornamenting Inductive Definitions) works directly on Coq code, rather than on an embedded language. DEVOID automates lifting functions and proofs along *algebraic ornaments* [**?**], a particular class of ornaments that represent fully-determined indexed types like lists and vectors. DEVOID implements an algorithm to search for ornaments between these types—to the best of our knowledge, the first search algorithm for ornaments—and an algorithm to lift functions and proofs along the ornaments it discovers.

We motivate (Section 5.2), specify (Section 5.3), and formalize (Section **??**) the search and lifting algorithms that DEVOID implements (Section **??**). A comparison to a more general proof reuse approach (Section 4.4) demonstrates the benefits of using ornaments: DEVOID imposes less of a proof burden on the user, and produces smaller terms and faster functions.

5.2 MOTIVATING EXAMPLE: PORTING A LIBRARY

DEVOID is a plugin for Coq 8.8; it can be found in the repository linked to as **Supplement Material** under the abstract of this paper. To see how it works, consider an example using the types from Figure 18, the code for which is in `Example.v`. In this example, we lift two list zip functions and a proof of a theorem relating them from the Haskell CoreSpec library [**?**]:

```
zip {T1 T2}: list T1 → list T2 → list (T1 * T2).
zip_with {T1 T2 T3} (f : T1 → T2 → T3): list T1 → list T2 →
    list T3.
zip_with_is_zip {T1 T2}: ∀(l1:list T1)(l2:list T2), zip_with
    pair l1 l2 = zip l1 l2.
```

DEVOID runs a preprocessing step before lifting, which we describe in Section **??**; we assume this step has already run. We use the cyan background color to denote tool-produced terms and the names that refer to them. We run DEVOID to lift functions and proofs from lists to vectors, but it can also lift in the opposite direction.

Step 1: Search. We first use DEVOID's `Find ornament` command to search for the relation between lists and vectors:

```
Find ornament list vector.
```

This produces functions which together form an equivalence (denoted $\simeq$):

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

Step 2: Lift. We then lift our functions and proofs along that equivalence using DEVOID's `Lift` command. For example, to lift `zip`, we run the command:

```
Lift list vector in zip as zipV_p.
```

This produces a function with this type:

```
zipV_p {T1 T2} : Σ n.vector T1 n → Σ n.vector T2 n → Σ n.
    vector (T1 * T2) n.
```

that behaves like `zip`, but whose body no longer refers to lists. We lift our proof similarly:

```
Lift list vector in zip_with_is_zip as zip_with_is_zipV_p.
```

This produces a proof of the analogous result (denoting projections by π_l and π_r):

```
zip_with_is_zipV_p {T1 T2} : ∀ (v1 : Σ n.vector T1 n) (v2 : Σ n.
    vector T2 n),
  zip_withV_p pair (∃ (πl v1) (πr v1)) (∃ (πl v2) (πr v2)) =
  zipV_p (∃ (πl v1) (πr v1)) (∃ (πl v2) (πr v2)).
```

that no longer refers to lists, `zip`, or `zip_with` in any way.

Step 3: Unpack. The lifted terms operate over vectors whose lengths are *packed* inside of a sigma type. While this lets `Lift` provide strong theoretical guarantees, it can make it difficult to interface with the lifted code. We can recover *unpacked* terms using DEVOID's `Unpack` command. For example, to unpack `zipV_p`, we run the command:

```
Unpack zipV_p as zipV.
```

This produces functions and proofs that operate directly over vectors, like `zipV`:

```
zipV {T1 T2} {n1} (v1 : vector T1 n1) {n2} (v2 : vector T2 n2) :
  vector (T1 * T2) (πl (zipV_p (∃ n1 v1) (∃ n2 v2))).
```

and `zip_with_is_zipV`:

```
zip_with_is_zipV : ∀ {T1 T2} {n1} (v1 : vector T1 n1) {n2} (v2 :
    vector T2 n2),
  eq_dep _ _ _ (zip_withV pair v1 v2) _ (zipV v1 v2).
```

Step 4: Interface. For any two inputs of the same length, `zipV` and `zipV_with` contain proofs that the output has the same length as the inputs. However, the types obscure this information. `Example.v` explains how to recover more user-friendly types, like that of `zipV_uf`:

```
zipV_uf {T1 T2} {n} : vector T1 n → vector T2 n → vector (T1 *
    T2) n.
```

and that of `zip_withV_uf`:

```
zip_withV_uf {T1 T2 T3} (f : T1 → T2 → T3) {n} :
  vector T1 n → vector T2 n → vector T3 n.
```

which both restrict input lengths. We can then use our lifted functions and proofs in client code. For example, we can write a different version of Coq's `BVand` function for bitvectors:

```
BVand {n} (v1 : vector bool n) (v2 : vector bool n) : vector
    bool n :=
  zip_withV_uf andb v1 v2.
```

By working over lists, we are able to reason about only the interesting pieces, thinking about indices only when relevant; in contrast, when writing proofs over vectors, even simple theorems can generate tricky proof obligations. With DEVOID, the programmer can use the lifted functions and proofs to interface with code that uses vectors, then switch back to lists when vectors are unmanageable. In essence, ornaments form the glue between these types.

5.3 SPECIFICATION

This section specifies the two commands that DEVOID implements:

1. `Find ornament` searches for ornaments (specified in Section 5.3.1, described in Section **??**).
2. `Lift` lifts along those ornaments (specified in Section **??**, described in Section **??**).

Algebraic Ornaments. DEVOID searches for and lifts along *algebraic ornaments* in particular. An algebraic ornament relates an inductive type A to an indexed version of that type B with a new index of type I_B, where the new index is fully determined by a unique fold over A. For example, `vector` is exactly `list` with a new index of type `nat`, where the new index is fully determined by the `length` function. Consequentially, there are two functions:

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

that are mutual inverses:

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

and therefore form the type equivalence from Section 5.2. Moreover, since the new index is fully determined by `length`, we can relate `length` to `ltv`:

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

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

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

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

```
promote : A i⃗ → Σ(n : I_B i⃗).B (index n i⃗).
forget  : Σ(n : I_B i⃗).B (index n i⃗) → A i⃗.
```

that are mutual inverses:[1]

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

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

```
indexer : A i⃗ → I_B i⃗.
```

which projects the promoted index:

```
coherence : ∀(a : A i⃗), indexer a = π_l (promote a).
```

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

`Find ornament` searches for algebraic ornaments between types and is, to the best of our knowledge, the first search algorithm for ornaments. `Lift` then lifts functions and proofs along those ornaments, removing all references to the old type. Both commands make some additional assumptions for simplicity; detailed explanations for these are in `Assumptions.v`.

5.3.1 *Find ornament*

In their original form, ornaments are a programming mechanism: Given a type A, an ornament determines some new type B. We invert this process for algebraic ornaments: Given types A and B, DEVOID searches for an ornament between them. This is possible for algebraic ornaments precisely because the indexer is extensionally unique. For example, all possible indexers for `list` and `vector` must compute the length of a list; if we were to try doubling the length instead, we would not be able to satisfy the equivalence.

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

1 The adjunction condition follows from section and retraction.

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

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

5.3.2 *Lift*

`Lift` lifts a term along the ornamental promotion isomorphism between A and B. That is, it lifts types to corresponding types and terms of those types to corresponding terms:

```
Lift list vector in list as vector_p. (* vector_p T := Σ (n :
    nat).vector T n *)
Lift list vector in (cons 5 nil) as v_p. (* v_p := ∃ 1 (consV 0
    5 nilV) *)
```

Furthermore, it recursively preserves this equivalence, lifting non-dependent functions like `zip` so that they map equivalent inputs to equivalent outputs:

```
∀ {T1 T2} l1 l2, promote (zip l1 l2) = zipV_p (promote l1) (
    promote l2).
```

This intuition breaks down with dependent types. With equivalence alone, we can't state the relationship between `zip_with_is_zip` and `zip_with_is_zipV_p`, since the unlifted conclusion:

```
zip_with pair l1 l2 = zip l1 l2.
```

does not have the same type as the conclusion of the lifted version applied to promoted arguments; any relation between these terms must be heterogenous.

In particular, `Lift` preserves the *univalent parametric relation* [**?**], a heterogenous parametric relation that strengthens an existing parametric relation for dependent types [**?**] to make it possible to state preservation of an equivalence: Two terms t and t' are related by the univalent parametric relation $[[\Gamma]]_u \vdash [t]_u : [[T]]_u\ t\ t'$ at type T in environment Γ if they are equivalent up to transport. The details of this relation can be found in the cited work.

`Lift` preserves this relation using the components that `Find ornament` discovers, and additionally guarantees that the lifted term does not refer to the old type in any way:

- **Inputs**: The inputs to and outputs from `Find ornament`, along with a term t, assuming:
  - the assumptions and guarantees from `Find ornament` hold,
  - I_B is not A,
  - t is well-typed and fully η-expanded,
  - t does not apply `promote` or `forget`, and
  - t does not reference B.
- **Outputs**: A term t', guaranteeing:
  - if t is $A\ \vec{i}$, then t' is $\Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i})$,
  - t' does not reference A, and
  - if in the current environment $\Gamma \vdash t : T$, then $[[\Gamma]]_u \vdash [t]_u : [[T]]_u\ t\ t'$.

`Lift` does not require a proof that the input components form the ornamental promotion isomorphism, but they must for the guarantees to hold. It can operate in either direction, promoting from A to packed B or forgetting in the opposite direction; the specification for the forgetful direction is similar, with extra restrictions on how B is used within t.

5.4 ALGORITHMS

This section describes the algorithms that implement the specifications from Section 5.3.

Presentation. We present both algorithms relationally, using a set of judgments; to turn these relations into algorithms, prioritize the rules by running the derivations in order, falling back to the original term when no rules match. The default rule for a list of terms is to run the derivation on each element of the list individually.

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