PROOF ENGINEERING TOOLS FOR A NEW ERA

TALIA RINGER

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

Doctor of Philosophy

University of Washington

2021

Reading Committee:
TODO, Chair
TODO
TODO

Program Authorized to Offer Degree:
Computer Science & Engineering

ABSTRACT

PROOF ENGINEERING TOOLS FOR A NEW ERA

Talia Ringer

Chairs of the Supervisory Committee:
TODO
Computer Science & Engineering

Abstract will go here.

To my parents. Hope this fits on your fridge.

CONTENTS

ACKNOWLEDGMENTS

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

We got other wonderful feedback on the paper from Cyril Cohen, Tej Chajed, Ben Delaware, Jacob Van Geffen, Janno, James Wilcox, Chandrakana Nandi, Martin Kellogg, Audrey Seo, James Decker, and Ben Kushigian. And we got wonderful feedback on e-graph integration for future work from Max Willsey, Chandrakana Nandi, Remy Wang, Zach Tatlock, Bas Spitters, Steven Lyubomirsky, Andrew Liu, Mike He, Ben Kushigian, Gus Smith, and Bill Zorn. The Coq developers have for years given us frequent and efficient feedback on plugin APIs for tool implementation.

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

Part I

OPENING

1

INTRODUCTION

2

READING GUIDE

Part II

PROOF ENGINEERING

3 PROOF DEVELOPMENT

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

4

PROOF MAINTENANCE

Context, user study. Examples. Related work.

4.1 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* [173].

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 [12, 218, 193, 27, 194, 135, 209, 15, 31, 214]. 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 [175]. 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 [55] to remotely collect fine-grained data on proof development processes. We have built a Coq plugin (Section 4.2) 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 [98, 154] and Idris [200].

We have used REPLICA to collect a month's worth of data on the proof developments of 8 intermediate to expert Coq users (Section 4.3). 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 4.4) and changes to programs and specifications (Section 4.5). 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 [185, 119, 96, 155, 87, 217], proof refactoring tools [208, 207, 78, 10, 31, 182, 179], and proof repair tools [175, 179].

Our experiences collecting and analyzing this data have helped us to identify three wishes (Section 6.17), 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.

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

4.2 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]

REPLica works by instrumenting Coq to listen to user interaction. Coq's interaction model (Section 4.2.1) 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` [56] to the IDEs CoqIDE [57] and Proof General [14], 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 4.2.2), then sends this data to the server (Section 4.2.3), 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 4.3) and analyze (Sections 4.4 and 4.5) data from hundreds of interactive sessions.

4.2.1 *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

2 `http://github.com/uwplse/coq-change-analytics`
3 Counted using David M. Wheeler's SLOCCount.

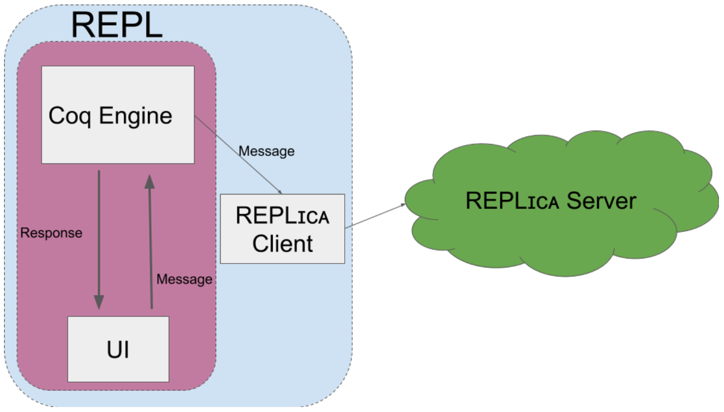


Figure 1: REPLICA design. The REPLICA client listens to the REPL and sends data to the REPLICA server.

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 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,

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

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 [85]) of this definition (again simplified for readability):

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

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 4.5). As a consequence, we were able to find this pattern of development of incrementally extending inductive types for four of our users (Section 4.5.2.1).

4.2.2 *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 [54] 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;

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	6	0	0
1	42	10	4
2	7	3	0
3	11495	101	8
4	1	0	0
5	41	15	16
6	1	0	0
7	229	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.

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 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.

4.2.3 *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

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

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.

4.3 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 4.4)
- **Q2**: What kinds of changes to programs and specifications do users make often, and do those changes reveal patterns amenable to automation? (Section 4.5)

To answer these questions, we recruited 12 proof engineers to install REPLica and use Coq normally for a month (Section 4.3.1). We received a month's worth of data containing granular detail on Coq development for 8 of 12 of those users (Section 4.3.2). After a month, we shut down the server, closed the study, and visualized and analyzed the data (Section 4.3.3).

4.3.1 *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,

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

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 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.

4.3.2 *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.

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

REPLica logged interactive sessions for only 8 of the 12 users. Section 6.17 discusses possible causes of, implications of, and remedies for this.

4.3.3 *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.

4.4 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 4.4.1)
2. Within proofs, application of lemmas was the main driver of proving. (Section 4.4.2)
3. Within proofs, over half of fixes to tactic mistakes fixed either an improper argument or wrong sequencing structure. (Section 4.4.3)

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.

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.

4.4.1 *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 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 [185, 119, 155, 87, 217] with tools for repairing definitions [175, 179].

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 4.5) reveals more about how specifications and programs changed.

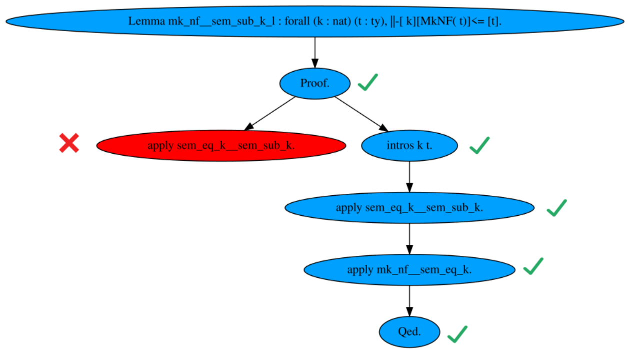

Lemma mk_nf__sem_sub_k_l : forall (k : nat) (t : ty), ||-[k][MkNF(t)]<= [t].
Proof.
apply sem_eq_k__sem_sub_k.
intros k t.
apply sem_eq_k__sem_sub_k.
apply mk_nf__sem_eq_k.
Qed.

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.

4.4.2 *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, 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 [119] for Coq already offers promising developments in this direction by understanding and providing hints about similar lemmas, as does the proof automation tool CoqHammer [61]; similar functionality may help improve the performance of tools that suggest tactics.

4.4.3 *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.

4.5 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 4.6.2), 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 4.5.1), then classify changes that we found (Section 4.5.2). 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 4.5.2.1, 4.5.2.2, 4.5.2.3, and 4.5.2.4), we identified benchmarks (examples of the pattern in our data) and lessons for automation.

4.5.1 *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 4.6.2 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:

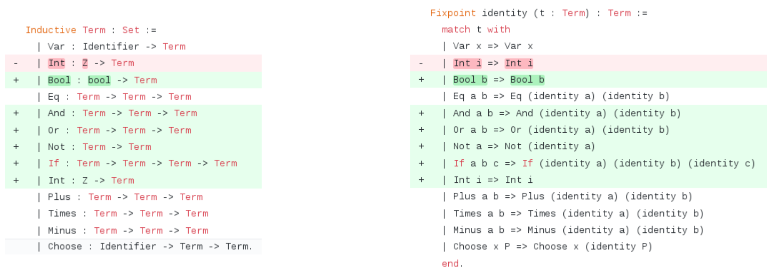
6. **Fun**: function in an application
7. **Typ**: type annotation

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

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

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.

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.

4.5.2 *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.

4.5.2.1 *Incremental Development of Inductive Types*

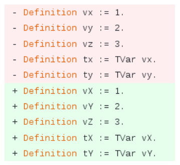
The most common changes for Users 1 and 5 were adding cases to match statements (**Add Cas**) and adding constructors to inductive types (**Add Ctr**). These changes corresponded to incremental development of inductive types, followed by corresponding extensions to match statements of functions that 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.



```
- Definition vx := 1.
- Definition vy := 2.
- Definition vz := 3.
- Definition tx := TVar vx.
- Definition ty := TVar vy.
+ Definition vX := 1.
+ Definition vY := 2.
+ Definition vZ := 3.
+ Definition tX := TVar vX.
+ Definition tY := TVar vY.
```

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

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 [30], and a proposed algorithm for generating proofs that satisfy those obligations [148], 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.

4.5.2.2 *Repetitive Refactoring of Identifiers*

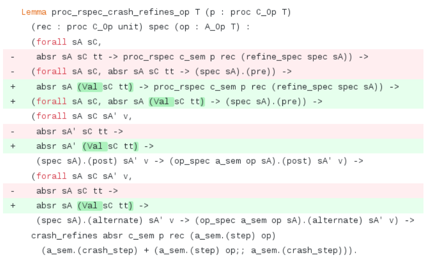
Users 1 and 7 showed a pattern of repetitive refactoring, through qualifying constants after changing imports (**Qfy Con**), and renaming identifiers (**Rnm Idn**) and the constants that referred to them (**Rnm Con**), respectively. This pattern sometimes spanned multiple sessions, and even simple refactorings 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



```
   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.

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 [208] and Chick [179] would have helped our users, but few refactoring tools for ITPs support this [173], 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.

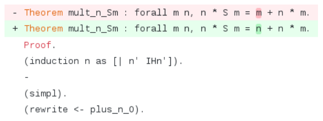
> **Takeaway**: Refactoring and renaming tools, similar to those available for programmers in languages like Java, could also help proof engineers, and could potentially be more powerful in ITPs.

4.5.2.3 *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



```
- 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_O).
```

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).

`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 **Content** [175] and **Structure** [178], 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.

4.5.2.4 *Interactive Discovery of Programs and Specifications*

In Q1 (Section 4.4), 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 [152]. 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 [162] for testing specifications, or with the `induct` tactic from FRAP [45] or the hypothesis renaming functionality from CoqPIE [182] for simple generalization of inductive hypotheses. Integrating tools for discovering lemma and theorem names [16] 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.

4.6 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 4.6.1)
2. More information about user interaction (Section 4.6.2)
3. More users (Section 4.6.3)

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.

4.6.1 *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 4.2). 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) [204] framework as its interaction model. With PIDE, UIs and ITPs communicate using an asynchronous protocol to manage versions of a document [205], annotating the document with new information when it becomes available [204]. 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.

4.6.2 *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 4.6.1, but it may be worth weighing the tradeoffs.

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

4.6.3 *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 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.

4.7 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 [110] monitors Java development activity and uses it to visualize codebases to make them easier to navigate. The Solstice [153] plugin instruments the IDE to make program analysis possible at this level of granularity. Several studies have used video recordings [181, 115], sometimes combined with IDE instrumentation [122], 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 [12, 218, 193], archives and libraries [27, 135, 209, 15], project logs [12, 218], artifacts [194, 135, 15], meeting notes [12, 218], and personal experiences [31, 214]. Collecting proof development data at the level of user interaction was, until now, an unaddressed opportunity to understand proof development processes [173]. This has not gone unnoticed. For example, one study [218] 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 [135] 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 4.5.2.4). 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 [144] describes a general framework for representing changes in algebraic datatypes. Existing work [79] analyzes a large repository and collects data on changes to quantify code decay. TreeJuxtaposer [149] 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 [116]. 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 [175] is evaluated on case studies identified using a manual change analysis of Git commits. The proof refactoring and repair tool Chick [179] 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 [80], CoqPIE [182], and PeaCoq [180] all have support for visualizing changes during development. The machine learning tool Proverbot9001 [185] 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.

Part III

TOOLS FOR A NEW ERA

5

PROOF REPAIR

5.1 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 [215, 19, 74].

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 [2] and rippling [36]). 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 [215, 19, 74], or through program-specific proof automation [43]. 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 [161].

```
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 [124] 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.

5.2 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 [140]. 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.X
```

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.

5.3 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 [55]. 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 5.7.

We motivate the core components (Section 5.3.1) and describe how they compose to find patches (Section 5.3.2). We find patches for real code in Section 6.6, and we describe our implementation in Section 5.5.

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

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

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

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

Figure 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.

5.3.1 *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 [2], 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.

5.3.2 *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 6.6 describes real-world examples that demonstrate more configurations.

5.4 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 5.4.1)

2. **Porting definitions** between libraries
   (Software Foundations, Section 5.4.2)

3. **Updating proof assistant versions**
   (Coq Standard Library, Section 5.4.3)

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 5.4.1 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 5.7.

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

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

Figure 14: Definitions of bin_to_nat for Users A (left) and B (right).

5.4.1 *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 [126] 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.X
```

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.X
```

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.✓
```

5.4.1.1 *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 corre-
   sponding case of the proof
3: if there are candidates then
4:     try to invert the patch to find an isomorphism
```

5.4.2 *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 [9, 21] 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.

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

Figure 15: Old (left) and new (right) definitions of `divide` in Coq.

5.4.3 *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 [127]. 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.X
```

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.X
```

Adding the patches PUMPKIN found to a hint database made the proof go through:

```
  ... split; intros (c,Hc); exists c; auto.✓
```

5.4.3.1 *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.

5.5 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 5.5.1) and the procedure (Section 5.5.2) 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 5.5.3).

5.5.1 *Inside the Components*

The interested reader can follow along in the repository.

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

5.5.1.1 *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 5.6.2 and 5.7.

6 These trees are inspired by categorical models of dependent type theory [97].

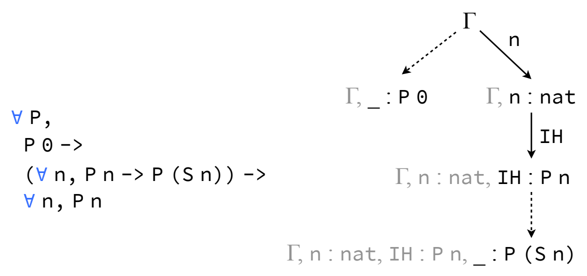


Figure 16: The type of (left) and tree for (right) the induction principle `nat_ind`. The solid edges represent hypotheses, and the dotted edges represent the proof obligations for each case in an inductive proof.

5.5.1.2 *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 [46] 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.

5.5.1.3 *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 5.6.2), 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 5.6.2.

5.5.1.4 *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 5.6.2.

5.5.1.5 *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.

5.5.2 *Inside the Procedure*

The implementation (`patcher.ml4`) of the procedure from Section 5.3.2 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 5.4.1, 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.

5.5.3 *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 [76]. 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 [120]. 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.

5.6 TESTING BOUNDARIES

In the case studies in Section 6.6, 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 5.6.1). 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 5.6.2)

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 5.6.2)

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 5.6.3)

5.6.1 *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 5.4.2. 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.

5.6.2 *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 [46]) `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.

5.6.3 *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.

5.7 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 [30, 148] and type-directed diffing [145] 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 [7]) or continuous integration (CI) system (such as Travis [8]) 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 [102, 17, 40] 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 [5] and Agda [1]). Proof assistants with non-constructive logics (for example, Isabelle/HOL [4]) 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 [6].

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 [2], and adding too many hints may impede performance. More generally, we can integrate PUMPKIN with a transfer tactic [99, 221], 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.

5.8 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 [22] or through an automatic method [132]. This is later generalized and implemented in Isabelle [99] and Coq [221, 196]; 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 [30] adapts proof obligations to structural changes in inductive types. Later work [148] 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 [81, 168, 105] 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` [43] can make proofs more resilient to changes. Hammers like Isabelle's sledgehammer [163] can make proofs agnostic to some low-level changes. Recent work [61] 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 [43, 42] can help prevent low-level maintenance tasks. Writing these tactics requires good engineering [90] 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 [36] 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 [142] may help improve these components. Lean [186] 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 [35] 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 [86] 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 [215] and designing theorems and proofs that make maintenance less of an issue. Design principles for specific domains (such as formal metatheory [19, 72, 74]) can make verification more tractable. CertiKOS [93] introduces the idea of a deep specification to ease verification of large systems. Ornaments [63, 212] 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 [102, 17] and to optimize build performance [40]. 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 [141]. The Haskell refactoring tool HaRe [3] automatically lifts definitions, and may be useful for improving abstraction. There is a growing body of work on refactoring in the context of ITP [206, 32]. The IDE CoqPIE [182] and the verification platform Why3 [29] 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 [145] finds differences in algebraic data types. Semantics-based change impact analysis [18] models semantic differences between documents. Differential assertion checking [121] analyzes different versions of a program for relative correctness with respect to a specification. Incremental λ-calculus [37] 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 [94]. 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 [164, 129, 123, 139, 146] 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.

6

OFTEN, PRACTICALLY

6.1 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 [103] 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" [52] and noted that "almost no one should be using [them] for anything" [53].

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* [136], 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 [213]; 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 [118, 136, 66].

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* [136], 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 6.2), specify (Section 6.3), and formalize (Section 6.4) the search and lifting algorithms that DEVOID implements (Section 6.14). A comparison to a more general proof reuse approach (Section 6.6) demonstrates the benefits of using ornaments: DEVOID imposes less of a proof burden on the user, and produces smaller terms and faster functions.

6.2 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 [192]:

```
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 6.14; 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.

6.3 SPECIFICATION

This section specifies the two commands that Devoid implements:

1. `Find ornament` searches for ornaments (specified in Section 6.3.1, described in Section 6.4.1).

2. `Lift` lifts along those ornaments (specified in Section 6.3.2, described in Section 6.4.2).

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 6.2. Moreover, since the new index is fully determined by `length`, we can relate `length` to `ltv`:

$\forall$ `(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 [202]:

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

that are mutual inverses:[1]

`section    :` $\forall\ (a : A\ \vec{i})$, `forget (promote` $a) = a$.
`retraction :` $\forall\ (b_\Sigma : \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i}))$, `promote (forget` $b_\Sigma) = b_\Sigma$.

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

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

which projects the promoted index:

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

Following existing work [118], 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`.

6.3.1 *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.

6.3.2 *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* [197], a heterogenous parametric relation that strengthens an existing parametric relation for dependent types [25] 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.

6.4 ALGORITHMS

This section describes the algorithms that implement the specifications from Section 6.3.

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$_\omega$ with primitive eliminators; this is a simplified version of the type theory underlying Coq. Figure 34 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

$\langle i \rangle \in \mathbb{N}$, $\langle v \rangle \in$ Vars, $\langle s \rangle \in$ { Prop, Set, Type$\langle i \rangle$ }	$\Gamma \vdash t : T$ `// type checking`
	$\Gamma \vdash t_1 \equiv_{\beta\delta\iota} t_2$ `// definitional equality`
$\langle t \rangle$ ::= $\langle v \rangle$ | $\langle s \rangle$ | Π ($\langle v \rangle$: $\langle t \rangle$) . $\langle t \rangle$ |	t_β `// beta-reduction`
λ ($\langle v \rangle$: $\langle t \rangle$) . $\langle t \rangle$ | $\langle t \rangle$ $\langle t \rangle$ |	$t_{\beta\delta\iota}$ `// normalization`
Ind ($\langle v \rangle$: $\langle t \rangle$){$\langle t \rangle$,...,$\langle t \rangle$} | Constr ($\langle i \rangle$, $\langle t \rangle$) |	$t\ [y\ /\ x]$ `// substitution`
Elim($\langle t \rangle$, $\langle t \rangle$){$\langle t \rangle$,...,$\langle t \rangle$}	$\xi\ (I,\ Q,\ c,\ C)$ `// type of eliminator`

Figure 19: CIC$_\omega$ syntax (left, from existing work [201]) and judgments and operations (right).

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

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

index := insert (off A B)
deindex :=
    remove (off A B)

Figure 20: Common definitions for both algorithms.

$\{t_1, \ldots, t_n\}$ to denote lists of terms, and we use $\vec{i}[j]$ to denote accessing the element of the list $\vec{i}$ at offset j.

Common Definitions. The algorithms assume list insertion and removal functions `insert` and `remove`, plus two functions DEVOID implements: `off` computes the offset of the new index of type I_B in B's indices, and `new` determines whether a hypothesis in a case of the eliminator type of B is new. Figure 20 contains other common definitions, the names for which are reserved: The `index` and `deindex` functions insert an index into and remove an index from a list at the index computed by `off`. Input type A expands to an inductive type with indices of types $\vec{X_A}$, sort s_A, and constructors $\{C_{A_1}, \ldots, C_{A_n}\}$. P_A denotes the type of the motive of the eliminator of A, and each E_{A_i} denotes the type of the eliminator for the ith constructor of A. Analogous names are also reserved for input type B.

6.4.1 *Find ornament*

The `Find ornament` algorithm implements the specification from Section 6.3.1. It builds on three intermediate steps: one to generate each of `indexer`, `promote`, and `forget`. Figure 21 shows the algorithm for generating `indexer`. The algorithms for generating `promote` and `forget` are similar; Figure 22 shows only the derivations for generating `promote` that are different from those for generating `indexer`, and the derivations for generating `forget` are omitted.

6.4.1.1 *Searching for the Indexer*

Search generates the `indexer` by traversing the types of the eliminators for A and B in parallel using the algorithm from Figure 21, which consists of three judgments: one to generate the motive, one to generate each case, and one to compose the motive and cases.

Generating the Motive. The $(T_A, T_B) \Downarrow_{i_m} t$ judgment consists of only the derivation INDEX-MOTIVE, which computes the indexer motive from the types A and B (expanded in Figure 20). It does this by constructing a function with A and its indices as premises, and the type I_B in the conclusion with the appropriate indices. Consider `list` and `vector`:

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

For these types, INDEX-MOTIVE computes the motive:

```
λ (l:list T) . nat
```

Generating Each Case. The $\Gamma \vdash (T_A, T_B) \Downarrow_{i_c} t$ judgment generates each case of the indexer by traversing in parallel the corresponding cases of the eliminator types for A and B. It consists of four derivations: INDEX-CONCLUSION handles base cases and conclusions of inductive cases, while INDEX-HYPOTHESIS, INDEX-IH, and INDEX-PROD recurse into products.

INDEX-HYPOTHESIS handles each new hypothesis that corresponds to a new index in an inductive hypothesis of an inductive case of the eliminator type for B. It adds the new index to the environment, then recurses into the body of only the type for which the index already exists. For example, in the inductive case of `list` and `vector`, `new` determines that `n` is the new hypothesis. INDEX-HYPOTHESIS then recurses into the body of only the `vector` case:

```
Π (tl:T) (l:list T) (IHl:pA l), ...      Π (tv:T) (v:vector T n)
    (IHv:pB n v), ...
```

INDEX-PROD is next. It recurses into product types when the hypothesis is neither a new index nor an inductive hypothesis. Here, it runs twice, recursing into the body and substituting names until it hits the inductive hypothesis for both types:

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

INDEX-MOTIVE

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

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

INDEX-CONCLUSION

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

INDEX-HYPOTHESIS

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

INDEX-IH

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

INDEX-PROD

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

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

INDEX-IND

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

Figure 21: Identifying the indexer function.

Π `(IH`$_l$`:p`$_A$` l), p`$_A$` (cons t`$_l$` l)` Π `(IH`$_v$`:p`$_B$` n l), p`$_B$
    `(S n) (consV n t`$_l$` l)`

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

`p`$_A$` (cons t`$_l$` l)`   `p`$_B$` (S IH`$_l$`) (consV IH`$_l$` t`$_l$` l)`

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

λ `(t`$_l$`:T) (l:list T) (IH`$_l$`:(`λ` (l:list T).nat) l).S IH`$_l$

Composing the Result. The $\Gamma \vdash (T_A,\ T_B) \Downarrow_i t$ judgment consists of only Index-Ind, which identifies the motive and each case using the other two judgments, then composes the result. In the case of `list` and `vector`, this produces a function that computes the length of a list:

λ `(l:list T).Elim(l, `λ` (l:list T).nat)`
  `{0, `λ` (t`$_l$`:T) (l:list T) (IH`$_l$`:(`λ` (l:list T).nat) l).S IH`$_l$`}`

6.4.1.2 *Searching for Promote and Forget*

Figure 22 shows the interesting derivations for the judgment $(T_A,\ T_B) \Downarrow_p t$ that searches for `promote`: Promote-Motive identifies the motive as B with a new index (which it computes using `indexer`, denoted by metavariable π). When Promote-IH recurses, it substitutes the inductive hypothesis for the term rather than for its index, and it substitutes the new index (which it also computes using `indexer`) inside of that term. Promote-Conclusion returns the entire term, rather than its index. Finally, Promote-Ind not only recurses into each case, but also packs the result.

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

6.4.1.3 *Core Search Algorithm*

The core search algorithm produces `indexer`, `promote`, and `forget`, then composes them into a tuple. This tuple is how Devoid represents ornaments internally. Devoid includes an option to generate a proof that these components form the ornamental promotion isomorphism;

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

Promote-Motive

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

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

Promote-Conclusion

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

Promote-IH

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

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

Promote-Ind

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

Figure 22: Identifying the promotion function.

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

Figure 23: Common definitions for the core lifting algorithm.

by default, this is disabled, since `Lift` does not need this proof. The implementation of this option gives intuition for correctness of the search algorithm, and is described in Section 6.5.3.

6.4.2 *Lift*

The `Lift` algorithm implements the specification from Section 6.3.2. We show only one direction of the algorithm, promoting from A to packed B; the forgetful direction is similar. The core algorithm (Figure 37) builds on a set of common definitions (Figure 23) and two intermediate judgments: one to lift eliminators (Figure 24) and one to lift constructors (Figure 25).

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

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

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

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

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

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

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

Figure 24: Lifting eliminators.

Common Definitions. The common definitions (Figure 23) define some useful syntax: $\uparrow$ applies `promote`, $\downarrow$ applies `forget`, and π_{I_B} applies `indexer`. $\exists_{I_B}$ packs a term of type B into an existential with the index at the appropriate offset. $\uparrow_B$ and $\uparrow_{I_B}$ promote and then project; $\downarrow_A$ packs and forgets, and $\downarrow_{I_B}$ packs, forgets, and then applies `indexer` to project the index.

6.4.2.1 *Lifting Eliminators*

The $\Gamma \vdash t \Uparrow_E t'$ judgment (Figure 24) defines rules for lifting the motive and case of an eliminator, changing the *domain of induction* from A to B. The intuition is that any term of type A is the result of forgetting some term of type packed B. Then, since A and B have the same inductive structure, we can lift the eliminator of A to the eliminator of B, and move that forgetfulness *inside of each case*. For example, the following terms are propositionally equal:

```
Elim((↓_A b), p_A){
  f_nil,
  (λ(t_l:T)(l:list T)(IH_l:
      p_A l).
    f_cons t_l l IH_l)
}
```

```
Elim(b, λ(n:nat)(v:vector T n).p_A
    (↓_A v)){
  f_nil,
  (λ(n:nat)(t_v:T)(v:vector T n)(IH_v:p_A
      (↓_A v)).
    f_cons t_v (↓_A v) IH_v)
}
```

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

NORMALIZE

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

Figure 25: Lifting constructors.

The induction rules implement this transformation. CASE lifts a case of the eliminator by first recursively lifting the motive, then using the lifted motive to compute the type of the new case, and then using that type to compute the body of the new case. In the example above, when lifting the inductive case, it first recursively lifts the motive p_A using MOTIVE, which drops the index, packs and forgets the argument of type B, and then β-reduces the result, eliminating references to B. This produces the new motive:

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

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

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

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

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

6.4.2.2 *Lifting Constructors*

The $\Gamma \vdash t \Uparrow_C t'$ judgment (Figure 25) lifts applications of constructors of A to applications of constructors of B. This judgment computes one step of the promotion, leaving the recursive lifting of the arguments to the final algorithm. Using the same types, in the base case:

```
↑ nil ≡_βδι ∃ O nilV
```

and in the inductive case:

```
↑ (cons t l) ≡_βδι ∃ (S (↑_I_B l)) (consV (↑_I_B l) t (↑_B l))
```

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

6.4.2.3 *Core Lifting Algorithm*

The core algorithm (Figure 37) builds on these intermediate judgments. The interesting derivations for correctness are the first six:

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 26: Core lifting algorithm.

Lift-Elim and Lift-Constr use the judgments for lifting eliminators and constructors of A. Internalize internalizes the explicit `promote` functions from the lifted constructors to recursive applications of the algorithm. Retraction and Coherence use the respective properties of the ornamental promotion isomorphism metatheoretically: the first to drop the explicit `forget` functions from the lifted eliminators, and the second to lift the `indexer` to a projection (in the forgetful direction, Section replaces Retraction). Finally, Equivalence lifts A along the equivalence to packed B. The remaining derivations recurse predictably.

6.5 IMPLEMENTATION

The DEVOID Coq plugin implements the algorithms from Section 6.4; the link to the code is in **Supplement Material**. DEVOID cannot produce an ill-typed term, since Coq type checks all terms that plugins produce and rejects ill-typed terms. The implementations of `Find ornament` (`search.ml`) and `Lift` (`lift.ml`) are mostly the same as the algorithms, but with changes to address implementation challenges that scale the algorithms to a Coq tool for proof engineers. This section describes a sample of these changes from each of three categories: addressing differences between Coq and the type theory that the algorithms assume (Section 6.5.1), optimizing for efficiency (Section 6.5.2), and improving usability (Section 6.5.3).

6.5.1 *Addressing Language Differences*

Fixpoints. Coq implements eliminators in terms of pattern matching and fixpoints. To handle terms that use these features, DEVOID includes a `Preprocess` command that translates these terms into equivalent eliminator applications. This command can preprocess a definition (like `zip` from Section 6.2) or an entire module (like `List`, as shown in `ListToVect.v`) for lifting. It currently supports fixpoints that are structurally recursive on only immediate substructures. To translate such a fixpoint, it first extracts a motive, then generates each case by partially reducing the function's body under a hypothetical context for the constructor arguments. This is enough to preprocess `List`; Section 6.8 discusses possible extensions.

Non-Primitive Projections. By default, projections in Coq are non-primitive. That is, this:

```
∀ (T : Type) (v : Σ (n : nat).vector T n), v = ∃ (π_l v) (π_r v).
```

cannot be proven by reflexivity alone (see `Projections.v`). Therefore, DEVOID must pack terms like `v` into existentials; otherwise, lifting will sometimes fail. This is why the type of `zip_with_is_zipV_p` in the example from Section 6.2 packs v_1 and v_2. For the sake of performance and readability of lifted code, DEVOID is strategic about when it packs.

Constants. Because Coq has constants, the implementation of NORMALIZE refolds [33] after normalizing. That is, it acts like the `simpl` tactic in Coq, but with special support for sigma types. For example, to lift the `cons` constructor of a list, after normalizing the promotion of `cons t l`, DEVOID substitutes the projections of the promotion of `l` for their normal forms, which determines and saves the following fact:

```
∀ {T} (l : list T), ↑ (cons t l) = ∃ (S (↑_{I_B} l)) (consV (↑_{I_B} l)
    t (↑_B l)).
```

Refolding helps produce more readable lifted code. It also improves lifting performance, since it occurs just once for each constructor.

6.5.2 *Optimizing for Efficiency*

Delayed Reduction. When lifting eliminators, DEVOID computes a list of arguments and delays reduction. It computes this list backwards, storing the new indices that inductive hypotheses refer to as it recurses. This removes the call to `new` in the premise of DROP-INDEX.

Lazy η-Expansion. The lifting algorithm assumes that all terms are fully η-expanded. Sometimes, however, η-expansion is not necessary. For efficiency, rather than fully η-expand ahead of time, DEVOID η-expands lazily, only when it is necessary for correctness.

Caching. To prevent extra recursion, DEVOID caches the outputs of search, as well as lifted constants, inductive types, and constructors. Since these are constants, lookup is low-cost.

6.5.3 *Improving Usability*

Correctness Proofs. DEVOID has options (used in `Example.v`) that tell search to generate proofs that its outputs are correct, thereby increasing confidence in and usefulness of those outputs. The proof of `coherence` is reflexivity. The intuition behind the automation to prove `section` and `retraction` (`equivalence.ml`) is that `promote` and `forget` map along corresponding constructors, so inductive cases preserve equalities. Thus, each inductive case of these proofs is generated by a fold that rewrites each recursive reference, with reflexivity as identity.

Unpacking. DEVOID includes an `Unpack` command (used in `Example.v`) that unpacks packed types in functions and proofs. This way, users may access unpacked terms without writing boilerplate code. For simple functions, this command packs arguments and projects results. It splits higher-order functions into two functions. For proofs that use equality, it applies one lemma convert to dependent equality, and one lemma to deal with non-primitive projections.

User-Friendly Types. `Example.v` describes how the user can recover user-friendly types after unpacking. For example, to recover a function with an output of type `vector T n`, the user lifts a proof that the length of the output of the unlifted `list` version of that function is `n`, then rewrites by that lifted proof. The intuition behind this is that this equivalence holds:

```
{ l : list T & length l = n } ≃ vector T n
```

Recovering a user-friendly type for a proof relating these functions is more complex, since it necessitates reasoning at some point about equalities between equalities. For some index types like `nat`, this follows simply from the fact that the type forms an h-set [202]: all proofs of equality between the same two terms of that type are equal. There is preliminary work on determining a general methodology for deriving user-friendly types for proofs that does not rely on any properties of the index type. The idea is to use the adjunction condition along with the proof of `coherence` by reflexivity; see GitHub issue #39 for the status of this work.

6.6 CASE STUDY

We used Devoid to automatically discover and lift along ornaments for two scenarios:

1. Single Iteration: from binary trees to sized binary trees
2. Multiple Iterations: from binary trees to binary search trees to AVL trees

For comparison, we also used the ornaments that Devoid discovered to lift functions and proofs using *Equivalences for Free!* [197] (EFF), a more general framework for lifting across equivalences. Devoid produced faster functions and smaller terms, especially when composing multiple iterations of lifting. In addition, Devoid imposed little burden on the user, and the ornaments Devoid discovered proved useful to EFF.

We chose EFF for comparison because Devoid is the only tool for ornaments in Coq, and because doing so demonstrates the benefits of specialized automation for ornaments. Devoid can handle only a small class of equivalences compared to EFF, and it can currently handle only incremental changes to types (one new index at a time). Our experiences suggest that it is possible to use both tools in concert. Section 6.16 discusses EFF in more detail.

Setup. The case study code is in the `eval` folder of the repository. For each scenario, we ran Devoid to search for an ornament, and then lifted functions and proofs along that ornament using both Devoid and EFF. We noted the amount of user interaction (Section 6.6.1), as well as the performance of lifted terms (Section 6.6.2). To test the performance of lifted terms, we tested runtime by taking the median of ten runs using `Time Eval vm_compute` with test values in Coq 8.8.0, and we tested size by normalizing and running `coqwc` on the result.[2]

In the first scenario, we lifted traversal functions along with proofs that their outputs are permutations of each other from binary trees

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

(`tree`) to sized binary trees (`Sized.tree`). In the second scenario, we lifted the traversal functions to AVL trees (`avl`) through four intermediate types (one for each new index), and we lifted a search function from BSTs (`bst`) to AVL trees through one intermediate type. Both scenarios considered only full binary trees.

To fit `bst` and `avl` into algebraic ornaments for DEVOID, we used boolean indices to track invariants. While the resulting types are not the most natural definitions, this scenario demonstrates that it is possible to express interesting changes to structured types as algebraic ornaments, and that lifting across these types in DEVOID produces efficient functions.

6.6.1 *User Experience*

For each intermediate type in each scenario, we used DEVOID to discover the components of the equivalence. These components were enough for DEVOID to lift functions and proofs with no additional proof burden and no additional axioms. To use EFF, we also had to prove that these components form an equivalence; we set the appropriate option to generate these proofs using DEVOID. In addition, to use EFF, we had to prove univalent parametricity of each inductive type; these proofs were small, but required specialized knowledge. To lift the proof of the theorem `pre_permutes` using EFF, we had to prove the univalent parametric relation between the unlifted and lifted versions of the functions that the theorem referenced; this pulled in the functional extensionality axiom, which was not necessary using DEVOID.

In the second scenario, to simulate the incremental workflow DEVOID requires, we lifted to each intermediate type, then unpacked the result. For example, the ornament from `bst` to `avl` passed through an intermediate type; we lifted `search` to this type first, unpacked the result, and then repeated this process. In this scenario, using EFF differently could have saved some work relative to DEVOID, since with EFF, it is possible to skip the intermediate type;[3] DEVOID is best fit where an incremental workflow is desirable.

6.6.2 *Performance*

Relative to EFF, DEVOID produced faster functions. Figure 27 summarizes runtime in the first scenario for `preorder`, and Figure 28 summarizes runtime in the second scenario for `preorder` and `search`.

3 The performances of the terms that EFF produces are sensitive to the equivalence used; for a 100 node tree, this alternate workflow produced a search function which is hundreds of times slower and traversal functions which are thousands of times slower than the functions that DEVOID produced. In addition, the lifted proof of `pre_permutes` using EFF failed to normalize with a timeout of one hour.

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Devoid	0.0	0.0	0.0	3.0 (1.00x)	35.0 (0.95x)
	EFF	0.0	1.0	27.0	486.5 (162.17x)	8078.5 (218.33x)

Figure 27: Median runtime (ms) of unlifted (`tree`) and lifted (`Sized.tree`) `preorder` over ten runs with test inputs ranging from about 10 to about 10000 nodes.

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Devoid	71.5	71.0	69.0	75.0 (25.00x)	109.0 (2.95x)
	EFF	1.0	11.0	152.0	2976.5 (992.17x)	56636.5 (1530.72x)
`search`	**Unlifted**	0.0	0.0	2.0 (1.00x)	3.0 (1.00x)	29.0 (1.00x)
	Devoid	12.0	14.0	12.0 (6.00x)	15.0 (5.00x)	50.0 (1.72x)
	EFF	1.0	5.0	67.0 (33.50x)	1062.0 (354.00x)	15370.5 (530.02x)

Figure 28: Median runtime (ms) of unlifted (`tree`) and lifted (`avl`) `preorder`, plus unlifted (`bst`) and lifted (`avl`) `search`, over ten runs with inputs ranging from about 10 to about 100000 nodes.

The `inorder` and `postorder` functions performed similarly to `preorder`. The functions DEVOID produced imposed modest overhead for smaller inputs, but were tens to hundreds of times faster than the functions that EFF produced for larger inputs. This performance gap was more pronounced over multiple iterations of lifting.

DEVOID also produced smaller terms: in the first scenario, 13 vs. 25 LOC for `preorder`, 12 vs. 24 LOC for `inorder`, and 17 vs. 29 LOC for `postorder`; and in the second scenario, 21 vs. 120 LOC for `preorder`, 20 vs. 119 LOC for `inorder`, 24 vs. 125 LOC for `postorder`, and 31 vs. 52 LOC for `search`. In the first scenario, the lifted proof of `pre_permutes` using DEVOID was 85 LOC; the lifted proof of `pre_permutes` using EFF was 1463184 LOC.

We suspect DEVOID provided these performance benefits because it directly lifted induction principles, whereas EFF produced lifted functions in terms of unlifted functions. The multiple iteration case in particular highlights this, since EFF's approach makes lifted terms much slower and larger as the number of iterations increases, while DEVOID's approach does not.

6.7 RELATED WORK

Ornaments. DEVOID automates discovery of and lifting across algebraic ornaments in a higher-order dependently typed language. In the decade since the discovery of ornaments [136], there have been a number of formalizations and embedded implementations of ornaments [65, 117, 66, 118, 64]. DEVOID is the first tool for ornamentation to operate over a non-embedded dependently typed language. It essentially moves the automation-heavy approach of Ornamentation in ML [213], which operates on non-embedded ML code, into the type theory that forms the basis of theorem provers like Coq. In doing so, it takes advantage of the properties of algebraic ornaments [136]. It also introduces the first search algorithm to identify ornaments, which in the past was identified as a "gap" in the literature [118].

Lifting Proofs. DEVOID identifies and lifts proofs along a specific equivalence similar to that from existing ornaments work [118]. The need to automatically lift functions and proofs across equivalences and other relations is a long-standing challenge for proof engineers [131, 23, 130, 100, 222, 51]. The univalence axiom from Homotopy Type Theory [202] enables transparent transport of proofs; cubical type theory [49] gives univalence a constructive interpretation.

Our work is closely related to *Equivalences for Free!* [197], which brings this full circle, using mathematical properties of univalence to enable lifting across equivalences in a substantial subset of CIC_ω without relying on the univalence axiom. In doing so, it introduces and formalizes the relation that our specification depends upon, and implements a framework for lifting in Coq. This framework is more general than DEVOID: It lifts along any equivalence, not just ornamental promotions, and can handle opaque terms, with the caveat that users must prove each equivalence themselves; DEVOID requires non-opaque terms and lifts along the class of equivalences that correspond to ornamental promotions, taking advantage of the mathematical properties of ornaments to eliminate the need for explicit applications of section and retraction, and to discover and prove certain equivalences automatically. These mathematical properties allow us to automatically lift the induction principle and eliminate references to old terms, which is beneficial for performance.

Similarly, our work is related to CoqEAL [51], which transfers functions along arbitrary relations between types. As these relations do not necessarily need to be equivalences, this framework is more general than our work. Similar tradeoffs between automation and generality apply: CoqEAL produces functions that refer to the old type, and does not yet support automatic inference of relations. In addition, CoqEAL currently only supports automatic transfer of functions, and does not yet handle proofs.

These tools may provide an alternative backend for DEVOID. Furthermore, our search algorithm may help discover relations that make these tools easier to use, and our lifting algorithm may help improve automation and efficiency for certain relations in these tools.

Program and Proof Reuse. The problem that we solve is fundamentally about proof reuse, which applies software reuse principles to ITPs. There is a wealth of work in proof reuse, from tactic languages [82] and logical frameworks [38], to tools for proof abstraction and generalization [169, 106], to domain-specific methodologies [71] and frameworks [73].

DEVOID focuses on the specific problem of reuse when adding fully-determined indices to types. Other approaches to this problem include combinators which definitionally reduce to desirable terms [77] in the language Cedille, and automatic generation of conversion functions in Ghostbuster [137] for GADTs in Haskell. Our work focuses on a type theory different from both of these, in which the properties that allow for such combinators in Cedille are not present, and in which dependent types introduce challenges not present in Haskell.

DEVOID is not the first tool to combine search with reuse. Optician [143] synthesizes bidirectional string transformations; a similar approach may help extend tooling to handle transformations for low-level data. PUMPKIN PATCH [176] searches the difference in proofs for patches that can be used to repair proofs broken by changes; DEVOID uses a similar approach to identify functions that form an equivalence. The resulting tools are complementary: DEVOID supports the addition of indices and hypotheses, which PUMPKIN PATCH does not support; PUMPKIN PATCH supports changes in values, which DEVOID does not support.

6.8 CONCLUSIONS & FUTURE WORK

We presented DEVOID: a tool for searching for and lifting across algebraic ornaments in Coq. DEVOID is the first tool to lift across ornaments in a non-embedded dependently typed language, and to automatically infer certain kinds of ornaments from types alone. Our algorithms give efficient transport across equivalences arising from algebraic ornaments; our case study demonstrates that such automation can make lifted terms smaller and faster as part of an incremental workflow.

Future Work. A future version may support other ornaments beyond algebraic ornaments, with additional user interaction as needed; this may help support, for example, the ornament between `nat` and `list`, where `list` has a new element in the `cons` case. A future version may loosen restrictions on input types to support adding constructors while preserving inductive structure, recursive references under products, and coinductive types. Integrating with PUMPKIN PATCH [176] may help remove restrictions DEVOID makes about the hypotheses of B. `Preprocess` currently supports only certain fixpoints; a more general translation may help DEVOID support more terms, and discussions with Coq developers suggest that the implementation of such a translation building on work from the equations [189] plugin is in progress. Extending DEVOID to generate proofs of coherence conditions for lifted terms may increase user confidence. Proofs that the commands that

Devoid implements satisfy their specifications may also increase user confidence. Better automating the recovery of user-friendly types may improve user experience.

6.9 introduction

Program verification with interactive theorem provers has come a long way since its inception, especially when it comes to the scale of programs that can be verified. The seL4 [114] verified operating system kernel, for example, is the effort of a team of proof engineers spanning more than a million lines of proof, costing over 20 person-years. Given a famous 1977 critique of verification [75] (emphasis ours):

> *A sufficiently fanatical researcher* might be willing to devote *two or three years* to verifying a significant piece of software if he could be assured that the software would remain stable.

we could argue that, over 40 years, either verification has become easier, or researchers have become more fanatical. Unfortunately, not all has changed (emphasis still ours):

> But real-life programs need to be maintained and modified. There is *no reason to believe* that verifying a modified program is any easier than verifying the original the first time around.

Tools that can automatically refactor or repair proofs [208, 207, 78, 10, 31, 182, 179, 177] give us reason to believe that verifying a modified program *can* sometimes be easier than verifying the original, even when proof engineers do not follow good development processes, or when change occurs outside of proof engineers' control [173]. Still, maintaining verified programs can be challenging: it means keeping not just the programs, but also specifications and proofs about those programs up-to-date. This remains so difficult that sometimes, even experts give up in the face of change [174].

The problem of automatically updating proofs in response to changes in programs or specifications is known as *proof repair* [173, 177]. While there are many ways proofs need to be repaired, one such need is in response to a changed type definition (Section 6.11). We make progress on two open challenges in proof repair in response to changes in type definitions:

1. Existing work supports very limited classes of these changes like non-structural changes [177] or a predefined set of changes [179, 208], and these are not informed by the needs of proof engineers [174].

2. Proof repair tools are not yet integrated with typical proof engineering workflows like tactics [173, 177, 179], and may impose additional proof obligations like proving relations corresponding to changes [178].

Our approach combines a configurable proof term transformation (Section 6.12) with a prototype decompiler from proof terms back to suggested tactics (Section 6.13). This is implemented (Section 6.14) in Pumpkin Pi, an extension to the Pumpkin Patch [177] proof repair plugin suite for Coq 8.8 that is available on Github.[4]

Addressing Challenge 1: Flexible Type Support. The case studies in Section 6.15—summarized in Table 1 on page 111—show that Pumpkin Pi is flexible enough to support a wide range of proof engineering use cases. In general, Pumpkin Pi can support any change described by an equivalence, though it takes the equivalence in a deconstructed form that we call a *configuration*. The configuration expresses to the proof term transformation how to translate functions and proofs defined over the old version of a type to refer only to the new version, and how to do so in a way that does not break definitional equality. The proof engineer can write this configuration in Coq and feed it to Pumpkin Pi (*manual configuration* in Table 1), configuring Pumpkin Pi to support the change.

Addressing Challenge 2: Workflow Integration. Research on workflow integration for proof repair tools is in its infancy. Pumpkin Pi is built with workflow integration in mind. For example, Pumpkin Pi is the only proof repair tool we are aware of that produces suggested tactic scripts for repaired proofs, a challenge highlighted in existing proof repair work [177, 179] and in a survey of proof engineering [173]. In addition, Pumpkin Pi implements search procedures that automatically discover configurations and prove the equivalences they induce for four different classes of changes (*automatic configuration* in Table 1), decreasing the burden of proof obligations imposed on the proof engineer. Our partnership with an industrial proof engineer has informed other changes to further improve workflow integration (Sections 6.14.1 and 6.15).

Bringing it Together. Figure 29 shows how this comes together when the proof engineer invokes Pumpkin Pi:

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

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

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

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

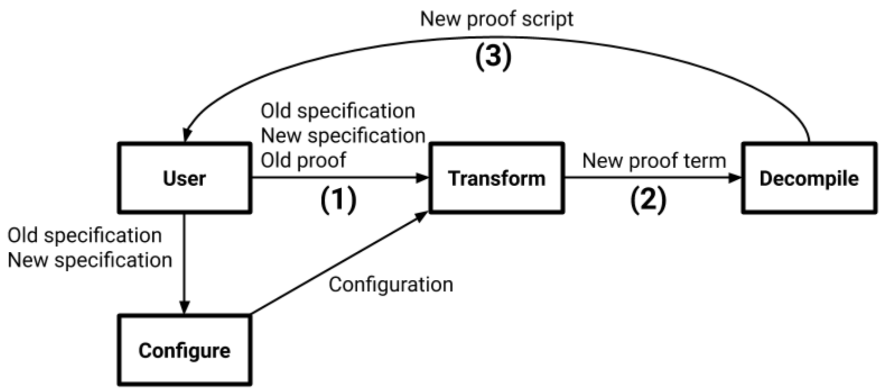


Figure 29: The workflow for Pumpkin Pi.

There are currently four search procedures for automatic configuration implemented in Pumpkin Pi (see Table 1 on page 111). Manual configuration makes it possible for the proof engineer to configure the transformation to any equivalence, even without a search procedure. Section 6.15 shows examples of both workflows applied to real scenarios.

6.10 A SIMPLE MOTIVATING EXAMPLE

Consider a simple example of using Pumpkin Pi: repairing proofs after swapping the list constructors (Figure 30). This is inspired by a similar change from a user study of proof engineers (Section 6.15). Even such a simple change can cause trouble, as in this proof from the Coq standard library:[5]

```
Lemma rev_app_distr {A} :
  ∀ (x y : list A), rev (x ++ y) = rev y ++ rev x.
Proof.
  induction x as [| a l IHl].
  induction y as [| a l IHl].
  simpl. auto.
  simpl. rewrite app_nil_r; auto.
  intro y. simpl.
  rewrite (IHl y). rewrite app_assoc; trivial.
Qed.
```

This theorem says that appending (++) two lists and reversing (`rev`) the result behaves the same as appending the reverse of the second list onto the reverse of the first list. When we change the `list` type, the proof no longer works. To repair this proof with Pumpkin Pi, we run this command:

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

assuming the old and new list types from Figure 30 are in modules `Old` and `New`. This suggests a proof script that succeeds (in light blue to denote Pumpkin Pi produces it automatically):

```
Proof.
  intros x. induction x as [a l IHl| ]; intro y0.
```

5 We use induction instead of pattern matching.

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

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

Figure 30: The updated `list` (bottom) is the old `list` (top) with its two constructors swapped (orange).

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

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

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

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

Figure 31: Two functions between `Old.list` and `New.list` (top) that form an equivalence (bottom).

```
  - simpl. rewrite IHl. simpl.
    rewrite app_assoc. auto.
  - induction y0 as [a l H| ].
    + simpl. rewrite app_nil_r. auto.
    + auto.
Qed.
```

where the dependencies (`rev`, `++`, `app_assoc`, and `app_nil_r`) have also been updated automatically ①. If we would like, we can manually modify this to something that more closely matches the style of the original proof:

```
Proof.
  induction x as [a l IHl|].
  intro y. simpl.
  rewrite (IHl y). rewrite app_assoc; trivial.
  induction y as [a l IHl|].
  simpl. rewrite app_nil_r; auto.
  simpl. auto.
Qed.
```

We can even repair the entire list module from the Coq standard library all at once by running the `Repair module` command ①. When we are done, we can get rid of `Old.list`.

The key to success is taking advantage of Coq's structured proof term language: Coq compiles every proof script to a proof term in a language called Gallina that is based on the calculus of inductive constructions—Pumpkin Pi repairs that term. Pumpkin Pi then decompiles the repaired proof term (with optional hints from the original proof script) back to a proof script that the proof engineer can maintain.

In contrast, updating the poorly structured proof script directly would not be straightforward. Even for the simple proof script above, grouping tactics by line, there are $6! = 720$ permutations of this proof script. It is not clear which lines to swap since these tactics do not have a semantics beyond the searches their evaluation performs. Furthermore, just swapping lines is not enough: even for such a simple change, we must also swap arguments, so `induction x as [| a l IHl ]` becomes `induction x as [a l IHl|]`. Valentin Robert's thesis [179] describes the challenges of repairing tactics in detail. Pumpkin Pi's approach circumvents this challenge.

6.11 problem definition

Pumpkin Pi can do much more than permute constructors. Given an equivalence between types $A$and B, Pumpkin Pi repairs functions and proofs defined over $A$to instead refer to B(Section 6.11.1). It does this in a way that allows for removing references to A, which is essential for proof repair, since Amay be an old version of an updated type (Section 6.11.2).

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

Figure 32: `J` is `I` with `A` and `B` factored out to `bool` (orange).

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

Figure 33: A vector is a list indexed by its length (orange).

6.11.1 *Scope: Type Equivalences*

Pumpkin Pi repairs proofs in response to changes in types that correspond to *type equivalences* [202], or pairs of functions that map between two types and are mutual inverses.[6] When a type equivalence between types $A$and Bexists, those types are *equivalent* (denoted $A\simeq B$). Figure 31 shows a type equivalence between the two versions of `list` from Figure 30 that Pumpkin Pi discovered and proved automatically ①.

To give some intuition for what kinds of changes can be described by equivalences, we preview two changes below. See Table 1 on page 111 for more examples.

Factoring out Constructors. Consider changing the type `I` to the type `J` in Figure 32. `J` can be viewed as `I` with its two constructors `A` and `B` pulled out to a new argument of type `bool` for a single constructor. With Pumpkin Pi, the proof engineer can repair functions and proofs about `I` to instead use `J`, as long as she configures Pumpkin Pi to describe which constructor of `I` maps to `true` and which maps to `false`. This information about constructor mappings induces an equivalence `I` $\simeq$ `J` across which Pumpkin Pi repairs functions and proofs. File ② shows an example of this, mapping `A` to `true` and `B` to false, and repairing proofs of De Morgan's laws.

Adding a Dependent Index. At first glance, the word *equivalence* may seem to imply that Pumpkin Pi can support only changes in which

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

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

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

Figure 34: Syntax [201] for CIC$_\omega$ from with (from left to right) variables, sorts, dependent types, functions, application, inductive types, inductive constructors, and primitive eliminators.

the proof engineer does not add or remove information. But equivalences are more powerful than they may seem. Consider, for example, changing a list to a length-indexed vector (Figure 33). Pumpkin Pi can repair functions and proofs about lists to functions and proofs about vectors of particular lengths ③, since `Σ(l:list T).length l = n` $\simeq$ `vector T n`. From the proof engineer's perspective, after updating specifications from `list` to `vector`, to fix her functions and proofs, she must additionally prove invariants about the lengths of her lists. Pumpkin Pi makes it easy to separate out that proof obligation, then automates the rest.

More generally, in homotopy type theory, with the help of quotient types, it is possible to form an equivalence from a relation, even when the relation is not an equivalence [13]. While Coq lacks quotient types, it is possible to achieve a similar outcome and use Pumpkin Pi for changes that add or remove information when those changes can be expressed as equivalences between Σ types or sum types.

6.11.2 *Goal: Transport with a Twist*

The goal of Pumpkin Pi is to implement a kind of proof reuse known as *transport* [202], but in a way that is suitable for repair. Informally, transport takes a term t and produces a term t' that is the same as t modulo an equivalence $A \simeq B$. If t is a function, then t' behaves the same way modulo the equivalence; if t is a proof, then t' proves the same theorem the same way modulo the equivalence.

When transport across $A \simeq B$ takes t to t' , we say that t and t' are *equal up to transport* across that equivalence (denoted $t \equiv_{A \simeq B} t'$).[7] In Section 6.10, the original append function ++ over `Old.list` and the repaired append function ++ over `New.list` that Pumpkin Pi produces are equal up to transport across the equivalence from Figure 31, since (by `app_ok` ①):

```
∀ T (l1 l2 : Old.list T),
  swap T (l1 ++ l2) = (swap T l1) ++ (swap T l2).
```

The original `rev_app_distr` is equal to the repaired proof up to transport, since both prove the same thing the same way up to the equivalence, and up to the changes in ++ and `rev`.

Transport typically works by applying the functions that make up the equivalence to convert inputs and outputs between types. This approach would not be suitable for repair, since it does not make it possible to remove the old type A. Pumpkin Pi implements transport in a way that allows for removing references to A—by proof term transformation.

7 This notation should be interpreted in a metatheory with *univalence*—a property that Coq lacks—or it should be approximated in Coq. The details of transport with univalence are in the HoTT book [202], and an approximation in Coq is in a recent paper [197]. For equivalent $A$and B, there can be many equivalences $A \simeq B$. Equality up to transport is across a *particular* equivalence, but we erase this in the notation.

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

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

Figure 35: The dependent constructors and eliminators for old (left) and new (right) `list`, with the difference in orange.

6.12 THE TRANSFORMATION

At the heart of Pumpkin Pi is a configurable proof term transformation for transporting proofs across equivalences ④. It is a generalization of the transformation from an earlier version of Pumpkin Pi called Devoid [178], which solved this problem a particular class of equivalences.

The transformation takes as input a deconstructed equivalence that we call a *configuration*. This section introduces the configuration (Section 6.12.1), defines the transformation that builds on that (Section 6.12.2), then specifies correctness criteria for the configuration (Section 6.12.3). Section 6.14.1 describes the additional work needed to implement this transformation.

Conventions. All terms that we introduce in this section are in CIC$_\omega$ [59, 60] (the core calculus of Coq) with primitive eliminators, the syntax for which is in Figure 34. The typing rules are standard. We assume inductive types Σ with constructor $\exists$ and projections π_l and π_r, and = with constructor `eq_refl`. We use $\vec{t}$ and $\{t_1, \ldots, t_n\}$ to denote lists of terms.

6.12.1 *The Configuration*

The configuration is the key to building a proof term transformation that implements transport in a way that is suitable for repair. Each configuration corresponds to an equivalence $A \simeq B$. It deconstructs the equivalence into things that talk about A, and things that talk about B. It does so in a way that hides details specific to the equivalence, like the order or number of arguments to an induction principle or type.

At a high level, the configuration helps the transformation achieve two goals: preserve equality up to transport across the equivalence between $A$and B, and produce well-typed terms. This configuration is a pair of pairs:

```
((DepConstr, DepElim), (Eta, Iota))
```

each of which corresponds to one of the two goals: `DepConstr` and `DepElim` define how to transform constructors and eliminators, thereby preserving the equivalence, and `Eta` and `Iota` define how to transform η-expansion and ι-reduction of constructors and eliminators, thereby producing well-typed terms. Each of these is defined in CIC$_\omega$ for each equivalence.

Preserving the Equivalence. To preserve the equivalence, the configuration ports terms over $A$to terms over $B$by viewing each term of type Bas if it is an A. This way, the rest of the transformation can replace values of $A$with values of B, and inductive proofs about $A$with inductive proofs about B, all without changing the order or number of arguments.

The two configuration parts responsible for this are `DepConstr` and `DepElim` (*dependent constructors* and *eliminators*). These describe how to construct and eliminate $A$and B, wrapping the types with a common inductive structure. The transformation requires the same number of dependent constructors and cases in dependent eliminators for $A$and B, even if $A$and $B$are types with different numbers of constructors ($A$and Bneed not even be inductive; see Sections 6.12.3 and 6.15).

For the `list` change from Section 6.10, the configuration that Pumpkin Pi discovers uses the dependent constructors and eliminators in Figure 35. The dependent constructors for `Old.list` are the normal constructors with the order unchanged, while the dependent constructors for `New.list` swap constructors back to the original order. Similarly, the dependent eliminator for `Old.list` is the normal eliminator for `Old.list`, while the dependent eliminator for `New.list` swaps cases.

As the name hints, these constructors and eliminators can be dependent. Consider the type of vectors of some length:

```
packed_vect T := Σ(n : nat).vector T n.
```

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

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

and the eliminator it discovers eliminates the projections:

```
DepElim(s, P) { f0 f1 } : P (∃ (πl s) (πr s)) :=
  Elim(πr s, λ(n : nat)(v : vector T n).P (∃ n v)) {
    f0,
    (λ(t : T)(n : nat)(v : vector T n).f1 t (∃ n v))
  }.
```

In both these examples, the interesting work moves into the configuration: the configuration for the first swaps constructors and cases, and the configuration for the second maps constructors and cases over `list` to constructors and cases over `packed_vect`. That way, the transformation need not add, drop, or reorder arguments. Furthermore, both examples use automatic configuration, so Pumpkin Pi's **Configure** component discovers `DepConstr` and `DepElim` from just the types $A$and B, taking care of even the difficult work.

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

Inductive nat :=              Inductive N :=
| O : nat                     | N0 : N
| S : nat → nat.              | Npos : positive → N.
```

Figure 36: Unary (left) and binary (right) natural numbers.

Producing Well-Typed Terms. The other configuration parts `Eta` and `Iota` deal with producing well-typed terms, in particular by transporting equalities. A naive proof term transformation may fail to generate well-typed terms if it does not consider the problem of transporting equalities. Otherwise, if the transformation transforms a term `t : T` to some `t' : T'`, it does not necessarily transform `T` to `T'` [198].

`Eta` and `Iota` describe how to transport equalities. More formally, they define η-expansion and ι-reduction of $A$and B, which may be propositional rather than definitional, and so must be explicit in the transformation. η-expansion describes how to expand a term to apply a constructor to an eliminator in a way that preserves propositional equality, and is important for defining dependent eliminators [159]. ι-reduction (β-reduction for inductive types) describes how to reduce an elimination of a constructor [158].

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

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

which is propositional and not definitional in Coq. Thanks to this, we can forego the assumption that our language has primitive projections (definitional η for Σ).

Each `Iota`—one per constructor—describes and proves the ι-reduction behavior of `DepElim` on the corresponding case. This is needed, for example, to port proofs about unary numbers `nat` to proofs about binary numbers `N` (Figure 36). While we can define `DepConstr` and `DepElim` to induce an equivalence between them ⑤, we run into trouble reasoning about applications of `DepElim`, since proofs about `nat` that hold by reflexivity do not necessarily hold by reflexivity over `N`. For example, in Coq, while `S (n + m) = S n + m` holds by reflexivity over `nat`, when we define `+` with `DepElim` over `N`, the corresponding theorem over `N` does not hold by reflexivity.

To transform proofs about `nat` to proofs about `N`, we must transform *definitional* ι-reduction over `nat` to *propositional* ι-reduction over `N`. For our choice of `DepConstr` and `DepElim`, ι-reduction is definitional over `nat`, since a proof of:

```
∀ P p0 ps n,
  DepElim(DepConstr(1, nat) n, P) { p0, ps } =
  ps n (DepElim(n, P) { p0, ps }).
```

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

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

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

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

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

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

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

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

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

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

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

PROD
$$\frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \Pi(t : T).b \Uparrow \Pi(t' : T').b'}$$

VAR
$$\frac{v \in \mathrm{Vars}}{\Gamma \vdash v \Uparrow v}$$

Figure 37: Transformation for transporting terms across $A \simeq B$ with configuration `((DepConstr, DepElim), (Eta, Iota))`.

```
(* 1: original term *)
λ (T : Type) (l m : list T) .
 Elim (l, λ(l: Old.list T).list
      T → list T)) {
   (λ m . m),
   (λ t _ IHl m . Constr(1, Old.
       list T) t (IHl m))
 } m.
```

```
(* 4: reduced to final term *)
λ (T : Type) (l m : list T) .
 Elim (l, λ(l: New.list T).list
      T → list T)) {
   (λ t _ IHl m . Constr(0, New.
       list T) t (IHl m)),
   (λ m . m)
 } m.
```

```
(* 2: after unifying with
    configuration *)
λ (T : Type) (l m : A) .
 DepElim (l, λ(l: A).A → A)) {
   (λ m . m)
   (λ t _ IHl m . DepConstr(1,
       A) t (IHl m))
 } m.
```

```
(* 3: after transforming *)
λ (T : Type) (l m : B) .
 DepElim (l, λ(l: B).B → B)) {
   (λ m . m)
   (λ t _ IHl m . DepConstr(1,
       B) t (IHl m))
 } m.
```

Figure 38: Swapping cases of the append function, with names fully qualified only when needed for clarity, counterclockwise, the input term: 1) unmodified, 2) unified with the configuration, 3) ported to the updated type, and 4) reduced to the output.

holds by reflexivity. But ι for `N` is propositional, since the corresponding theorem over `N` does not. `Iota` for `nat` in the `S` case is a rewrite by the proof of the above, with type:

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

and similarly for `N` (5). The transformation replaces rewrites by reflexivity over `nat` to rewrites by propositional equalities over `N`, so that `DepElim` behaves the same over `nat` and `N`.

Taken together over both $A$and B, `Iota` describes how the inductive structures of $A$and Bdiffer. The transformation requires that `DepElim` over $A$and over Bhave the same structure as each other, so if $A$and B*themselves* have the same inductive structure (if they are *ornaments* [136]), then if ι is definitional for A, it will be possible to choose `DepElim` with definitional ι for B. Otherwise, if $A$and B(like `nat` and `N`) have different inductive structures, then definitional ι over one would become propositional ι over the other.

6.12.2 *The Proof Term Transformation*

Figure 37 shows the proof term transformation $\Gamma \vdash t \Uparrow t'$ that forms the core of Pumpkin Pi. The transformation is parameterized over equivalent types $A$and B(Equivalence) as well as the configuration. It assumes η-expanded functions. It implicitly constructs an updated context Γ' in which to interpret t', but this is not needed for computation.

section: $\forall$ (a : A), g (f a)
  = a.
retraction: $\forall$ (b : B), f (g
  b) = b.

constr_ok: $\forall$ j $\vec{x_A}$ $\vec{x_B}$, $\vec{x_A}$
  $\equiv_{A \simeq B}$ $\vec{x_B}$ $\rightarrow$
 DepConstr(j, A) $\vec{x_A}$ $\equiv_{A \simeq B}$
  DepConstr(j, B) $\vec{x_B}$.

elim_ok: $\forall$ a b P$_A$ P$_B$ $\vec{f_A}$ $\vec{f_B}$,
 a $\equiv_{A \simeq B}$ b $\rightarrow$
 P$_A$ $\equiv_{(A \rightarrow s) \simeq (B \rightarrow s)}$ P$_B$ $\rightarrow$
 $\forall$ j, $\vec{f_A}$[j] $\equiv_{\xi(A, P_A, j) \simeq \xi(B, P_B, j)}$
  $\vec{f_B}$[j]$\rightarrow$
 DepElim(a, P$_A$) $\vec{f_A}$
  $\equiv_{(Pa) \simeq (Pb)}$ DepElim(b, P
  $_B$) $\vec{f_A}$.

elim_eta(A): $\forall$ a P $\vec{f}$, DepElim(a, P)
  $\vec{f}$: P (Eta(A) a).
eta_ok(A): $\forall$ (a : A), Eta(A) a = a.

iota_ok(A): $\forall$ j P $\vec{f}$ $\vec{x}$ (Q: P(Eta(A)
  (DepConstr(j, A) $\vec{x}$)) $\rightarrow$ s),
 Iota(A, j, Q) :
  Q (DepElim(DepConstr(j, A) $\vec{x}$, P)
   $\vec{f}$) $\rightarrow$
  Q (rew $\leftarrow$ eta_ok(A) (DepConstr(j
   , A) $\vec{x}$) in
   ($\vec{f}$[j]...(DepElim(IH$_0$, P) $\vec{f}$)...(
    DepElim(IH$_n$, P) $\vec{f}$)...)).

Figure 39: Correctness criteria for a configuration to ensure that the transformation preserves equivalence (left) coherently with equality (right, shown for A; $B$is similar). f and g are defined in text. s, $\vec{f}$, $\vec{x}$, and $\vec{\text{IH}}$ represent sorts, eliminator cases, constructor arguments, and inductive hypotheses. ξ (A, P, j) is the type of DepElim(A, P) at DepConstr(j, A) (similarly for B).

The proof term transformation is (perhaps deceptively) simple by design: it moves the bulk of the work into the configuration, and represents the configuration explicitly. Of course, typical proof terms in Coq do not apply these configuration terms explicitly. PUMPKIN Pi does some additional work using *unification heuristics* to get real proof terms into this format before running the transformation. It then runs the proof term transformation, which transports proofs across the equivalence that corresponds to the configuration.

Unification Heuristics. The transformation does not fully describe the search procedure for transforming terms that PUMPKIN Pi implements. Before running the transformation, PUMPKIN Pi *unifies* subterms with particular A(fixing parameters and indices), and with applications of configuration terms over A. The transformation then transforms configuration terms over $A$to configuration terms over B. Reducing the result produces the output term defined over B.

Figure 38 shows this with the list append function ++ from Section 6.10. To update ++ (top left), PUMPKIN Pi unifies Old.list T with A, and Constr and Elim with DepConstr and DepElim (bottom left). After unification, the transformation recursively substitutes $B$for A, which moves DepConstr and DepElim to construct and eliminate over the updated type (bottom right). This reduces to a term with swapped constructors and cases over `New.list T` (top right).

In this case, unification is straightforward. This can be more challenging when configuration terms are dependent. This is especially pronounced with definitional `Eta` and `Iota`, which typically are implicit (reduced) in real code. To handle this, Pumpkin Pi implements custom *unification heuristics* for each search procedure that unify subterms with applications of configuration terms, and that instantiate parameters and dependent indices in those subterms ⑥. The transformation in turn assumes that all parameters and indices are fixed.

Pumpkin Pi falls back to Coq's unification for manual configuration and when these custom heuristics fail. When even Coq's unification is not enough, Pumpkin Pi relies on proof engineers to provide hints in the form of annotations ⑤.

Specifying a Correct Transformation. The implementation of this transformation in Pumpkin Pi produces a term that Coq type checks, and so does not add to the trusted computing base. As Pumpkin Pi is an engineering tool, there is no need to formally prove the transformation correct, though doing so would be satisfying. The goal of such a proof would be to show that if $\Gamma \vdash t \Uparrow t'$, then t and t' are equal up to transport, and t' refers to Bin place of A. The key steps in this transformation that make this possible are porting terms along the configuration (Dep-Constr, Dep-Elim, Eta, and Iota). For metatheoretical reasons, without additional axioms, a proof of this theorem in Coq can only be approximated [197]. It would be possible to generate per-transformation proofs of correctness, but this does not serve an engineering need.

6.12.3 *Specifying Correct Configurations*

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

While we cannot control for intentions, we *can* specify what it means for a chosen configuration to be correct: Fix a configuration. Let `f` be the function that uses `DepElim` to eliminate $A$and `DepConstr` to construct B(`g` similar). Figure 39 specifies the correctness criteria for the configuration. These criteria relate `DepConstr`, `DepElim`, `Eta`, and `Iota` in a way that preserves equivalence coherently with equality.

Equivalence. To preserve the equivalence (Figure 39, left), `DepConstr` and `DepElim` must form an equivalence (`section` and `retraction` must hold for `f` and `g`). `DepConstr` over $A$and Bmust be equal up to trans-

$\langle v \rangle \in$ Vars, $\langle t \rangle \in \text{CIC}_\omega$

$\langle p \rangle$ ::= intro $\langle v \rangle$ | rewrite $\langle t \rangle$ $\langle t \rangle$ | symmetry | apply $\langle t \rangle$ | induction $\langle t \rangle$ $\langle t \rangle$ { $\langle p \rangle, \ldots, \langle p \rangle$ } | split { $\langle p \rangle$, $\langle p \rangle$ } | left | right | $\langle p \rangle$. $\langle p \rangle$

Figure 40: Qtac syntax.

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

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

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

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

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

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

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

Figure 41: Qtac decompiler semantics.

port across that equivalence (`constr_ok`), and similarly for `DepElim` (`elim_ok`). Intuitively, `constr_ok` and `elim_ok` guarantee that the transformation correctly transports dependent constructors and dependent eliminators, as doing so will preserve equality up to transport for those subterms. This makes it possible to avoid applying `f` and `g`, instead porting terms from Adirectly to B.

Equality. To ensure coherence with equality (Figure 39, right), `Eta` and `Iota` must prove η and ι. That is, `Eta` must have the same definitional behavior as the dependent eliminator (`elim_eta`), and must behave like identity (`eta_ok`). Each `Iota` must prove and rewrite along the simplification (*refolding* [33]) behavior that corresponds to a case of the dependent eliminator (`iota_ok`). This makes it possible for the transformation to avoid applying `section` and `retraction`.

Correctness. With this correctness criteria for a configuration, we get the completeness result (proven in Coq ⑧) that every equivalence induces a configuration. We also obtain an algorithm for the soundness result that every configuration induces an equivalence. The algorithm to prove `section` is as follows (`retraction` is similar): replace `a` with `Eta(A) a` by `eta_ok(A)`. Then, induct using `DepElim` over A. For each case i, the proof obligation is to show that `g (f a)` is equal to `a`, where `a` is `DepConstr(A, i)` applied to the non-inductive arguments (by `elim_eta(A)`). Expand the right-hand side using `Iota(A, i)`, then expand it again using `Iota(B, i)` (destructing over each `eta_ok` to apply the corresponding `Iota`). The result follows by definition of `g` and `f`, and by reflexivity.

Automatic Configuration. PUMPKIN Pi implements four search procedures for automatic configuration ⑥; we do not discuss these in detail. The algorithm above is essentially what **Configure** uses to generate functions `f` and `g` for these configurations ⑨, and also generate proofs `section` and `retraction` that these functions form an equivalence ⑩. To minimize dependencies, PUMPKIN Pi does not produce proofs of `constr_ok` and `elim_ok` directly, as stating these theorems cleanly would require either a special framework [197] or a univalent type theory [202]. If the proof engineer wishes, it is possible to prove these in individual cases ⑧, but this is not necessary in order to use PUMPKIN Pi.

6.13 DECOMPILING PROOF TERMS TO TACTICS

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

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

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

First Pass: Basic Proof Scripts. The first pass takes Coq terms and produces tactics in Ltac, the proof script language for Coq. Ltac can be confusing to reason about, since Ltac tactics can refer to Gallina terms, and the semantics of Ltac depends both on the semantics of Gallina and on the implementation of proof search procedures written in OCaml. To give a sense of how the first pass works without the clutter of these details, we start by defining a mini decompiler that implements a simplified version of the first pass. Section 6.14.2 explains how we scale this to the implementation.

The mini decompiler takes CIC_{ω} terms and produces tactics in a mini version of Ltac which we call Qtac. The syntax for Qtac is in Figure 40. Qtac includes hypothesis introduction (`intro`), rewriting (`rewrite`), symmetry of equality (`symmetry`), application of a term to prove the goal (`apply`), induction (`induction`), case splitting of conjunctions (`split`), constructors of disjunctions (`left` and `right`), and composition (`.`). Unlike in Ltac, `induction` and `rewrite` take a motive explicitly, rather than relying on unification. Similarly, `apply` leaves the arguments to proof obligations.

The semantics for the mini decompiler $\Gamma \vdash t \Rightarrow p$ are in Figure 41 (assuming $=$, `eq_sym`, $\wedge$, and $\vee$ are defined as in Coq). As with the real decompiler, the mini decompiler defaults to the proof script that applies the entire proof term with `apply` (Base). Otherwise, it improves on that behavior by recursing over the proof term and constructing a proof script using a predefined set of tactics.

For the mini decompiler, this is straightforward: Lambda terms become introduction (Intro). Applications of `eq_sym` become symmetry of equality (Symmetry). Constructors of conjunction and disjunction map to the respective tactics (Split, Left, and Right). Applications of equality eliminators compose symmetry (to orient the rewrite direction) with rewrites (Rewrite), and all other applications of eliminators become induction (Induction). The remaining applications become apply tactics (Apply). In all cases, the decompiler recurses, breaking into cases, until only the Base case holds.

While the mini decompiler is very simple, only a few small changes are needed to move this to Coq. The generated proof term of `rev_app_distr` from Section 6.10, for example, consists only of induction, rewriting, simplification, and reflexivity (solved by `auto`). Figure 42 shows the proof term for the base case of `rev_app_distr` alongside the proof script that Pumpkin Pi suggests. This script is fairly low-level and close to the proof term, but it is already something that the proof engineer can step through to understand, modify, and maintain. There are few differences from the mini decompiler needed to produce this, for example handling of rewrites in both directions (`eq_ind_r` as opposed to `eq_ind`), simplifying rewrites, and turning applications of `eq_refl` into `reflexivity` or `auto`.

```
fun (y0 : list A) =>
  list_rect _ _ (fun a l H =>
    eq_ind_r _ eq_refl (app_nil_r (rev l) (a::[])))
    eq_refl
    y0

- intro y0. induction y0 as [a l H|].
  + simpl. rewrite app_nil_r. auto.
  + auto.
```

Figure 42: Proof term (top) and decompiled proof script (bottom) for the base case of `rev_app_distr` (Section 6.10), with corresponding terms and tactics highlighted the same color.

Second Pass: Better Proof Scripts. The implementation of **Decompile** first runs something similar to the mini decompiler, then modifies the suggested tactics to produce a more natural proof script (11). For example, it cancels out sequences of `intros` and `revert`, inserts semicolons, and removes extra arguments to `apply` and `rewrite`. It can also take tactics from the proof engineer (like part of the old proof script) as hints, then iteratively replace tactics with those hints, checking the result. This makes it possible for suggested scripts to include goal-solving custom tactics and decision procedures.

6.14 IMPLEMENTATION

The transformation and mini decompiler abstract many of the challenges of building a tool for proof engineers. This section describes how we solved some of these challenges.

6.14.1 *Implementing the Transformation*

Termination. When a subterm unifies with a configuration term, this suggests that PUMPKIN Pi *can* transform the subterm, but it does not necessarily mean that it *should*. In some cases, doing so would result in nontermination. For example, if $B$is a refinement of A, then we can always run EQUIVALENCE over and over again, forever. We thus include some simple termination checks in our code (12).

Intent. Even when termination is guaranteed, whether to transform a subterm depends on intent. That is, PUMPKIN Pi automates the case of porting *every* $A$to B, but proof engineers sometimes wish to port only *some* As to Bs. PUMPKIN Pi has some support for this using an interactive workflow (13), with plans for automatic support in the future.

From CIC$_\omega$ to Coq. The implementation (4) of the transformation handles language differences to scale from CIC$_\omega$ to Coq. We use the existing `Preprocess` [178] command to turn pattern matching and

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

Table 1: Some changes using Pumpkin Pi (left to right): class of changes, kind of configuration, examples, and Coq tools we know of that support repair along (Repair) or automatic proof of (Search) the equivalence corresponding to each example. Tools considered are Devoid [178], the Univalent Parametricity (UP) white-box transformation [198], and a classic tool from 2000 [131]. Pumpkin Pi is the only one that supports tactic suggestions. More nuanced comparisons to these and more are in Section 6.16.

fixpoints into eliminators. We handle refolding of constants in constructors using `DepConstr`.

Reaching Real Proof Engineers. Many of our design decisions in implementing Pumpkin Pi were informed by our partnership with an industrial proof engineer (Section 6.15). For example, the proof engineer rarely had the patience to wait more than ten seconds for Pumpkin Pi to port a term, so we implemented optional aggressive caching, even caching intermediate subterms that we encounter in the course of running the transformation ⑭. We also added a cache to tell Pumpkin Pi not to δ-reduce certain terms ⑭.

The experiences of proof engineers also inspired new features. For example, we implemented a search procedure to generate custom eliminators to help reason about types like Σ`(l : list T).length l = n` by reasoning separately about the projections ⑮. We added informative error messages ㉒ to help the proof engineer distinguish between user errors and bugs. These features helped with workflow integration.

6.14.2 *Implementing the Decompiler*

From Qtac to Ltac. The mini decompiler assumes more predictable versions of `rewrite` and `induction` than those in Coq. **Decompile** includes additional logic to reason about these tactics ⑪. For example, Qtac assumes that there is only one `rewrite` direction. Ltac has two rewrite directions, and so the decompiler infers the direction from the motive.

Qtac also assumes that both tactics take the inductive motive explicitly, while in Coq, both tactics infer the motive automatically. Consequentially, Coq sometimes fails to infer the correct motive. To handle induction, the decompiler strategically uses `revert` to manipulate the goal so that Coq can better infer the motive. To handle rewrites, it uses `simpl` to refold the goal before rewriting. Neither of these approaches is guaranteed to work, so the proof engineer may sometimes need to tweak the suggested proof script appropriately. Even if we pass Coq's induction principle an explicit motive, Coq still sometimes fails due to unrepresented assumptions. Long term, using another tactic like `change` or `refine` before applying these tactics may help with cases for which Coq cannot infer the correct motive.

From CIC$_\omega$ to Coq. Scaling the decompiler to Coq introduces let bindings, which are generated by tactics like `rewrite in`, `apply in`, and `pose`. **Decompile** implements ⑪ support for `rewrite in` and `apply in` similarly to how it supports `rewrite` and `apply`, except that it ensures that the unmanipulated hypothesis does not occur in the body of the let expression, it swaps the direction of the rewrite, and it recurses into any generated subgoals. In all other cases, it uses `pose`, a catch-all for let bindings.

Pretty Printing. After decompiling proof terms, **Decompile** pretty prints the result ⑪. Like the mini decompiler, **Decompile** represents its output using a predefined grammar of Ltac tactics, albeit one larger that is larger than Qtac, and that also includes tacticals. It maintains the recursive proof structure for formatting. Pumpkin Pi keeps all output terms from **Transform** in the Coq environment in case the decompiler does not succeed. Once the proof engineer has the new proof, she can remove the old one.

6.15 CASE STUDIES: PUMPKIN PI EIGHT WAYS

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

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

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

Figure 43: A simple language (left) and the same language with two swapped constructors and an added constructor (right).

Algebraic Ornaments: Lists to Packed Vectors. The transformation in Pumpkin Pi is a generalization of the transformation from Devoid. Devoid supported proof reuse across *algebraic ornaments*, which describe relations between two inductive types, where one type is the other indexed by a fold [136]. A standard example is the relation between a list and a length-indexed vector (Figure 33).

Pumpkin Pi implements a search procedure for automatic configuration of algebraic ornaments. The result is all functionality from Devoid, plus tactic suggestions. In file ③, we used this to port functions and a proof from lists to vectors of *some* length, since `list T` ≃ `packed_vect T`. The decompiler helped us write proofs that we had found too hard to write by hand, though the suggested tactics did need massaging.

Unpack Sigma Types: Vectors of Particular Lengths. In the same file ③, we then ported functions and proofs to vectors of a *particular* length, like `vector T n`. Devoid had left this step to the proof engineer. We supported this in Pumpkin Pi by chaining the previous change with an automatic configuration for unpacking sigma types. By composition, this transported proofs across the equivalence from Section 6.11.

Two tricks helped with workflow integration for this change: 1) have the search procedure view `vector T n` as Σ`(v : vector T m).n = m` for some `m`, then let Pumpkin Pi instantiate those equalities via unification heuristics, and 2) generate a custom eliminator for combining list terms with length invariants. The resulting workflow works not just for lists and vectors, but for any algebraic ornament, automating manual effort from Devoid. The suggested tactics were again helpful for writing proofs that we had struggled with manually, but only after massaging. More effort is needed to improve tactic suggestions for dependent types.

Tuples & Records: Industrial Use. An industrial proof engineer at the company Galois has been using Pumpkin Pi in proving correct an implementation of the TLS handshake protocol. Galois had been using a custom solver-aided verification language to prove correct C programs, but had found that at times, the constraint solvers got stuck. They had built a compiler that translates their language into Coq's specification language Gallina, that way proof engineers could finish stuck proofs interactively using Coq. However, due to language differences, they had found the generated Gallina programs and specifications difficult to work with.

The proof engineer used Pumpkin Pi to port the automatically generated functions and specifications to more human-readable functions and specifications, wrote Coq proofs about those functions and specifications, then used Pumpkin Pi to port those proofs back to proofs about the original functions and specifications. So far, they have used at least three automatic configurations, but they most often used an automatic configuration for porting compiler-produced anonymous tuples to named records, as in file ⑱. The workflow was a bit nonstandard, so there was little need for tactic suggestions.

Permute Constructors: Modifying a Language. The swapping example from Section 6.10 was inspired by benchmarks from the Replica user study of proof engineers [174]. A change from one of the benchmarks is in Figure 43. The proof engineer had a simple language represented by an inductive type `Term`, as well as some definitions and proofs about the language. The proof engineer swapped two constructors in the language, and added a new constructor `Bool`.

This case study and the next case study break this change into two parts. In the first part, we used Pumpkin Pi with automatic configuration to repair functions and proofs about the language after swapping the constructors ①. With a bit of human guidance to choose the permutation from a list of suggestions, Pumpkin Pi repaired everything, though the original tactics would have also worked.

Add new Constructors: Extending a Language. We then used Pumpkin Pi to repair functions after adding the new constructor in Figure 43, separating out the proof obligations for the new constructor from the old terms ⑲. This change combined manual and automatic configuration. We defined an inductive type `Diff` and (using partial automation) a configuration to port the terms across the equivalence `Old.Term + Diff` $\simeq$ `New.Term`. This resulted in case explosion, but was formulaic, and pointed to a clear path for automation of this class of changes. The repaired functions guaranteed preservation of the behavior of the original functions.

Adding constructors was less simple than swapping. Most notably, the repaired terms were (unlike in the swap case) inefficient compared to human-written terms. For now, they make good regression tests for the human-written terms—in the future, we hope to automate the discovery of the more efficient terms, or use the refinement framework CoqEAL [50] to get between proofs of the inefficient and efficient terms.

Factor out Constructors: External Example. The change from Figure 32 came at the request of a non-author. We supported this using a manual configuration that described which constructor to map to `true` and which constructor to map to `false` ②. The configuration was very simple for us to write, and the repaired tactics were immediately useful.

Permute Hypotheses: External Example. The change in ⑳ came at the request of a different non-author (a cubical type theory expert), and shows how to use Pumpkin Pi to swap two hypotheses of a type, since `T1` $\rightarrow$ `T2` $\rightarrow$ `T3` $\simeq$ `T2` $\rightarrow$ `T1` $\rightarrow$ `T3`. This configuration was manual. Since neither type was inductive, this change used the generic construction for any equivalence. This worked well, but necessitated some manual annotation due to the lack of custom unification heuristics for manual configuration.

Change Inductive Structure: Unary to Binary. In ⑤, we used Pumpkin Pi to support a classic example of changing inductive structure: updating unary to binary numbers [131]. In Coq, a binary number `N` (Figure 36) is either zero or a positive binary number. A positive binary number is either 1 (`xH`), or the result of shifting left and adding 1 (`xI`) or 0 (`xO`). This allows for a fast addition function, found in the Coq standard library. In the style of the classic example [131], we used Pumpkin Pi to derive a slow binary addition function that does not refer to `nat`, and to port proofs from unary to slow binary addition. We then showed that the ported theorems hold over fast binary addition.

The configuration for `N` used definitions from the Coq standard library for `DepConstr` and `DepElim` that had the desired behavior with no changes. `Iota` over the successor case was a rewrite by a lemma from the standard library that reduced the successor case of the eliminator that we used for `DepElim`:

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

The need for nontrivial `Iota` comes from the fact that `N` and `nat` have different inductive structures. By writing a manual configuration with this `Iota`, it was possible for us to implement this transformation that had been its own tool.

While porting addition from `nat` to `N` was automatic after configuring Pumpkin Pi, porting proofs about addition took more work. Due to the lack of unification heuristics for manual configuration, we had to annotate the proof term to tell Pumpkin Pi that implicit casts in the inductive cases of proofs were applications of `Iota` over `nat`. These annotations were formulaic, but tricky to write. Unification heuristics would go a long way toward improving the workflow.

After annotating, we obtained automatically repaired proofs about slow binary addition, which we found simple to port to fast binary addition. We hope to automate this last step in the future using CoqEAL. Repaired tactics were partially useful, but failed to understand custom eliminators like `N.peano_rect`, and to generate useful tactics for applications of `Iota`; both of these are clear paths to more useful tactics.

6.16 RELATED WORK

Proof Repair. The search procedures in **Configure** are based partly on ideas from the original PUMPKIN PATCH prototype [177]. The PUMPKIN PATCH prototype did not apply the patches that it finds, handle changes in structure, or include support for tactics beyond the use of hints.

Proof repair can be viewed as a form of *program repair* [147, 88] for proof assistants. Proof assistants like Coq are a good fit for program repair: A recent paper [171] recommends that program repair tools draw on extra information such as specifications or example patches. In Coq, specifications and examples are rich and widely available: specifications thanks to dependent types, and examples thanks to constructivism.

Proof Refactoring. The proof refactoring tool Levity [31] for Isabelle/HOL has seen large-scale industrial use. Levity focuses on a different task: moving lemmas. Chick [179] and RefactorAgda [208] are proof refactoring tools in a Gallina-like language and in Agda, respectively. These tools support primarily syntactic changes and do not have tactic support.

A few proof refactoring tools operate directly over tactics: POLAR [78] refactors proof scripts in languages based on Isabelle/Isar [203], CoqPIE [182] is an IDE with support for simple refactorings of Ltac scripts, and Tactician [10] is a refactoring tool for switching between tactics and tacticals. This approach is not tractable for more complex changes [179].

Proof Reuse. A few proof reuse tools work by proof term transformation and so can be used for repair. A 2004 paper [105] describes a transformation that generalizes theorems in Isabelle/HOL. PUMPKIN Pi generalizes the transformation from DEVOID [178], which transformed proofs along algebraic ornaments [136]. A 2000 paper [131] implements a proof term transformation between unary and binary numbers. Both of these fit into PUMPKIN Pi configurations, and none implements tactic support in Coq like PUMPKIN Pi does. The expansion algorithm from the 200 paper [131] may help guide the design of unification heuristics for `Iota` in PUMPKIN Pi.

CoqEAL [50] transforms functions across relations, and these relations can be more general than PUMPKIN Pi's equivalences. However, while PUMPKIN Pi supports both functions and proofs, CoqEAL supports only simple functions due to the problem that Iota addresses. CoqEAL may be most useful to chain with PUMPKIN Pi to get faster functions. Both CoqEAL and recent ornaments work [211] may help with better workflow support for changes that do not correspond to equivalences.

The PUMPKIN Pi transformation implements transport. Transport is realizable as a function given univalence [202]. UP [197] approximates it in Coq, only sometimes relying on functional extensionality. While powerful, neither approach removes references to the old type.

Recent work [198] extends UP with a white-box transformation that may work for repair. This imposes proof obligations on the proof engineer beyond those imposed by PUMPKIN Pi, and it includes neither search procedures for equivalences nor tactic script generation. It also does not support changes in inductive structure, instead relying on its original black-box functionality; Iota solves this in PUMPKIN Pi. The most fruitful progress may come from combining these tools.

Proof Design. Much work focuses on designing proofs to be robust to change, rather than fixing broken proofs. This can take the form of design principles, like using information hiding techniques [216, 112] or any of the structures [**?**, 191, 184] for encoding interfaces in Coq. Design and repair are complementary: design requires foresight, while repair can occur retroactively. Repair can help with changes that occur outside of the proof engineer's control, or with changes that are difficult to protect against even with informed design.

Another approach to this is to use heavy proof automation, for example through program-specific proof automation [44] or general-purpose hammers [26, 163, 109, 62]. The degree to which proof engineers rely on automation varies, as seen in the data from a user study [174]. Automation-heavy proof engineering styles localize the burden of change to the automation, but can result in terms that are large and slow to type check, and tactics that can be difficult to debug. While these approaches are complementary, more work is needed for PUMPKIN Pi to better support developments in this style.

6.17 CONCLUSIONS & FUTURE WORK

We combined search procedures for equivalences, a proof term transformation, and a proof term to tactic decompiler to build PUMPKIN Pi, a proof repair tool for changes in datatypes. The proof term transformation implements transport across equivalences in a way that is suitable for repair and that does not compromise definitional equality. The resulting tool is flexible and useful for real proof engineering scenarios.

Future Work. Moving forward, our goal is to make proofs easier to repair and reuse regardless of proof engineering expertise and style.

We want to reach more proof engineers, and we want PUMPKIN Pi to integrate seamlessly with Coq.

Three problems that we encountered scaling up the PUMPKIN Pi transformation were lack of type-directed search, ad hoc termination checks, and inability for proof engineers to add custom unification heuristics. We hope to solve these challenges using *e-graphs* [157], a data structure for managing equivalences built with these kinds of problems in mind. E-graphs were recently adapted to express path equality in cubical [89]; we hope to repurpose this insight.

Beyond that, we believe that the biggest gains will come from continuing to improve the prototype decompiler. Two helpful features would be preserving indentation and comments, and automatically using information from the original proof script rather than asking for hints. Some improvements could come from better tactics, or from a more structured tactic language. Integration with version control systems or with integrated development environments could also help. With that, we believe that the future of seamless and powerful proof repair and reuse for all is within reach.

7

UNDER THE HOOD

Part IV

CLOSING

8 CONCLUSION

9

THE NEXT ERA

BIBLIOGRAPHY

[1] Agda, 2017.

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

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

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

[5] Lean theorem prover, 2017.

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

[7] Proof general, 2017.

[8] Travis ci, 2017.

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

[10] Mark Adams. Refactoring proofs with tactician. In Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, pages 53–67, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

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

[12] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. Large-scale formal verification in practice: A process perspective. In *International Conference on Software Engineering*, pages 1002–1011, Zurich, Switzerland, June 2012. ACM.

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

[14] David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25 – April 2, 2000 Proceedings*, pages 38–43, Berlin, Heidelberg, 2000. Springer.

[15] David Aspinall and Cezary Kaliszyk. Towards formal proof metrics. In *Fundamental Approaches to Software Engineering*, pages 325–341, Berlin, Heidelberg, 2016. Springer.

[16] David Aspinall and Cezary Kaliszyk. What's in a theorem name? In *Interactive Theorem Proving*, pages 459–465, Cham, 2016. Springer International Publishing.

[17] Serge Autexier, Dieter Hutter, and Till Mossakowski. Verification, induction termination analysis. chapter Change Management for Heterogeneous Development Graphs, pages 54–80. Springer-Verlag, Berlin, Heidelberg, 2010.

[18] Serge Autexier and Normen Müller. Semantics-based change impact analysis for heterogeneous collections of documents. In *Proceedings of the 10th ACM Symposium on Document Engineering*, DocEng '10, pages 97–106, New York, NY, USA, 2010. ACM.

[19] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008. ACM.

[20] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: the poplmark challenge. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2005.

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

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

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

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

[25] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22:107–152, 03 2012.

[26] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector

for Isabelle/HOL. *Journal of Automated Reasoning*, 57(3):219–244, Oct 2016.

[27] Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In *Intelligent Computer Mathematics: International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings.*, pages 3–17, Cham, 2015. Springer International Publishing.

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

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

[30] Olivier Boite. Proof reuse with extended inductive types. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*, pages 50–65, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[31] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In *Intelligent Computer Mathematics*, pages 32–48, Berlin, Heidelberg, 2012. Springer.

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

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

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

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

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

[37] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA, 2014. ACM.

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

[39] Stuart Card and David Nation. Degree-of-interest trees: A component of an attention-reactive user interface. 01 2002.

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

[41] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM.

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

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

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

[45] Adam Chlipala. Formal reasoning about programs, 2017.

[46] Adam Chlipala. Library equality, 2017.

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

[48] Andy Cockburn. Supporting tailorable program visualisation through literate programming and fisheye views. *Information and Software Technology*, 43(13):745 – 758, 2001.

[49] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint arXiv:1611.02108*, 2016.

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

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

[52] coq-club. [Coq-Club] Dealing with equalities in dependent types. `http://sympa.inria.fr/sympa/arc/coq-club/2017-01/msg00099.html`, 2017. Accessed: 2019-03-25.

[53] coq-club. [Coq-Club] Trouble with dependent induction. `http://sympa.inria.fr/sympa/arc/coq-club/2017-12/msg00079.html`, 2017. Accessed: 2019-03-25.

[54] Coq Development Team. The Coq commands: Customization at launch time, 1989-2021.

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

[56] Coq Development Team. The coq commands, 1999-2018.

[57] Coq Development Team. Coq integrated development environment, 1999-2018.

[58] Coq Development Team. A short introduction to coq, 1999-2019.

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

[60] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

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

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

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

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

[65] Pierre-Evariste Dagand and Conor McBride. A categorical treatment of ornaments. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 530–539, Washington, DC, USA, 2013. IEEE Computer Society.

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

[67] Simon P. Davies. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2):237 – 267, 1993.

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

[69] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

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

[72] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM.

[73] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM.

[74] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330, New York, NY, USA, 2013. ACM.

[75] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 206–214, New York, NY, USA, 1977. ACM.

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

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

[78] Dominik Dietrich, Iain Whiteside, and David Aspinall. Polar: A framework for proof refactoring. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 776–791, Berlin, Heidelberg, 2013. Springer.

[79] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan 2001.

[80] Jim Fehrle. Pull request: Highlight differences between successive proof steps (color, underline, etc.), 2018.

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

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

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

[84] Leah Findlater and Joanna McGrenere. A comparison of static, adaptive, and adaptable menus. In *Proceedings of the SIGCHI*

Conference on Human Factors in Computing Systems, CHI '04, pages 89–96, New York, NY, USA, 2004. ACM.

[85] Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for Coq. Technical report, MINES ParisTech, October 2016.

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

[87] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.

[88] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1219–1219, New York, NY, USA, 2018. ACM.

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

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

[91] Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, and Megan Squire. Repositories with public data about software development. *Int. J. Open Source Softw. Process.*, 2(2):1–13, April 2010.

[92] Jason Gross. Coq Github issue #6609: Primitive projections should be on by default. `https://github.com/coq/coq/issues/6609`, 2018.

[93] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, GA, 2016. USENIX Association.

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

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

[96] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-pattern recognition and lemma discovery in ACL2. In *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pages 389–406, Berlin, Heidelberg, 2013. Springer.

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

[98] HOL Development Team. Running hol, 2016-2018.

[99] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs: Third International Conference*, CPP 2013, pages 131–146, Cham, 2013. Springer International Publishing.

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

[101] Jasper Hugunin. Transfer indexed parameter, 2018.

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

[103] Inria. The Coq standard library. `http://coq.inria.fr/distrib/current/stdlib`. Accessed: 2019-03-15.

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

[105] Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*, pages 152–167. Springer, Berlin, Heidelberg, 2004.

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

[107] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6):967–989, November 2006.

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

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

[110] Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM.

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

[112] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014.

[113] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014.

[114] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[115] A. J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32, 2006.

[116] Amy Ko and Brad Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16:41–84, 02 2005.

[117] Hsiang-Shang Ko and Jeremy Gibbons. Relational algebraic ornaments. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming*, pages 37–48. ACM, 2013.

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

[119] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine learning in Proof General: Interfacing interfaces. In *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012.*, pages 15–41, 2012.

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

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

[122] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM.

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

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

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

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

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

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

[129] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

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

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

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

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

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

[135] D. Matichuk, T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples. Empirical study towards a leading indicator for cost of formal software verification. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 722–732, May 2015.

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

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

[138] James McKinna and Conor McBride. One song to the tune of another.

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

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

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

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

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

[144] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. Type-directed diffing of structured data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 2–15, New York, NY, USA, 2017. ACM.

[145] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. Type-directed diffing of structured data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 2–15, New York, NY, USA, 2017. ACM.

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

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

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

[149] Tamara Munzner, Francois Guimbretiere, Serdar Tasiran, Li Zhang, and Yunhong Zhou. Treejuxtaposer: Scalable tree comparison using focus+context with guaranteed visibility. *ACM Trans. Graph.*, 22:453–462, 07 2003.

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

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

[152] Toby Murray and P. C. van Oorschot. BP: Formal proofs, the fine print and side effects. In *IEEE Cybersecurity Development (SecDev)*, pages 1–10, Sep. 2018.

[153] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Reducing feedback delay of software development tools via continuous analysis. *IEEE Transactions on Software Engineering*, 41(8):745–763, August 2015.

[154] Magnus O. Myreen. Guide to hol4 interaction and basic proofs, 2008-2018.

[155] Yutaka Nagashima and Yilun He. PaMpeR: Proof method recommendation system for Isabelle/HOL. In *Proceedings of the International Conference on Automated Software Engineering*, ASE 2018, pages 362–372, New York, NY, USA, 2018. ACM.

[156] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery.

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

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

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

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

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

[162] Zoe Paraskevopoulou, Cătălin Hriţçu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 325–343, Cham, 2015. Springer International Publishing.

[163] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *International Workshop on the Implementation of Logics (IWIL 2010)*, volume 2 of *EPiC Series*, pages 1–11. EasyChair, 2012.

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

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

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

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

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

[169] Olivier Pons. Generalization in type theory based proof assistants. In *International Workshop on Types for Proofs and Programs*, pages 217–232. Springer, 2000.

[170] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings*

of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[171] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. ACM.

[172] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

[173] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[174] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. Replica: Repl instrumentation for coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 99–113, 01 2020.

[175] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 115–129, New York, NY, USA, 2018. ACM.

[176] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129. ACM, 2018.

[177] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 115–129, New York, NY, USA, 2018. Association for Computing Machinery.

[178] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[180] Valentin Robert and Sorin Lerner. Peacoq, 2014-2016.

[181] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, December 2004.

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

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

[184] Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types: application à la Théorie des Catégories*. PhD thesis, Université Paris VI, Paris, France, 1999.

[185] Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. Generating correctness proofs with neural networks. *CoRR*, abs/1907.07794, 2019.

[186] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning: 8th International Joint Conference*, IJCAR 2016, pages 99–115, Cham, 2016. Springer International Publishing.

[187] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. *CoRR*, abs/1701.04391, 2017.

[188] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. *CoRR*, abs/1701.04391, 2017.

[189] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 419–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

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

[191] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 278–293, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[192] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. hs-to-coq. `https://github.com/antalsz/hs-to-coq`, 2018-2019. Accessed: 2019-03-12.

[193] Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. Productivity for proof engineering. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 15:1–15:4, New York, NY, USA, 2014. ACM.

[194] Mark Staples, Rafal Kolanski, Gerwin Klein, Corey Lewis, June Andronick, Toby Murray, Ross Jeffery, and Len Bass. Formal specifications better than function points for code sizing. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1257–1260, Piscataway, NJ, USA, 2013. IEEE Press.

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

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

[197] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: Univalent parametricity for effective transport. *Proc. ACM Program. Lang.*, 2(ICFP):92:1–92:29, July 2018.

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

[199] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 264–276, New York, NY, USA, 2009. Association for Computing Machinery.

[200] The Idris Community. The idris repl, 2017.

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

[202] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[203] Makarius Wenzel. Isabelle/Isar–a generic framework for human-readable proof documents. *From Insight to Proof–Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.

[204] Makarius Wenzel. Isabelle/jEdit – a prover IDE within the PIDE framework. In *Intelligent Computer Mathematics*, pages 468–471, Berlin, Heidelberg, 2012. Springer.

[205] Makarius Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 515–530, Cham, 2014. Springer International Publishing.

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

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

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

[209] Freek Wiedijk. Statistics on digital libraries of mathematics. *Studies in Logic, Grammar and Rhetoric*, (18(31)):137–151, 2009.

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

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

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

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

[214] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs*

and Proofs, CPP 2016, pages 154–165, New York, NY, USA, 2016. ACM.

[215] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. ACM.

[216] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. ACM.

[217] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*, 2019.

[218] He Zhang, Gerwin Klein, Mark Staples, June Andronick, Liming Zhu, and Rafal Kolanski. Simulation modeling of a large scale formal verification process. In *International Conference on Software and Systems Process*, pages 3–12, Zurich, Switzerland, June 2012. IEEE.

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

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

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

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