REPLica: REPL Instrumentation for Coq Analysis

Talia Ringer
University of Washington
USA
tringer@cs.washington.edu

Alex Sanchez-Stern
University of California, San Diego
USA
alexss@eng.ucsd.edu

Dan Grossman
University of Washington
USA
djg@cs.washington.edu

Sorin Lerner
University of California, San Diego
USA
lerner@cs.ucsd.edu

Abstract

Proof engineering tools make it easier to develop and maintain large systems verified using interactive theorem provers. Developing useful proof engineering tools hinges on understanding the development processes of proof engineers. This paper breaks down one barrier to achieving that understanding: remotely collecting granular data on proof developments as they happen.

We have built a tool called REPLica that instruments Coq's interaction model in order to collect fine-grained data on proof developments. It is decoupled from the user interface, and designed in a way that generalizes to other interactive theorem provers with similar interaction models.

We have used REPLica to collect data over the span of a month from a group of intermediate through expert proof engineers—enough data to reconstruct hundreds of interactive sessions. The data reveals patterns in fixing proofs and in changing programs and specifications useful for the improvement of proof engineering tools. Our experiences conducting this study suggest design considerations both at the level of the study and at the level of the interactive theorem prover that can facilitate future studies of this kind.

CCS Concepts • **Human-centered computing → User studies**; • **Software and its engineering → Software notations and tools**; **Formal software verification**; **Software evolution**;

Keywords proof engineering, user interaction, study methodologies

CPP '20, January 20–21, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7097-4/20/01…$15.00
https://doi.org/10.1145/3372885.3373823

ACM Reference Format:
Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL Instrumentation for Coq Analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372885.3373823

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* [36].

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 [2, 4, 6, 8, 27, 44, 45, 51, 52, 54]. 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 [37]. 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 [11] to remotely collect fine-grained data on proof development processes. We have built a Coq plugin (Section 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 [21, 33] and Idris [46].

We have used REPLica to collect a month's worth of data on the proof developments of 8 intermediate to expert Coq users (Section 3). 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) and changes to programs and specifications (Section 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 [19, 20, 25, 34, 43, 53], proof refactoring tools [1, 8, 15, 39, 42, 49, 50], and proof repair tools [37, 39].

Our experiences collecting and analyzing this data have helped us to identify three wishes (Section 6), the fulfillment of which may facilitate future studies of proof development: better abstraction of user environments, more information about user interaction, and more users. We discuss important design considerations for both study designers and ITP designers who hope to grant each wish.

In summary, we contribute the following:

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

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 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` [12] to the IDEs CoqIDE [13] and Proof General [3], communicate with Coq through the REPL. REPLica instruments the REPL to collect fine-grained data while remaining decoupled from the UI.

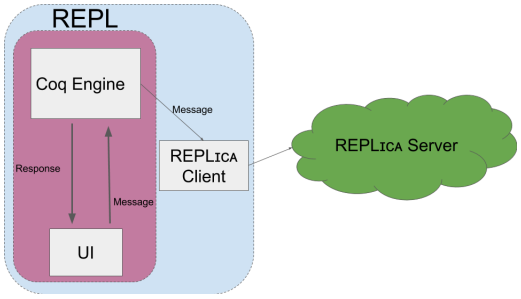

Figure 1. REPLica design. The REPLica client listens to the REPL and sends data to the REPLica server.

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 2.2), then sends this data to the server (Section 2.3), which stores it for analysis.

The implementation effort for REPLica was modest, with just 412 LOC of OCaml for the client and 178 LOC of Python for the server.[3] This modest effort was enough for us to collect (Section 3) and analyze (Sections 4 and 5) data from hundreds of interactive sessions.

2.1 User-Coq Interaction

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

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

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

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

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

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, 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 [18]) 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 5). As a consequence, we were able to find this pattern of development of incrementally extending inductive types for four of our users (Section 5.2.1).

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 [10] so that Coq always loads it. On the first build, REPLica asks the proof engineer for consent and for demographic information for the study. From then on, the proof engineer uses Coq normally.

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

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

2.3 Client-Server Interaction

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

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

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

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

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

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

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

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

3 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)
- **Q2**: What kinds of changes to programs and specifications do users make often, and do those changes reveal patterns amenable to automation? (Section 5)

To answer these questions, we recruited 12 proof engineers to install REPLica and use Coq normally for a month (Section 3.1). We received a month's worth of data containing granular detail on Coq development for 8 of 12 of those users (Section 3.2). After a month, we shut down the server, closed the study, and visualized and analyzed the data (Section 3.3).

3.1 Recruiting

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

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

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

3.2 Collection

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

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

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

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

REPLica logged interactive sessions for only 8 of the 12 users. Section 6 discusses possible causes of, implications of, and remedies for this.

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 6).

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

[7] http://github.com/uwplse/analytics-data

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	23	15	0
11	17	8	0

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

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 4.1)
2. Within proofs, application of lemmas was the main driver of proving. (Section 4.2)
3. Within proofs, over half of fixes to tactic mistakes fixed either an improper argument or wrong sequencing structure. (Section 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.

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 [19, 25, 34, 43, 53] with tools for repairing definitions [37, 39].

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 5) reveals more about how specifications and programs changed.

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 [25] for Coq already offers promising developments in this direction by understanding and providing hints about similar lemmas, as does the proof automation tool CoqHammer [14]; similar functionality may help improve the performance of tools that suggest tactics.

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

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.

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 to the end of 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.

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 6). 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 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 5.1), then classify changes that we found (Section 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 5.2.1, 5.2.2, 5.2.3, and 5.2.4), we identified benchmarks (examples of the pattern in our data) and lessons for automation.

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 6.2 discusses some of the challenges.

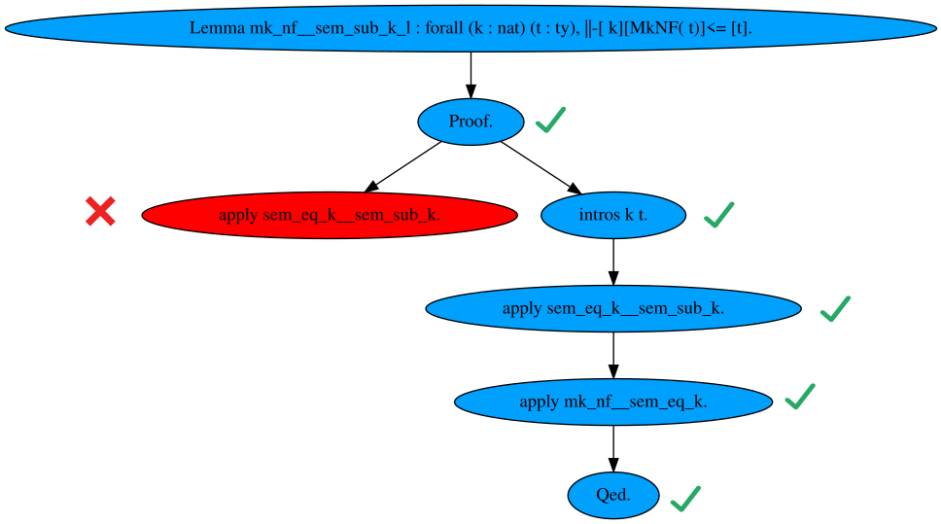

Figure 5. An example search tree, generated from the collected data by REPLica. It shows the user attempting to apply a lemma, which fails until they first run the `intros` tactic.

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

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

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

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

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

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

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

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

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

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

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 7 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.

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 6 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 6 came from User 5, Sessions 18, 19, 27, 33, and 35. There, over the course of three weeks, the user incrementally developed the inductive type `Term` along with a record `EpsilonLogic` and fixpoints `simplify` (later renamed to `identity`) and `free_vars`.

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

Lessons for Automation Given that several users show this pattern, this is one use case for which better automation may help. Automation may help proof engineers adapt other inductive types, match statements, and proofs after extending inductive types with new constructors. We are not aware of any work on adapting related inductive types and match statements. There is some work on adapting proof obligations to new constructors [7], and a proposed algorithm for generating proofs that satisfy those obligations [29], 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.

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.

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 7. Top changes, by user.

```
- 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.

```
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.

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

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

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

Lessons for Automation Refactoring terms (rather than proof scripts) as in RefactorAgda [50] and Chick [39] would have helped our users, but few refactoring tools for ITPs support this [36], 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.

5.2.3 Repetitive Repair of Specifications

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

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

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

```
- 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).

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** [37] and **Structure** [38], 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.

5.2.4 Interactive Discovery of Programs and Specifications

In Q1 (Section 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 [31]. 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 [35] for testing specifications, or with the `induct` tactic from FRAP [9] or the hypothesis renaming functionality from CoqPIE [42] for simple generalization of inductive hypotheses. Integrating tools for discovering lemma and theorem names [5] 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.

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 6.1)
2. More information about user interaction (Section 6.2)
3. More users (Section 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.

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 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) [47] framework as its interaction model. With PIDE, UIs and ITPs communicate using an asynchronous protocol to manage versions of a document [48], annotating the document with new information when it becomes available [47]. 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.

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.

[8] http://github.com/coq/coq/issues/8989

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 6.1, but it may be worth weighing the tradeoffs.

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.

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 the development processes of proof engineers. Outside of the context of ITPs, REPLica is not the first tool to collect data at this level of granularity. Mylar [22] monitors Java development activity and uses it to visualize codebases in order to make them easier to navigate. The Solstice [32] IDE plugin instruments the IDE to make program analysis possible at this level of granularity. Several studies have used video recordings [24, 41], sometimes combined with IDE instrumentation [26], 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 [2, 44, 54], archives and libraries [4, 6, 27, 51], project logs [2, 54], artifacts [4, 27, 45], meeting notes [2, 54], and personal experiences [8, 52]. Collecting proof development data at the level of user interaction was, until now, an unaddressed opportunity to understand proof development processes [36]. This has not gone unnoticed. For example, one study [54] 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 [27] 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 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 [28] describes a general framework for representing changes in algebraic datatypes. Existing work [16] analyzes a large repository and collects data on changes to quantify code decay. TreeJuxtaposer [30] 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 [23]. 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 [37] is evaluated on case studies identified using a manual change analysis of Git commits. The proof refactoring and repair tool Chick [39] 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 [17], CoqPIE [42], and PeaCoq [40] all have support for visualizing changes during development. The machine learning tool Proverbot9001 [43] 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.

Acknowledgments

We thank Emilio J. Gallego Arias for designing the initial version of the hooks into the REPL that enabled this study. We thank Enrico Tassi, Gaëtan Gilbert, Maxime Dénès, Vincent Laporte, and Théo Zimmermann for help improving, finalizing, and merging the hooks for distribution in Coq 8.10. We thank Jason Gross for alpha testing REPLica before the study began and reporting bugs. We thank Dimitar Bounov, Martin Kellogg, Chandrakana Nandi, Doug Woos, and Karl Palmskog for help writing and filming the study recruitment video. We thank all who participated in the study or helped distribute promotional materials. We thank the UW PLSE lab and the UCSD Programming Systems group for feedback. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256082. Any opinions, findings, and conclusions or recommendations expressedin this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mark Adams. 2015. Refactoring Proofs with Tactician. In *Software Engineering and Formal Methods*, Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 53–67. https://doi.org/10.1007/978-3-662-49224-6_6

[2] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. 2012. Large-Scale Formal Verification in Practice: A Process Perspective. In *International Conference on Software Engineering*. ACM, Zurich, Switzerland, 1002–1011. https://doi.org/10.1109/ICSE.2012.6227120

[3] David Aspinall. 2000. 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*. Springer, Berlin, Heidelberg, 38–43. https://doi.org/10.1007/3-540-46419-0_3

[4] David Aspinall and Cezary Kaliszyk. 2016. Towards Formal Proof Metrics. In *Fundamental Approaches to Software Engineering*. Springer, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/978-3-662-49665-7_19

[5] David Aspinall and Cezary Kaliszyk. 2016. What's in a Theorem Name?. In *Interactive Theorem Proving*. Springer International Publishing, Cham, 459–465. https://doi.org/10.1007/978-3-319-43144-4_28

[6] Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. 2015. Mining the Archive of Formal Proofs. In *Intelligent Computer Mathematics: International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*. Springer International Publishing, Cham, 3–17. https://doi.org/10.1007/978-3-319-20615-8_1

[7] Olivier Boite. 2004. Proof Reuse with Extended Inductive Types. In *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*, Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65. https://doi.org/10.1007/978-3-540-30142-4_4

[8] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. 2012. Challenges and Experiences in Managing Large-Scale Proofs. In *Intelligent Computer Mathematics*. Springer, Berlin, Heidelberg, 32–48. https://doi.org/10.1007/978-3-642-31374-5_3

[9] Adam Chlipala. 2017. Formal Reasoning About Programs. http://adam.chlipala.net/frap/

[10] Coq Development Team. 1989-2018. The Coq Commands: Customization at launch time. http://coq.inria.fr/refman/practical-tools/coq-commands.html#customization-at-launch-time

[11] Coq Development Team. 1989-2019. The Coq Proof Assistant. http://coq.inria.fr

[12] Coq Development Team. 1999-2018. The Coq Commands. http://coq.inria.fr/refman/practical-tools/coq-commands.html

[13] Coq Development Team. 1999-2018. Coq Integrated Development Environment. http://coq.inria.fr/refman/practical-tools/coqide.html

[14] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (01 Jun 2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

[15] Dominik Dietrich, Iain Whiteside, and David Aspinall. 2013. Polar: A Framework for Proof Refactoring. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, Berlin, Heidelberg, 776–791. https://doi.org/10.1007/978-3-642-45221-5_52

[16] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 27, 1 (Jan 2001), 1–12. https://doi.org/10.1109/32.895984

[17] Jim Fehrle. 2018. Pull Request: Highlight differences between successive proof steps (color, underline, etc.). http://github.com/coq/coq/pull/6801

[18] Emilio Jesús Gallego Arias. 2016. *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq*. Technical Report. MINES ParisTech. https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408

[19] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. 2017. TacticToe: Learning to Reason with HOL4 Tactics. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Vol. 46. EasyChair, 125–143. https://doi.org/10.29007/ntlb

[20] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. 2013. 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*. Springer, Berlin, Heidelberg, 389–406. https://doi.org/10.1007/978-3-642-45221-5_27

[21] HOL Development Team. 2016-2018. Running hol. https://hol-theorem-prover.org/guidebook/#running-hol

[22] Mik Kersten and Gail C. Murphy. 2005. Mylar: A Degree-of-interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development (AOSD '05)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/1052898.1052912

[23] Amy Ko and Brad Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16 (02 2005), 41–84. https://doi.org/10.1016/j.jvlc.2004.08.003

[24] A. J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32 (2006).

[25] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. 2012. 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*. 15–41. https://doi.org/10.4204/EPTCS.118.2

[26] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. 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)*. ACM, New York, NY, USA, 361–370. https://doi.org/10.1145/1287624.1287675

[27] D. Matichuk, T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples. 2015. Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 722–732. https://doi.org/10.1109/ICSE.2015.85

[28] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. 2017. Type-directed Diffing of Structured Data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2017)*. ACM, New York, NY, USA, 2–15. https://doi.org/10.1145/3122975.3122976

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

[30] Tamara Munzner, Francois Guimbretiere, Serdar Tasiran, Li Zhang, and Yunhong Zhou. 2003. TreeJuxtaposer: Scalable Tree Comparison Using Focus+Context with Guaranteed Visibility. *ACM Trans. Graph.* 22 (07 2003), 453–462. https://doi.org/10.1145/1201775.882291

[31] Toby Murray and P. C. van Oorschot. 2018. BP: Formal Proofs, the Fine Print and Side Effects. In *IEEE Cybersecurity Development (SecDev)*. 1–10. https://doi.org/10.1109/SecDev.2018.00009

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

[33] Magnus O. Myreen. 2008-2018. Guide to HOL4 interaction and basic proofs. http://hol-theorem-prover.org/HOL-interaction.pdf

[34] Yutaka Nagashima and Yilun He. 2018. PaMpeR: Proof Method Recommendation System for Isabelle/HOL. In *Proceedings of the International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 362–372. https://doi.org/10.1145/3238147.3238210

[35] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Springer International Publishing, Cham, 325–343. https://doi.org/10.1007/978-3-319-22102-1_22

[36] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281. https://doi.org/10.1561/2500000045

[37] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting Proof Automation to Adapt Proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 115–129. https://doi.org/10.1145/3167094

[38] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for Proof Reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:19. https://doi.org/10.4230/LIPIcs.ITP.2019.26

[39] Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs*. Ph.D. Dissertation. UC San Diego.

[40] Valentin Robert and Sorin Lerner. 2014-2016. PeaCoq. http://goto.ucsd.edu/peacoq/

[41] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Softw. Eng.* 30, 12 (Dec. 2004), 889–903. https://doi.org/10.1109/TSE.2004.101

[42] Kenneth Roe and Scott Smith. 2016. 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*. Springer International Publishing, Cham, 491–499. https://doi.org/10.1007/978-3-319-43144-4_32

[43] Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. 2019. Generating Correctness Proofs with Neural Networks. *CoRR* abs/1907.07794 (2019). arXiv:1907.07794 http://arxiv.org/abs/1907.07794

[44] Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. 2014. Productivity for Proof Engineering. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, New York, NY, USA, Article 15, 4 pages. https://doi.org/10.1145/2652524.2652551

[45] Mark Staples, Rafal Kolanski, Gerwin Klein, Corey Lewis, June Andronick, Toby Murray, Ross Jeffery, and Len Bass. 2013. Formal Specifications Better Than Function Points for Code Sizing. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1257–1260. https://doi.org/10.1109/ICSE.2013.6606692

[46] The Idris Community. 2017. The Idris REPL. http://docs.idris-lang.org/en/latest/reference/repl.html

[47] Makarius Wenzel. 2012. Isabelle/jEdit – A Prover IDE within the PIDE Framework. In *Intelligent Computer Mathematics*. Springer, Berlin, Heidelberg, 468–471. https://doi.org/10.1007/978-3-642-31374-5_38

[48] Makarius Wenzel. 2014. 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*. Springer International Publishing, Cham, 515–530. https://doi.org/10.1007/978-3-319-08970-6_33

[49] Iain Johnston Whiteside. 2013. *Refactoring proofs*. Ph.D. Dissertation. University of Edinburgh. http://hdl.handle.net/1842/7970

[50] Karin Wibergh. 2019. *Automatic refactoring for Agda*. Master's thesis. Chalmers University of Technology and University of Gothenburg.

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

[52] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. 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)*. ACM, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

[53] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. In *International Conference on Machine Learning*.

[54] He Zhang, Gerwin Klein, Mark Staples, June Andronick, Liming Zhu, and Rafal Kolanski. 2012. Simulation Modeling of A Large Scale Formal Verification Process. In *International Conference on Software and Systems Process*. IEEE, Zurich, Switzerland, 3–12. https://doi.org/10.1109/ICSSP.2012.6225979