QED in Context

An Observation Study of Proof Assistant Users

ANONYMOUS AUTHOR(S)

Interactive theorem provers, or *proof assistants*, are important tools across many areas of computer science and mathematics, but even experts find them challenging to use effectively. To improve their design, we need a deeper, user-centric, understanding of proof assistant usage.

We present the results of an observation study of proof assistant users. We use contextual inquiry methodology, observing 30 participants doing their everyday work in Coq and Lean. We qualitatively analyze their experiences to surface four observations: that proof writers iterate on their proofs by reacting to and incorporating feedback from the proof assistant; that proof progress often involves challenging conversations with the proof assistant; that proofs are constructed in consultation with a wide array of external resources; and that proof writers are guided by design considerations that go beyond simply "getting to QED." Our documentation and analysis of these four themes clarifies what proof work really looks like with current tools as well as potential design opportunities that tool builders and researchers should consider when working to improve the usability of proof assistants.

Additional Key Words and Phrases: Proof Assistants, Contextual Inquiry, Human Factors

1 Introduction

Mechanized proofs are increasingly important in many branches of computer science and mathematics. For example, a 2020 report showed that POPL saw about a quarter of published papers in recent years accompanied by mechanized proofs [25]. But mechanized proofs remain quite hard to produce, requiring both substantial expertise and effort.

One way to make proofs easier to write is to improve the experience of using the *proof assistants*, or interactive theorem provers, in which many of them are built. Indeed, tool builders have devised many techniques to improve proof assistants, including better automation [14, 27, 34]; more direct means for constructing complex proofs, like graphical editing [23, 48]; more intelligible views of proof states [12, 32]; automatic suggestions for tactics [45, 49], premises [37, 45], and whole proofs [1, 16]; and utilities for proof repair [20, 40] and reuse [43].

In moments of accelerated technological development, such as the present one for proof assistants, it is important to ensure that researchers are on the same page. Advances need to be calibrated to user needs and aligned with realistic user workflows; without these checks, we risk wasted effort and missed opportunities. Happily, established research methods from human-computer interaction (HCI) can provide just the insights we need [11]. Prior OOPSLA studies have shed light on user needs in functional programming [29] and code-generation tools [7], providing valuable insight to shape research. User-centered research can also generate new research ideas: in property-based testing, recent user studies [17, 18] have led to new tools [19, 33] that address real developer needs. The same kinds of insights can guide research and development around proof assistants.

We present a study of proof assistant use that paints a rich picture of the realities of proof mechanization, with a focus on cataloging and interpreting the easy-to-miss, moment-to-moment interactions of real-world proof work. We observed 30 users of Coq [47] and Lean [30] for one hour, each doing their own work, and spoke with them in real time about the strategies they deployed and the obstacles they encountered (§3). We offer the following contributions:

- We make four observations about proof assistant usage — specifically, proof writing involves:
  - continual iteration through reaction to and incorporation of feedback (§5),

2024. ACM 2475-1421/2024/1-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- challenges when conversing with the proof assistant about minutiae (§6),
- consultation of an array of resources beyond the current proof (§7), and
- consideration of design aspects beyond simply "getting to QED" (§8).

We describe these phenomena using examples from our observation sessions, concretizing what might otherwise remain "folklore" knowledge and enriching it with descriptions of real situations proof writers encountered in their work. From these examples, proof assistant researchers can reconstruct scenarios of usage situations and replay user rationales.

- To further support future research, we identify five opportunities to capitalize on our observations and advance proof assistant usability, including ideas for how future versions of proof assistants can support exploratory proving, help proof writers manage details, and align better with users' values (§9).

2 Terminology

We assume in this paper familiarity with the basic functionality of proof assistants, but do not assume deep knowledge of a particular proof assistant — the reader should know what tactics and proofs states are, for instance, but they need not know the behavior of specific tactics in Coq or Lean. This section briefly reviews how we use common terms such as these.

A *proof assistant* is a programming environment that helps users write mechanized (i.e., machine-checked) proofs. Proof assistant users could write *proof terms* by hand, but more commonly, they might instead construct proof terms by applying a series of *tactics*. Consider this example in Coq:

```
Lemma add_0_l : forall (n : nat), 0 + n = n.
Proof. intros. simpl. reflexivity. Qed.
```

The code between the `Proof` and `Qed` is a *proof script*, and the `intros`, `simpl`, and `reflexivity` within are tactics. A user would likely write this proof script incrementally; at each step marked with a period, they can evaluate the proof to this point and see what *goal* they should prove next. The goal is visually presented to the user as a *proof state*.

Zooming out beyond individual proof scripts, a *proof development* consists roughly of *definitions*, *lemmas*, and their *proofs*. Definitions describe the structures under discussion. Lemmas are the facts being proven about these structures. (Some lemmas are labeled theorems, propositions, corollaries, etc.; we use the term "lemma" generically for all of these.) We call the union of the definitions and the lemma statements the *specification* of a development.

We use the phrase *proof writers* to refer to users writing mechanized proofs in proof assistants. We sometimes juxtapose mechanized proofs with *paper proofs* — informal proofs written in natural language, whether literally on paper or in a text editor.

3 Methodology

In order to understand the realities of everyday proof work, we designed our study to bring us close to that work. We followed the methodology of *contextual inquiry* [24] from human-computer interaction, which focuses on observing real users doing their own work and speaking with them in real time to understand what they are doing. This methodology complements prior work on proof assistant usability — such as focus groups [8], experience reports [10], predefined tasks [2, 3], and log analyses [41, 46] — by offering a detailed real-time picture of the *process* of proof writing.

3.1 Setting

Scope. We carefully chose the scope of this study — the proof assistants, the specific users, and the kinds of proof work that we chose to observe — to achieve our goals of understanding the usage and usability of proof assistants as they are leveraged to support actual work.

We chose to study two of the most widely used proof assistants: Coq[1] and Lean. These proof assistants share the fundamental interaction model described in §2, but the contexts where they are typically used differ: Coq, first released in 1989, has a strong user base centered in the programming languages community; Lean, released in 2013, has a rapidly growing user base centered in the mathematics community. This choice of proof assistants allowed us to ground the study in observations common to both tools, while also observing some experiential aspects unique to each.

We decided to recruit only proof assistant users who were presently engaged in an open-ended project, excluding, for example, students using proof assistants for homework assignments. This ensured that we observed realistic proof work. Our participants still had varying levels of experience.

While participants worked on tasks of their choice, we often encouraged participants to choose a task where they would be engaging primarily with proof scripts. That is, our observation sessions (and thus the results we present here) focused primarily on proof-centric work, where users are writing or modifying proof scripts, as opposed to specification- or infrastructure-centric work, where they might instead be primarily setting up definitions, notations, etc.

The process of building a proof development can take months or even years. Within the mere hour we had to observe each participant, we wanted to see the aspects of their work that are most specialized to the setting of a proof assistant — where they actively interact with the proof assistant to gradually produce proof scripts. Having said this, what we observed was proof-*centric* but far from proof-*exclusive*. For example, we saw many instances of participants revisiting their specifications in the middle of a proof (see §5.1).

Participants. We conducted study sessions with 30 participants, 15 using Coq and 15 using Lean. We recruited via Zulip forums, personal contacts, and mailing lists. After each session, we asked participants to report some details about their background. The level of experience and occupation of each participant is summarized in Appendix A. Coq participants tended to be more experienced (median four years) than Lean users (median two years) overall, which is expected given that Lean is newer. Twenty-six of the participants identified as male, three female, and one non-binary.

3.2 Protocol

Interviews were 90 minute sessions with a few minutes of short questions and instructions at the start, about 60 minutes of observation in the middle, and an interview at the end. We did not set rigid time boundaries, so the exact breakdown varied from participant to participant; for example, when feasible, we tried to end observations at a natural stopping point. We arrived at this format after some iteration: for the first participant we separately scheduled an interview a few weeks after the observation; for the second and third participants, we ran closer to 60-minute sessions.

Observation. For *contextual inquiry* studies, Beyer and Holtzblatt [24] recommend that the relationship between study participant and study facilitator should be analogous to that of a craftsperson and their apprentice. In particular, the apprentice observes the work in the context it occurs — rather than hearing it summarized later. The apprentice should also interject with questions that clarify their understanding, including by offering *interpretations* of what they observe and having the craftsperson either agree with or refine these interpretations.

Concretely, participants were asked to (a) share their screen and (b) narrate what they were doing and thinking as they worked; they were warned that we would interrupt with questions. Otherwise, they were instructed to do work that they would do regardless of our presence, in the way they normally would. For example, if they would normally use the internet to search how to do something, they should (and did) do so.

[1]Coq is in the process of being renamed to Rocq, but nearly all participants referred to it as Coq, so we will do so here.

Interview. After the observation, we conducted a short interview. We did not use a uniform set of questions across all participants, so that we could instead tailor the questions to what we observed. Some questions simply followed-up to clarify points that were raised during the observation. Others sought to connect the observed incidents with the participant's broader experience. If, for example, we saw a participant spending significant time searching for lemmas, we might ask: We saw that you used X and Y methods for searching — is there a reason you did not also use method Z? How do you find the search process generally? Are there aspects you find especially tedious or challenging?

3.3 Qualitative Analysis

In total, we collected about 43 hours of session recordings, with automatic audio transcriptions produced by Zoom. Two authors heavily revised the transcriptions to correct transcription errors and embed notes about the actions participants took and code snippets participants worked with.

The first author then conducted a thematic analysis [9] of the transcripts. This analysis involved an initial *open coding* pass using the Delve qualitative coding tool. In this pass, the author reviewed transcripts for interesting patterns of usage and tagged them with *codes*. These codes were updated throughout analysis for consistency and completeness. The entire authoring team gave feedback on the codes by reviewing examples and the code book as a whole during meetings and asynchronous review. When the first author had coded approximately 75 percent of the transcripts and informally reviewed the others, another author audited the analysis by spot-checking excerpts for all codes. The first author then revised the code book to incorporate this author's feedback and completed a second (*axial*) coding pass to apply the new code book.

The resulting organization of themes informs the organization of the paper. All examples were also checked against the video recordings for accuracy.

4 Overview of Findings

In the four sections that follow, we explore four different views of proof assistants, each contributing a perspective on what successful interaction with them should look like. These perspectives are overlapping, representing processes that are often taking place simultaneously, though at different levels of resolution and involving complementary features of the proof assistant.

The first two sections characterize the work involved in developing proofs. In §5 we examine what it looks like when proof work is moving along. We call attention to the ways in which proof writers leverage feedback from the proof assistant to guide their progress and the ways that they seek out actionable feedback from the environment. In §6 we identify challenges that arose — frequently at the very lowest levels of interaction — as proof writers attempted to communicate their intent and interpret the outcomes of their actions.

The following two sections characterize usability considerations beyond the proof itself. In §7 we show that proving requires frequent use of external resources such as lemmas, prior proofs, and reference paper proofs. In §8 we describe how decisions in proof implementation are often influenced by concerns beyond simply getting the proof to compile, such as the desire to make proofs more easily maintainable or to make proof intent more transparent.

Readers who are frequent proof assistant users will likely find many of the described processes familiar; our goal is to take these processes that might otherwise be second nature and illuminate aspects that may have future relevance for work on improving proof assistants.

5 Proofs in Motion

By watching proof writers engaged in their work, we can observe what it actually looks like to navigate the complexity and uncertainty of the proving process. Viewing their work not in terms of finished products but as "proofs in motion" gives us insight into how proof writers use proof assistant feedback to iterate on their proofs, switch between tasks, and try out different solutions.

5.1 Iteration

Writing a mechanized proof is an iterative process. Proof writers frequently realized, due to proof assistant feedback, that they should change earlier parts of the proof effort, and then propagated the changes, again with the help of the proof assistant, by replaying and repairing the proof.

5.1.1 Realization. We start by examining moments of realization — when proof writers realized, in the middle of a proof attempt, that they should not continue the proof as-is but instead iterate in some way, such as by revising the current specification or extracting a new lemma. We observe in particular how proof writers used the proof state to arrive at these realizations.

Specification Revision. One situation, encountered by 14 participants, was realizing while writing a proof that the specification being proven should be revised. This often occurred when proof writers detected that a proof has entered into a bad state. Sometimes, the badness was self-evident: L4 and L13 both encountered base cases that simplified to the unprovable goal `False`, leading them to find and fix errors in their lemma statements.

But other times, proof writers needed to notice something more subtle about their proof state. C3 narrated that they had a relation `RR` in an assumption and `RR'` in the conclusion of incompatible types. This mismatch told them that their goal was not provable. C3 returned to their lemma statement to revise it, and continued to write and rewrite different versions of the statement, often stepping back into the proof to gauge how it did or did not progress, for the next thirty minutes.

Lemma Extraction. Another situation requiring iteration, encountered by 15 participants, was realizing mid-proof that a new lemma would be useful. Because they were at or near a place in the proof where they intended to actually apply the lemma, proof writers sometimes *extracted* snippets of the proof state and directly used them in the lemma statement.

For example (see note[2]), L10 copied this subexpression from the proof state and pasted it as a starting point for a new lemma:

```
(fun i => Polynomial.coeff (Polynomial'.toPoly (head :: tail) i))
```

They modified the expression and placed it in an equality, where the right-hand side was what they wanted the expression to simplify into. The eventual lemma statement said

```
Polynomial.coeff (Polynomial'.toPoly p) i = p.getD i 0.
```

L10 thus used the proof state snippet as an input for adding new structural elements to their proof.

In fact, some study participants deliberately advanced their proofs to seek proof states that aided extraction. C1 realized they needed a lemma near the beginning of a proof, but, after struggling to write its statement directly, returned to progressing the proof until they reached the "interesting" part of their proof. They then temporarily copy-pasted a few lines of the proof state below their lemma so that they could reference it while writing the lemma statement. C12 also did a complex series of maneuvers to extract a lemma: first, they copy-pasted a segment of a proof state, including an assumption H, to form the basis of a new lemma statement; then they realized they wanted H to be phrased differently, so they returned to the proof and unfolded a definition; finally, they copy-pasted the new H and edited the proof state excerpt into their desired lemma statement.

C8 went a step further — they set up a "bare-bones, ugly script," where the explicit purpose was to discover what lemmas they should extract for future use. "I'm going to see if I can sort of reverse

[2]Throughout this paper, we provide code snippets to supplement our examples. They are primarily intended to help the reader visualize what is happening, and relevant details will be called out. It is not important to understand every detail.

engineer what I would expect to get as a lemma," they explained. They started drafting a lemma in the middle of the script, while the proof state was still visible, before moving the lemma to its final location. Similarly, C8 also intentionally began proving a statement that lacked necessary assumptions, so that they could wait for the assumptions to "jump out" as they progressed the proof and add them then. Further examples of proof writers setting up environments to receive desirable feedback from the proof assistant can be found in §5.4.

Participants almost always did lemma extraction manually, with one notable exception. C6 used Coq's `clear` tactic to remove all but one assumption from the context. They then used `Company-Coq`'s `lemma-from-goal` command to generate a lemma from the proof state. Later, they cleaned up the lemma, e.g., by renaming argument names `i` to `id` and `i0` to `instr`, and realized during proving that they needed an additional assumption. That is, `lemma-from-goal` replaced some of the manual aspects of extraction, though naturally it could not eliminate the reasoning required to determine when a lemma is needed and what shape it should take.

5.1.2 Propagation. We next examine what happens when proof writers finished an iteration cycle by replaying and repairing proofs after a change. Consider C5, who spent their observation session experimenting with different ways to resolve a problem with their specification. As is typical to using a proof assistant, each time C5 made a change, they replayed their file and the proof assistant informed them where proofs succeeded or failed. In fact, they described the ability to receive such feedback as "the magic" of working in a proof assistant.

Proof writers experienced this magic whenever they propagated a change to the proofs it affects. After fixing an error in their lemma statement, for example, L4's previously failing proof refreshed to now succeed, providing immediate feedback that the fix was correct. Of course, changes sometimes instead caused failures, and some of these indicated that the change needs further iteration. When C1 added a new instance of a typeclass and recompiled the files in their development, they realized previously working proofs now broke; they needed to limit the scope of the instance so that it would not be used in those proofs.

Failures also indicated places requiring proof repair. After swapping the sides of a bidirectional lemma statement, L11 tweaked proofs that broke because they relied on the wrong direction of the lemma. After a more substantial change to their specification, L9 commented out their broken proofs and interleaved writing new proof snippets, copy-pasting old proof snippets, and repairing those old snippets to reflect the changes. This process has many similarities to proof reuse (§7.3).

5.2 Context Switching

Proof assistant feedback is helpful in other situations too — in particular, *context switching*. Proof writers frequently switched away from a goal and later switched back again. In doing so, they relied on the proof assistant to maintain the proof state, so they could resume right where they paused.

Twelve participants used Coq's `admit` or Lean's `sorry` to temporarily skip a goal and work on a different one, allowing them to prioritize the goals they want to work on. Some prioritization occurred on the fly, as when proof writers decided to start with an easier case of a proof. L3, for example, narrated after generating two subgoals, "[The second] goal is easier, so we'll take care of that first." Prioritization sometimes also reflected higher-level strategization. C11 explained that they use `admit`s until they can establish the "general skeleton" of their proof. C7 explained they chose to prove a more complex lemma before proving the simpler lemma it depended on to prioritize ensuring the complex lemma statement was correct.

Implicit in these cases is the fact that proof writers know proof assistants can re-supply the proof state at a skipped goal when the proof writer returns to it. L6 noted this explicitly, saying they found context switching in a proof assistant to be "much cheaper" than it would be on paper. When writing paper proofs, the contextual information is "just in the back of your head, and then if you switch contexts, you forgot what exactly was in the back of your head." When writing mechanized proofs, the context is automatically provided.

5.3 Small-Stakes Trials

Given the complexity of mechanization, proof writers sometimes had an inexact understanding of what would or would not be accepted by a proof assistant. In these cases, to figure out the right way to formulate their next step, they engaged in small-stakes trial and error — by interacting with the proof assistant as an oracle that provides reliable, instantaneous feedback.

In our study, 18 participants engaged in clear instances of trial and error, which we identified as such because they narrated their uncertainty about whether their attempt would work and/or tried a series of similar solutions in rapid succession. These trials were performed in tightly scoped ways, often at the granularity of a single tactics or arguments to that tactic. Participants sometimes succeeded in a matter of seconds; other times, after they had cycled through the immediately obvious solutions, they had to pause and try a different approach.

Though trial and error behavior suggests the proof writer's knowledge of the proof assistant is inexact, knowing which potential solutions to try and how to respond to errors still reflects substantial expertise. Consider, for example, when C4 was trying to prove an equality where the only difference between the two sides was $n - 0$ versus n. They made this series of attempts over the course of 30 seconds:

```
1. change (n - 0) with n. ✗          2. change (n - 0) with n%Z. ✗
3. change (n - 0)%Z with n%Z. ✗      4. replace (n - 0)%Z with n%Z. ✓
```

That is, they realized that they had type-checking issues after Attempts 1 and 2. Then, they realized that in fact the `change` tactic, which tries to automatically convert one expression into another, would not work at all and that they instead needed to switch to `replace`, which generates a new goal $n = n - 0$. To prove this goal, they made another series of attempts over the course of 40 seconds, (here, the ✗ indicates either a tactic that failed or did nothing):

```
1. auto. ✗   2. done. ✗   3. simpl. ✗   4. auto. ✗   5. lia. ✓
```

While C4 did not immediately remember that the `lia` tactic would solve the goal, they did know that tactics such as `auto` and `done` might plausibly solve trivial goals like this one. We observe through examples such as this one that proof writers are able to use the proof assistant's feedback to turn their substantial but inexact knowledge into a working proof.

5.4 Sandboxing

We saw previously how C8 deliberately set up their proof environment so that they could use the proof state to assist with extracting lemmas and assumptions. Four participants additionally created *temporary environments* that would better elicit the proof assistant feedback they wanted.

C14, for example, created a temporary lemma whose stated purpose was to "test whether [the proof assistant] knows" that a particular object is an instance of a particular typeclass. They tried proving the lemma by `exact _`, which should succeed if Coq does know this, but the test failed. They realized they needed to add an import, the test then succeeded, and they erased the lemma.

In addition to quick fact-checking, temporary environments also assisted with figuring out proof steps without the clutter of the larger proof context. L4 struggled to show in the middle of a proof that $10^1 \leq 10^{2^i}$, given $1 \leq 2^i$ and a number of other, unnecessary assumptions. They wrote a temporary goal with just the essentials: $10^a \leq 10^b$ if $a \leq b$. Once they figured out this goal, they ported the proof over to where they were originally. L4 noted that this strategy of separating out a "self-contained example" helps to "remove some of the distracting hypotheses and syntax," and has the additional benefit of sometimes making automated library search tactics (§7.4) work better.

6 Conversing with the Prover

Proof writers constantly encountered challenges conversing with their proof assistants. In one direction, they needed to speak to the proof assistant in a way it can understand, at sometimes gruesome levels of precision. In the other, they needed to understand the proof assistant's feedback, which could include unwieldy proof states and confusing error messages.

These communication challenges were significant sources of friction. C3 expressed their frustration with what they called "overhead," saying, "Fifty percent of my time is figuring out why doesn't Coq do the thing that is very obvious, and fifty percent of the time is actually reasoning about the things which are important." L1 spent the observation session dealing with "nonsense," saying that only after an hour of this were they finally ready to address "mathematically meaningful questions."

6.1 Speaking to the Prover

When speaking to the prover, a proof writer needs to translate the ideas in their head into a highly precise and sometimes unnatural language.

6.1.1 Precision. Proof assistants demand extreme precision, which often led to proof writers fussing with details that are incidental to the main ideas of the proof.

Proof writers faced precision problems when they understood, conceptually, what tactic they wanted to use to advance their proof, but had trouble invoking the tactic in exactly the right way. For example, L10 wanted to do a case analysis on the expression `p.length`. Writing

```
cases p.length
```

did not work as intended: it led to an impossible goal where the conclusion was only true if `p` was empty, but the fact that `p.length` was zero was missing from the context. During the session, L10 solved the problem by doing a case analysis directly on `p`, but we determined later in the interview that it would have worked if they had written

```
cases h : p.length
```

to force the necessary information to be retained in an assumption `h`. That is, L10 knew what they wanted to do (a case analysis on the length), they knew the tactic to do it (`cases`), and they correctly identified the issue that arose (that length was not retained as an assumption), but they had to pivot to a different approach because they didn't realize they could just add two characters.

In another example, C4 briefly encountered an issue where they tried to unfold the definition of `len`, which was implicitly used in an assumption. Nothing happened. After some investigation, they realized they needed to first unfold an outer definition; only then could they unfold the inner `len`. C4 remarked, "Coq really needs little steps by little steps always."

Another source of precision problems was when proof writers decided to use a lemma but needed to figure out exactly how to use the lemma as desired. This could involve pinpointing where in the goal they wanted to do a rewrite. L15 commuted specific summands in their goal by writing

```
conv =>     lhs     lhs     rw [add_comm]
```

(with tabs representing newlines); the two `lhs` commands tell Lean precisely where to rewrite using `add_comm`. When applying lemmas, participants often needed to provide arguments manually; for example, C8 had to instantiate the seventh argument to a lemma:

```
iApply (ewp_sitem_open _ _ _ _ _ _ (ieq ?[y]))
```

They described this as a "weird hack" for unification to work. From these examples, we see that it was not enough for proof writers to know the lemma they wanted to use and how, conceptually, to use it; they also had to carefully express the usage in the precise language of the proof assistant.

6.1.2 Naturalness. Styles of reasoning that are natural to how proof writers want to think about their proof or write it on paper sometimes feel unnatural in a proof assistant.

Mathematical intent that is easy to express in paper proofs can require extra effort to mechanize. For example, Lean provides a proof mode, `calc` blocks, to facilitate proofs involving chains of equalities. However, while it is very natural in a paper proof to also, say, subtract or divide a term from both sides of an equation, L13 found that they have to "jump through a few hoops to make that style of proof fit in a `calc` block well." For C14, one challenge that seemed "fundamental" to the formal proof setting was that it can be "quite non-trivial" to convert between equivalent representations of a definition. By contrast, "When you do things on paper, you can fluidly jump between different design choices, as long as you know how to make up for it."

A few participants commented on the difference between *forwards reasoning*, where the proof proceeds from the assumptions towards the conclusion, and *backwards reasoning* from the conclusion towards the assumptions. L8 said, "When I think of a properly written proof in mathematics, I think you should be always going forward and trying to justify what you're doing. This implies this, and this is true because of this." C11 said that forward reasoning seems "a little bit more natural to human brain reasoning," but Coq is "constructed to make backwards reasoning super easy."

C15 also said that, in a proof assistant, it is "usually more tempting" to do backwards reasoning. A downside of backwards reasoning is that one might start "using the proof assistant like a video game" and become narrowly focused on making incremental progress. Forward reasoning, by contrast, requires active thinking about what the "intermediate assertions" of the proof are. C15 cited ease of forward reasoning as a reason they like `SSReflect` tactics.

6.2 Listening to the Prover

Listening to the prover requires just as much translation as speaking: proof states and error messages can be unwieldy and confusing for those who are not already fluent.

6.2.1 Unwieldy States. Proof states are critical for tracking proof progress, but they can be long and complex. C1, for example, had at one point 44 lines of variables and assumptions in the context but noted that only a few lines were relevant.

One common challenge with managing proof state complexity is determining which definitions to unfold and which simplifications to perform so that the proof operates at a desired level of abstraction. Too much unfolding leads to proof states that are too verbose. C8 said that, at the start of their project, they wanted to let Coq "compute as much as it can," but this led to an "explosion" of the proof state, to the extent that goals became over 30 lines long. C4 shared that they found it difficult to find the "sweet spot" when unfolding definitions, where the right details are exposed but before the proof states become "way too big for you to even understand what it says."

Oversimplification can also cause problems later in the proof. L9 encountered a complex proof state that they transformed into a shorter, simpler state with the `simp` tactic. The next step of the proof failed, so they tried removing the `simp`, and the failing step worked! "Wait, what?" they wondered. Upon further examination, it appeared that the seemingly helpful simplification rearranged their state so that typeclass inference failed. That is, even actions that make the proof state nicer to look at can have bad downstream effects.

Despite challenges with proof state management, we were surprised at how adept participants could be at interpreting the domain specific details of their proof states. For example, C10 encountered a proof state excerpted below, describing it as a "big ugly thing":

```
  (holds
    (set_nth 0 ([seq row_mx u v ord0 i0 | i0 <- enum 'I_(n + 1)] ++ r)
      (n.+1 + i) x)
  ... 7 more lines
```

After considering for just a moment, they concluded that it "essentially gives me what I want."

6.2.2 Confusing Errors. When a proof assistant rejects a proof step, it displays an error message to help the proof writer debug. Sometimes this message is not actually so helpful.

Indeed, proof writers can be misled by error messages, especially when there is some distance between the root cause and the trigger of the error message. C3 modified an `intros` tactic to manually provide assumption names, instead of using auto-generated names. But then a previously successful `apply` tactic later in the script, which did not refer to any of those names, now gave a "failed to unify" error. After a moment of bewilderment, C3 found that they had accidentally provided only two of the three assumption names to `intros`, causing the proof state to have the wrong shape.

Proof writers can also struggle with error messages that expose low-level details that they do not want to think about. L9 had a rewrite that failed with "motive is not type correct,"which they were "never really sure how to deal with." They fixed the issue with some trial and error. "I'm very much a mathematician," L9 said. "I don't know much about ... the underlying stuff going on in Lean. I just try to work around it." Similarly, C7 recounted difficulties with debugging typeclass issues due to unhelpful error messages:

> "The error message just says a whole bunch of stuff – something evars, and lots of shelved stuff — and you have no idea what's going on. It doesn't tell you what's missing. ... It tells you at a very low level, oh, we [the proof assistant] can't unify this, we can't find something. But the thing they tell you is not really close to what you actually need, and that gets really frustrating."

Sometimes, these low-level details reveal that portions of the proof state that appear the same are in fact not. C6 found themselves with a goal of `t1 && t2` and a seemingly identical lemma with a conclusion of the form `t1 && t2`. They tried to solve the goal by applying the lemma, only to encounter an error message of this form:

```
Unable to unify "if ?M17926 && ?M17927 then True else False" with t1 && t2 = true
```

Where did the `if then else` and `= true` come from? As C6 realized, the issue was that the goal and the lemma were implicitly relying on two different, incompatible ways of coercing booleans into propositions, despite the fact that they looked exactly the same on the surface.

7 Proof Sources and Resources

Mechanized proofs are rarely written completely from scratch. Instead, the proof writer might adapt a proof from an existing source, such as by translating a paper proof or by reusing a previously mechanized proof; they might also rely on existing resources, by searching for relevant lemmas or by seeking information about proof techniques.

7.1 Translating Paper Proofs

Some mechanized proofs originate in whole or in part on paper — a more malleable, less rigorous medium. Five participants explicitly referenced a paper proof during the observation session, and a few others mentioned that they do so in the interview. The sources of these proofs ranged from mathematical papers to textbooks to olympiad problem solutions.

Mechanization generally requires making many implicit details in a paper proof explicit. L4 remarked that paper proofs are often ambiguous as to whether a variable should be universally quantified or whether it specifically refers to something in the current context. For example, in a paper proof they translated during the observation session, the proof inducted on n, but also made intermediate statements that were implicitly true of all n. L4 chose in this case to err on the side of universally quantifying these statements, so they could more flexibly apply the statements later.

Details about the structure of the proof, especially if they differ from the proof assistant's built-in support, can also be tricky to handle. L2, for example, described a textbook proof that proceeded by "induction on the absolute difference of these two numbers," subject to some bounds, noting that, though complex, this induction strategy is "understandable pretty quickly for a human." But they had difficulty expressing it in Lean, needing to rely on (and justify) a custom induction principle.

Though challenging, the particularities of the mechanization process can also be precisely why mechanization is valuable. L1 observed that, in a mathematical paper, the author might claim the existence of an algorithm that is implicit in the proof, but the proof might not make explicit what that algorithm is. "When you migrate that proof into Lean, you actually need to construct it, and then the construction actually produces the computational content," L1 said.

L5 noted that one reason they derived value from mechanization is that using Lean is "very clarifying in terms of taking my intuition for how these objects should work and turning them into actual, proper mathematical definitions." The difficulties in mechanizing certain kinds of definitions are not inherently bad and can instead lead to better design choices. For example, they explained, they had something "essentially coinductive" but wanted to avoid coinduction both because Lean did not support it well, "*and relatedly*" because it is uncommon among mathematicians — Lean tends to focus on supporting techniques that are in common usage. That is, L5 said, "The expositional problem of how should I write this down in a way that will be accessible to mathematicians is sort of correlated to what did Lean actually make the effort to support."

7.2 Doing Scratch Work

Proof writers sometimes did scratch work alongside mechanization, either on paper or in a code comment. For example, C1 (on paper) and L4 (in a comment) worked out examples of how a definition should work on small inputs. C1 used their examples to guide the Coq definition, while L4 used the examples after writing the Lean definition, to think through and fix an off-by-one error.

L13 intermittently wrote on paper during the observation session. "I don't have a strict set of equations that I'm following step by step to translate into Lean, and so I'm swapping between thinking about the proof in Lean and then going back to think about how I'd write this as an informal math proof," they explained. On paper, they explained, it is easier to do simplifications and read notation such as fractions.

C3 left temporary notes such as this one about what their lemma statement meant, using code comments as a kind of scratch pad:

```
(* [k] will terminate with postcondition [RR] and invariant [φ] *)
```

C3 explained that the "big expression here" (the lemma statement) just says that k will terminate. Writing this fact down "helps me retrieve this fact so I don't have to reparse" the entire lemma.

7.3 Reusing Proofs

Of course, mechanized proofs can resemble not only paper proofs, but also prior mechanized proofs. An important aspect of this task was finding a chunk of code elsewhere in their development that the participant believed would sufficiently resemble the situation at hand. This process involved recognizing conceptual similarities between parts of the development.

Proof writers drew connections between lemmas based on commonalities in the statements, despite some differences. L3 identified a relevant prior lemma because the underlying "patterns of computation” in the functions involved were essentially the same, even though the literal “computational object”s differed. C1 said if they had a proof where, for example, lists are an instance of the `functor` typeclass, they might adapt this to prove that the same data structure is also an instance of a different typeclass (e.g., lists are traversible functors) or that a different data structure is an instance of the same typeclass (e.g., trees are functors).

In addition to identifying similarities in proof structure across lemmas, participants also identified opportunities for reuse within the same lemma. C2’s task, for example, was to extend an existing proof after new typing rules were added. They frequently looked to previously proven cases, chosen based on their type theoretic knowledge of which “derivations sort of have the same shape,” such as when “substitutions are in the same places.”

7.4 Searching for Lemmas

An extremely common activity when writing a mechanized proof is *lemma search*: finding a suitable, previously established fact to advance a proof. Proof writers leveraged existing tools for lemma search in combination with their — often quite specific — assumptions about the target lemma.

7.4.1 Engines. We start with an overview of the kinds of lemma search features (“engines”) participants used, which differed substantially between the two proof assistants.

Ten Coq participants used the `Search` command (either directly, or indirectly using an editor shortcut) during the observation session. Queries contained substrings of the lemma name, identifiers appearing in the lemma statement, patterns that the lemma statement must obey, or a combination. As an example of searching by lemma name and text, C1 ran the command `Search binddt “rw” letin`, which returns lemmas whose name contains the substring “rw” (due to the quotes) and whose statement contains `binddt` and `letin`. As an example of searching by pattern, C10 ran a command of the form `Search ?a + _ == ?a + _`, which returns lemmas whose statement contains as a subexpression the `==` equality of two sums whose first arguments are the same.

On the Lean side, participants used several search engines: Nine participants used question-mark tactics, which automatically search for and suggest lemmas that can be used in the current proof state, subject to some criteria. For example, `simp?` tries to return a chain of simplification lemmas. Five participants looked for a lemma through the `mathlib` online documentation, whether by querying substrings of the lemma name through the search bar or by navigating to a relevant definition and browsing nearby. Three participants used `Moogle.ai` to do lemma search; Moogle describes itself as a “semantic search engine” for `mathlib` that accepts natural language queries. Two participants used `Loogle`, which has similar functionality as Coq’s `Search` command.

Lean participants sometimes mixed-and-matched these mechanisms, moving to an alternative when they were unable to find the lemma they were looking for. L10 spent 15 minutes searching for a lemma before determining that it was not yet in `mathlib`, and writing a pull request to add it. During this time, they used approaches including `simp?`, the documentation, and Moogle.

7.4.2 Specificity vs. Fuzziness. When choosing a search engine and formulating a search query, proof writers had to consider what assumptions they have about the lemma’s name or contents, and how accurate they think these assumptions are. Searches with a high degree of *specificity* often accelerated the process, where the target lemma was one of just a few results, but even subtle inaccuracies could cause the lemma to not be included at all. On the flip side, search strategies that permitted *fuzziness* were more forgiving of inaccuracies, but also less effective at narrowing results.

Search by Name. An especially specific approach is predicting the lemma name based on naming conventions. One common convention is for the lemma name to be tied to the definition names within the lemma statement. C10 reasoned they had “`nth` in front of `set_nth`” in their goal, so the lemma they needed was probably `nth_set_nth`. (It was!) Similarly, L10 noted that `mathlib` lemmas are "named for the sequence of functions that are applied as they appear."

L15 found eight lemmas directly through the `mathlib` documentation's search bar, often by trying variations in rapid succession; for example:

searching for...	`multiplicity.finite_prime_left`		*searching for...*	`Nat.lt_of_succ_le`	
queried...	`finite_of_prime`	✗	*queried...*	`Nat.lt_off`	✗
	`multiplicity.prime`	✓		`Nat.lt_iff_succ`	✗
				`Nat.lt_of_succ`	✓

The `mathlib` search bar allows queries to be a non-contiguous subsequence of the lemma name (e.g., `multiplicity.prime` above) but is unforgiving of other discrepancies. We see above that fatal discrepancies included both typos ("off" instead of "of") and conceptual errors about the contents of the lemma ("iff," used in `mathlib` for bi-implications, instead of "of," for single implications).

L15 explained that contributing factors to their success in searching for lemmas by name were that they had seen many of the lemmas previously and that they were familiar with `mathlib`'s naming conventions. Such conventions can require quite fine-grained knowledge: L13, for example, described difficulties knowing conventional abbreviations, such as "coe" for "coercions," where searching for the full word would often not elicit the target lemma.

Search by Pattern. Another approach that enables a high degree of specificity involves searching by the shape of the lemma, in particular with the usage of search patterns in Coq. (Although Loogle does support patterns, we did not observe any Lean participants search by pattern.)

Patterns can fail to match a lemma in subtle ways. When C10 sought a lemma that contained the subexpression `\poly_(i <? n) E1 i`, their attempt to search via the pattern `\poly_(_ < _) _` did not return that lemma. They speculated that while the notations match visually, the underlying terms might differ. C4 said they found it challenging that a lemma that is conceptually equivalent to their search might be excluded, such as if they flip the two sides of an equality.

Search by Subject. A fuzzier approach is to search by definition names that should appear within the lemma statement, but not details about how the definitions should be related.

From L14's perspective as both a library designer and user, when thinking about what kinds of lemmas they tend to reach for, they explained that "the most common thing that happens is I have two concepts, and I want to see how they interact." They used the Loogle search engine to search for pairs of definitions. As we saw above, Coq's `Search` command also supports this kind of search.

Combining a more specific approach such as patterns with this approach can greatly narrow a search. C15 searched for just `Permutation` before eventually adding a pattern and searching for `Permutation (_ :: nil)`. With the new query, their desired lemma `Permutation_singleton_inj` was the first of eight results, whereas previously it had been the 36th in a long list.

Search by Natural Language. The fuzziest approach of all is natural-language queries.

One situation where support for natural language search *would* have been useful was that of C4, who searched for the substring "range" in the source code of the library they were using. They did not find the lemma they were looking for; later, it turned out that the lemma was in the file they were browsing, but it did not contain the string "range" in its name or body, since the range was instead written in the form an inequality. C4 said in the interview that they wished for an engine that is "much closer to the intention of the search rather than what's strictly written."

Moogle supports natural language queries for `mathlib` lemmas, though it appears to be sensitive to small differences in phrasing. L4, who said they usually reach for Moogle first, searched for "power of sum equals product of powers." Unfortunately, this query returned results about power sets, whereas they wanted lemmas about powers in the sense of exponents. They then changed the first word of their query to "pow"; the first result was the lemma they needed.

Search in Context. Search strategies vary not only in query specificity but in context specificity, the extent a search engine is aware of where a lemma will be used when deciding what results to include. For example, among Lean's `simp?`, Coq's `Search` command, and Lean's `mathlib` documentation, `simp?`, which returns lemmas only if they can be used in the current proof state, is the most context-aware; then `Search`, which returns lemmas only if they are in scope in the current file; then the documentation, which, as an online resource, returns lemmas regardless of context.

While context awareness certainly helps with reducing the quantity of results, lack of context awareness can also be useful. L10 found a lemma through the `mathlib` documentation that they did not initially find through `simp?`, since it was not yet imported. C8 found a lemma with the `Search` command at one point in the proof and then did not use the lemma until minutes later, after they had performed the necessary rewriting. When C4 searched for lemmas about one definition, they noticed that many results contained expressions where this definition was composed with another. They changed their goal to match this form and then were able to use a lemma from the query.

7.5 Seeking Other Information

Proof assistant users, even experienced users, may need to learn about tactics and techniques they are unfamiliar with while writing a proof. This can be a difficult process: L8, for example, said they find tactics to be "a black box" and that they need to learn them on a "tactic-by-tactic basis," as opposed to being able to learn "general principles." To facilitate information seeking, participants leveraged a combination of documentation, examples, and community channels.

7.5.1 (Not) Using the Documentation. C8 said that in the past few months they had used the Coq reference manual "a dozen times a week at least." C2, on other hand, said, "I generally don't tend to read a lot of the documentation and just sort of figure out what's going on with whatever tactics just by experimentation." C2 briefly accessed documentation about the syntax of the `SSReflect` library during the observation, but then decided, "I don't think I need to understand it."

It can be quite challenging to guess what a query for an unknown tactic or technique would look like. When asked about their experience with having an abstract idea of what they want to accomplish and finding the tactics to do it, C8 said they found this process to be "very difficult" and that they "spent maybe a week trying to do something once."

An alternative to querying the documentation is simply reading it. L13 shared, "I will sometimes just scroll through the tactics list and read about some that exist, and then hopefully in the future, I will remember that I have learned about a new tactic, and maybe I'll be able to use it." In fact, they said, "Rarely do I discover tactics while I'm actively coding."

7.5.2 Using Examples. Proof writers made use of code examples, both by reading textbook examples and by adapting snippets from library source code.

L10 and L13 used examples of the `cases` and `induction`' tactics to help them figure out the correct syntax. In L10's case, they found the example directly in the documentation, by hovering over the `cases` tactic in the `VS Code` editor, and in L13's case, they found the example in the *Mathematics in Lean* online textbook.

When creating a new typeclass, L12 copy-pasted a mathematically adjacent typeclass definition already in `mathlib` and modified it for their use case, since they did not know all of the syntax "off the top of [their] head." L12 said that because they are relatively new to Lean, they tend to rely on "looking at the patterns that other people who are writing this code are using."

The process of adapting an example can be elaborate, as in the case of L8, who was seeking to learn how to develop a custom induction principle to reduce repetition in their proofs. Since they remembered seeing something similar in the Lean source code for division, they navigated to that file and located a proof that used `div.inductionOn`. They copy-pasted the proof into a new file and developed a modified version that did not use `div.inductionOn`. Indeed, they explained that their strategy was to take an example using the desired, "idiomatic" approach and work their way *back* to the "wrong" approach, so that they could see a connection that would then help them move their code that used the wrong approach towards the idiomatic approach. (The observation session ended while they were in the midst of the second step.)

Part of the complexity of this situation can perhaps be attributed to the fact that the example L8 used was not an intentional, pedagogical example of the technique they were trying to learn, but rather a piece of source code that they happened to know existed. They said they wished there were more examples, especially ones that are not too basic, as they would be difficult to generalize to more complex situations, but also not too advanced, as they would be difficult to understand. Similarly, C8 wished for more examples of "more hardcore tactic language uses." They cited the repository `coq-tricks`[3] as the kind of resource they wished there were more of.

7.5.3 Asking Others. Seven participants explicitly said they seek help from other proof assistant users, whether in their local community or by asking (or browsing) on forums such as Zulip.

C8 commented that they are "very lucky" to have more experienced colleagues to ask for assistance. "Without the people in my office building, I would have had a lot more trouble going from novice to intermediate," they said. At one point during the observation, L13 was trying to unfold a definition only on the left-hand side of their equation. They first went to the documentation for the `unfold` tactic and then, not seeing a solution, they searched in the Lean Zulip for "unfold left hand side." From a thread asking the same question, they found the suggestion to use `conv_lhs`, which worked. L12 said they often post "silly questions" on the new members stream in the Lean Zulip, acknowledging that they feel comfortable doing so, but this may not be the case for everyone.

7.5.4 Copilot. A few participants had Copilot, an AI assistant in Visual Studio Code that suggests code snippets, enabled as they worked, and occasionally accepted its suggestions. Because the number of usages was small, and because programmer-AI interactions are an area of research worthy of separate examination [7], we do not discuss Copilot further here.

8 Beyond QED

While the headline benefit of a mechanized proof is ensuring that some top-level theorem is true, we saw that proof writers also cared deeply about the proof itself. Throughout our study sessions, participants described the numerous, subtle, and sometimes conflicting qualities they value in proofs, beyond simply "it compiles." In this section, we focus particularly on maintainability, communication of mathematical ideas, and compliance with conventions.

8.1 Maintainability

Proof scripts change over time, in response to a wide variety of factors. In light of this, 13 participants expressed concern about *maintainability* of their proofs as they evolve.

A core aspect of maintainability is *robustness* to changes. L2 made a proof robust to changes in its underlying definitions by enforcing strict abstraction boundaries. During the session, they initially wrote a proof using the `rcases` tactic, whose behavior relies on the implementation details of the

[3] https://github.com/tchajed/coq-tricks. Its README notes, "Some tips, tricks, and features in Coq that are hard to discover."

definitions involved; then, they refactored the proof to use a lemma that preserved the abstraction barrier between the proof and these details.

But participants wanted more than just robustness to failure — they also wanted to ensure that, if the proof does fail, it does so in a way that is conducive to understanding and fixing the failure. C5 said they prioritize structuring a proof so that it "breaks exactly in the place where things actually break." C5 showed an example of a proof in their development structured like the one below on the right and explained why they preferred it to the alternative on the left.

```
   Proof.                   Proof.
     induction H.             induction H.
*    1: constructor.          1: now constructor.
     (rest of proof)          (rest of proof)
```

In this context, `constructor` alone solves the first goal, so the `now` is unnecessary. But suppose a change were made so `constructor` no longer solved the goal. The right proof would fail precisely at Line *, since `now` fails if the goal is not solved, while the left proof may fail at some unknown point later in the proof. C5 proactively uses "terminators" such as `now` to make failure localization easier.

The relationship between automation, robustness, and ease of fixing failures is complex: in some cases, automation improves robustness, obviating the need to fix failures; in other cases, it causes proofs to break in ambiguous and less local ways. Consider, for example, Lean's `simp` tactic and variants. The `simp` tactic automatically applies known lemmas in a black-box way. Alternatively, `simp only [lemma1, lemma2, ...]` applies only explicitly provided lemmas.

Is `simp` or `simp only` preferable for maintainability? It depends! L9 encountered an error in the last line of a previously working proof:

```
simp only [vcomp_eq_comp, comp_app, id_app', id_comp].
```

Upon examination, they realized that they now needed to refer to the lemma `comp_app` as `NatTrans.comp_app` instead, likely due to a namespace change. They could just make the fix within the `simp only`, but they opted to instead replace the line with just a `simp`, which automatically figures out the correct name. (Immediately after, they further refactored their proof to replace `simp` and the line preceding in their proof with the proof search tactic `aesop_cat`.) That is, more automated tactics are sometimes preferable because they are more robust to certain kinds of changes.

Conversely, `simp only` may be preferable in other cases. L10 briefly had a `simp` in the middle of their proof, but they converted it into a `simp only`. They explained that they did not want to leave a "raw `simp` in the middle", which simplifies the goal but does not solve it, since if the behavior of `simp` changes internally, then this can cause problems later in the proof. (This is consistent with Lean's official recommendation[4] to avoid "non-terminal" `simp`s.) That is, more restrictive, less automated tactics like `simp only` may assist with failure localization.

8.2 Communication

Participants also cared what their proofs communicated mathematically.

Organization. One means of communication is to indicate the *logical units* of a proof. The following examples demonstrate how considerations around how to organize these units might play out at the granularity of tactics, logical proof steps, and lemmas.

At the tactic level, we asked C6 why they wrote `intros ; cbn` despite the fact that the first tactic only generated one subgoal (so the semicolon was not needed). They answered that they liked to chain together series of tactics that represent "one chunk of thought" so that they would be evaluated as a single unit when stepping through the proof.

[4] https://leanprover-community.github.io/extras/simp.html#non-terminal-simps

At the level of logical proof steps, L7 showed us the proof outlined below, which was the result of a refactoring to better communicate its structure:

```
have H1 i := by
  ... 11 lines of proof
have H2 i :=
  ... 7 lines of proof, which use H1
2 lines of proof, which use H2
```

Originally, the proof was written "upside down." They started with the last line above, which created a goal corresponding to what is now `H2`, and in the course of proving `H2`, they needed to prove what is now `H1`. They noted that the "bulk of the work" happened in the proof of `H1`, and this version of the proof allowed them to "logically separate" that work from the rest of the proof.

C1 showed an example where they had comments `(* Merge LHS *)` and `(* Merge RHS *)` interspersed between the lines of the proof. Many of their proofs naturally proceed in "stages," where they performed rewrites on one side of an equality and then on the other side. The rewrites could be done in any order, but instead of "freewheeling," C1 said they had learned to structure their proofs in this organized way to improve understanding. Similarly, they chose not to incorporate heavy-duty automation, since they wanted to be able to step through the steps of their proof. "I really specifically am trying to record *how* that equation gets proved," they said.

Proofs could also be organized at the level of lemmas. C5 focused on communicating the content of their proof not through their tactic scripts but rather through their lemma statements; they aimed to separate out "readable and self-contained" lemmas that form "sensible logical units." Then, they explained, "The big proof by induction is not very interesting, usually. It's about combining all of the things that you have already."

Intent. Beyond structural considerations, proof writers also tried to make stylistic decisions that signal their mathematical intent, such as choosing between multiple viable tactics. L4 discussed goals that could be solved by the `omega`, `linarith`, or `positivity` decision procedures. Of these, "`positivity` is less powerful, but it expresses more intent," since proofs by `positivity` must use "straightforward" reasoning. Indeed, they elaborated, "If I'm reading a proof and I see `linarith` or `omega`, I'm like, I don't want to try to dig into why this is true. I'm just going to trust it. Whereas this `positivity`, it's telling me that you can definitely just glance at this and see what's going on here."

As another example, L2 discussed why they opt to use the `exact` tactic to supply a lemma in certain situations instead of using `apply`, which is more powerful overall. "The idea is, when you `exact`, that's signaling that you've got the final thing that you want," they explained. "You can `apply` a number of theorems, but your last step should almost always be an `exact`. It just signals to whoever's reading the code, now we have the thing."

The fact that proof writers sometimes avoid maximally powerful tactics in favor of ones that convey their intention is reminiscent of what we saw above, where proof writers sometimes eschewed maximally powerful automation in favor of techniques that streamline understanding and fixing failures.

Concision. Sometimes proof writers care not only about the content of a proof but also how concisely it is written. To achieve better concision proof writers may retroactively rearrange their code to avoid unnecessary steps. L3, for example, was writing a proof of roughly this form:

```
match h with
| case1 => simp [h, defn] ...
| case2 => simp [h, defn] ...
```

They tweaked it to unfold defn before the match, allowing them to remove the two highlighted steps. L3 described this as a “neat little trick” to do some “proof golf” (cf. code golf).

Concision can itself be the end goal — some participants expressed that shorter proofs simply feel nicer and cleaner — but it can also tie in with the above theme of signaling intent. Throughout the observation, L9 regularly sought opportunities to decrease the length of their proofs. They explained that the proofs they worked on that day correspond to just one line of mathematical reasoning, so they wanted the mechanized version to be similarly concise. For more complex proofs, they still valued concision, since they want their mechanized proofs to match the number of high-level steps in the “normal math proof” as closely as possible. “A shorter proof signifies that every step I’m taking really matters. It’s really quite crucial that I do all of these [steps],” L9 said.

8.3 Conventions

Mechanized proofs do not exist in isolation; neither do proof writers’ values about what makes a good proof. In particular, they may hope to integrate their work into a larger project, which may in turn come with conventions that contributors are encouraged or required to follow. Even in self-contained projects, users still find utility in good conventions.

Sources of Conventions. For Lean, the dominant context proof writers work in is its mathematical library, mathlib. Nine participants explicitly mentioned that they have contributed or plan to contribute to mathlib — indeed, three participants modified their code in response to reviewer comments on a previous pull request during their observation sessions.

For Coq, the landscape of libraries is more fragmented. C10’s work builds on top of and should eventually be integrated into mathcomp, a mathematical components library, and C14 was doing the same but for Coq-HoTT, a homotopy type theory library. Several participants used SSReflect tactics, rather than built-in tactics, to varying extents; this influenced their proof style.

Proof writers have also established and followed their own conventions. Doing so might involve being consistent with collaborators on the same project or, even in the absence of collaborators, being consistent with themselves, to keep a large development organized.

Examples of Conventions. Many of the conventions involved maintaining consistency in how lemmas within a library were named, formatted, and organized. As described in §7.4, participants benefited from their knowledge of naming conventions when searching for lemmas. When writing new lemmas that they hoped to merge into libraries like mathlib, participants were also careful to follow library conventions.

Not only can conventions aid lemma search, conventions (or the lack thereof) can also affect lemma usage. C15 said they found it frustrating when libraries are inconsistent about, say, which arguments to a lemma are implicit or explicit, since this forces them to look up the statement of the lemma when using it rather than just proceeding on instinct. Lack of consistency “makes it harder to fit the library in your head.”

Conventions can also support specific technical aims. In response to comments on their mathlib pull request, L11 reversed some lemmas so that the simpler side was on the right of an equality or if-and-only-if. Since rewrites in Lean are by default from left to right, this convention means that when simplifying using such lemmas, “things actually become simpler.” C9 wrote a proof using tactics from a library called Iris, but then decided to rewrite the proof to not use these tactics. Doing so allowed them to remain consistent with their convention to not use Iris tactics in this part of their proof development (and avoid the extra import to access the tactics).

Cultures of Proof. When projects such as mathlib establish conventions about what constitutes good style, they affect not only the proofs that are written but also the proof writers themselves.

One positive impact of this phenomenon is that proof writers might be alerted through reviewer comments to new techniques and tricks. For example, one comment to L12 informed them that `have` statements could take parameters, allowing them to refactor the proof snippet on the left into the one on the right.

```
have hU : ∀ z, ...            have hU (z) : ...
  intro z                       // no intro
```

L12 said they liked the new version "much better," since it allowed them to rid the proof of the "boilerplate-y" line with the `intro` tactic.

Conventions can also suggest to proof writers the cultural values of a proof community. At multiple points during their observation, L2 experimented with refactoring certain tactic-based segments of their proof script to be term-based instead. The proof below on the left, for example, was changed to the one on the right with the help of Lean's `show_term` command:

```
intro _            exact fun _ => Or.inl (Eq.refl zero)
left
rfl
```

Based on L2's experiences with `mathlib` and reading others' code, their understanding is that proofs that are term-based or otherwise shorter (i.e., fewer lines of code) are often preferred by the community. L2 said that, while proof terms can signal "don't read this" when "nothing interesting" is happening in a section of the proof, they find that tactics are sometimes much clearer.

L2's impression is that preferences in the `mathlib` community are rooted not only in practical and aesthetic considerations but also in cultural considerations. "I think in some sense there's this idea that you're smarter if you use proof terms," they explained. They felt that there is a "bro culture" around "how unreadable can I make my code."

We want to be careful not to overfit on the precise case of tactics versus terms, since participants overall expressed a range of views on when they might prefer one over the other. But L2's experiences illustrate how a community's norms, while useful for standardization, can also have unintended negative effects on the community's culture. A more targeted examination of this topic would be an interesting avenue for future work (see §9).

8.4 Other Considerations

Easy maintenance, clear communication, and consistent conventions were the proof values participants espoused most frequently and enthusiastically, but other values were mentioned as well.

One such value was *performance* of proof-checking. L4 noted, for example, that they sometimes take into consideration the performance penalty of using high-powered search tactics like `omega` and `aesop`. Performance can be difficult to gauge accurately: for example, L8 said that they assumed term-based proofs should always be faster than their tactic-based counterparts, but were informed by Lean developers that this was not necessarily the case, though L8 was not sure exactly why.

Proof writers may also care not just about how the proof script behaves but also the characteristics of the underlying proof term. C14, for example, was working with homotopy type theory, where the *path* of equalities between terms matters, not just the fact that the terms are equal. For this reason, C14 said, they preferred to avoid tactics like `rewrite`, which sometimes produced a path of equalities that solved the goal but made the proof term difficult to work with in later proofs.

9 Discussion

We now distill our study findings into some high-level takeaways about patterns of proof assistant usage. For each observation (OBS), we provide one or more recommendations (REC) for future directions of proof assistant improvement.

OBS1: *Proof states inform more than just the next step: proof writers interpret the state within the broader context of their proof effort and leverage it to direct iteration.*

Prior to this study, we might have said that proof writers interact with proof assistants by writing a proof step, seeing the updated proof state, and using that state to determine the next step. Certainly this is part of the picture, but not the whole picture.

We saw in §5 the iterative nature of proof writing. Proof writers may look at a proof state and realize that, rather than continue to make local, linear progress, they should redirect their attention elsewhere. They may realize, for example, that the goal is unsolvable, so they need to revise their specification, or that it should be solved elsewhere, so they should extract a lemma. When they are ready to return, they return to their proof to see the (perhaps changed) proof state again.

These iterative cycles happen so often that it is easy to take them for granted, but they are worthy of a closer look! Proof writers and proof assistants work in harmony: the proof writer conducts the process, deciding what to focus on and how, while the proof assistant provides feedback on demand.

REC1: *Center iteration on proof ideas as a core strength of proof assistants.*

The proof assistant's constant feedback helps proof writers explore, clarify, and refine their reasoning as they progress in their proof. We suggest centering and building on this strength in future proof assistant development.

We should re-examine proof assistant affordances in light of the observation that proof writing occurs non-linearly. For example, modern IDEs often provide proof state diffs, which highlight what is new about each proof state since the previous step. But, due to the non-linear nature of much proof writing, the actual previous step in the proof writer's workflow is not always the literal preceding step in the proof script; the most recent edit — which the proof writer may instead want to see the effects of — could have been to an earlier part of the proof or to the specification itself.

We should also seek opportunities to give the proof assistant a more proactive role in the iteration process. We saw that proof writers regularly need to draw connections between parts of their proof state and parts of their proof development. If something looks askew, they need to locate the source of the issue. The proof assistant could assist the proof writer with drawing these connections, for example by allowing them to query why their proof state looks a particular way, à la Whyline [26]. More broadly, we believe the proof state should not be viewed as a static projection of information, but rather as something that the proof writer may want to interactively probe and query.

OBS2: *Dealing with the minutiae of mechanization is tedious, but moreover, it diverts the proof writer from the conceptually interesting facets of their proof.*

We saw in §6 that it is not enough for the proof writer to know what, conceptually, the next step of their proof should be; they must also know how to express this step in the proof assistant's language, often via tactics. And it is not enough for them to know what tactic to use, they must also know how to invoke it with precisely the right arguments. Moreover, if the assistant rejects the proof attempt, the error message it returns may be yet another cause of difficulty. These complications demand additional time and effort from the proof writer — even when they understand at a high level what they need to do next to advance their proof.

Indeed, as we witnessed participants battling with the proof assistant over the details of their proofs, we were especially struck by comments about how this "overhead" of mechanization (C3) prevents the proof writer from focusing their attention on the "mathematically meaningful questions" (L1) surrounding their proof.

REC2: *Make the level of detail proof writers have to deal with less overwhelming.*

Of course, automation that solves goals outright would be ideal — reducing the level of detail the proof writer has to deal with to zero — so we should continue to invest in improving push-button automation. But when fully automating a proof is not feasible, we should also invest in approaches that take advantage of the considerable knowledge proof writers possess about how to progress their proofs. For example, if a proof writer knows which tactic they want to use and roughly what they want to do with the tactic, but not how to instantiate it, the proof assistant could collaboratively help fill in these details.

We should also support proof writers in deferring uninteresting details while they outline the mathematical core of their proof. Currently, proof writers can use `admit` or `sorry` to skip goals. To go further, perhaps they could specify entire classes of goals they want to skip — anything about substitution, for example. Allowing users to construct a barrier between minutiae and core proof content could further alleviate the friction associated with using proof assistants.

OBS3: *Effective proof writing requires effective reuse of prior work.*

Proofs build on proofs. As we saw in §7, proof writers take advantage of prior work by themselves or others, often in the context of searching for applicable lemmas and reusing related proof snippets. While the participants we observed were generally adept at these tasks, they implicitly relied on specialized knowledge of the proof developments or libraries they were working with. With lemma search, for example, participants often relied on their knowledge about fine-grained naming conventions of the libraries they are using.

REC3: *Ensure proof writers are equipped to work within larger proof developments.*

Effective search, reuse, and other information seeking within a proof development can be a significant accelerant to proof writing, but these skills require experience and familiarity. When addressing barriers to proof assistant competency, we should consider not just "local" proof writing skills such as tactic usage but also proof engineering [39] skills that grapple with the larger-scale contexts that proof writing occurs within.

For example, the popular *Software Foundations* textbook [36] for Coq proceeds from first principles by re-implementing standard definitions and lemmas, and commands like `Search` are only briefly introduced. This pedagogical approach make sense when teaching the basics of formal reasoning, but to prepare users to enter real-world proof developments, they should additionally be taught how not to reinvent the wheel but instead find and reuse it.

OBS4: *Although their primary product is machine-checked proofs, proof writers still value how their design choices impact the humans (themselves included!) that interact with their proofs.*

Writing natural language proofs is an expressive, often creative endeavor. One might describe such proofs as clear, convincing, or elegant — or the opposite. What about mechanized proofs? Mechanization is primarily a communication process that unfolds between the proof writer and the proof assistant, not between the proof writer and potential readers. We might suppose, then, that proof writers do not care about such aesthetic or communicative concerns.

But they do care! In more subtle, technical ways, yes, but we saw in §8 that proof writers make intentional choices that reflect what they value in a proof. They may opt, for example, to write proofs so that maintainers — often their future selves — can understand and repair failures. And they may consider what their proof conveys mathematically, at various levels of granularity — from its high-level organization and lemma structure to low-level decisions about which of several interchangeable tactics to use.

Moreover, proof writers may care about whether their proofs conform to the conventions established by a larger proof project. For many Lean users, this project is the mathematical library `mathlib`. Participants viewed `mathlib` both as a source of lemmas and tactics that they could use within their developments and also as a guide on how they should style their proofs.

REC4: *When designing tools, value what proof writers value in their proofs.*

There is a recent push towards tools that make humans responsible for less of the mechanization effort, such as tools for automatic proof generation [1, 16] and repair [20, 40]. When designing and evaluating such work, we should consider not only whether the produced proofs compile, but also whether their contents match the user's preferences. It might matter, for example, what tactics the proof uses, how concisely it is written, or how well it matches conventions.

REC5: *Remember that mechanization is also an expressive endeavor.*

Throughout the mechanization process, the proof writer makes decision after decision about how best to express their proof. While many of these are for the benefit of the proof assistant, some are for their future selves or others in their community. We advocate that mechanized proofs be recognized not just as an compilable chunk of code but as what could be the carefully crafted artifact of a substantial undertaking. To this end, we are excited by projects such as the Archive of Formal Proofs [15], Alectryon [38], Lean widgets [6], coq-lsp's Markdown and LaTeX support[5], and more.

Suggestions for Further Studies. Our study required participants to be working on an open-ended project, which enforced a minimum level of experience with proof assistants. A future study could instead observe novice proof assistant users, e.g., completing homework for a class or following a tutorial. What barriers do they encounter? Are present pedagogical methods and materials adequate for addressing those barriers? (We suggested in REC3 that they may not be.) How do the challenges they face resemble and differ from those encountered by experienced users?

Our study observed participants doing everyday proof work of their choosing, leading to a wide range of observed tasks that, in turn, allowed us to take a broad view of proof assistant usage. Future studies could examine specific aspects in further depth. For example, because we observed individual proof writers within a set block of time, the interpersonal interactions were asynchronous — reading an old Zulip thread, or making a note that they should ask a colleague later, for example. What do these interactions look like live? If, say, a proof writer is asking someone to help debug an issue with their proof, how do they explain the issue, and how does the other person load the necessary context about a proof they did not write?

More broadly, one might further examine the cultures of proof assistant communities. What encourages or discourages proof writers to be active members of such communities, and how does this match up with demographics? When proof assistant communities intersect with existing proof communities — e.g., among mathematicians who already have norms in place for writing, disseminating, and evaluating proofs — what collisions and fusions occur?

10 Related Work

Our findings deepen those from prior studies of proof assistant usability.

Some prior studies have observed users on provided tasks. Aitken et al. [2] observed seven users of HOL [21] proving the same theorem about lists. They found that participants that were more frequent users of HOL tended to finish the task faster *and* interact with the proof assistant more. They noted that there was some, but not much, revision of prior proof steps, which may reflect the simplicity of the task. Aitken and Melham [3] ran trials with six users of HOL and six users of Isabelle [35], each on a single task. They cataloged user errors, such as writing invalid syntax, unintended failures of proof steps, and difficulties recalling correct function and theorem names. We observe these errors as well in our observations of real work, highlighting how developers use search functionality and convention to overcome challenges in recalling names, and additional minutiae of working with the proof assistant like understanding state and error messages.

[5]https://github.com/ejgallego/coq-lsp/blob/main/etc/doc/USER_MANUAL.md

Other studies have analyzed real-world proof work through log analyses. Ringer et al. [42] collected a month's worth of data from eight Coq users. Their findings, like ours, emphasize the iterative nature of proof development, where the logs showed that users revised specifications after a failed proof attempt, for example. Our section on Proofs in Motion (§5) describes these patterns of iteration and revision, detailing the work involved and the aspects of the proof assistant that users lean on. Staples et al. [46] analyzed project management data from developments related to the L4.verified development [5] in Isabelle. Their main observation is that proof size is highly-correlated with "effort" (as reported weekly by managers). Our study offers a deeper look at how proof writers progress, or fail to, during a particular session of proof work.

Another approach to investigating usability is focus groups. Beckert et al. [8] conducted a focus group of five Isabelle users. Their participants reported difficulties with figuring out the right tactic and tactic arguments. They also said that they often wanted to refactor proofs to improve understandability, though the situations where this occurs were not elaborated on. We offer concrete examples of issues around precision and communication.

Other prior work has shared its authors' own experiences working with proof assistants and those of their team. Andronick et al. [5] and Bourke et al. [10] reflect on the challenges posed by large-scale proof development. They also point out challenges around local trial and error, debugging broken proofs, and enforcing conventions, as do we. The projects in these papers (L4.verified and Verisoft [4]) have several hundred thousands lines of code, which is substantially larger than the developments our participants typically worked on. As a result, important considerations for them, such as proof-checking performance and domain specific automation, are not core considerations of the present paper.

QED at Large [39] surveys work related to proof engineering — the intersection of proof assistants and software engineering. Several of the processes we mention in this paper are also noted in broad terms in their survey, such as proof repair and proof reuse.

Lincroft et al. [28] mined data from the implementation repositories and community forums for Coq, Lean, and Isabelle. Their study focuses on broader contribution patterns (e.g., the lack of cross-pollination between proof assistants) rather than individual user experiences.

Zooming out from proof assistants, there has been extensive qualitative user research on the software development process generally — e.g., on refactoring [31], code search [44], and code generation [7]. There is also a blossoming literature on this kind of work in the area of formal methods — e.g., on characterizing the experience of working with property-based testing methodology [17, 18], on specification languages [22], on correctness-oriented languages like Rust [13, 50], and on the functional language paradigm upon which they are based [29]. This literature has identified both obstacles to picking up this tooling and evidence of its successful adoption. Our intent in the present paper has been to help map out the space of challenges and possibilities for proof assistants with an analogous study.

11 Conclusion

We conducted an observation study of 30 users of Coq and Lean doing their own proof work, using the methods of contextual inquiry. Through this study, we developed a nuanced understanding of what usage of these proof assistants looks like in practice, leading to recommendations that we hope will help improve the future usability of proof assistants.

Data-Availability Statement

There is no artifact associated with this paper, since the data from our study — e.g., the recordings and transcripts of study sessions — cannot be made publicly available, per our consent protocols.

References

[1] Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, Talia Ringer, and Yuriy Brun. 2023. Proofster: Automated Formal Verification. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 26–30. https://doi.org/10.1109/ICSE-Companion58688.2023.00018

[2] J. Stuart Aitken, Phil Gray, Tom Melham, and Muffy Thomas. 1998. Interactive theorem proving: An empirical study of user activity. *Journal of Symbolic Computation* 25, 2 (1998), 263–284.

[3] Stuart Aitken and T Melham. 2000. An analysis of errors in interactive proof attempts. *Interacting with computers* 12, 6 (2000), 565–586.

[4] Eyad Alkassar, Mark A Hillebrand, Dirk Leinenbach, Norbert W Schirmer, and Artem Starostin. 2008. The Verisoft approach to systems verification. In *Verified Software: Theories, Tools, Experiments: Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings 2*. Springer, 209–224.

[5] 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 *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1002–1011.

[6] Edward W. Ayers, Mateja Jamnik, and W. T. Gowers. 2021. A Graphical User Interface Framework for Formal Verification. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:16. https://doi.org/10.4230/LIPIcs.ITP.2021.4

[7] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (apr 2023), 27 pages. https://doi.org/10.1145/3586030

[8] Bernhard Beckert, Sarah Grebing, and Florian Böhl. 2015. A Usability Evaluation of Interactive Theorem Provers Using Focus Groups. In *Software Engineering and Formal Methods*, Carlos Canal and Akram Idani (Eds.). Springer International Publishing, Cham, 3–19. https://doi.org/10.1007/978-3-319-15201-1_1

[9] Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. *Analysing Data*. Springer International Publishing, Cham, 51–60. https://doi.org/10.1007/978-3-031-02217-3_5

[10] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. 2012. Challenges and experiences in managing large-scale proofs. In *Intelligent Computer Mathematics: 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings 5*. Springer, 32–48.

[11] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (Aug. 2021), 98–106. https://doi.org/10.1145/3469279

[12] Shardul Chiplunkar and Clément Pit-Claudel. 2023. Diagrammatic notations for interactive theorem proving. In *4th International Workshop on Human Aspects of Types and Reasoning Assistants*. EPFL.

[13] Will Crichton. 2020. The usability of ownership. *arXiv preprint arXiv:2011.06171* (2020).

[14] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453.

[15] Manuel Eberl, Gerwin Klein, Peter Lammich, Andreas Lochbihler, Tobias Nipkow, Larry Paulson, René Thiemann, and Dmitriy Traytel (Eds.). 2024. . https://www.isa-afp.org/

[16] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1229–1241.

[17] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581

[18] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. 2022. Some Problems with Properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*.

[19] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of Property-Based Testing Effectiveness. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. 16. https://doi.org/10.1145/3654777.3676407

[20] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI, Article 107 (June 2023), 25 pages. https://doi.org/10.1145/3591221

[21] Michael JC Gordon and Tom F Melham. 1993. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press.

[22] Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. 2022. Little tricky logic: misconceptions in the understanding of LTL. *arXiv preprint arXiv:2211.01677* (2022).

[23] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. 2013. A graphical language for proof strategies. In *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings 19*. Springer, 324–339.

[24] Karen Holtzblatt and Hugh Beyer. 1997. *Contextual design: defining customer-centered systems* (1 ed.). Morgan Kaufmann.

[25] Talia Ringer on Jan 29 and 2020. 2020. Mechanized Proofs for PL: Past, Present, and Future. https://blog.sigplan.org/2020/01/29/mechanized-proofs-for-pl-past-present-and-future/

[26] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/985692.985712

[27] Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-box best-first proof search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 253–266.

[28] Gwenyth Lincroft, Minsung Cho, Katherine Hough, Mahsa Bazzaz, and Jonathan Bell. 2024. Thirty-Three Years of Mathematicians and Software Engineers: A Case Study of Domain Expertise and Participation in Proof Assistant Ecosystems. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 1–13. https://ieeexplore.ieee.org/document/10555745/?arnumber=10555745 ISSN: 2574-3864.

[29] Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 155 (oct 2021), 30 pages. https://doi.org/10.1145/3485532

[30] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[31] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How we refactor, and how we know it. In *2009 IEEE 31st International Conference on Software Engineering*. 287–297. https://doi.org/10.1109/ICSE.2009.5070529

[32] Wojciech Nawrocki, Edward W Ayers, and Gabriel Ebner. 2023. An extensible user interface for Lean 4. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[33] Ernest Ng, Harrison Goldstein, and Benjamin C. Pierce. 2024. Mica: Automated Differential Testing for OCaml Modules. https://doi.org/10.48550/arXiv.2408.14561 arXiv:2408.14561 [cs].

[34] Lawrence Paulson. 2010. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In *PAAR-2010: Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning (EPiC Series in Computing, Vol. 9)*, Renate A. Schmidt, Stephan Schulz, and Boris Konev (Eds.). EasyChair, Edinburgh, Scotland, 1–10. https://doi.org/10.29007/tnfd

[35] Lawrence C Paulson. 1994. *Isabelle: A generic theorem prover.* Springer.

[36] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2024. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook. Version 6.7, http://softwarefoundations.cis.upenn.edu.

[37] Bartosz Piotrowski, Ramon Fernández Mir, and Edward Ayers. 2023. Machine-learned premise selection for Lean. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 175–186.

[38] Clément Pit-Claudel. 2020. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 155–174.

[39] 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 (Sept. 2019), 102–281. https://doi.org/10.1561/2500000045 Publisher: Now Publishers, Inc..

[40] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 112–127. https://doi.org/10.1145/3453483.3454033

[41] 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 2020)*. Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3372885.3373823

[42] 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*. 115–129.

[43] 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)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[44] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*.

Association for Computing Machinery, New York, NY, USA, 191–201. https://doi.org/10.1145/2786805.2786855
[45] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2024. Towards large language models as copilots for theorem proving in lean. *arXiv preprint arXiv:2404.12534* (2024).
[46] 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*. 1–4.
[47] Coq Development Team. 1989-2024. The Coq Proof Assistant. http://coq.inria.fr
[48] Matej Urbas and Mateja Jamnik. 2014. A framework for heterogeneous reasoning in formal and informal domains. In *Diagrammatic Representation and Inference: 8th International Conference, Diagrams 2014, Melbourne, VIC, Australia, July 28–August 1, 2014. Proceedings 8*. Springer, 277–292.
[49] Sean Welleck and Rahul Saha. 2023. LLMSTEP: LLM proofstep suggestions in Lean. *arXiv preprint arXiv:2310.18457* (2023).
[50] Anna Zeng and Will Crichton. 2019. Identifying barriers to adoption for Rust through online discourse. *arXiv preprint arXiv:1901.01001* (2019).

A Backgrounds of Participants

ID	Exp. *(years)*	Occupation
C1	5	PhD student
C2	2	PhD student
C3	5	PhD student
C4	2	postdoc
C5	8	postdoc
C6	4	postdoc
C7	1	PhD student
C8	3	PhD student
C9	5	PhD student
C10	4	PhD student
C11	5	PhD student
C12	4	PhD student
C13	3	undergraduate student
C14	2	*(no response)*
C15	10+	professor

(a) Coq Participant Backgrounds

ID	Exp. *(years)*	Occupation
L1	1	PhD student
L2	2	PhD student
L3	3	PhD student
L4	4	software engineer
L5	2	professor
L6	3	PhD student
L7	7	PhD student
L8	1	software engineer
L9	2	PhD student
L10	3	PhD student
L11	<1	PhD student
L12	<1	professor
L13	<1	PhD student
L14	4	research software engineer
L15	<1	graduate student

(b) Lean Participant Backgrounds