QED in Context: Observations of Proof Assistant Users

ANONYMOUS AUTHOR(S)

Additional Key Words and Phrases: Keywords

1 Introduction

Contributions roughly as follows:

- We present the first large-scale, in-context observational study of proof assistant users. [Question: Is 30 participants considered large-scale?]
- We describe, in more concrete and colorful detail than "folklore" wisdom, what it looks like to use a proof assistant.
- We make recommenations on future research to improve proof assistant usability.

2 Study Methodology

- Thirty participants: 15 using Coq, and 15 using Lean.
- Some demographics in the table on the next page.
- Mostly 90 minute sessions with around 5 minutes of intake, 60 minutes of observation, and around 25 minutes of post-observation interview.
- For observation, participants asked to do their "normal, everyday work" on their proof development — i.e. work they would complete regardless of our presence.
- Also asked participants to narrate out loud what they were doing and thinking, and interjected periodically with clarifying questions.
- Interview questions followed up in more detail about things we saw / asked about their broader experiences with related things. For example, if they did a lot of lemma search, we asked them about their experiences with lemma search.
- Coded the transcripts in Delve / consulted other authors as I iterated the codebook.

3 Proofs in Motion

When proof writers are engaging in their real, everyday proof work, we can observe, in vivid detail, what it looks like to navigate the complexity and uncertainty that permeates the proving process. Throughout this process, proof writers carefully interact with the proof assistant, seeking and utilizing its feedback in complex and even creative ways.

3.1 Iteration

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

3.1.1 Realization. We start by examining moments of realization — when proof writers realize, in the middle of a proof attempt, that they should not continue the proof as-is but instead iterate, such as by revising the current specification, extracting a new lemma, or backtracking to an earlier proof step. We observe in particular how proof writers use the proof state to arrive at these realizations.

2024. ACM XXXX-XXXX/2024/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ID	Exp. *(years)*	Occupation
C1	5	PhD student
C2	2	PhD student
C3	5	PhD student
C4	2	postdoc
C5	4	postdoc
C6	8	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 Demographics

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 Demographics

Specification Revision. One situation, encountered by *n* participants, is realizing while writing a proof that the specification being proven should be revised. This often occurs when proof writers detect that a proof has entered into a bad state. Sometimes, the badness is 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. Other times, proof writers implicitly rely on their expertise about how the proof state should look to notice that it has deviated from that expectation.

C3, for example, narrated that they had a `RR` in an assumption and `RR'` in the conclusion, but the `RR` was a relation on `I` while `RR'` was a relation on `I + R`, so the goal was "definitely not true." The proof state at this point is shown below:

```
RR : rel I (World E)
H : RR x w
RR' : rel (I + R) (World E)
H0 : forall (x : I) (w : World E), RR x w ->
    <( {k, x}, {w} |= vis {φ} AU (AX done {RR}) )>
______________________________________________________
<( {k, x}, {w} |= vis {φ} AU (AX done {RR'}) )>
```

We simplified the proof state by coloring the mentioned subterms and abridging it to exclude eight lines preceding and a second subgoal following — simplifications not afforded to C3, though they were able to realize the issue after a very brief examination. 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 *n* participants, is realizing mid-proof that a new lemma would be useful. C11 performed a quick search that showed they were missing the lemma they wanted. "I will write this lemma now," C11 reasoned. "It is simple enough, and I feel like I should have it, and I'm going to need it later on." Lemmas can also improve modularity; L13 said one reason they might separate out a lemma is if they do not expect the details of the lemma's proof to be relevant to the current proof.

When the need for the lemma arises organically during a proof, proof writers can *extract* snippets of the proof state and use them in the lemma statement. L10 copied this subexpression of the goal from the proof state and pasted it as a starting point for a new lemma:

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

Then, they modified the expression, such as by changing (head :: tail) to just a variable p, and placed it in an equality, where the right-hand side was what they wanted the expression to simplify into. The eventual lemma statement, with the modified original expression highlighted, was:

```
Lemma Polynomial'.coeff_toPoly
  {R : Type} [Semiring R] (p : Polynomial' R) (i : ℕ) :
  Polynomial.coeff (Polynomial'.toPoly p) i = p.getD i 0.
```

That is, L10 used the proof state as an input for adding new structural elements to their proof.

In fact, some proof writers deliberately advance their proofs to seek proof states that aid extraction. C1 realized they needed a lemma near the beginning of a proof, but after struggling to write its statement directly, they returned to progress the proof until they reached the "main interesting thing." They then temporarily copy-pasted a few lines of the proof state at this point into the editor so that they could reference it while writing the lemma statement. C12 also did a 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 their 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 the reasoning rule lemmas should look like. "I'm going to see if I can sort of reverse 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 actual location. Similarly, C8 also intentionally began proving a statement that lacked the necessary assumptions, so that they could wait for the the assumptions to "jump out" as they progressed the proof and add them then. We share further examples of proof writers engineering their environments to receive desirable feedback from the proof assistant in §3.4.

Participants almost always performed this 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 automatically generate a lemma from the proof state. Later, they spent some time cleaning up the lemma statement, such as by renaming an argument name i to id and i0 to instr. When proving the lemma, they realized they erroneously cleared a necessary assumption, which they quickly added. That is, the lemma-from-goal automation replaced some of the copy-pasting aspects of lemma extraction, though naturally it cannot eliminate the reasoning required to determine when a lemma is needed and what shape it should take.

Going Backwards. Proof writers may also realize they need to revisit and revise earlier points in their proof script. This could be because they made a mistake previously, such as when L4 realized after encountering an impossible proof state they had used the wrong variable in an intermediate let-statement four lines earlier. C10 realized that they had some "stray" goals that should have been solved earlier in the proof. To locate the source, they rapidly evaluated the proof to a series of intermediate locations, marked in order below:

```
② rewrite lemA lemB lemC ④ lemD ⑤ lemE lemF. ③
four lines...
...of rewrites. ①
```

Note that since they were using `ssreflect` to chain a series of rewrites together, they needed to temporarily add periods, Coq's intermediate evaluation markers, at ④ and then ⑤. After determining that `lemE` was the issue, they modified the step to avoid the stray goal.

3.1.2 Propagation. We next examine propagation — when proof writers complete the iteration cycle by recompiling and repairing proofs after a change. We again focus on how the proof writer interacts with and benefits from the proof assistant during this process.

Consider the experience of C5, who spent their observation session experimenting with different ways to resolve a problem with their specification. Each time they made a change, they could recompile their file, and the proof assistant would inform them where the proof succeeds or fails. In fact, they described the ability to receive such feedback as "the magic" of being in a proof assistant.

Proof writers in general experience this magic whenever they propagate a change to the proofs it affects. After fixing an off-by-one error in their lemma statement, for example, L4's previously failing proof immediately refreshed to now succeed, providing immediate feedback that the fix was correct. Of course, changes often instead cause failures. Some of these failures indicate that the change needs further iteration. When C1 added a new instance of a type class and recompiled the files in their development, they realized previously working proofs now broke, so they needed to limit the scope of the instance so that it would not be erroneously used in those proofs.

Failures can also identify locations requiring *proof repair*. The proof writer might directly edit the broken proof. After switching the sides of an if-and-only-if in a lemma, L11 fixed proofs that relied on what was now the wrong direction of the lemma (using Lean's `.mpr` projection) to use the other direction instead (using `.mp`). Or, the proof writer may comment out the broken proof, starting the proof anew while referencing the old proof as needed. Indeed, L9 interleaved writing new proof snippets, copy-pasting old proof snippets, and repairing those old snippets to reflect the changes they had made to the specification. This process has many similarities to proof reuse (§4.2).

3.2 Context Switching

We have seen how proof writers use proof assistant feedback to support iteration, but it is helpful in other situations too — in particular, when *context switching*. Proof writers frequently switch away from a goal and later switch back again. In doing so, they rely on the proof assistant to maintain the proof state, so they can resume right where they last paused.

Proof writers, including n participants during their observation sessions, use Coq's `admit` or Lean's `sorry` to temporarily skip a goal and continue working on a different one, allowing them to prioritize the goals they want to work on.

Some prioritization occurs on the fly, such as when proof writers decide 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 can also reflect higher-level strategization depending on the status of the proof effort. 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 all of these cases and more 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 re-supplied.

Indeed, when proof writers return to a skipped goal, to refresh their memory on what the goal was, or even to choose which skipped goal to work on, they can examine the proof states at those points in the proof. After a brief inspection, C11 opted to work on the second `admit` in their proof before the first, since it seemed more straightforward. They explained that in general, they like to prioritize filling in `admit`s that appear to require a "smaller lemma" to fill, in the hopes that this lemma will be useful when filling the complex `admit`s.

The ease of revisiting `admit`s is a further accelerant for the iterative processes described in the previous section. Returning to C5's experience, each time they made a experimental change to their specification, in addition to fixing broken proofs, they also reassessed whether previously `admit`-ed goals could now be solved. Critically, not only does the proof assistant automatically re-supply the context at these `admit`s, the context is automatically synced with the new specification.

3.3 Small-Stakes Trials

[This would be a section about the small-stakes trial-and-error that participants often engage in when they're trying to figure out a detail of their proof. By small-stakes, we mean that they're not randomly trying things throughout the proof, but rather, e.g. cycling through a few tactics that each solve certain kinds of trivial goals to figure out the one that works in a particular situation.]

3.4 Environment Engineering

[This would be a section about some of the zany things participants did to elicit better feedback from the proof assistant, such as by creating a fake goal to step through a computation.]

4 Proof Sources and Resources

Mechanized proof writing rarely proceeds from scratch. Instead, the proof writer might adapt the proof from an existing source, such as by translating a paper proof or by reusing previously mechanized proofs, and they might rely on existing resources, such as by searching for relevant lemmas or by seeking information about proof techniques. Effectively carrying out these tasks requires careful attention to detail.

4.1 Using Paper

Some mechanized proofs originate in whole or in part on paper, which is a more malleable, less rigorous medium for proof writing than a proof assistant.

4.1.1 Translations. Participants [TODO: list which ones] explicitly referenced a paper proof during the observation session. [TODO: share a few details about the sources of the proofs, e.g. textbook, their own paper, math olympiad solutions.]

Mechanization requires making implicit details in a paper proof explicit. L4 said that paper proofs are often ambiguous about whether a variable should be universally quantified or specifically refers to something in the current context. For example, in the solution 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 the statements, so that they could more flexibly apply the statements later, but said in general they also need to be careful not to overly strengthen a statement such that it is no longer true.

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 proceeds by "induction on the absolute difference of these two numbers, knowing that this one is bounded by that and the other one is bounded by this." L2 noted that though complex, this induction strategy is "understandable pretty quickly for a human." In Lean, however, they had difficulty expressing this induction, needing to rely on (and justify) their own custom induction principle.

While challenging, the particularities of the mechanization process can be precisely why mechanization is valuable. L1 observed that in a mathematical paper, the author might claim the existence of an algorithm that "flows around" the proof, but this 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." For example, they explained, they had something "essentially coinductive" but wanted to work around the need for coinduction because Lean did not support it well, "*and relatedly*" — since Lean tends to focus on supporting techniques that are in common usage — it is uncommon among mathematicians. 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."

4.1.2 Scratch Work. Proof writers sometimes do scratch work on paper alongside mechanization. Note that in this section, we use paper loosely here to include both tree-origin paper and temporary comments within the proof assistant.

C1 (on paper) and L4 (in a comment) worked out examples of how a definition should work on small inputs. C1 used the examples prior to writing the Coq definition, to determine how it should be phrased and avoid off-by-one errors, 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, so I'm swapping between thinking about the proof in Lean and then going back to think about how I write this as an informal math proof," they later explained as their process. L13 said that on paper, it is easier to do simplifications and to read notation such as fractions.

We saw in §3.2 examples of proof writers using comments to remember their thought process. C3, as a further example, left notes such as this one about what their lemma statement meant:

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

C3 explained that "this big expression here" (m lines of code) just says that k will terminate. Writing it down as a comment "helps me retrieve this fact so I don't have to reparse" the entire lemma.

4.2 Reusing Proofs

[This would be a section about how participants reused proofs written by themselves by others, often by copy-pasting snippets from the proof and adapting it to the local context.]

4.3 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 leverage existing tools for lemma search in combination with their — often quite specific — assumptions about the target lemma.

4.3.1 Engines. We start with an overview of the kinds of lemma search engines participants used, which differed substantially between the two proof assistants.

n 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, definitions appearing in the lemma statement, patterns that the lemma statement must obey, or a combination.

As an example of the first two, `C1` ran the command «`Search binddt "rw" letin`», which returns lemmas whose name contains the substring "rw" and whose statement contains `binddt` and `letin`. And as an example of searching by pattern, `C10` ran the command «`Search ?a + _ == ?a + _`», which returns lemmas whose statement contains as a subexpression the `==` equality of two sums, the first arguments to which must be the same.

On the Lean side, several search engines were used:

- k 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 specially marked simplification lemmas.
- k participants found 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.
- k participants used `Moogle.ai` to do lemma search, which describes itself as a "semantic search engine" for `mathlib` that accepts natural language queries.
- k participants used `Loogle`, which has similar functionality as Coq's `Search` command.

4.3.2 *Specificity vs. Fuzziness.*

When choosing a search engine and formulating a search query, the proof writer must 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 accelerate the process, where the target lemma is one of the only results returned, but the risk is that even subtle inaccuracies can cause the lemma to not be included at all. On the flip side, search strategies that permit *fuzziness* are more forgiving of inaccuracies, but may be less effective at narrowing results.

Search by Name. An especially specific approach to lemma search is predicting the lemma name. A common naming 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		*searching for*	
`multiplicity.finite_prime_left`		`Nat.lt_of_succ_le`	
queried		*queried*	
`finite_of_prime`	✗	`Nat.lt_off`	✗
`multiplicity.prime`	✓	`Nat.lt_iff_succ`	✗
		`Nat.lt_of_succ`	✓

The search bar allows queries to be a non-contiguous subsequence of the lemma name (e.g., `multiplicity.prime`) but is unforgiving of other discrepancies. This 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 impliciations).

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 they were familiar with `mathlib`'s naming conventions. These conventions can require quite fine-grained knowledge: L13, for example, described difficulties due to 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 patterns. (Note that although Loogle does support patterns, we did not observe any Lean participants search by pattern.)

Patterns can be used to supply a notation directly, instead of the underlying definitions — for example, C6 alternated between searching for «`(_ && _)`» and the corresponding function name «`andb`». They can also be used to supply constants that should appear at some position in the lemma — for example, C7 ran a search for «`(_ * _) = 0`». And they can be used to enforce that two or more values need to be the same — for example, C10 ran a search for «`?a + _ == ?a + _`».

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. This requires the user to specify the subject of the lemma but not what the lemma should say.

As we saw in the previous section, Coq's `Search` command supports this kind of search. When C8 had one assumption in their proof state about `PureExec` and another about `LanguageCtx`, they ran the command «`Search PureExec LanguageCtx`», which returns lemmas that mention both terms. This led to precisely the lemma they needed. 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 this effect.

Combining this approach with the previous can greatly narrow a search. C15 searched for just «`Permutation`» before eventually 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 result.

Search by Natural Language. Perhaps the fuzziest approach of all is natural language queries.

One example of a situation where support for natural language search could 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 how a query is phrased. In the case of L4, who said they usually reach for Moogle first, they searched for "power of sum equals product of powers." Unfortunately, this query returned results mostly about power sets, whereas they were looking for lemmas about powers in the sense of exponents. They then changed the first word of their query to "pow" and the first result was immediately the lemma they were seeking.

Search in Context. Search strategies not only vary in query specificity but also 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, observe that `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 the lemma 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 that composed that definition with another. C4 changed their goal to match this form, and then was able to use a lemma from the query.

4.4 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.

4.4.1 (Not) Using the Documentation. Of course, both Coq and Lean have documentation about their languages. C8 said that in the past few months they have used the reference manual "a dozen times a week at least." C2, on other hand, said, "I generally don't tend to 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 SSReflect syntax during the observation, but then decided, "I don't think I need to understand it," and continued onwards.

Unlike in the context of lemma search, it can be less clear 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."

4.4.2 Using Examples. Proof writers make use of code examples, both by reading textbook examples and by adapting snippets from library source code. This process has some similarities to reusing proofs, though the source proof may be used more indirectly.

L10 and L13 used examples of the cases and induction' tactics, respectively, to help them figure out the correct syntax. In L10's case, they found the example directly from the documentation, by hovering over the cases tactic in the VS Code editor, and in L13's case, they found the example through 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 incredibly complex, 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 the 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 ended while they were in the midst of the second step.)

Perhaps part of the complexity of this situation can be attributed to the fact that the example L8 was using was not an intentional, pedagogical example of the technique they were trying to learn, but was instead a piece of source code that they happened to know existed. L8 said they wished there were more examples, especially examples that are not too basic, as they would be difficult to generalize to more complex situations, and 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`[1] as the kind of resource they wish there was more of.

4.4.3 *Asking Others.*

n participants said they seek help from other proof assistant users, whether in their local community or by asking (or searching for) questions in 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 frequently post “silly questions” on the new members stream in the Lean Zulip, acknowledging, “I feel comfortable doing that. I know not everyone can do that.”

5 Conversing with the Prover

Proof writers are in constant conversation with their proof assistants. In one direction, proof writers need to speak to the proof assistant in a way it can understand, at sometimes gruesome levels of precision. In the other direction, proof writers need to attend to the proof assistant’s feedback, which may include proof states and errors that are difficult to decipher.

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

5.1 Speaking to the Prover

5.1.1 *Precision.*

Proof assistants demand precision from proof writers, which often leads to fussing with details that are orthogonal to the main ideas of the proof.

Proof writers may face precision problems when they understand conceptually what tactic they want to use to advance their proof, but then have trouble invoking the tactic in exactly the right way. For example, L10 wanted to do a case analysis on the expression `p.length`, and so they wrote

```
cases p.length
```

But this did not work as intended, since it led to an impossible goal where the conclusion was only true if `p` was empty, but no information was retained in the context that `p.length` was zero. During the observation, L10 had to instead do 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
```

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

which forces the necessary information to be retained in an assumption h. That is, L10 knew what they wanted to do (do a case analysis on the length), they knew the tactic to do it (cases), and they correctly identified the issue that arose (the length was not retained as an assumption), but they had to pivot to a different approach due to the absence of two characters.

As another example, C4 briefly encountered an issue where they tried to unfold the definition of len in an assumption of the form c : t. Nothing happened, so they printed the definition of t and saw that it indeed contained len. They realized they needed to first unfold the outer t, and only then could they unfold the inner len. C4 remarked, “Coq really needs little steps — little steps always.”

Another source of precision problems is when proof writers decide to use a lemma, but then need to figure out how exactly to do so. Participants used a variety of techniques to pinpoint the rewrite they wanted; for example, L15 commuted specific summands within a larger goal by writing

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

where the two lhs commands tell Lean to do the rewrite within the left-hand side of the left-hand side of the goal. With lemma applications, participants often needed to manually provide arguments; for example, C8 had to instantiate the seventh argument to a lemma:

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

C8 described this as a “weird hack that we need to restort to,” since otherwise unification would fail later in the proof.

5.1.2 *Naturality.*

Styles of reasoning that are more natural to how a proof writer wants to think about their proof — or write it on paper — may instead feel unnatural in a proof assistant.

Mathematical intent that is easy to express in paper proofs can require careful effort to mechanize. Lean provides a proof mode, calc blocks, which facilitates 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 has found that they have to “jump through a few hoops to make that style of proof fit in a calc block well.” And for C14, one challenge that seems “fundamental” to the formal proof setting is that it can be “quite non-trivial” to convert between equivalent representations of a definition. By contrast, C14 said, “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.”

L8 shared that when they started using Lean, they mostly wrote term-based rather than tactic-based proofs, since they felt that term-based proofs more closely resembled paper proofs. “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, and so on and so forth,” L8 said. “And that works out naturally doing things in a term-based way.”

Two other participants similarly commented on *forwards reasoning*, where the proof proceeds from the assumptions towards the conclusion, versus *backwards reasoning*, from the conclusion towards the assumptions. 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 the proof assistant, it is “usually more tempting” to do backwards reasoning. However, 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. They cited their preference for forward reasoning as a reason they like to use SSReflect tactics instead of built-in tactics, since they find forward reasoning is more convenient in SSReflect.

5.2 Listening to the Prover

5.2.1 Unwieldy States. Proof states, critical for tracking proof progress, can be long and complex. Proof writers thus need to leverage considerable expertise to understand and manage proof states.

[TODO: briefly include an example of a proof state that the participant called "ugly."]

One challenge with handling proof states is determining which definitions to unfold and which simplifications to perform so that the proof is operating at the desired level of abstraction. Too much unfolding can lead to proof states that are much 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 where goals became over 30 lines long, in their recollection. C4 shared that they found it difficult to find the "sweet spot" when unfolding, where the right details are exposed while stopping short of the proof state getting "way too big for you to even understand what it says."

Oversimplification can also cause problems later in the proof. During their observation session, L9 encountered a proof state with this conclusion:

```
V.p.IsHomLift (1 (X.p.obj a))
  ({app := fun x |-> 1 (((F.comp G).comp H).obj x),
   naturality := ...}.app a)
```

Using the `simp` tactic, the proof state was transformed into the shorter, simpler state below:

```
V.p.IsHomLift (1 (X.p.obj a))
 (1 (H.obj (G.obj (F.obj a))))
```

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 led the compositions of F, G, and H to be rearranged so that type class inference failed.

Proof writers sometimes adopt specific strategies to facilitate unfolding and simplification of proof states. C8, for instance, sought greater control by making all of their definitions `Opaque` by default, so that they would not be unfolded during simplification. C4 used a custom tactic `unfold_cap`, that was capable of unfolding over 50 specific definitions of interest, though this tactic was not yet comprehensive, so they still needed to intersperse it with manual unfoldings.

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

Proof writers can be misled by error messages. L10 remarked when reflecting on error messages they found challenging as a beginner Lean user, "It's something that you learn with experience, that it's not always the case that what the error message is leading you to believe is exactly true." C2 wrote a lemma statement of the form

```
exists A B i, ... exists j, ... tUniv i
```

and received the error message that Coq "cannot infer the implicit parameter `A` of `ex`." Understandably, they tried adding a type annotation to `A` to assist with type inference, but this was not the issue, so the error persisted. After about two minutes of debugging, they finally realized that they had made a typo: they should have written `tUniv j`. That is, the text of the error message can cause proof writers to develop incorrect assumptions about the root cause of the error.

The distance within a proof script between the error message and the root cause can be confusing as well. C3 modified their `intros` tactic to manually provide assumption names, instead of using the auto-generated names. But then a previously working `apply` tactic later in the proof, which did not refer to any of the names involved, now gave a "failed to unify" error. After a brief moment of bewilderment, C3 figured out that they had accidentally provided only two of the three assumption names, causing the proof state to have the wrong shape.

Proof writers can also struggle with error messages that rely on low-level details that the proof writer might otherwise not want to think about. L9 had a rewrite that failed with "motive is not type correct," which they are "never really sure how to deal with." They fixed the issue through 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, it's not really close to what you actually need, and that gets really frustrating."

Sometimes, the low-level details may reveal that portions of the proof state that appear the same are in fact not. C6 found themselves in the observation session with a goal of `t1 && t2` and a lemma of the form `t1 && t2`. They tried to solve the goal by applying the seemingly identical 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 do the `if then else` and `= true` come from? As C6 found, the issue was that the goal was implicitly using the coercion `is_true`, while the lemma was implicitly using the coercion `Is_true` both of which converted booleans into propositions, but in a way that was ultimately incompatible.

6 Beyond QED

While the headline benefit of a mechanized proof is ensuring that a theorem is true, proof writers also care deeply about the content of the proof itself. Throughout the sessions, participants described the numerous, nuanced, and sometimes conflicting qualities they value in a proof, beyond simply "it compiles." In this section, we focus particularly on maintainability, communication of mathematical ideas, and compliance with conventions.

6.1 Maintainability

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

A core aspect of maintainability is *robustness*, since a proof that does not break is especially easy to maintain. L2 made a proof robust to changes in underlying definitions by enforcing strict abstraction boundaries. They had originally written a proof using a tactic called `rcases`, whose behavior relies on the implementation details of the definitions involved, and they refactored the proof to use a lemma that preserved the abstraction barrier between the proof and these details.

Participants wanted more than just robustness to failure — they also wanted to ensure that if the proof does fail, it fails in a way that is conducive to understanding and fixing the failure. C5 said they prioritze structuring their proofs so that it "breaks exactly at the point where things actually break." C5 showed an example of a proof in their development structured like the one on the right, and explained why they prefer 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 especially non-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` preferrable for maintainability? 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. That is, `simp` is sometimes preferrable because it is more robust to certain kinds of changes.

However, `simp only` may be preferrable 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 the Lean's official recommendation[2] to avoid "non-terminal" `simps`.) That is, the more restrictive, less automated `simp only` may assist in ensuring a proof fails fast.

6.2 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 for organizing these units might play out at the granularity of tactics, proof steps, and lemmas.

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

Moving up to proof step organization, L7 showed us the proof outlined below, which was the result of a refactoring to better communicate the proof structure:

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

[2]todo

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 perform rewrites on one side of an equality and then rewrites on the other side. The order of the rewrites is not fixed, but instead of "freewheeling," C1 said they had learned to structure their proofs in this organized way to improve their 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," C1 said.

For C5, the focus is on communicating the content of their proof not through their tactic scripts but rather through their lemmas, where they aim 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, participants also try to make stylistic decisions that signal their mathematical intent. In L4's case, they said they often had goals that could be solved by the `omega`, `linarith`, or `positivity` decision procedures. Of these, "`positivity` is less powerful, but it expresses more intent," L4 said, 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 the `apply` tactic, 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 prefer to use not the maximally powerful tactic, but rather the one that conveys their intention, is reminiscent of what we saw in the previous section with maintainability: proof writers sometimes eschew maximally powerful automation in favor of techniques that streamline understanding and fixing failures.

Concision. [This is a subsection about how several participants also had a strong desire to make their proofs as short as possible, sometimes in ways that supported intent (e.g. mathematically easier proofs should take less space).]

6.3 Conventions

Formal proofs do not exist in isolation; neither do users' values about what makes a good proof. Users may be seeking to integrate their work into a larger project, which may in turn come with conventions that contributors are encouraged or required to follow. Even within more 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`, and as a result, a dominant presence in our discussion of conventions will be the stylistic guidelines that `mathlib` enforces. *n* participants explicitly mentioned during the session that they intend to submit their work as a pull request to `mathlib`, and in fact, three particpants modified their code in response to reviewer comments during the observation session.

For Coq, the landscape of libraries is more fragmented. C10's work built 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 also used ssreflect tactics, rather than built-in tactics, to varying extents, which also influenced their proof style.

Proof writers can also establish and follow their own conventions. Doing so could involve being consistent with collaborators on the same project, or even in the absence of collaborators, being consistent with onself, in order to keep a large development organized.

Examples of Conventions. [This subsection describes some of the conventions participants followed. This included lemma naming / file organization conventions, which makes it easier to locate lemmas. Lemma setup conventions, such as the order of arguments. And so on.]

Cultures of Proof. When projects such as mathlib establish conventions about what constitutes good style, these 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 for L12 informed them that have statements could take parameters, allowing them to refactor the proof snippet on the left into the snippet on the right.

```
have hU : forall 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 on the left, for example, was changed to become the proof 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 working with mathlib and reading others' code, their understanding is that proofs that are term-based or otherwise shorter are often preferred by the mathlib community. L2 said that while proof terms can help 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. "It's kind of a bro culture of, how unreadable can I make my code so the unworthy can't understand it?" They added later that they felt people wanted to demonstrate, "Look at all the cool computer tricks I can do to like get it into this completely unreadable form. There's definitely an idea of whoa, dude, that's so awesome."

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 provide a perspective on how a community's norms, while useful for standardization, can also shape the community's culture. A more targeted examination of this topic would be an interesting avenue for future work (§).

6.4 Other Considerations

[This is a short section about other considerations that a small number of participants articulated, such as performance.]

7 Discussion

7.1 Observations

OB1: *Proof states inform more than just the next step: proof writers interpret the state within the broader context of their proof effort and use it to direct iteration.* Colloquially, we may say that proof writers interact with the proof assistant by writing a proof step, evaluating it to see the updated proof state, and then using that state to determine the next step. Certainly, this is part of the picture, but it is not the whole picture.

We saw in §3 the iterative nature of proof writing. That is, proof writers may see a proof state and realize that rather than continue to make local, linear progress, they should instead interrupt the flow of their proof and redirect their attention to elsewhere in their development. They may have realized, for example, that the goal is unsolvable, so they revise their specification, or that it should be solved elsewhere, so they extract a lemma. Then, when they are ready to return, they simply re-evaluate their proof to see the new proof state.

These iterative cycles happen so frequently that it is easy to take them for granted, but they are no small feat! We can think of the proof writer and proof assistant as working in harmony with each other: the proof writer conducts the process, deciding what to focus on and how, while the proof assistant provides up-to-date feedback on demand.

OB2: *Effective proof writing requires effective usage of prior work.* Proofs build on proofs: as we saw in §4, proof writers take advantage of prior work by themselves or others, most often in the context of finding 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, even the smallest of discrepancies between the search query and the target lemma can cause the search to fail. And with proof reuse, the reuser needs to have an accurate mental model of how proof techniques used in disparate proofs align.

OB3: *The minutiae of mechanization distract from the mathematically interesting aspects of a proof.* When the proof writer converses with the proof assistant, we saw in §5 that frictions abound.

The proof writer not only needs to know what, mathematically, the next step of their proof is, but also how to express it in the precise language the proof assistant will accept. If the proof assistant rejects the attempt, the error message it returns may be a further cause of difficulty, such as when it requires the proof writer to grapple with inner workings of the proof assistant. We were particularly struck by comments saying that this friction causes frustration because it prevents the proof writer from spending their time on the aspects of the proof they care most about.

OB4: *Proof writers, though producing machine-checked proofs, still value how their design choices impact the humans (themselves included!) that interact with their proof.* We saw in §6 that proof writers make intentional choices that reflect what they value in a proof — beyond mere compilation.

Proof writers may care about the proof as it evolves over time, and whether maintainers including their future selves can easily understand and fix failures. Proof writers may also care about what their proof communicates, sometimes in quite subtle ways. Analogously to fine-tuning prose in a paper proof, a mechanized proof writer might, for example, select amongst otherwise interchangeable tactics and choose the one that best conveys their intention. Moreover, proof writers may care about whether their proofs conform to the norms established by a larger proof project, so that those working on the project benefit from the consistency.

7.2 Research Opportunities

RO1: *Reduce the friction of proof exploration.* Proof writers may want to engage in open-ended exploration, such as by experimenting with different specification variants or different proving techniques. On the one hand, the proof assistant is great for exploration: it tracks the proof state, and prevents the proof writer from writing invalid proofs. On the other hand, the proof assistant is terrible for exploration: it also prevents the proof writer from writing all sorts of proofs that are phrased imperfectly. Can we reconcile this contradiction, in favor of exploration?

When proving on paper, the proof writer can opt to "handwave" details that they find unimportant. We think that proof assistants should also better support (temporary) handwaving. Limited support is currently available, such as using `admit` to skip a goal. But what about, say, allowing the proof writer to indicate they want to skip an entire class of details? Or that they want the proof assistant to accept their proof if it works up to some margin of error? Of course, these are not easy things to accomplish, but in general, we see an opportunity to enable proof writers to, during exploration, take advantage of a proof assistant without becoming encumbered by it.

RO2: *Allow users to control the fuzziness of lemma search.* We realized through our observations of lemma search that there is not, in our opinion, a one-size-fits-all approach. Users sometimes benefit from levers for specificity, which allows them to pinpoint precisely the lemma they want, but they sometimes wish for fuzziness, which allows them to still find their lemma despite inaccuracies in their assumptions about the lemma name or contents.

By mixing different search techniques, users can, with enough effort, achieve their desired balance, but we see an opportunity to make this balance more readily available. For example, the user should be able to search for substrings that appear in the lemma name, and return precisely the lemmas that match; if their target lemma doesn't, they should then be able to increase the fuzziness and return, say, lemmas within a certain edit distance. Or, the user should be able to search for a specific pattern that the lemma statement should obey; if their target lemma isn't returned, they should be able to ask to see lemmas that match the pattern fuzzily, up to some transformations.

RO3: *Ensure proof generation techniques respect proof values.* Given what is likely to be an increasing proportion of mechanized proofs produced by generative artificial intelligence or other automatic techniques, what say should humans have in what these proofs look like?

As long as human proof writers remain part of the proof writing process, we predict they will want to exercise control over the contents over the final proof script, so that they can better maintain, understand, and share their proofs. This suggests that generation techniques should take into account proof values, which could range from inferring naming conventions from other proof scripts nearby to allowing the user to specify properties they want their proofs to have.

RO4: *Examine implications for proof assistant pedagogy.* In our study, we observed participants working on open-ended proof developments, where there are many more degrees of freedom – and as a result, much more uncertainty — than, say, a homework assignment for a proof assistant class.

How do novice proof-assistant users make the jump between a carefully scaffolded homework and a real-world project? And how can proof assistant instructors facilitate this jump? We saw that our participants demonstrated not only a command of the mechanics of proof writing, such as tactic syntax and usage, but also of the *skills* needed to incrementally and iteratively build proof developments. Future studies could assess how best to instill these skills in novice users.

RO5: *Examine cultures of proof assistant communities.* In our study, we saw glimpses of the broader communities proof writers belong to, which may be small clusters of colleagues, or huge projects such as Lean's `mathlib`. Proof writers learn from the prior work of others, turn to them for assistance, and look to them for guidance on what constitutes a good proof.

Future studies could conduct a more targeted examination on these proof assistant communities. Because we were observing 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, what encourages or discourages proof writers to be active members of a proof assistant community? And when these proof assistant communities intersect with existing proof communities — for example, among mathematicians who already have norms in place for writing, disseminating, and evaluating proofs — what collisions and fusions occur?

8 Related Work

9 Conclusion

References