Towards a Science of Interactive Proof Reading

Jessica Shi [1], Benjamin Pierce [1] and Andrew Head [1]

[1] *University of Pennsylvania, Philadelphia, PA*

Abstract

Proof assistants such as Coq are powerful tools for formally verifying the correctness of software. We are interested in the process of *reading* the proofs produced in this mechanized context, with a goal of building tools to reduce sources of friction and misunderstanding. In this paper, we summarize the early steps we have taken to explore the design space for proof reading support, the results of a pilot study to better understand the proof reading process, and our future plans for this ongoing research.

Keywords: Proof Assistants. Proof Reading. Automation. Development Environments.

1 Introduction

Mechanized proofs are formal mathematical proofs produced with the help of a machine, which checks the proof to ensure its validity. One kind of machine for mechanized proofs is the *proof assistant*. Proof construction proceeds interactively: the user provides information about the proposed next step, while the assistant responds with feedback about how the proof has progressed. Significant work has been and continues to be done to make writing mechanized proofs easier, leading to powerful abstractions, libraries, and automation tools.

We believe there is also significant opportunity to clarify and improve the complementary process of *proof reading*. Whereas pen-and-paper proofs need to convince the reader of their correctness, mechanized proofs need to convince the machine. This leads to the inclusion of details that can be elided on paper and, conversely, the exclusion of details through automation that must otherwise be supplied manually. As a result, human readability of mechanized proofs suffers, especially due to gaps between the reader's needs and the content of the proof.

But do humans actually read mechanized proofs? Yes, and several situations come to mind. The reader might study example proofs to learn a new technique, perhaps for the purpose of reusing that technique in their own proofs. Or, the reader might examine a broken proof as part of a larger debugging process. And beyond individual proofs, the reader might peruse a proof development to understand how definitions and lemmas connect and how to evolve the development when needed.

Our long-term goal is to work towards a science of proof reading, with three major components: theories on what factors contribute to successful proof reading; tools for reducing sources of friction during proof reading; and evaluations of these theories and tools against the experiences of users. We have collected some initial thoughts in this paper.

2 Background

2.1 Scope of Present Inquiry

There are usability problems for myriad formal verification tools, but each of them requires solutions fit to their particular contexts. We focus on *interactive theorem provers*, where the user drives proof construction, as opposed to *automated theorem provers*, where a constraint solver plays the central role. The rest of this paper grounds its discussion by considering the specific environment of Coq [1]; in the future, we are also interested in other provers such as Lean [2] and Isabelle [3].

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

2.2 Definitions

We next discuss two intertwined components that are involved when reading proofs in proof assistants: the proof script and the proof state. Suppose a reader is trying to understand the proof of this theorem:

```
∀ (P Q : Prop), P → Q → P ∧ Q.
```

The theorem says, if we assume that propositions `P` and `Q` hold, then we can show that `P ∧ Q` hold, where ∧ is logical conjunction. A proof may proceed by splitting into two cases and proving each of `P` and Q separately using the given assumptions. In a mechanized version of this proof, the *proof script*, supplied by the author, might look like this:

```
Proof.
  intros. split.
  - assumption.
  - assumption.
```

Proof scripts in Coq consist of a series of *tactics* — in this case, intros, split, and assumption. Tactics are commands that guide the proof assistant on how to approach the next part of the proof. For example, right before the split tactic, the *proof state* looks like this:

```
P, Q : Prop
HP : P
HQ : Q
--------------
P ∧ Q
```

Above the dividing line are the *hypotheses* in our *local context*. Below is the *conclusion*, the statement that is being proven. Together, the hypotheses and the conclusion form a *goal*. After the split tactic is evaluated, the proof state becomes

```
...
--------------
(1/2)
P
(2/2)
Q
```

The purpose of the split tactic is to break apart logical conjunctions; that is, the proof assistant separates goals of the form P ∧ Q into two goals P and Q. Crucially, the proof state is visible to users! In fact, the reader can *step through* the proof by evaluating the proof script incrementally and observing how the proof state changes in response.

2.3 Automation and Readability

Part of a science of proof reading is to understand the mechanisms that can affect the readability of the proof. Continuing with the previous example, the assumption tactics in the second and third line solve their respective subgoals by observing that the conclusion matches a hypothesis in the context. When using assumption, the writer does not need to specify the name of the hypothesis; instead, Coq automatically finds the right one. But if the reader wants to know that assumption uses HP in the second line and HQ in the third, they would need to manually reason about the proof state.

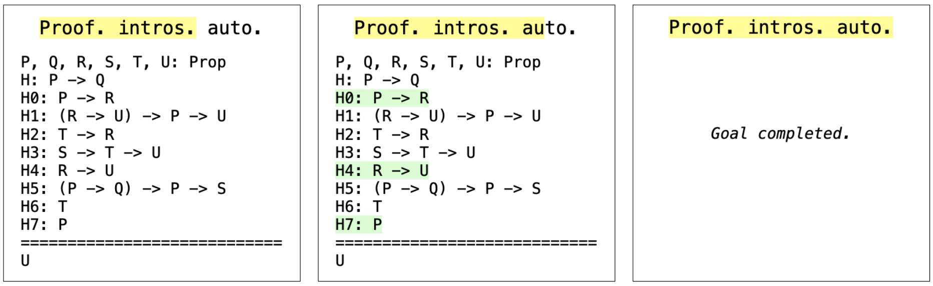
This proof could be further automated. Instead of repeating the same tactic twice, the writer could have split; assumption. The semicolon applies the tactic that follows to every goal generated by the tactic that precedes — it is an example of a *tactical*, or a tactic whose arguments are other tactics. An even more compact alternative is to complete the entire proof with just auto., which can solve certain goals by trying a limited number of tactics. From the perspective of the reader, the process of stepping through the proof becomes quite coarse in the face of automation. Coq evaluates one sentence — a chunk of the script generally ending in a period — at a time. In these single-sentence proofs, the proof state immediately jumps from the original statement to a message along the lines of “no goals remaining,” obscuring the intermediate steps along the way.

3 Design Ideas

In this section, we present two ideas for when tooling could aid proof reading. We focus on lightweight interventions that convey otherwise hidden information about *how* the tactics (3.1) and tacticals (3.2) in a mechanized proof work the way they do.

3.1 Tactic Previews

When a user reads a proof in Coq, they can observe how the proof state changes, but this is only a partial view of what is happening beneath the surface. For example, the auto tactic often solves



```
Proof. intros. auto.

P, Q, R, S, T, U: Prop
H: P -> Q
H0: P -> R
H1: (R -> U) -> P -> U
H2: T -> R
H3: S -> T -> U
H4: R -> U
H5: (P -> Q) -> P -> S
H6: T
H7: P
============================
U
```

```
Proof. intros. auto.

P, Q, R, S, T, U: Prop
H: P -> Q
H0: P -> R
H1: (R -> U) -> P -> U
H2: T -> R
H3: S -> T -> U
H4: R -> U
H5: (P -> Q) -> P -> S
H6: T
H7: P
============================
U
```

```
Proof. intros. auto.

Goal completed.
```

Figure 1. This proof involves a complicated-looking goal that can be solved with the `auto` tactic. From left to right, the panels show (1) the proof state before `auto`, (2) the hypotheses used by `auto`, and (3) the proof state after `auto`. Coq would step directly from the first panel to the third panel, but by adding *half steps*, the reader can view the information in the second panel and get further clarity on how `auto` solves the goal.

a goal by finding some combination of hypotheses to apply, but which combination is hidden to the reader unless they look for it manually, such as by examining the *proof term* that Coq builds from the proof script. (And, as a participant in a later study put it, "I would never look at the proof term.")

We propose *tactic previews*, which highlight the hypotheses in the context that are about to be used by a particular tactic. When should this visualization appear? Consider a proof script marked with three potential locations for previewing the `current` tactic:

```
previous.(A) cur(B)rent.(C)
```

If we chose location `(A)`, the new information would coincide with the existing information about how `previous` changed the proof state; this is unlikely to reduce the cognitive burden. If we chose location `(C)`, we would rapidly run into problems, since tactics often cause hypotheses — or the entire goal — to disappear, so there would nothing left to highlight. Hence we chose `(B)`.

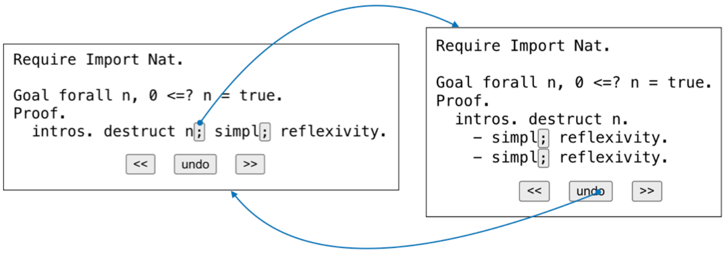
Figure 1 shows an example, where the reader can optionally preview the `auto` tactic by taking a *half step* or skip the preview by taking the usual full step. At the half step, the proof state is augmented to show which hypotheses are used by `auto`, thus directing the reader's attention to the relevant parts of the proof state and helping them understand the proof at a greater level of granularity.

One way to implement such a tool is to examine the proof term before and after a tactic is applied. We can then count the number of times each hypothesis is used in the term and indicate to the reader the hypotheses that increased in count. IDEs can already show a visual diff of *what* a tactic does to the proof state; we are instead asking *how* a tactic is able to work, so we compute a syntactic diff of the proof term. This approach also allows us to be agnostic to what the tactic is. We prototyped a version of this approach that computes the diff by performing rudimentary string parsing of the proof term, which is capable of producing the assistance shown in Figure 1; a robust solution supporting general cases would require more nuanced integration with how the proof assistant builds proof terms.

3.2 Tactical Refactoring

Coq tacticals such as the semicolon alter the structure of a proof script, which in turn alters what the proof reader sees as they step through the proof. In *Dynamic Proof Presentation* [4], Jackson advocates for allowing users to read proofs at the level of detail suitable to their needs, independently of how the proof was written. One line of research focuses on *proof refactoring* [5]. For instance, Tactician [6] is a tool for the HOL Light theorem prover [7] that can, in their terminology, *unravel* a tactical-heavy proof into one with step-by-step tactics and *package* a proof in the other direction.

We can build on and extend this work in at least two ways. First, our understanding is that Tactician's transformations take the whole proof from one style to another; readers may instead want more localized, fine-grained control over which tacticals to refactor. Figure 2 shows an example for semicolon unraveling, implemented by querying Coq about the number of subgoals remaining. The reader can click on a semicolon to expand that segment of the proof script into the individual cases that have been collapsed, so that they can step through the proof at a slower pace.



```
Require Import Nat.

Goal forall n, 0 <=? n = true.
Proof.
  intros. destruct n; simpl; reflexivity.
```

```
Require Import Nat.

Goal forall n, 0 <=? n = true.
Proof.
  intros. destruct n.
    - simpl; reflexivity.
    - simpl; reflexivity.
```

Figure 2. This proof uses the semicolon tactical, collapsing cases that can be handled with the same tactics into a single step. With the help of semicolon *unraveling*, readers can toggle between a more compact view of the proof script (left) and a more step-by-step view (right) by clicking on the appropriate buttons. Unraveled scripts enable the reader to see intermediate proof states when stepping through the proof.

Related tacticals create ambiguous refactoring situations. The `repeat` tactical applies a tactic until it fails to make progress, while the do tactical takes an additional parameter `n` and applies a tactic `n` times. We could either package the script `tac. tac. tac.` as `repeat tac.` or `do 3 tac.` The former runs the risk of causing an infinite loop, while the latter is more brittle. In scenarios like this one, we could make and justify a heuristic, or we could leave the decision up to the reader. Indeed, the second way we envision extending prior proof refactoring work in the future is by evaluating the usability and effectiveness of different points in the design space through user studies.

4 Pilot Study

We conducted a pilot study with three goals in mind: (1) develop preliminary observations about how users read Coq proofs; (2) gather feedback on the ideas for proof reading aids described in Section 3; and (3) prepare for future studies.

4.1 Methods

- **Participants:** We recruited four PhD students from our labs. We chose two participants (P1 and P2) whose only experience with Coq is through classes using *Software Foundations* [8] and two participants (P3 and P4) who currently use Coq for their research.
- **Tasks:** We asked each participant to complete three tasks, with a goal of seeing how they interacted with Coq when presented with situations that probed their understanding of the proof. The participants used their own editors on their own devices.
- **Interviews:** We then conducted semi-structured interviews. In order to help readers better understand proofs, we need to clarify how users define understanding. So, each interview started with the question, "What is your personal threshold for what counts as understanding a proof?," and the conversation proceeded based on their answers, with a focus on why this was their threshold and difficulties they encountered on the road towards understanding.
- **Feedback:** Finally, we solicited feedback on the half-step tactic previews from Section 3. This part of the study was deliberately non-standardized; we iterated on the explanations and mockups between participants.

4.2 Results: Task Performance

One task was to fix a proof about binary search trees from *Verified Functional Algorithms* [9]. The proof used semicolons and a custom tactic `bdall`, and we changed it to fail due to a subtle bug in the `insert` definition where a `v'` should have been a `v`. P2 and P3 found the bug in approximately twelve and three minutes, respectively. P1 and P4 did not find the bug.

P1, P2, and P4 all engaged in some level of manually de-automating the proof script. P3 did not modify the proof script but reflected afterwards that undoing some of the automation would have made the error more obvious. P1 unraveled the semicolons into the separate cases but kept `bdall`. P4 further unraveled the definition of `bdall` by slowly building from the smaller tactics it was composed of. P2 reconstructed the proof from scratch, without noticeably referencing the definition of `bdall`.

Hence the automation of the proof script often stood in the way of the participants locating the source of the bug, who needed to understand the proof at a greater level of granularity than the one provided by the author. This lends credence to the usefulness of proof refactoring tooling that facilitates de-automation.

4.3 Results: Interviews on Understanding Proofs

Some themes we identified in this interview phase include:

Users seek high-level understanding. Participants tended to define understanding a Coq proof in terms of whether they could write a pen-and-paper version capturing the core ideas. P4 told us, "I think a proof is about explaining why something is true and if you can't recreate it, then you can't really explain why it's true." P3 specified that they did not mean translating each tactic but rather producing a "very simple proof sketch" with the important parts. P1 phrased understanding as being able to "put things in English."

Pen-and-paper proofs complement mechanization. When probed further, P3 and P4 identified specific affordances of pen-and-paper proofs that they had previously sketched in the context of their research. P4 emphasized necessity: since the contexts of their proofs are quite complex, it is difficult to write a mechanized proof without already having an intuition from a pen-and-paper proof of which hypotheses to use and when. P3 emphasized flexibility: changing the definitions and fixing the proofs that break is a tedious process in Coq; on paper, they can iterate more rapidly.

These themes on the importance of high-level understanding and pen-and-paper proofs suggest that writers of mechanized proofs may have developed useful, conceptual intuitions about their proofs along the way, and so proof readers may benefit if these intuitions are conveyed to them.

Automation clashes with reading. Perhaps influenced by the tasks, participants frequently critiqued the effect of automation on readability. P1 described how automation can "make things really confusing to follow," citing the semicolon as a roadblock to stepping through the proof. P2 reflected that in proofs using automation, they sometimes instead wanted to "understand every single case" and "why those tactics worked."

Automation clashes with debugging. As an example, P3 criticized tactical patterns of the form `all: try solve tac`, which attempts to solve each goal with `tac`. When successful, the goal disappears; when unsuccessful, the goal remains untouched. But this pattern does not convey information about which goals are expected to be solvable by `tac`. P3 observed that if the proof later breaks, it is hard to determine whether the goals that remain should have been solved by the pattern or are new ones that should be solved from scratch.

Proofs are written in more than one pass. P1 described an example proof writing situation in terms of phases "when I'm first working through a proof" and "when I'm cleaning things up," where the latter involved more compactly writing a series of repetitive tactics produced during the former. Similarly, P4 felt automation can help make a proof smaller and more robust to change, but they do not use it to "actually find the proof ... in the first place." If proof writers are already in the habit of cleaning their proofs, perhaps tooling can nudge and support them in making their proofs more human-readable during this second pass.

Users read example proofs to learn tactics. P4 discussed their experience learning Iris [10], a Coq framework. They have found it hard to learn how to use its specialized tactics, such as what must be done to "get the proof state into a situation where they can work." P4 observed, "Tactics have very complicated specifications, and so we don't usually bother specifying them at all." Instead of the tactic definition, in P4's experience, reading proofs where the tactic is used can be more helpful.

4.4 Results: Feedback on Tactic Previews

Participants generally agreed that tactic previews of the sort we showed them could be useful when reading proofs written by other people. As a result of feedback from P1, who initially thought the half-step was a glitch, we more carefully explained the mode of interaction. Subsequent feedback about the half-step was positive; P3 said they liked that it did not collide with existing IDE functionality, thus avoiding clutter. However, participants did not think the tool would be useful when they were writing the proofs themselves, since they use tactics like `assumption` or `auto` when they already know (or do not care to know) which hypotheses are needed.

4.5 Considerations for Studying Proof Reading

Our findings from the pilot study provide important opportunities for further exploration.

In the task-based portion, we also asked participants questions about simple, completed proofs; after reviewing the transcripts, we realized that we did not know how to interpret the implications of their answers. For example, when asked which hypotheses were used by `auto`, three participants manually found a successful combination that was different from the one found by `auto`. This answer could be considered incorrect, but it may not be incorrect in a meaningful way.

The challenges in task design reflect some of the unique aspects of proof assistants. Because Coq prevents invalid proofs, the user can productively engage in "trial and error" approaches to resolve ambiguities in whether, for example, a tactic works or not. This in turn may lower the level of comprehension accuracy and precision needed. Hence in future studies, task design should center situations where difficulty in reading proofs clearly causes difficulty in writing related proofs.

Our open-ended question about "understanding" at the start of the interview led to conversations that tended to focus on participants understanding their own proofs, often while they were in the process of writing them. Future studies could additionally target situations when the user is reading proofs written by other people, especially when the goal is to understand the mechanized steps rather than just the high-level concepts.

P3 and P4 both use Coq for programming language metatheory; by contrast, other labs may verify the correctness of complex software systems, which likely leads to different considerations about, say, the role of automation. Future studies will ideally include a broader range of users.

5 Related Work

This research direction connects to the umbrella topic of *proof engineering* [11], which explores how traditional software engineering tasks transfer over to the proof assistant setting. We discussed in Section 3.2 prior work on proof refactoring [5], [6] and *Dynamic Proof Presentation* [4].

A popular interface for Coq is Proof General [12] and the accompanying Company-Coq extension [13], with features such as proof state diffs and proof script folding. Similarly popular is CoqIDE [1], which recently added a tactic debugger. PeaCoq [14] is a novice-oriented interface that has a proof-tree view to visualize proofs hierarchically rather than sequentially. Other tools relevant to proof reading include Alectryon [15], which supports displaying the proof states in literate Coq proofs in the browser, and ProofWidget [16], which supports custom user interfaces for Lean.

There have also been a few user studies about proof assistants. For example, Ringer *et al.* [17] observed how Coq users write and modify their definitions and proofs; Beckert *et al.* [18] conducted focus groups to probe the gap between users' mental models and the proof state. Many of the results of our pilot study echo previous results. Both studies identified the lack of proof refactoring support as a tooling weakness. In the latter study, Isabelle users observed that understanding why certain tactics fail is difficult and that automation can ease proof writing but can also be "unintuitive."

The initial work presented in this paper and the future work we have planned builds upon this body of existing work by centering the proof reading process, identifying specific readability problems in Coq caused by automation, and proposing tools to help the reader make sense of automation.

6 Conclusion

The process of proof reading is a demanding one, requiring knowledge of both the high-level content of the proof and the low-level mechanization details. Difficulties are compounded when automation factors in — the proof writer can skip steps that it expects the proof assistant to fill in on its behalf, and the proof assistant tends not to communicate these details to the proof reader in an easily digestible way. We hope to shift some of the burden off of the proof reader by teasing apart the subtleties of the readings process and building tooling to bolster understanding, and we welcome the community's input on this research direction during these formative stages.

References

[1] "The Coq proof assistant," *url: https://coq.inria.fr/*, 1989–2022.

[2] "Lean theorem prover," *url: https://leanprover.github.io/*, 2013–2022.

[3] "Isabelle," *url: https://isabelle.in.tum.de/*, 1994–2022.

[4] P. B. Jackson, "Dynamic proof presentation," in *Mathematical Reasoning: The History and Impact of the DReaM Group*, Springer, 2021, pp. 63–86.

[5] I. J. Whiteside, "Refactoring proofs," 2013.

[6] M. Adams, "Refactoring proofs with Tactician," in *SEFM 2015 Collocated Workshops*, Springer, 2015, pp. 53–67.

[7] "The HOL light theorem prover," *url: https://www.cl.cam.ac.uk/∼jrh13/hol-light/*, 1996–2022.

[8] B. C. Pierce, A. Azevedo de Amorim, C. Casinghino, *et al.*, *Software Foundations*. Electronic textbook, 2017.

[9] A. Appel *et al.*, *Verified Functional Algorithms*. Electronic textbook, 2016.

[10] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, 2018.

[11] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, Z. Tatlock, *et al.*, "QED at large: A survey of engineering of formally verified software," *Foundations and Trends® in Programming Languages*, vol. 5, no. 2-3, pp. 102–281, 2019.

[12] D. Aspinall, "Proof General: A generic tool for proof development," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2000, pp. 38–43.

[13] C. Pit-Claudel and P. Courtieu, "Company-Coq: Taking Proof General one step closer to a real IDE," 2016.

[14] V. Robert and S. Lerner, "PeaCoq," *url: http://goto.ucsd.edu/peacoq*, 2014.

[15] C. Pit-Claudel, "Untangling mechanized proofs," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, 2020, pp. 155–174.

[16] E. W. Ayers, M. Jamnik, and W. T. Gowers, "A graphical user interface framework for formal verification," in *12th International Conference on Interactive Theorem Proving (ITP 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[17] T. Ringer, A. Sanchez-Stern, D. Grossman, and S. Lerner, "REPLica: REPL instrumentation for Coq analysis," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020, pp. 99–113.

[18] B. Beckert, S. Grebing, and F. Böhl, "A usability evaluation of interactive theorem provers using focus groups," in *International Conference on Software Engineering and Formal Methods*, Springer, 2015, pp. 3–19.