Towards a Science of Interactive Proof Reading

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

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

Abstract

Proof assistants such as the Coq theorem prover 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 understand the design space of comprehension aids and conduct a pilot study, as well as our future plans for this ongoing research project.

Keywords: Proof Assistants. Program Comprehension.

1 Introduction

Proof assistants are formal verification tools. Constructing mechanized proofs in proof assistants is a collaborative process: the user provides information about the proposed next step, while the tool responds with feedback about how the proof progresses. Significant work has been and continues to be done to make proof writing easier, including through increasingly powerful abstractions, libraries, and automation tools. We believe there is also significant opportunity to clarify and improve the complementary process of *proof reading*.

If we consider mechanized proofs as mathematical artifacts, then we encounter the first challenge. In mathematics, proofs are written to convince the reader of its correctness. In verification, proofs are written to convince the computer. This leads to the inclusion of details that we elide on paper and vice versa, the exclusion of details through automation that we otherwise need to supply manually. Hence human-readability may become a lower priority.

If we consider proofs as programs, we arrive at the second challenge. A magical model that takes a mechanized proof and outputs a natural language proof would not solve the problem of proof reading, since we also need to address the use case of *reading proofs to write proofs*. We should support, for example, learning the language by reading example proofs and debugging by reading broken proofs.

In reality, proof assistants sit at a precarious but exciting intersection between a medium for mathematical communication and a complex programming environment. A proof reader may want to understand high-level concepts, low-level mechanization techniques, or some combination of both. Our long-term goal is to tease apart the subtleties of this process and to build tooling to bolster comprehension; in the short-term, we have collected some initial thoughts in this paper.

2 Background

2.1 Scope

There are usability questions throughout the world of formal verification, but they are unlikely to have uniform answers. 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. This paper is conducted entirely within Coq [cite], but in the longer term, we are also interested in other tactic-based provers such as Lean [cite] and Isabelle [cite].

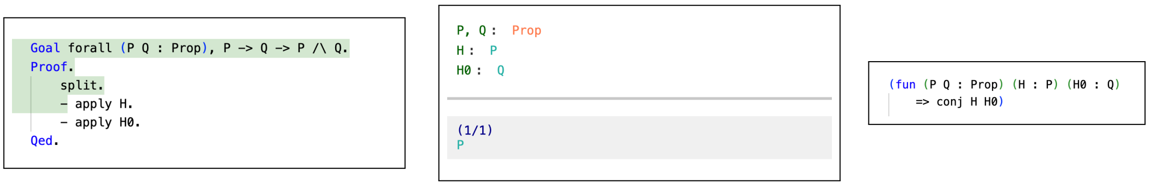
PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

2.2 Definitions

We next discuss three intertwined components: the proof script, the proof state, and the proof term. The user incrementally writes a *proof script*, which is composed of a series of *tactics* — commands that indicate how to approach the next step. Depending on the contents of the script, Coq updates the *proof state*. Each *goal* is visually separated into what is above and below a dividing line. Above are the *hypotheses* in our *local context*. Below is the *conclusion*, the statement we are trying to prove. We finish the proof when we have discharged all goals. Under the hood, Coq is also building the *proof term*, which is not visible to the user except upon request.



```
Goal forall (P Q : Prop), P -> Q -> P /\ Q.
Proof.
    split.
    - apply H.
    - apply H0.
Qed.
```

```
P, Q :  Prop
H :  P
H0 :  Q
────────────────
(1/1)
P
```

```
(fun (P Q : Prop) (H : P) (H0 : Q)
   => conj H H0)
```

Figure 1. A proof script (left) about ∧, the proof state (middle) partway through, and the proof term (right).

2.3 Extended Example

Figure 1 contains a proof about logical conjunction. Even in this simple example, the script already reflects implicit stylistic choices. Consider these alternatives, in increasing order of automation:

- `Proof. split. - assumption. - assumption. Qed.`
  The second and third lines of the original proof script solve the goal by observing that the conclusion matches a hypothesis in the context. Instead of specifying the hypothesis name, we can use the `assumption` tactic, which automatically finds the right one.
- `Proof. split; assumption. Qed.`
  Instead of repeating the same tactic, we can more compactly write the previous proof using `;`, which applies the tactic that follows to every goal generated by the tactic that precedes. The semicolon is an example of a *tactical*, or a tactic whose arguments are other tactics.
- `Proof. auto. Qed.`
  We could have completed the entire proof with just `auto`, which can handle certain goals by trying a limited number of tactics.

How do these choices made by the proof writer affect the proof reader? In the text of the proof script, details are obscured as the proof becomes more automated: without additional background, we do not know the hypotheses used by `assumption` or `auto`. And for the proof state, Coq evaluates one sentence — a chunk of the script generally ending in a period — at a time. Hence *stepping through* the proof, a common reading approach, becomes quite coarse in the face of automation: in the second and third alternatives, which are single-sentence proofs, the proof state immediately jumps from the original statement to a message along the lines of "no goals remaining."

Yet, automation serves an important communicative purpose. From the perspective of reading proofs to write proofs, knowing that a statement of a particular shape can be proved with `auto`, for example, is valuable information. More broadly, all tactics can be considered a form of automation, since Coq translates each tactic into a corresponding change in the underlying proof term. The scope of the automation performed by a tactic simply reflects the extent of the abstraction barrier. The interesting issues arise, then, when the proof reader wants to peek beneath or otherwise perturb the barriers set by the proof writer.

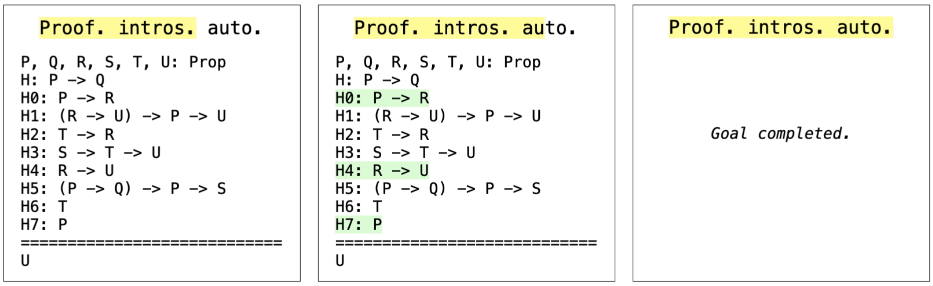
3 Design Ideas

In this section, we present two aspirational directions for building tooling to improve the proof reading process. We focus on lightweight interventions that convey otherwise hidden information about *why* 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 actually happening beneath the surface. For example, the `auto` tactic often solves the goal by finding some combination of hypotheses to apply, but which combination is hidden to the user unless they either reconstruct it manually or read the proof term. And, as a participant in our pilot 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? Suppose we have a proof script



```
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
```

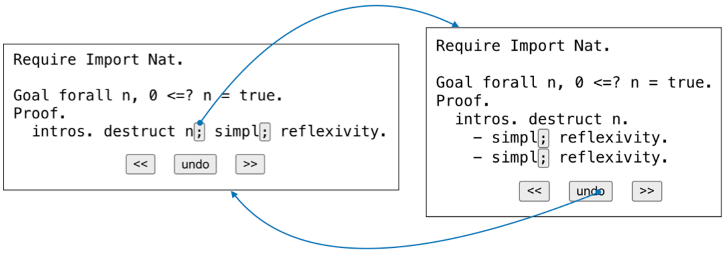
```
Proof. intros. auto.

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 2. Coq would step directly from the first panel to the third panel, while our "half-steps" allow users to optionally view what hypotheses are being used by `auto` in the second panel.



```
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 3. Users can toggle between a more compact view (left) and a more step-by-step view (right) by clicking on the appropriate buttons.

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, we chose `(B)`. The user can optionally preview `current` by taking a *half-step* or skip the preview by taking the usual full step. Figure 2 shows an example.

One way to implement such a tool would be 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 mark the ones that increased in count. IDEs can already show a visual diff of *what* a tactic does to the proof state; we are instead asking *why* 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 using rudimentary string parsing, which is strong enough to handle Figure 2, but a much more careful implementation is needed in general. There is not a one-to-one correspondence between the names used in the proof state versus the proof term; for example, if we `apply H1 in H2`, the proof state will update `H2` in place, while the proof term will generate a new name. This is a significant problem that will take some head-scratching, but we are optimistic that a nice (or at least interesting) solution exists.

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* [cite], the authors advocate for allowing users to read proofs at the level of detail suitable to their needs, independently of how the proof was written. Part of their work has been on *proof refactoring* [cite]. Tactician [cite] is a tool for the HOL Light theorem prover [cite] 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 proof from one style to another; users may instead want more localized, fine-grained control over which tacticals to refactor. Figure 3 shows a toy example for semicolon unraveling, implemented by querying Coq about the number of subgoals remaining. Many design decisions remain about the interface and how to perform the refactoring.

Related tacticals create ambiguous refactoring situations. The tactical `repeat` applies a tactic until it fails to make progress, while do 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 user. 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 ran a pilot study with three goals in mind: (1) develop preliminary observations about how users read Coq proofs; (2) gather feedback on comprehension aid ideas; 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* [cite] and two participants (P3 and P4) who currently use Coq for their research.
- **Tasks:** We asked each participant to complete a few tasks, with a goal of seeing how they interacted with Coq when presented with situations that probed their understanding. The participants used their own editors on their own devices.
- **Interviews:** We then transitioned to semi-structured interviews. 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 an emphasis on delving into why this was their threshold and difficulties they encountered on the road towards understanding.
- **Feedback:** Finally, we asked for feedback on our tooling ideas. This portion was deliberately non-standardized; we iterated on the explanations and implementations between participants.

4.2 Results: Tasks

One task was to fix a proof about binary search trees adapted from *Verified Functional Algorithms* [cite]. The proof used semicolons and also a custom tactic `bdall`, and it failed 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`.

4.3 Results: Interviews

In the interview phase, we engaged in discussions about understanding proofs. Some themes:

Users seek high-level understanding. Participants tended to define understanding a Coq proof in terms of whether they could write a paper version. P4 said, "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."

Paper proofs support exploration. When probed further, P3 and P4 identified specific affordances of paper proofs in the context of their research. 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. P4 emphasized necessity: the contexts of their proofs are quite complex, so it is difficult to proceed without already having an intuition for which hypotheses to use and when.

Automation can clash with reading. Perhaps influenced by the tasks, participants had a lot to say about automation. P1 said it can "make things really confusing to follow," citing the semicolon as a roadblock to stepping through the proof. P2 said that in proofs that used automation, they sometimes instead wanted to "understand every single case" and "why those tactics worked."

Automation can clash with debugging. As an example, P3 criticized tactical patterns of the form `all: try solve tac`, which attempts to discharge each goal with `tac`. When successful, the goal disappears; when unsuccessful, the goal remains untouched. 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 a scenario 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 said 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."

Learning tactics is difficult. On the novice side, P1 and P2 acknowledged that they do not always understand the tactics they use; P1 said they sometimes perform a "random search" of tactics to see what sticks. And on the expert side, P4 discussed their experience learning Iris [cite], a Coq framework. P4 has found it hard to know when and 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 also observed, "Tactics have very complicated specifications, and so we don't usually bother specifying them at all." Instead of the tactic definition, P4 said that proofs where the tactic is used can be more helpful references.

4.4 Results: Feedback

We showed participants mockups of the half-step tactic previews.

Participants seemed to generally agree that such a tool 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. P4 did note an interesting exception: some tactics introduce *existential variables*, which lead to ambiguity until these variables are instantiated by a later tactic. Proof writers can sometimes run into issues if they misunderstand the instantiation.

4.5 Discussion

Our results — or in some cases, lack of results — 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. To improve our tasks, we need better awareness of 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. In hindsight, we should have structured the questions to also target situations where the user is reading completed proofs 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 to prove properties about the languages they are designing; by contrast, other labs may focus on verifying the correctness of large systems, which likely leads to different considerations about, say, the role of automation. Future studies will ideally include a greater number and broader range of users.

5 Related Work

We are adjacent to the umbrella topic of *proof engineering* [cite], since we are interested in exploring how traditional software engineering tasks transfer over to the proof assistant setting. We discussed in Section 3.2 prior work on proof refactoring [cite, cite] as part of *Dynamic Proof Presentation* [cite].

In addition to IDEs in mainstream usage like Proof General [cite], some environments specifically target novices. For example, PeaCoq [cite] provides diffs of proof states (now widely available), proof-tree views, and automatic tactic suggestions. Other tools include Alectryon [cite], which generates literate Coq proofs where the proof state can be directly shown in the browser, and ProofWidgets [cite], which allows users to write custom user interfaces for Lean.

There have also been a few user studies about proof assistants. For example, [cite] collected data about patterns in how Coq users write and modify their definitions and proofs; [cite] conducted focus groups to probe the gap between users' mental models and the proof state. Indeed, many of the results of our pilot study echo previous results. Both [cite] and [cite] 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."

6 Conclusion

We are eager to receive feedback about this proposed research, especially as we continue to improve our sense of where we fit into the much larger landscape of programming language usability. To conclude, we emphasize three themes that articulate why the domain of proof assistants is unique and their implications for proof reading:

1. Proof assistants support mathematics and programming, so proof reading needs to consider both.
2. The proof writer can use automation to transform the structure and granularity of a proof, so tooling should help the proof reader adapt accordingly.
3. The proof writer does not need to be perfect, since the proof assistant will prevent mistakes, so evaluation of the proof reader should be relaxed accordingly.