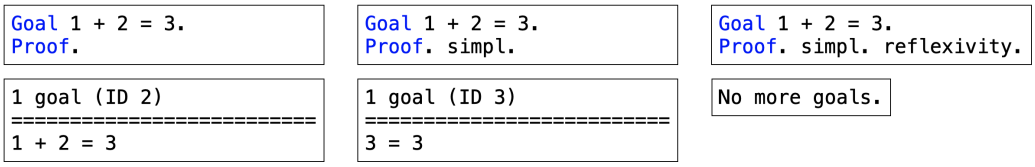
Proposal:

Most programming environments make it easy to write the wrong code. I view breakdowns in correctness as breakdowns in communication. The user thought they were instructing the computer to do one thing, but they conveyed something else entirely. These gaps between intention and implementation become sources of friction at best or sources of bugs at worst.

Imagine if the computer prevented the user from making mistakes. Enter formal verification, which asks us to mathematically prove our program correct. One increasingly popular tool in this space is the *proof assistant*, where the user and the computer collaboratively construct a proof.

What do I mean by "collaboratively"? In the figure below, we prove $1 + 2 = 3$ in the Coq proof assistant. The boxes in the top row are the *proof script*, which the user provides in the form of *tactic* commands, and the boxes in the bottom row are the *proof state*, which Coq updates in response. If we instead attempted to input a tactic that does not work on the current proof state, such as if we try to prove a false statement, Coq would output an error.



I want to move towards a world where it is hard to convey the wrong code and easy to convey the right code. Proof assistants like Coq bring us only halfway. While an incorrect proof will not be accepted, constructing a correct proof requires significant expertise.

As a teaching assistant for a course on Coq, I have observed the learning barriers firsthand. For example, many students tend to write proofs by trial and error. Because users do not need to check *if* their proofs work, they can progress without knowing *why* they work. Such progress is often superficial; brute force does not scale.

My approach will be to build tools that facilitate what I call **proof comprehension**: borrowing from *program comprehension*, which in turn borrows from *text comprehension*, these terms refer to the activity of understanding the medium at hand.

I propose three projects, which if successful, would lead to these usability improvements, respectively. First, users would more easily comprehend existing proofs due to annotations that direct their attention to the relevant details at each step. Second, users would more easily navigate proof developments due to automated suggestions connecting similar proofs. Third, users would more easily write readable proofs due to evidence-backed rules of what constitutes good style.

Proof Annotations. Reading a proof requires processing large amounts of information. If we categorize the information as explicit versus implicit and as Coq-provided versus user-provided, we can discover several opportunities for enhancement.

The proof state is explicit, Coq-provided information. As our proofs become more intricate, proof states can become quite uniweldy: a colleague once remarked that after extensive simplification, her state finally fit on one screen. Building on the concept that effective visualizations do not have to be elaborate interventions and can just tell users where to look[1], we can support annotations that highlight the portions of the proof state that a particular tactic relies on, and vice versa.

Documentation is usually explicit, user-provided information. Some forms of documentation may need to be changed when the proof is changed — for example, comments outlining each case of a case analysis may become out of order. I believe that tooling can automate parts of this maintenance process, leading to better accuracy.

Finally, there is also implicit information, especially due to *automation*: proof writers often omit details for convenience, but these omissions may hurt proof readers. Uncovering some details during the reading process would shed light on why a proof works.

Proof Navigation. Beyond localized comprehension tasks, I also want to support users as they navigate an entire development, which can contain numerous definitions, theorems, and even custom tactics, as well as proofs showing these elements in action, spread across various files and directories. The search functionality provided by Coq centers fact-finding: it is most effective when we know roughly what we are seeking and only need to recall, say, a theorem name.

I envision additional search mechanisms that emphasize proactive exploration. In particular, given a particular proof state — such as in the middle of a proof the user is stuck on — the ideal tool would automatically identify similar states in previously completed proofs. To do so, we need to establish heuristics that capture the shape and salient characteristics of a proof state.

Proof Style. There are existing ideas about what it means for a program to have "good style." Some conventions can be formalized into *rules of discourse*, and program comprehension studies[2] show that programs deviating from these rules are harder to comprehend.

I wish to find the rules of discourse for proof assistants, likely via observational studies of Coq experts and content analyses of artifacts. Equipped with a better understanding of how best to write a mechanized proof, tooling can then nudge users towards refactorings.

Timeline:

I started graduate school by joining my lab's ongoing research on *property-based testing* (PBT), a form of software testing where the inputs are randomly generated and the expected behavior is encoded as properties. I am co-leading a project to popularize robust empirical evaluation of PBT tools. Our experiments will help practitioners by clarifying best practices, especially when navigating tradeoffs, and our infrastructure will help researchers easily run their own experiments.

We are on track to submit a paper in early 2023, and we hope this work will change the way the PBT community thinks about evaluation. From a personal perspective, this has also served as an important opportunity to learn how to push a research project from start to (almost) end, especially with respect to refining and elevating ideas through discussions with collaborators.

On the proof assistant side, I have been gradually concretizing my research plan, which has evolved into the proposal above. Last semester, I led a reading group on *proof engineering*, the intersection of proof assistants and software engineering. This semester, I have embarked on the early stages of the proof annotation project.

I am optimistic that with a careful combination of research methods in programming languages and human-computer interaction, I can establish a clearer picture of how to effectively communicate with a proof assistant. I hope this will be one step towards a programming future where we simply cannot help but correctly translate our intention into implementation.

[1] *Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools.*
[2] *Empirical Studies of Programing Knowledge.*