Proposal:

In most programming environments, it is easy to produce code that is wrong. 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 an environment that prevents 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.

```
Goal 1 + 2 = 3.
Proof.
```

```
1 goal (ID 2)
==========================
1 + 2 = 3
```

```
Goal 1 + 2 = 3.
Proof. simpl.
```

```
1 goal (ID 3)
==========================
3 = 3
```

```
Goal 1 + 2 = 3.
Proof. simpl. reflexivity.
```

```
No more goals.
```

My research interest is to move towards a world where it is hard to produce code that is wrong and easy to produce code that is right. With proof assistants like Coq, we are 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, since 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 can more easily comprehend existing proofs due to annotations that direct their attention to the relevant details at each step. Second, users can more easily navigate proof developments due to automated suggestions connecting similar proofs. Third, users can more easily write readable proofs due to evidence-backed rules of what constitutes good style.

Proof Annotations. Statements rarely hold true in a vacuum; rather, they tend to rely on a set of assumptions, which Coq displays in the *context* — the portion of the proof state above the line. These can become quite unwieldy as our proofs become more intricate: a colleague once remarked that after extensive simplification, her context finally fit on just one screen.

As they stand now, proof assistant environments tend to only provide information about what a tactic did and not why it worked. A concept in programming tools research that I have found captivating is that interventions do not need to be elaborate to be effective; instead, visualizations can, for example, just tell users where to look[1].

Simple annotations of the proof script and proof state can thus improve the status quo. In one direction, tactics may rely on certain assumptions in the context — a tool should draw attention to these. In another direction, the assumption shown at a particular stage in the proof may have been changed by previous tactics — again, a tool should point these out.

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 support 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 of these 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 users. One interesting aspect will be automation: proof writers can omit details for convenience, but these omissions may hurt proof readers. 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 work on *property-based testing*, 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 support empirical evaluation of input generation strategies. This includes experiments to understand what factors inform a strategy's bug-finding ability and infrastructural support for executing experiments in a reproducible, extensible way.

We are on track to submit a paper in early 2023, and we hope that our work will be valuable to the PBT community. In particular, practitioners can have more clarity on tradeoffs in choosing among existing techniques, and researchers can access a well-calibrated system for assessing new techniques. From a personal perspective, this project 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 interests, which have evolved into the proposal above. Last semester, I led a reading group on *proof engineering*, the research area that studies the intersection of proof assistants and software engineering. This semester, I have embarked on the early stages of the proof annotation project and am excited to see where it takes me.

I am optimistic that with a careful combination of research methods in programming languages and human-computer interaction, we can achieve a clearer picture of how to effectively communicate with a proof assistant and build the tools that make this picture a reality.

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