I am a second-year computer science PhD student at the University of Pennsylvania. I am a member of the programming languages research group, and my advisor is Prof. Benjamin Pierce.

Research Interests

My research interests fit under the generous umbrella of *program correctness*. That is, how do we ensure that our code behaves according to its specification? From a practical standpoint, the benefits of having good answers to this question are clear: software permeates every aspect of our lives, and bugs in this software can interfere with their safe operation.

But what draws me to this research area can be framed a bit differently. I believe that breakdowns in program correctness are fundamentally breakdowns in communication. The user thought they were instructing the computer to do one thing, but they conveyed something else entirely. To address situations like this, I am excited to explore approaches that center usability and to build tools that bridge the gap between intention and implementation.

Research Experience: Property-Based Testing

One rather large bucket in the program correctness toolbox is testing. We test our code by running it on a number of inputs and confirming that the outputs were as expected. Our research group focuses on *property-based testing*, where inputs are randomly generated and expectations are encoded as properties the program should satisfy.

Currently, 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 these experiments in a reproducible and extensible way. We are on track to submit the work as a conference paper in early 2023.

Research Proposal: Proof Comprehension

A limitation of testing is that even if we use, say, a thousand inputs, we still cannot be sure whether the thousand-and-first input will also pass. Enter formal verification, which instead asks us to mathematically prove our program correct. Among the tools within this space, I am interested in *proof assistants*, where the user and the computer collaboratively construct a proof.

These proof assistants are increasingly popular in both mathematics and computer science domains, used in complex projects ranging from a proof of the four color theorem to a verified C compiler. To support and improve the verification process, there has also been a recent flurry of activity in *proof engineering*, described in a survey paper[1] as "software engineering for proofs."

Last semester, I led a proof engineering reading group, and of the problems identified in our discussions, I felt most compelled to action by the difficulty of reading proofs written in proof assistants, having experienced this myself as a formidable learning barrier. As a result, my desired research focus is to improve what I call *proof comprehension*: the intersection between the unique, evolving setting of proof assistants and the well-established field of program comprehension.

The sections that follow will describe projects to (1) augment the reading process with proof annotations, (2) probe how proof style affects readability, and (3) navigate a large proof development. I am currently in the early stages of the first project, so naturally the latter projects will be adapted and concretized as time goes on.

Proof Annotations. 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.

```
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.
```

So what does comprehending this entail? A common refrain is that one should "step through the proof," which typically involves processing one tactic in the proof script at a time and observing how the proof state changes. Indeed, the figure has been arranged so that each column represents a snapshot of what would be displayed at each step.

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[2], visualizations can, for example, simply help users know where to look.

I believe that simple annotations of the proof script and proof state can improve the status quo. In one direction, tactics may rely on certain assumptions in the context — a reader might want to know which. In another direction, the assumption shown at a particular point in the proof may have been changed by previous tactics — again, a reader might want to know which.

Proof Style. There are existing, strongly-held ideas about what it means for a program to have "good style" and be void of "code smells." Some of these conventions can be formalized into *rules of discourse*, and studies[3] in the program comprehension literature have shown that programs that deviate from these rules are harder to comprehend.

By contrast, there is a sprinkling of "folklore" wisdom about how to write a mechanized proof that is easy to understand. I would like to evaluate these claims and determine what the rules of discourse are for proof assistants. One particularly interesting aspect will be *automation*: proof writers can omit details for convenience, but these omissions may hurt proof readers.

Proof Navigation. We have thus far focused on localized comprehension tasks, where we have a specific proof or tactic that we want to understand. 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 completed proofs that show these elements in action.

One way to make progress in a new proof is to know how old proofs were written, but absorbing and remembering this information can be cumbersome. I would like to build a tool to automatically suggest proof states that resemble the one at hand, likely by determining useful heuristics for proof similarity.

***

I am excited to move towards a world where we can be confident in our software's correctness. I hope to contribute by improving the usability of proof assistants, especially by facilitating clearer communication between the user and the computer. In this statement, I have outlined my current agenda, and I look forward to evolving this plan as I evolve as a researcher.

[1] *QED at Large: A Survey of Engineering of Formally Verified Software.*
[2] *Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools.*
[3] *Empirical Studies of Programing Knowledge.*