How do people learn to use interactive theorem provers?

AUDREY SEO, Paul G. Allen School of Computer Science & Engineering, University of Washington, USA
YADI WANG, Paul G. Allen School of Computer Science & Engineering, University of Washington, USA
ZHENNAN ZHOU, Paul G. Allen School of Computer Science & Engineering, University of Washington, USA

ACM Reference Format:
Audrey Seo, Yadi Wang, and Zhennan Zhou. 2022. How do people learn to use interactive theorem provers?. In *CSE 510: Advanced Topics in HCI, Seattle, WA*. ACM, New York, NY, USA, 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION AND MOTIVATION

In the past, verification tools were considered to be a frivolous, unrealistic pursuit that would be of little use to programmers and the software industry. De Millo et al. famously ridiculed the desire to produce verified software, stating "A sufficiently fanatical researcher might be willing to devote two or three years to verifying a significant piece of software if he could be assured that the software could remain stable" [5]. Of course, software is almost never stable, especially if it is in active development. In addition, many software engineers believe that verification has a steep learning curve and requires a monumental effort to verify even small pieces of code [12]. However, the status quo has recently changed due to numerous factors.

Software programs become increasingly pervasive and critical in every aspect of our modern social activities. Software crashes, malfunctions, and low performance can be incredibly expensive for companies, which increases the importance of verifying that a piece of software does what it is supposed to do. While testing is the most common approach in software engineering of ensuring that software meets the spec it is supposed to implement, formal verification through machine-checked proofs by *interactive theorem provers* (ITPs) provides stronger guarantees and meets a higher standard of correctness. Researchers and industry professionals have developed many ITPs, such as Coq [18], Lean [11], Isabelle [7], and Agda [17], to support various proof tasks. For instance, ITPs have been used to verify seL4 [10], an operating system microkernel, and Raft, a distributed computing consensus algorithm that is equivalent to Paxos in fault tolerance [21].

Among them, Coq is one of the most widely-used interactive theorem provers. Coq has a unique live programming interaction model. In most conventional programming languages, programmers write a whole program, which is then executed in its entirety by an interpreter or compiled to an executable. With Coq, however, programs can be executed incrementally, stepped forward or backward one command at a time, and constant feedback is provided by a *read-eval-print loop* (REPL). The REPL provides important context for completing proofs in Coq, such as the *proof state*, which contains of available hypotheses that are known to be true as well as the remaining cases of the proof that need to be proved. Coq programs consist of a *specification language*, called Gallina, and a *tactic language*, called Ltac, which is used to write *proof scripts* (see Figure 1 for an example of proof state and proof scripts). Researchers have used it to

Manuscript submitted to ACM

develop Verdi, a framework used to formally verify distributed systems [20], verify an optimizing C compiler [8], a disjoint set data structure [4], and the four color map theorem [6]. Coq is also the language that is formally taught in CSE 505, the graduate-level programming languages course at the University of Washington, for the past five offerings.

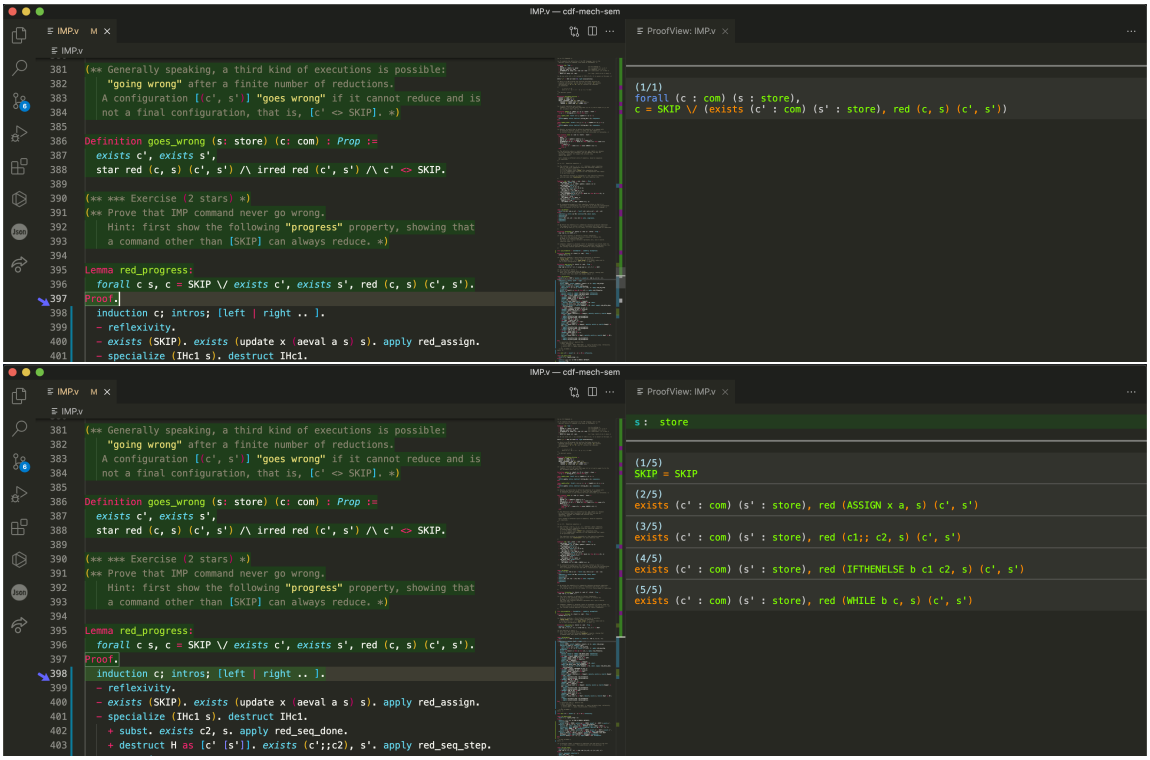

Fig. 1. Screenshots of a Coq program being run in the popular lightweight IDE Visual Studio Code, using the extension VSCoq to provide support for running Coq programs. On the left of each screenshot is the actual Coq program. The `Definition` command is used to define non-recursive variables and functions. The `Lemma` command defines a lemma or theorem. On the right of each screenshot is the proof state. In the second screenshot, the proof has been stepped from the beginning of the proof script to the right after the first line of the proof script (which is represented by the green highlighting), and goes from having one goal to having five. Lines 381-396 are written in Gallina. Lines 398-403 are written in Ltac.

Despite being proven to be widely useful for guaranteeing a high level of security, integrity, and reliability, ITPs are infamous for their steep learning curve. Even within the fields of programming languages (PL) and formal methods, ITPs have a fierce reputation for being notoriously difficult to use. There have been no studies that actually evaluate this commonly accepted folk knowledge, nor any studies that assess the usability of ITPs. Given that verification and proving are only increasing in their power, usefulness, and popularity, evaluating the barriers to learning how to use ITPs — as well as identifying what has worked for those who do use ITPs — is important for improving the accessibility of ITP tools as well as making safety-critical software more secure.

In this paper, we focus on Coq to demonstrate potential learning barriers and ways to improve its general accessibility and usability for both beginner and more advanced users of ITPs. While there are many other ITPs, Coq is an appropriate subject for this study considering the time constraints of this course project, the specific expertise and insight into

Coq that the first author has to offer, and the course CSE 505, which is often taught using Coq, including in its latest iteration in Spring 2021.

In section 2, we give important background and show how other researchers have studied similar problems. We then present our methods and procedures (section 3) for investigating potential learning barriers of using Coq. Then we reflect on our interview and analysis results according to the conceptual model of six learning barriers [9] and identify additional, potentially Coq-specific learning barriers (section 4). We then discuss the implications of our identified learning barriers in improving accessibility and usability of Coq, and more broadly, all ITPs (section 5). Finally, we conclude our work by discussing its limitations and opportunities for future work (section 6). In this report, we make the following contributions:

- A set of possible learning barriers for interactive theorem provers that will guide future investigations into our main research question.
- Initial findings for improving the overall accessibility and usability of interactive theorem provers, in particular Coq, in the fields of verification and formal methods.
- Additional ideas for tools and resources that beginner-intermediate Coq users might find particularly useful.

2 RELATED WORK

ITPs have typically been a niche tool since their inception. Due to their status as research software and the relatively small communities that use these tools, there has not been any work that studies how people learn to use ITPs. However, there has been preliminary work on improvements for ITPs, how instructors can use ITPs to teach other subjects, how to study how people use ITPs, as well as important background work in the areas of computer science education and learning.

Using HCI Methods to Evaluate PL Tools. Chasins et al. describe an initial methodology to organically combine human-computer interaction (HCI) and programming languages (PL) so that together they can make programming easier [3]. In particular, PL researchers and practitioners can make iterative, user-centered design decisions by observing and understanding how users doing complex computing tasks and interacting with programming systems. Feedback from users at multiple design stages can enable language and tool designers make more user-friendly features that solves real user problems. Combining human cognition, behavioral science, and design heuristic, PL researchers can maximize language ergonomics that matches existing mental models of target users, and make programming-specific functionalities that users feel most comfortable with. Usability evaluation provides PL researchers a concrete and rigorous way to measure the effectiveness and efficacy of their language-specific designs, tools, and improvements.

Improvements for ITPs. Due to the high effort needed to use interactive theorem provers, various systems have been developed to support Coq programmers. Proof General is an Emacs package that provides support for the REPLs of multiple interactive theorem provers, including Coq [2]. It is one of the main "IDEs" that Coq programmers use, due to its numerous features and keyboard shortcuts. Another Emacs package, company-coq, provides additional support for Coq akin to that of modern IDEs, including code completion, code-folding, documentation help, code snippets and templates, etc. [14] For an even more automated approach, Yang et al. used machine learning to create a model that attempts to automatically generate a proof of a given theorem [22]. Combined with other proof automation techniques, their model was able to prove 30% of the theorems in their testing set of about 13,000, which outperforms any other technique taken alone.

Studying Coq Users. Tavante describes a potential approach for doing a data-centered user study of ITPs [16]. By utilizing a web-based implementation of Coq, called jsCoq [1], a server could log events in jsCoq to a database, collecting massive amounts of data on users. Ringer et al. instrumented the Coq REPL to gather data on how twelve proof engineers, whose level of expertise ranged from intermediate to advanced [15]. They wanted to understand where proof engineers ran into problems so that automated tools could better assist proof engineers in tasks such as refactoring and proof repair.

Learning Barriers for Programming. Ko et al. describe six barriers to learning a programming system, which consists of all of the supporting software and tools around the usage of a programming language [9]. These barriers include design, selection, coordination, use, understanding, and information. While Visual Basic.NET is often considered an end-user programming system, we decide that the six learning barriers are general enough to be applied to our study on how people learn to use Coq. Moreover, since Coq consists of not only the languages Gallina and Ltac but also the live programming REPL interaction, this paper may provide a more comprehensive framework of Coq as a system. We expand on this work further in Section 4 of our work.

This work is a part of a larger effort to study how people use ITPs. However, there has been no prior work in this project since it is in its beginning stages.

3 METHODS

In order to explore what students found were barriers to learning Coq, their feelings about proof assistants, and their perceived benefits, we conducted interviews with students who took CSE 505 in Spring 2021[1] by Zachary Tatlock and James Wilcox, who were both a part of the Programming Languages and Software Engineering (PLSE) lab at the University of Washington. The class was taught remotely over Zoom and consisted of lectures twice a week where programming languages concepts were introduced and the instructors would write Coq code live in front of students, several readings on various programming languages and verification topics, and six homework assignments where students were expected to write code in Coq in pairs. The homework assignments involved a number of "core" problems, as well as extra "challenge problems" that students could gain additional points for completing.

We chose to interview students who had taken CSE 505 in Spring 2021 in particular for a number of reasons. First, the first author took CSE 505 in Spring 2021, and therefore would have insight into the class structure and added rapport with their former classmates. Second, Spring 2021 is the most recent quarter that CSE 505 has been offered. Ideally, it would have been better to have interviewed students who were currently taking the course so that their memories would be more fresh, but CSE 505 is not planned to be offered at all in the 2021-2022 academic school year at the University of Washington since the main instructor is on sabbatical. Finally, we recruited participants who took the course all in the same quarter so that their experiences would be more standardized, since they all saw the same content and had the exact same homework problems.

3.1 Participants

We recruited participants by emailing past students of CSE 505 via course mailing lists. Our email included a link to a screening survey in which potential participants were asked about their demographic information and their background in mathematics, programming languages, and Coq.

We received 7 responses for our survey, and we invited all respondents to participate in our interview study (see Table 1). In our invitation email, we included a copy of our information sheet for them to preview. All our participants

[1]The course website can be found at https://sites.google.com/cs.washington.edu/cse-505-spring-2021.

Table 1. Demographic data and study logistics for our study participants. P1 and P2 are our pilot study participants.
In the Gender column, PND = "Prefer not to disclose", F = "Female", GQ/NB/GF = "Genderqueer, nonbinary, or genderfluid", and M = "Male";
In the Background in Math column, Little to None = "Little to none experience in mathematical proofs", Undergrad = "Took undergraduate-level course(s) in mathematical proofs", Grad = "Took graduate-level course(s) in mathematical proofs".

Participant	Gender	Interview Format	Occupation	Background in Math	Background in PL
P1	PND	In-person	PhD Student in CS	Undergrad	PL/HCI researcher
P2	F	In-person	Researcher in CS	Grad	PL researcher
P3	GQ/NB/GF	Remote	PhD Student in CS	Grad	Took some courses
P4	M	Remote	Software Engineer	Undergrad	Took some courses
P5	M	In-person	PhD Student in CS	Grad	PL researcher
P6	M	Remote	Software Engineer	Little to None	Took some courses
P7	M	In-person	PhD Student in CS	Grad	PL researcher

were 18 years or older. Four of them identified as male, one identified as genderqueer, nonbinary, or genderfluid, one identified as female, and one preferred not to disclose their gender. All of our participants all reported to at least have some experience taking undergraduate-level mathematics and programming languages courses prior to taking CSE 505, although most of them (4 out of 7) had never used Coq before taking the class.

3.2 Interview Procedure

For ease of interviewing and to accommodate interviews during the COVID-19 pandemic, we decided to let our participants choose whether they wanted to be interviewed in-person or remotely. We ended up interviewing 3 participants via Zoom, and the rest in-person at the University of Washington. Each interview session lasted around an hour, and we recorded audio for all of them. During each interview, one of the authors would take the lead in asking questions, while at least one other author would be present to take notes and ask additional questions.

Our interview protocol began with an introduction of our study objectives, followed by a series of questions regarding their academic and/or professional background, with an emphasis on their experience associated with mathematical proofs, software verification and programming languages. We then asked them about their experience taking CSE 505, particularly their experience learning to use Coq. We concluded our session by asking their opinion about proof assistants in general. We employed a semi-structured interview model, so that we had the flexibility of asking follow-up questions of our participants.

3.3 Qualitative Analysis

After we finished recording all the interviews, we transcribed the interviews using the tool otter.ai [13], which uses machine learning techniques to transcribe audio. After transcription, we manually audited the transcripts to ensure accuracy and to fix misspellings and punctuation. We randomly split up the participants' transcripts into two groups of two transcripts and one group of three transcripts, and each author coded one group. Unfortunately, the audio for P7 was unusable due to audio interference and microphone difficulties, so we instead coded our notes for that participant.

After coding the transcripts, the authors compared the codes they generated and consolidated them. Ideally, each transcript should be coded by at least two different coders. However, we were unable to do so due to time constraints. Once we had a finalized list of codes, we performed a thematic analysis on the data. Using the tool Miro [19], we copied and pasted all of the quotes of interest as sticky notes and tagged them with the various codes we generated. After

Table 2. Ko et al.'s six learning barriers [9]

Learning Barrier	Description
Design	"Inherent cognitive difficulties of a programming problem, separate from the notation used to represent a solution."
Selection	"Properties of an environment's facilities for finding what programming interfaces are available and which can be used to achieve a particular behavior."
Coordination	"A programming system's limits on how programming interfaces in its language and libraries can be combined to achieve complex behaviors."
Use	"Properties of a programming interface that obscure (1) in what ways it can be used, (2) how to use it, and (3) what effect such uses will have."
Understanding	"Properties of a program's external behavior (including compile- and run-time errors) that obscure what a program did or did not do at compile or runtime."
Information	"Properties of an environment that make it difficult to acquire information about a program's internal behavior."

Table 3. Additional learning barriers we identified

Learning Barrier	Description
Dead End	A Coq learner may reach a point in their proof where they do not know how to proceed.
Environment	A Coq learner may feel unwelcomed in Coq's social environment and community.
Applicability	A Coq learner may question why learning to use ITPs can benefit them in real life.

we input all of our quotes, we organized the quotes into general themes, and mapped them to Ko et al.'s six learning barriers [9] if appropriate. We give a brief summary of these barriers in Table 2.

4 RESULTS

In this section, we identify six common themes we observed across all of the interviews. As outlined in Table 4, they are: language deficiencies, learning strategies, language resources, motivation, background knowledge, and pair programming. In the interest of providing a more in-depth, careful analysis for fewer themes rather than a cursory examination of more themes, we decided to leave analysis of the pair programming and language deficiencies themes for future work.

We provide an in-depth analysis of the four other themes, and then evaluate how these themes benefited or detracted from how participants learned to use Coq. When participants experienced difficulties, we additionally analyze whether the framework of six learning barriers applies to the participants' situations. When they fail to fully capture our observations about participants' learning barriers, we propose our own: *Dead End*, *Applicability*, and *Environment* (See Table 3).

Dead End occurs when a Coq user reaches a point in their proof where they do not know how to proceed. This is different from the learning barriers that Ko et al. described, because, as we will see in subsection 4.1, Learning Strategies, the process of proof writing is actually the search of the space of possible proofs, instead of the design and implementation of an algorithm for solving a problem. In conventional programming languages, programmers do not typically try every possible combination of language constructs and functions on a particular line of code before proceeding. Due to the fact that participants reported frequently using a *trial and error* strategy, however, this is

Table 4. The themes generated in our qualitative analysis, and learning barriers that we observed as playing a role in that theme. Learning barriers in bold are ones that we discovered in the process of performing the qualitative analysis.

Theme	Description	Observed Learning Barriers
Language Deficiencies	Issues with syntax, the Ltac language, the Gallina language, and the REPL interaction model.	Design, Selection, Coordination, Use, Understanding, Information, **Dead End**
Learning Strategies	Strategies participants used for learning Coq, which included *trial and error* and *learning by example*.	Selection, Coordination, **Dead End**
Langauge Resources	Use of the official docs, Google, StackOverflow, textbooks, tutorials, cheatsheets, etc. for learning Coq	Use, Understanding, **Environment**
Motivation	Participants' goals and motivating factors for learning Coq	**Applicability**
Background Knowledge	How participants' specific backgrounds (in math, PL, or software engineering) helped or hurt them when they were learning and programming in Coq.	Design, Use, **Applicability**, **Environment**
Pair Programming	Participants' experience learning Coq and working on Coq assignments together with another partner.	**Environment**

precisely what they did in Coq. Due to the exploratory nature of this kind of programming, users can work themselves into a *dead end*.

Our second Coq-specific learning barrier, Applicability, arises when users are not sure how verification using ITPs is useful, do not know how ITPs and the problem solving methods used with ITPs can be applied to problems that they care about, or believe that it would be too much effort to use an ITP to verify software. We frequently saw Applicability associated with the themes Motivation and Background Knowledge, though it may also affect how participants perceived the the documentation (Langauge Resources) or quirks of the Coq language (Language Deficiencies).

The final learning barrier we identified, Environment, refers to the social environment and community surrounding Coq, and how that environment can become a barrier to learning and adoption. This problem is particularly profound in the Background Knowledge theme, since background knowledge can be used as a gatekeeping mechanism in the PL community. It is also applicable in the themes Langauge Resources and Pair Programming, since language resources, which are frequently online for many programmers, are usually created by a community, and pair programming inherently creates a social environment.

4.1 Learning Strategies

Besides using the resources discussed above, participants also named a number of different strategies they used to learn how to use and prove theorems in Coq, such as watching and taking notes of the CSE 505 instructors programming live in class, or explaining tactics to other people such as their homework partner (P1, P4, P5, P6). In this section, however, we will focus on two strategies that are most common among our participants when they tried to learn how to use Coq and wrote proofs on the CSE 505 homework.

Trial and Error. Here, *trial and error* refers to the process of experimenting with a set of different tactics until finding the one with expected behaviors. Part of the reason why participants used a trial and error approach is simply because it was easier, and was facilitated by the Coq REPL. For P3, "it was definitely easier to guess and check than...to check more by reading first and just ...self-proving everything to start with." For other participants, the Coq REPL allowed them to see how parts of the Coq code worked, which helped build up a mental model for programming in Coq. P6 stated that the through the process of trial and error, he "started to get the idea of like, 'Okay, here's...what proving a lemma does, here's...why you might do something...just assert something and then move on, and then come back and prove it

later," and after using trial and error for a while, he "could...see...[how] the building blocks [of the proof could be] put together and imagine how it could be used really, in a really complex way." Similarly, P5 reported, "I loved stepping through the [Coq code]," which allowed him "to work...with [his] partner and...understand...her proofs by stepping through." In conventional programming languages, programmers frequently have problems with reading other's code. P5 however found that even when "there [were] tactics that [he] didn't know...[he] could just step through and see, 'ah, this is what it's doing on this [piece of code]."' He believed that "having that interactive view [was] absolutely critical." In this way, Coq actually *avoids* the Understanding barrier, since the program's external behavior is always displayed by the REPL. Additionally, when the "trial and error" approach was successful, this allowed participants to avoid many of the difficulties associated with the Design, Selection, and Coordination barriers identified by Ko et al.

Participants often compared Coq to playing a video game (P1, P3, P5). Sometimes, this gamification of proof writing was helpful. P5 reported that he "[played Coq] like a video game. So [he] basically...took [a proof goal], and [he] just tried to whack at it with tactics until [the proof] worked." This gamified experience created a "cycle of gratification" for P2, who said "it made [her] feel...like those mice in experiments where they...press the button and get the cookie." In this way, theorem proving in Coq could actually be fun for participants. However, this could also lead to limited understanding for some participants. P2 recalled "if you [couldn't] figure out what the right [tactic] is to make the proof assistant happy then" you would be frustrated. Even so, P2 said "it was almost like, addicting to like, keep trying different things," so she continued to work at the proofs even when she couldn't figure out the problem. For the most part, the trial and error approach worked for participants, but when it didn't, participants were at a loss. P5 said, "The difficulty that arose...when this [tactic] isn't enough, *why*[2] isn't enough?" This Dead End barrier provided a great source of frustration for participants.

Using the trial and error strategy lead participants to develop very ad hoc mental models of Coq's tactics. P5 said he while he "mechanically...picked up [how to use the tactics] eventually, by the end," before he did figure out the tactics, he said, "I never knew which one to use...So I would just use one. And then if it worked, I would go on, and if not, I would use the other. But sometimes, it was really frustrating in figuring out like, why isn't [it working]?...It does look like it's semantically gonna like, work out. Why isn't [the tactic] doing [what I think it should]?" So in addition to running up against a Dead End barrier, P5 was also encountering the Use barrier as well.

Learning by Examples. Some participants also mentioned that seeing examples often helped them learn better, especially when they were not able to write Coq programs completely by themselves as beginners. For example, P1 noted that he "cannot sit and build something from scratch", which is similar to P4, who commented that "I can use it. But could I write it from scratch? Maybe not. Maybe not without a lot more practice." Therefore, many participants agreed that going through examples was an important way for them to go beyond tactic descriptions and to gain more in-depth understandings on tactics. P4 noted that "I guess I understood what they meant by just beyond the terse description, what you see in the in the actual comments in the official quote, unquote, documentation. So it helped me dig a little deeper look at other examples." Moreover, P1 and P6 considered this learning-by-examples strategy the most efficient way for them to learn Coq. P6 noted that "the only way I was really absorbing it at that point was like, here's some examples of code". Similarly, P1 commented, "in terms of tactics, I've never remember, I need to see examples". P1 further argued that by providing more examples, Coq would become a much more friendly programming language from the "unusable tool" it is right now. Specifically, P1 describe the desired improvement as such: "every part of the program should have its own sorts of like, examples of inputs and visualization of outputs".

[2]Emphasis ours.

However, one participant, P5, noted that he found it hard to fully understand examples written by other people. He described how he was not able to extrapolate other's code to his own situation just by reading it online, saying that "just looking at someone's Coq proof doesn't tell you what's going on. You have to step through it." The proof state (see Figure 1) is a rich source of information for Coq programmers, and without it, proof scripts in Coq are nearly incomprehensible, especially if they are long.

4.2 Learning Resources

When learning to use Coq, all of our participants reported that they made use of online or offline resources in addition to attending the CSE 505 lectures when they were working on their assignments. Those resources include but are not limited to: the official documentation written by the Coq development team, documentation and examples curated by other institutions, textbooks, lecture code, and community notes collectively developed by their classmates in CSE 505.

Official Documentation. All of our participants noted that the official documentation produced by the Coq development team was lacking in multiple aspects, such as readability, accessibility, and usability.

P1, for instance, said "the documentation is...just like all jargon...using these, like complicated [terms], not providing really simple examples, like it just makes it, you know, very, very hard for people to use." P1 gave up on using the documentation, and said, "I don't think I would have used the documentation just because the documentation is like, terrible." P2 thought that "it's a problem that the docs are written for Coq experts, not Coq beginners," and observed that "especially compared to some other programming language docs, they're not approachable." She questioned "who are doc's for? And what purpose do they serve? And what are the goals of whoever's writing the docs?" Use and Understanding capture these readability and usability issues in the official documentation, given that it obscures the meanings, running behaviors, and usages of Coq tactics, making it hardly approachable, particularly for beginners. Additionally, the use of jargon produces an Information barrier, since in order to understand the internal behavior, users must understand the jargon.

Multiple participants emphasized the need for some kind of search tool for tactics. As outlined in subsection 4.1, all of our participants engaged in some kind of brute-force trial-and-error method of proof writing. For simpler proofs, this would often complete the proof. But for more complex proofs, participants(P3, P4, P5) reported sometimes hitting a wall that was difficult for them to circumvent. P2 said that "she [thought] she never figured out how to use `Search`[3] very effectively," so she only knew the tactics covered in class. P5 reported that while he "[knew] conceptually what [he] [wanted] to do," he "would have a [proof] hit a wall and then like, not know what [tactic] to hit [the proof] with." At this point, he would "[turn] to the documentation, and [continue] to struggle." Responses from our participants reinforced Use as a substantial learning barrier described by Ko et al.

External Resources. Given how lacking the official documentation is, many of the participants we interviewed used non-official Coq resources such as lecture notes and code, community notes developed by classmates, and posts on Stack Overflow.

Even though some of these resources were helpful, participants still reported having problems finding the right tactic. P5 recalled that "[he] would hit a wall, at which point [he] would really just scroll through the list of tactics that [the course instructors] gave." He recalled that the list was from a "Cornell website." P5 is likely referring to a Coq

[3]In Coq, `Search` is a command that allows programmers to look for lemmas. It is difficult to use however, and often pulls up hundreds of results for a given query.

tactics cheatsheet that was published by a course at Cornell University[4], which includes much more detailed examples than the official documentation. P5 really appreciated this external resource, calling it "pretty clutch." However, this cheatsheet only includes examples and usages for 27 of the *hundreds* of tactics in Coq. P5 remembered one time where he needed one tactic in particular, and that tactic was not listed on the cheatsheet. He said it "just did not emerge from the search...I killed, like eight hours [trying to find it]." This again constitutes an example of the Use barrier in Coq.

4.3 Motivation

Our findings suggests that there are several motivating factors for people to learn Coq. While Ko et al. did not include motivation in their framework, we observed that motivation was an important factor in our participants' learning experiences. The logic here is straightforward: those who find learning Coq interesting and beneficial for their own field of study or work are more likely to put more time and effort into it despite of the high learning curve, which may counteract the negative effect of learning barriers they have encountered.

A major motivation emerged from our interviews with P4 and P6 was formalizing skepticism about programs during development. In industry, programs are rarely built from scratch. As a common practice, software developers usually build their programs on top of dependencies, such as other programs, libraries, and frameworks. Each program has its own assumptions and guarantees. It can be beneficial to be skeptical about whether your code violates the assumptions or inappropriately assumes facts about the guarantees of the code dependencies. This way of thinking can help increase software quality by formalizing program states, specifications, inputs and outputs, and guarantees. This tangential benefit motivates our participants to continue learning Coq to develop more holistic view about programs. P6 said in his interview:

> The question is, as we change those systems, are we violating the previous assumptions? That's where I think having that model available … this feature we're asking for, isn't really possible? or it wouldn't be wise to do that, because the assumption was, you know, x, and now you're trying to create this other path that shouldn't exist.

Another motivating factor to learn Coq for engineers is the overlapping between theoretical concepts in Coq and actual applications in their everyday tasks. Knowing that there are some real-world applications of Coq in their day job helps them conceptualize the knowledge, which helps them perform better and contribute more in their work. Then the rewarding results from their work continue to stimulate them to learn more about Coq. This positive feedback loop creates constant motivations for our participants to learn Coq. P4, who has over two decades of industry experience in software testing and now in distributed system, find that learning Coq helped him build more confidence when encountering problems by excluding a huge amount of incorrect direction. He illustrated the example by saying,

> Someone was telling me that they saw two primaries at once, and I was like, "I don't think you saw that. I think what you saw was a program didn't check whether it was primary or not, and then assumed it was primary and did an action that only a primary could do. And since there was no check on it, it seemed to succeed, but what you actually have is a program that's not checking its error conditions." And I was pretty *confident*[5] saying that, because I was like, if that were the case, this would be broken 99% of the time in production, that's not happening...I felt pretty confident.

[4]Cornell is known, at least in the PL research community, for having very good PL faculty.
[5]Emphasis ours.

On the other hand, when motivation was missing, participants found that they struggled more. P3 didn't find programming in Coq as compelling as programming in a conventional language, saying, "I guess for intro CS courses, it felt easier to relate to the code...to what we were actually trying to achieve." As an example, they said that with conventional programming, they might "want to make a game. Here are some ideas that I kind of have a better intuition for how to make a game." But with PL, P3 thought "that link...didn't seem as clear." This issue of Applicability continually came up as a factor for our participants on why they would not want to use Coq again in the future.

4.4 Background Knowledge

As a language that enables proofs of mathematical and algorithmic theorems, it is important that its programmers have some understanding of related knowledge. As shown in Table 1, all of our participants have had some experience taking undergraduate-level or even graduate-level classes that taught them about proof strategies in math and CS. However, some of them reported that they still found difficulties applying their prior knowledge when writing Coq programs.

Coq Programs and Math. When given a complex programming task to work on, some programmers do not begin working on it in code immediately. Instead, they may choose to start with high-level ideas of what they want their program to achieve, and further develop their ideas in pseudo-code before translating it into a specific language. Similarly, in Coq's case, many participants mentioned that, when given a programming task to solve in Coq, their first reaction was usually to think about how they would solve it mathematically or in plain English, and then tried to translate it into a Coq program.

However, some participants mentioned that this translation process was difficult for them, even though they have relatively strong math background. P6 noted:

> Maybe not even specific to Coq necessarily, just the idea of thinking about programming and programs, like in terms of... being something you can represent mathematically, is [in itself] already kind of a barrier.

This constitutes an instance of Ko et al.'s Design barrier, since the task of translating programs to mathematical language was very cognitively intensive. Some participants reported that Coq's tactic names are confusing. P3 mentioned that they noticed a mismatch between some of Coq's tactic names and the behaviors they can achieve mathematically: "I see some [tactics] that are commonly used that have names, [but] the names that they have do not match up with what I would call them in more...pure math terms." This is a problem related to Ko et al.'s learning barrier Use, since the tactic name is obscuring the effect of the tactic.

Coq Programs and PL Concepts. The Coq source code is written in OCaml, a popular functional programming language in the ML (which stands for *metalanguage*) language family, which includes support for typical functional programming language constructs such as pattern matching in addition to having systems programming modules. Several participants noted that this made it especially easy for them to understand the syntax and concepts, such as P5, who said "it was fairly legible because it's basically OCaml." He added that he noticed "a lot...of the [syntax] from ML family languages were like, kind of around," such as "arguments by spaces" and arguments being "curried," which refers to partial evaluation of a function on just a few of its required parameters[6].

Almost all of our participants had some level of familiarity with functional programming, which may have helped them adapt to using Coq. Modern object-oriented programming languages have incorporated many concepts from

[6]Currying is named for Haskell Curry. More information on currying can be found here: https://en.wikipedia.org/wiki/Currying. Outside of the ML family and Haskell, there is not much native support for in other programming languages.

functional programming, such as in the case of Java adopting advances made by the Scala development team. P6 mentions his recent experience with functional programming using C#[7]:

> .NET basically [takes] all the really nice features from F#, like the function. The whole functional language they've added them to C# ... this is really nice, you know, all the type all the different types of like pattern matching and things like that

Having some background with functional programming seemed to help participants approach concepts in Coq.

We also observed that when participants had advanced experience in PL theory, they were able to develop a deeper understanding of Coq. P5, who is a PL researcher, observed that

> applied to prove goals, and you can like, prove sub goals kind of using them? And lemmas felt a lot like functions? Because it's like, oh, well, I have this fact that I want to prove. Right? And then I prove it. And then I can use that fact, because I've had this little lemma theorem and I can just hit stuff with that [theorem]. (lemmas are)

Interestingly, P5 is exactly right here — lemmas and theorems in Coq are literally represented as functions internally. P5 went further to reference the Curry-Howard Isomorphism, which is usually summarized as stating that "programs are proofs, and proofs are programs." In Coq, this is realized as "functions are proofs, and proofs are functions." These connections give P5 a deeper understanding of Coq that might be more difficult for students without a background in PL to grasp. This suggests that having programming language theory knowledge is not only helpful for absorbing the material in the class, but also for understanding Coq itself.

However, this may provide an even greater barriers to learning for students who are not well-versed in programming languages theory. Indeed, P3 described how they struggled with learning both the PL theory concepts *and* Coq at the same time. They said that "[their] impression of Coq...was like, 'this is a language that has been written with the intention of helping write proofs in general."' They found it difficult to translate the lectures about PL theory to the "syntax of exactly what they're writing out in the code." The lack of a background in programming languages theory may be a confounding factor in P3's case.

Coq Programs and Software Engineering. While Coq has extensive use in academia, it is rarely used in the industry. As two of our participants with extensive industry experience (P4, P6) mentioned, testing is intrinsically not fun for software engineers. P6 commented during his interview that

> for a lot of programmers, testing can be the least fun part. So they're not motivated to write normal tests that are very easy to use in their language...A lot of people also find it challenging because not everyone who's in industry writing code every day

Engineers instead would rather design some new features and write more code. Since verification can be seen as super-powered testing, it can put off software engineers. Moreover, human errors in programming cannot be easily avoided even with the use of static analysis and verification tools. Malfunctions due to human errors will still increase the financial costs, time used to fix the issues, and human labors. For engineering teams and the entire company, these extra costs bring no benefit even though the program is formally verified by an interactive theorem prover, thus further discouraging applying Coq to daily tasks and limiting Coq to a niche for a small group of people.

[7] C# is a general-purpose programming language developed by Microsoft.

Gatekeeping. Several participants made references to how they felt that the resources, error messages, lack of community, and background knowledge required excluded them or others, and made it more difficult to learn Coq. Such gatekeeping practices are common to computer science in general, such as GitHub installation instructions that presume that you know how to use certain command line tools, or simply the lack of documentation in general. However, given the small community and high learning curve required to learn Coq, this is exacerbated further.

In subsection 4.2, we describe how participants found the official documentation to be unapproachable, which can be a form of gatekeeping. P2 specifically pointed out that the official docs "use a lot of terms...from math theory or...that are just way beyond the level of...what a beginner user of Coq would know or understand." As an example, she mentioned the `lia` tactic, which stands for "linear integer arithmetic." P2 didn't think that the documentation was very good, since it said "something about like a ring[8]" that was at a "higher level of math theory than what I think a beginner Coq [user] would want." She added that "a better version of the doc would be like, you can use this to solve basic algebra...expressions." These issues create Selection and Use learning barriers, since it is difficult to find the right tactic to use (Selection, i.e., looking for "algebra" in the Coq tactic index, which does not bring up the `ring` tactic) and from the description, it is difficult to tell what the tactic actually does (Use).

P5, who is a PhD student in programming languages with extensive background knowledge and experience in PL, said there's "not enough of a community of people using [ITPs]," and that as a "part of the formal methods research community...you just sort of understand that [the lack of beginner-friendly documentation] is a part of the hazing that you have to do to be in the community." He did not think this was the right thing the PL community to encourage, and mentioned that "[he thought] people are kind of reluctant to ask for help...broadly in PL." This Environment barrier may prevent a broader range of people from learning to use Coq.

Even P7, who was one of our most knowledgeable participants in the area of PL theory, found the official documentation daunting with too many ways of stating the notation for tactics, and was often too abstract to be useful. He thought that descriptions of the essentials for Coq were lacking, even as an intermediate user himself. The authors have anecdotally noted that sometimes people assume that Coq is a tool that is just for people who know programming languages, but not even this much is true, as even the PL researchers in our study felt that they experienced difficulties when using Coq.

Participants even seemed cognizant of how these gatekeeping factors made them particularly well-suited to learning Coq. P5 qualified his observations about what he would like to see as improvements for Coq by stating "[he was] coming at [the potential improvements] from a biased perspective, because like, all of the math notation was fine for me. Like I could read things. Like I could read what the proof goal meant, and sort of understand all of that." In this statement, P5 is acknowledging that the problems that he saw were not necessarily problems that other students in the class, or perhaps users in general, would experience, since he was fine with the math notation used in Coq (see Figure 2 for an example of Coq's logic notation, which is similar to what is used in mathematics).

5 DISCUSSION

In section 4, we provided evidence for three additional learning barriers, in addition to Ko et al.'s original six, that we observed in our interviews with participants: Dead End, Applicability, and Environment. We will now discuss implications of each of these identified barriers, which may be particular to Coq and/or ITPs in general.

[8]Here, a *ring* refers to a mathematical object in abstract algebra: https://en.wikipedia.org/wiki/Ring_(mathematics).

```
88  Lemma and_example2' :
89    forall n m : nat,
90      n = 0 /\ m = 0 -> n + m = 0 .
91  Proof.
92    intros n m [Hn Hm].
93    rewrite Hn. rewrite Hm. reflexivity.
94  Qed.
```

Fig. 2. An example lemma in Coq containing ASCII-fied math symbols. Written in English (or at least the kind of English one finds in a math textbook), this would read: for all natural numbers n and m, if $n = 0$ and $m = 0$, then $n + m = 0$. One version using fairly standard mathematical and logical notation would look like: $\forall n, m \in \mathbb{N}, n = 0 \land m = 0 \Rightarrow n + m = 0$. If you are familiar with either reading math textbooks or the mathematical notation, you may be more comfortable with reading Coq lemmas.

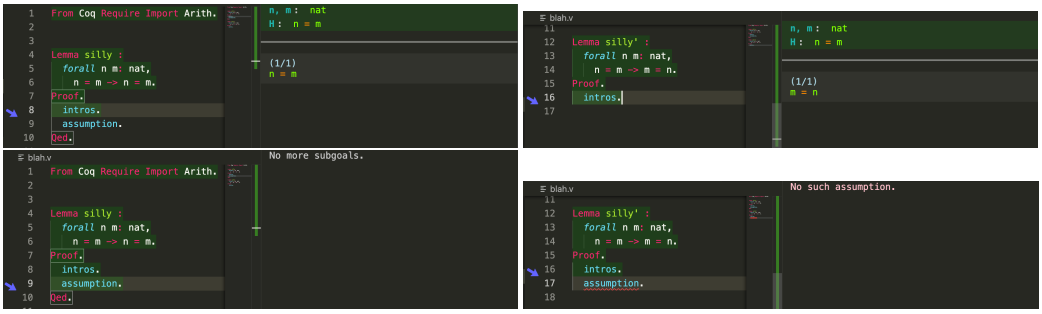

Fig. 3. A simple illustration of how proof states that look very similar to humans are not solvable by the same tactics. On the top left, we have a proof state where the we have a hypothesis `H` that states that $n = m$, and our goal is to show that $n = m$. The `assumption` tactic, whose effect is displayed in the bottom left screenshot, leverages the hypothesis `H` to provee the goal. The lemma `silly'` is very similar to `silly`, except that you are supposed to prove $m = n$, given that $n = m$. In the top right screenshot, the proof state looks very similar to the proof state in the top left screenshot. Since n and m are both natural numbers, a human would probably say that our proof goal, $m = n$, is obviously true, since we have $n = m$. It would seem reasonable, therefore, to try the `assumption` tactic in this case once more, but in the bottom right screenshot, the tactic fails with the error: "No such assumption." If the user does not know that the `assumption` tactic requires an exact, syntactic match, they may be frustrated by this error and not know how to proceed.

5.1 Dead End

Due to Coq's unique model of proof writing, as compared to traditional computer programming, users cannot plan out their proof scripts in advance. As the proof unfolds, users may uncover proof goals that they may not have anticipated that they would have to prove. Unlike conventional programming languages, in which it's important to implement an algorithm of some sort, with proof scripting, users are searching a space of possible proofs. Each proof state may be amenable to a number of different tactics, which simply makes the search space enormous. Over time, it is possible that Coq users may get a better intuition for which tactic to use in particular circumstances, which is something that the first author has observed in their own work with Coq. However, building up that intuition is a process that takes an

extended period of time, even despite the first author's extensive programming and proof writing experience[9]. Until intuition about the various tactics is at a high enough fidelity, users may be frustrated when their intuition fails, as illustrated by the simple example in Figure 3.

When a Coq user's human-based proof search algorithm fails, they run into the Dead End barrier. This problem is exacerbated by how difficult it is to search for the right tactic to use. As an example, consider the scenario in Figure 3, where the user is trying to prove the lemma `silly'`. They have already proved a similar lemma, `silly`, which went through after just two tactics. But the same two tactics don't work for `silly'`. What should the user look up? Perhaps the user could describe what they want to do as "flipping" the equality in either the proof goal or the hypothesis, and then they could use `assumption`. However, a search in the Coq tactic index for "flip" or "flipping" doesn't reveal any promising results.

One way to fix this problem would be to provide a way to search for tactics based on how you would like to transform your proof goal. Our preliminary findings suggest that participants would find this useful, since several participants explicitly asked for a way to do such a search. While other programming languages do not need this sort of search tool, they also benefit from hundreds of thousands of StackOverflow questions, blog posts, and other learning resources that may come up in a Google search. Due to the small nature of the Coq community, this sort of tool would help to make the most of the available documentation.

5.2 Environment

Unlike many other programming languages with millions of users, Coq's specificity makes it very difficult to have a large userbase. As a result, there does not exist a large, stable Coq community where Coq learners can communicate their experiences with each other or get help from Coq experts, either online or offline. Instead, when someone starts learning Coq, not only would they be intimidated by Coq's high learning curve, but also they would usually feel unwelcome in Coq's user community. They may get such impression from their first attempt of trying to read Coq's official documentation, seeing a ton of mathematical terminology, and realize that the official documentation is not written *for them*. Or, when their search of a specific Coq error message brings them to a StackOverflow post that has gone unanswered for months, they may also be discouraged by the thought that nobody would be willing to help them. Over time, the lack of support for Coq newcomers shapes the idea that Coq is a tool for a very small set of experts, and the path for learners to become experts is unknown and undocumented.

Therefore, we hope to advocate for change in the Coq community, including its development team and users, to put effort into making the Coq community a more newcomer-friendly place. Of course, we admit that such change cannot be achieved in short time, or without the collective effort of Coq instructors, experts, and beginners.

- For Coq's development team, one goal should be to make the official documentation more readable for non-experts. As in any other programming language, the official documentation is often the number one go-to resource for learners. Therefore, we should make sure that Coq's new learners are welcomed by easily understandable language and numerous examples when they seek help from the official documentation, instead of being overwhelmed by jargon that only very few people can understand.
- For Coq instructors, try to create a more collaborative atmosphere for your students. New learners are easily scared away if they perceive Coq community as unapproachable. Therefore, we can build small Coq communities within student body, and encourage them to help each other out. Hopefully by the end, students would feel

[9]Relative to the average computer science bachelor's degree holder

comfortable enough with learning Coq with others, and bring this positive atmosphere to the broader Coq community.

- For Coq users, start talking to other Coq users if possible! Many new Coq users, at some point, have had the feeling that they were the only person who could not understand Coq. However, as soon as you start exchanging conversation with other Coq learners, you will realize that you are not alone. By learning from each other, you will find the learning experience much more enjoyable.

5.3 Applicability

Coq sits at a particularly interesting intersection between coding and proving. Coq programs consist of two major components: 1) a specification of the system to be proved and 2) proof scripts that show that desired properties hold for the system. Such a system could be an operating system, a compiler, an algorithm, a data structure, etc. Constructing a valid proof requires a solid amount of background in mathematics and programming languages, making Coq a tool with considerably high threshold. The high threshold reflects that Coq is not relevant or useful to the majority of people in their everyday lives, which drastically limits the suitable applications to ones where not verifying the software would endanger lives or cost billions of dollars. Similar to other mathematical proofs, the actual applications of the program proofs will not shine out instantly. Although the Coq program can be used to demonstrate strict mathematical correctness, its direct impacts are often considered subsidiary. People care more about the effects and values brought by the program that Coq verifies than Coq itself.

In addition, unlike other general-purpose programming languages that have specific areas of applications or tangible results, such as JavaScript for creating web pages, C to interact with low-level systems, or Java to write applications that can be run on any system that has a JVM, programs written in Coq are very hard to materialize. Results from Coq are opaque, abstract, and intangible. This abstractness and low applicability makes it less motivating and interesting for people to try out and map the theoretical concepts to practical usages. Many concerning practical questions remain unresolved even if the program has been proved safe by Coq. The learning process becomes dry and tedious. The Applicability barrier, as a result, leads to a infinite inapplicability cycle.

While it requires a tremendous amount of work to link Coq with practical applications, it is still educationally beneficial to demonstrate tangible examples where learners can find relatable. Our initial findings suggest that applicability will be improved after the other barriers are resolved. By making Coq more usable and approachable and the surrounding communities more welcoming and supportive, more people and industries will use Coq to verify their software. Then, a larger number of use cases and applications will arise.

6 FUTURE WORK

Because of time constraints, we were not able to include as many participants as we would have liked in our interview study. While we found more interesting themes in our interviews with the seven participants in this study than could even fit within the scope of this class project, we would have preferred to have more subjects. In addition, we would ideally follow up our interview study with a think-aloud study, observing participants as they use an ITP. However, given the limits of this course project, we decided to focus on just the interview study at this point, and we leave a think-aloud study for future work. Additionally, we acknowledge that not all of our participants may have wanted to learn how to use an ITP in the first place (i.e., since they may have taken CSE 505 for reasons other than to learn Coq), which affects our results.

Furthermore, in the interest of having a focused contribution in this paper, the we were unable to include a thorough analysis of two of our identified themes: Pair Programming and Language Deficiencies. These themes involve participants' feelings about the pair programming model in CSE 505, which was often critical to participants' experiences with learning Coq and how participants felt in the course, and the peculiarities of Coq's syntax, REPL interaction model, and the two languages involved, Gallina and Ltac. We hope to flesh out these ideas further in a full paper. Another possible future direction would be to explore how community and social interaction in the course, such as through the instructors, the CSE 505 community notes, or an assigned homework partner, affects students' feelings about gatekeeping in PL or how much they belong in PL research.

In the future, we might also consider exploring the suggestions from our participants that could possibly lower the threshold for learning Coq. For example, P4 and P6 both suggest that having a modern and interactive programming environment would dramatically enhance the coding and learning experience[10]. P6 praised the interactive programming environment (REPL) used for Coq: " I was using VS code with that VS code extension, [where it] just kind of like, proves as you go along. That was actually really nice and really helpful." Many participants (P1, P2, P3) commented that they could not see any practical use cases of Coq in their professional/academic career, which sometimes makes them hard to grasp some concepts in Coq. Meanwhile, P4 and P6, as professional software engineers, consider mappings from conceptual ideas in Coq to their applications in their day job a significant contributing factor for their Coq learning process. P4 said that "[he] was somewhat inspired by how much richness there is in that aspect of the industry.". Some other valuable suggestions from our participants include making the official documentation more discoverable, improving Coq's community engagement, and making information about Coq more beginner friendly. But due to the time constraints in this course, we were unable to investigate these suggestions further, or follow-up with participants to flesh out these ideas.

To answer the entire question of how people learn to use ITPs, it would also be beneficial to interview people who use ITPs professionally about their learning experiences. This, however, would likely require several months of planning and scheduling, and thus could not be completed in the scope of this course project.

Finally, in order to rule out the effects of programming knowledge and other factors, we only recruited participants who took CSE 505 at the University of Washington and had a decent amount of programming experience, i.e., PhD or masters students or senior undergraduate students in computer science. Therefore, we were unable to produce any generalizable results and do not have a full picture of how people in general learn to use ITPs, but given that this study is the first of its kind, this is already a step forward, and we hope to build upon it in the future.

7 CONCLUSION

In this paper, we have explored how several people who took a programming languages course learned to use Coq, an interactive theorem prover, as a part of that course. After conducting interviews with the participants and analyzing their data, we found that preliminary evidence for applying Ko et al.'s six learning barriers to Coq as well as initial suggestions for three new learning barriers, (Dead End, Applicability, and Environment), that may be more specific to Coq or interactive theorem provers in general. These findings will help shape future research into how people learn to use interactive theorem provers, which we hope to explore further in future studies.

[10]While Emacs has the most full-powered coding experience for Coq, and is certainly widely used at least in the programming languages community, Emacs is also known for its own steep learning curve.

8 ACKNOWLEDGMENTS

We would first like to thank Prof. James Fogarty, the TA Anant Mittal, and the rest of our classmates in CSE 510 for their valuable input and suggestions that helped to shape this research. Second, we would like to thank Prof. Molly Feldman of Oberlin College, who has been collaborating with the first author on this project and gave vital feedback on our interview questions, as well as Prof. Zach Tatlock and James Wilcox, for sending along the call for participants to former students of CSE 505 and answering our emails within one business day. We would also like to thank Claire Seo, a PhD student in psychology at George Washington University, and Eunice Jun, a PhD candidate at the University of Washington, as well as our pilot participants for their amazing suggestions that helped to improve our screening questionnaire. Lastly, the first author in particular would like to thank Yadi Wang and Zhennan Zhou for being willing to learn about this incredibly niche research area and bringing in new insights from their own fields of interest.

REFERENCES

[1] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. 2017. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science*, 239, (January 2017), 15–27. arXiv: 1701.07125. ISSN: 2075-2180. DOI: 10.4204/EPTCS.239.2. Retrieved 03/16/2022 from http://arxiv.org/abs/1701.07125.

[2] David Aspinall. 2000. Proof general: A generic tool for proof development. In *Tools and algorithms for the construction and analysis of systems*. Susanne Graf and Michael Schwartzbach, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 38–43. ISBN: 978-3-540-46419-8.

[3] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Communications of the ACM*, 64, 8, 98–106. Publisher: ACM New York, NY, USA.

[4] Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on ML*, 37–46. https://dl.acm.org/doi/abs/10.1145/1292535.1292541?casa_token=WsA29F8dXbYAAAAA:7omVG3pTFKWdtrztw7MRoSK10nJFyWLDNZNSkWrkS8FiLBVPbC9ZiOvBzmQWwvvnxLlQM--6hblA.

[5] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. 1979. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22, 5, (May 1979), 271–280. Number of pages: 10 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.issue_date: May 1979. ISSN: 0001-0782. DOI: 10.1145/359104.359106. https://doi.org/10.1145/359104.359106.

[6] Georges Gonthier. 2008. Formal proof–the four-color theorem. *Notices of the AMS*, 55, 11, 1382–1393.

[7] Isabelle Development Team (last). [n. d.] Isabelle. (). Retrieved 03/07/2022 from https://isabelle.in.tum.de/.

[8] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS2 2018-9th European Congress Embedded Real-Time Software and Systems*, 1–9.

[9] Amy Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[10] LF Projects. [n. d.] seL4. (). Retrieved 03/07/2022 from http://sel4.systems/.

[11] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*. André Platzer and Geoff Sutcliffe, editors. Springer International Publishing, Cham, 625–635. ISBN: 978-3-030-79876-5.

[12] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How amazon web services uses formal methods. *Communications of The Acm*, 58, 4, (March 2015), 66–73. Number of pages: 8 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.issue_date: April 2015. ISSN: 0001-0782. DOI: 10.1145/2699417. https://doi.org/10.1145/2699417.

[13] Otter.ai. [n. d.] Otter.ai - Voice Meeting Notes & Real-time Transcription. en. (). Retrieved 03/16/2022 from https://otter.ai/.

[14] Clément Pit-Claudel and Pierre Courtieu. 2016. Company-Coq: Taking Proof General one step closer to a real IDE. en_US. *Pit-Claudel*, (January 2016). Accepted: 2021-09-23T17:56:26Z. Retrieved 03/16/2022 from https://dspace.mit.edu/handle/1721.1/101149.2.

[15] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL instrumentation for Coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 99–113.

[16] Hanneli C.A. Tavante. 2021. Towards an Incremental Dataset of Proofs. In Association for Computing Machinery, Chicago, Illinois, (October 2021). https://hannelita.com/download/submission-hatra21.pdf.

[17] The Agda development team. (last). [n. d.] The Agda Wiki. (). Retrieved 03/07/2022 from https://wiki.portal.chalmers.se/agda/pmwiki.php.

[18] The Coq development team. [n. d.] The Coq Proof Assistant. (). Retrieved 03/16/2022 from https://coq.inria.fr/.

[19] The Miro development team. [n. d.] The Visual Collaboration Platform for Every Team. en. (). Retrieved 03/16/2022 from https://miro.com/.

[20] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '15). Association for Computing Machinery, New York, NY, USA, (June 2015), 357–368. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737958. Retrieved 03/07/2022 from https://doi.org/10.1145/2737924.2737958.

[21] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (CPP 2016). Association for Computing Machinery, New York, NY, USA, (January 2016), 154–165. ISBN: 978-1-4503-4127-1. DOI: 10.1145/2854065.2854081. Retrieved 03/07/2022 from https://doi.org/10.1145/2854065.2854081.

[22] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *Proceedings of the 36th international conference on machine learning* (Proceedings of machine learning research). Kamalika Chaudhuri and Ruslan Salakhutdinov, editors. Volume 97. tex.pdf: http://proceedings.mlr.press/v97/yang19a/yang19a.pdf. PMLR, (June 2019), 6984–6994. https://proceedings.mlr.press/v97/yang19a.html.