Learner-Centered Design Criteria for Classroom Proof Assistants

MATTHEW KEENAN, University of Michigan, USA
CYRUS OMAR, University of Michigan, USA

We survey publications that report on the experience of introducing proof assistants into classrooms, identifying observed advantages and challenges. From this, we synthesize a series of design criteria for classroom proof assistants, and suggest methods for evaluating future designs against these criteria. Finally we discuss how we are using these criteria to guide design decisions in ongoing work on designing the Hazel Prover.

1 INTRODUCTION

Proofs are integral to all areas of math, science, and technology. Depending on the topic of study, these proofs may take the form of equational reasoning, logical derivations, geometric derivations, or program correctness proofs. Students are most commonly taught to write proofs by hand, with little to no direct feedback or mechanical assistance during the proof writing process. It is only much later, during grading, that course staff laboriously check the student's proof steps. This leads to delayed, erroneous, and inconsistent feedback, and ultimately limits learning.

In contrast, students solving programming problems enjoy access to educational programming environments and autograders that provide live feedback and assistance throughout the problem solving process, resulting in a tighter feedback loop and minimizing manual human effort, errors, and inconstencies.

Is it possible to bring this sort of automated feedback and assistance to proof problems? Over the past several decades, the formal methods community has developed proof assistants that provide feedback and assistance to experts as they write proofs. Several authors, given in Table 1, have tried using these proof assistants in the classroom. Others, concerned with the steep learning curve of proof assistants designed for experts, have designed new proof assistants specifically for classroom use. Some of these interfaces are given in Table 2.

Theorem Prover [9]	**Examples**
Coq [9]	Henz and Hobor [19], Knobelsdorf et al. [26], Gallego Arias et al. [13], Hendriks et al. [18], Delahaye et al. [11], Pierce [34]
Lean [10]	Avigad [3], Thoma and Iannone [42]
Isabelle [31]	Jacobsen and Villadsen [20], Nipkow [30]

Table 1. Experiences using full-scale proof assistants with students.

Layout	**Examples**
Tactic-based	Wemmenhove et al. [43] Bentkamp et al. [4] Karsten et al. [22]
Close to pen and paper	Rognier and Duhamel [36] Billingsley and Robinson [5] O'Connor and Amjad [32] Lodder et al. [28]
Proof trees	Staudacher et al. [40] Cerna et al. [8] Geck et al. [15] Gasquet et al. [14] Machın and Sierra [29] O'Connor and Amjad [32]

Table 2. A selection of proof assistants designed for education.

Authors' addresses: Matthew Keenan, University of Michigan, Ann Arbor, USA; Cyrus Omar, University of Michigan, Ann Arbor, USA.

Many of these publications have reported positive experiences for teachers and students, and found specific advantages that we will detail below. We need to be cautious with positive reports, however, as we are likely to see a bias in the literature towards success stories. Many of these reports also detail the challenges they faced introducing these tools to the classroom.

We are not yet aware of any efforts to introduce Large Language Model technology into proof classrooms, so we will not discuss their use, but we recognise that there is much future work that could be done with LLMs, especially with regards to feedback.

In the next section we will survey the papers in Table 1 and those in Table 2 with classroom deployments, and later we report on our ongoing work on the Hazel Prover, which is a theorem prover integrated into the Hazel live functional programming environment.

2 DESIGN CRITERIA AND RELATED WORK

In this section, we synthesize a set of design criteria for classroom proof assistants. We do not aim to be comprehensive in our survey of classroom proof assistants. Instead we focus on papers with substantial and interesting experience reports or experiments; most of these are recent papers. We intersperse these criteria with qualitative and quantitative research methods that may be used to evaluate candidate educational interventions against each design criteria. Again, we do not aim to be comprehensive or fully detailed here, but rather to encourage designers of these tools to think about how they could evaluate their claims.

2.1 Immediate and Comprehensive Feedback

2.1.1 Advantages of Computer Feedback. As students write or typeset proofs manually, they receive little to no immediate feedback or mechanical assistance [1]. Instead, their tentative proof is only checked days or weeks later, at great institutional expense, by course staff. The feedback the student ultimately receives after this laborious grading process can sometimes be vague, erroneous, or inconsistent between students. This time spent grading work can also take away time from more valuable formats of instructor feedback, such as office hours. Students report confusion and a sense that these proof-heavy courses "did little to aid students" [38, 41]. This has been observed to contribute to the disproportionate withdrawal of under-represented students [37].

In stark contrast, students solving programming problems nearly always have access to (1) programming environments that continuously report syntax and type errors and provide helpful completions, and (2) autograders that compare actual and intended program behavior. This tight feedback loop allows students to iterate rapidly to refine their mental models and quickly correct their own misconceptions [24, 25]. This has motivated the development of classroom proof assistants too. Students using SaSyLF [2] "felt they benefited from earlier feedback from the tool".

2.1.2 Design Considerations for Good Feedback. In order to make use of this feedback, it is important it is interpretable and helpful. The feedback given by full-scale proof assistants is often not tailored to students learning proof for the first time. Avigad [3] noted that Lean's "error messages were not sufficient for [students] to figure out what they were doing". Likewise Aldrich et al. [2] reported that their tool's messages "indicate that there is a problem but are not always able to point to why the problem exists."

Design Criterion 1 (Explanation of errors). *Classroom proof assistants should provide error messages that make it clear what specific mistake a student has made, and offer guidance for how to correct it.*

The Diproche system [7] checks students' proofs with an automated theorem prover, and when the reasoning is incorrect, tries adding certain incorrect axioms representing typical mistakes, and if successful tells the user which mistake they likely made.

Experiment 1.1 (Log Analysis). *To test which errors students receive, and whether students are able to resolve them we can log the errors students receive while using a proof assistant. We can log the number and types of errors encountered by students, as well as recording the time it takes for students to resolve these errors.*

Experiment 1.2 (A/B Testing). *We can measure the usefulness of errors by A/B testing. We can give some users full feedback and others reduced feedback [27]. By comparing these two groups we can see if there are advantages of full feedback.*

As well as reporting errors in a proof script, it is also helpful if a checker can recover from errors and continue checking the rest of the proof. This allows the checker to mark parts of the proof as correct, and find other errors in the proof script before the first ones are resolved, allowing an autograder to award partial credit.

This recovery from errors also makes feedback more useful for students. One student using SaSyLF [2] said they wanted the tool to provide assurance by marking judgments that were correct. This assurance would both make it easier to find errors by ruling out correct parts of the proof, and help contribute to the sense of satisfaction in section 2.3.2.

Design Criterion 2 (Error Tolerance). *The feedback from a classroom proof assistant should, where possible, be able to recover from an error in one part of the proof and mark the rest of the proof as either correct, or containing more errors.*

2.2 Scaffolding Building Proofs

In pen-and-paper proof, new students can often be intimidated by seeing a blank sheet of paper, unsure how to begin. An interactive tool can help students see what next steps are available to them and how to use them at any given stage in a proof. Seeing the available options can provide inspiration, allow students to try out the options to see what they do, and can rule out some nonsense steps [30] students might otherwise attempt. We refer to these tools that allow students to write proofs that they wouldn't be able to produce without the tool as *supports* or *scaffolds*.

Design Criterion 3 (Supporting Exploration). *Learners should be able to use the tool without already knowing all the details of the proof they want to produce. The tool should help give them ideas for next steps, and explain what each of the steps the user could take would do.*

Experiment 3.1 (Homework Analysis). *We can record what proportion of questions are attempted by students when using the tool, and compare this to the attempts from written exams the year before the tool was introduced.*

Scaffolds [17] are distinct from supports in that students must not become dependent a the scaffold, whereas when using a support, a user is never expected to learn how to perform the same task without a support.

Whether we want a proof assistant to serve as a scaffold or a support depends on the learning objectives of the course. Kerjean et al. [23] ask whether the purpose of using Coq in education is to teach Coq itself, or to teach mathematics using Coq. In the first case Coq is used as a *support*, and students should learn how to use all the automation features of Coq and follow all the coding standards expected in the Coq community. In the second case, Coq is used as a *scaffold* and you need to make sure that students can reproduce everything they do on pen and paper. This means

the tool should make sure users are familiar with the kinds of notation they would be expected to use on paper, and the proof layout they would be expected to produce.

In this report we are going to mostly assume the second, or more specifically that we are trying to teach proof skills using proof assistants, and that the classroom proof assistant must therefore be able to fade away. In practice there may be certain domains of computer science and math where students' use of industry-standard proof assistants is desired.

Results from past work are clear that proof assistant tools are able to act as a support. For example, Kerjean et al. [23] conclude that students using a proof assistant were more likely to search for proofs instead of giving up.

Previous work has a more mixed experience of how dependent students become on the proof assistant. Certain authors have reported transfer to pen-and-paper [30] [42]. Other authors have found it hard to transfer student's skills to pen-and-paper [26] [6].

A first step to transfer to pen-and-paper is to ensure that students engaging with the proof assistant are actually engaging in proof. Guillot and Narboux [23] found 47% of respondents who used Edukera in their introductory proof course reported that they were able to complete several exercises in Edukera without understanding anything, by clicking randomly. Henz and Hobor [19] found that students who didn't have a clear model of what Coq tactics were doing would "try tactics at random until for some unknown reason they hit on the right combination." They remedied this by providing their students with diagrams explaining exactly what the tactic does.

Design Criterion 4 (Require Engagement). *It should not be possible to complete exercises by automation or random clicking alone, the assistant must force students to learn in order to proceed. This can be done either by making it easier to learn necessary concepts, or making it harder to proceed.*

Experiment 4.1 (Survey). *To determine whether students are able to complete tasks randomly, we can survey students and ask whether they were able to complete exercises randomly. It may be worth also creating a baseline by asking students whether they were also able to complete written exercises by guessing.*

Experiment 4.2 (Interview). *To determine whether users understand the proof tactics they have used, we can allow users to write a proof using the tool, and then ask them to explain the tactics they have used.*

In order to facilitate transfer to pen-and-paper proofs, we should consider whether the proofs done on the computer are similar to those written on pen-and-paper. After introducing Coq to their classroom, Knobelsdorf et al. [26] observed that "students constantly performed better when using Coq in comparison to pen & paper". They rule out the possibility that students had just been solving Coq problems randomly because of the complexity of some of their exercises. They conclude instead that the skills learned in proving with Coq didn't naturally transfer to pen-and-paper because Coq proofs are so different to handwritten proofs.

Design Criterion 5 (Notation similar to written proofs). *Proofs in the tool should be laid out similarly to how students would be expected to produce proofs without the tool to make it easier to transfer proof competency.*

Böhne and Kreitz [6] build on Knobselsdorf's work, but argue that specializing proof assistant syntax to their course would take too much time. They therefore attempt a generic method for transferring competency from proof assistants to paper proofs, no matter how dissimilar the notation.

There have been many other attempts at making the proof assistant's notation more closely match pen-and-paper proofs. These vary widely, since their interface depends on the kind of proof

teachers want students to produce. For example, several interfaces have been created for producing tree-style proofs (examples in Table 2).

Experiment 5.1 (Assessment). *We can check that proof competency has actually transferred to pen-and-paper by seeing whether students can reproduce proofs they have created in the tool on pen-and-paper.*

Guillot and Narboux [36], observed that some students, after completing all their exercises on the Edukera platform, would struggle to reproduce their answers on paper because they could remember the structure of the proof but not the definitions they used. This suggests that providing definitions for the user to click on is an example of a support, but not a scaffold as users become reliant on it. Guillot and Narboux solved the definition problem by interleaving on-tool proof exercises with off-tool definition recall exercises. We hypothesize that it should also be possible to design a tool such that students are required to practice producing definitions as part of the proof process.

Design Criterion 6 (Teaches new definitions). *If students will need to memorize new definitions and rules to use on paper later, the tool should make users practice providing these definitions.*

2.3 Ease of Interaction

Full-scale proof assistants have historically targeted users with a graduate-level education and significant programming experience. Especially in lower-level STEM courses, asking students to write proofs directly as, e.g., dependently typed programs in Lean or using Coq's tactics is untenable: discrete math courses often do not have programming prerequisites nor enough time to teach the necessary programming skills.

Design Criterion 7 (Simple learning curve). *Classroom proof assistants must be approachable to students (and to instructors in many cases) with no programming background after minimal training—perhaps no more than a portion of a 50-minute lab or discussion section.*

To minimize training, we require an input method that students with no prior experience can get comfortable with quickly. Avigad [3] anecdotally reports that "Lean's syntax takes getting used to" when used in their introductory logic course. Knobelsdorf et al. [26] perform an analysis of the questions asked by students, noting that the "working with Coq" category of questions was largely limited to the first half of the course, with questions in the latter half being more content-related. The authors conclude that "students required surprisingly little training to get used to work with the theorem prover", but it should be noted that the course was a two-week elective full-time course, with only 21 participants, which may have selected students with a particular interest in Coq.

Experiment 7.1. *We can measure the time taken to get familiar with the tool by recording, over time, the proportion of questions asked to instructors/teaching assistants that are about the tool, compared to questions about the content of the course, as done by Knobelsdorf et al. [26].*

2.3.1 Distractions. Along with the steep learning curve, another problem with using full-featured proof assistants in the classroom is their distracting complexity. Henz and Hobor [19] remark that "It is amazing how easily one runs into all kinds of didactically-inconvenient topics at awkward moments" when using Coq, noting that they have to teach concepts they would otherwise elide from their course so that students don't get stuck. There are many steps that a proof assistant may require which is not usually required on paper, and designers should carefully consider how to hide these steps.

Design Criterion 8 (Customized elision). *Classroom proof assistants must not obligate proof steps that would be elided on paper. Problem designers must be able to customize elisions.*

2.3.2 Satisfaction. One well-attested advantage of using proof assistants in the classroom is that students enjoy it. One of Avigad [3]'s students "worked on unassigned problems in Lean in order to procrastinate writing English essays". Henz and Hobor [19] saw that after introducing a proof assistant, their course had a small increase in sudent opinion of the course, despite students also rating it as harder. It is important not to lose this sense of fun in designing custom tools for the classroom.

2.3.3 Robustness. Finally, it should go without saying that tools used by students should be robust and not prone to errors. Greenberg and Osborn [16] name the IDE's tendency to crash and mangle Unicode characters to be a significant problem in their adoption of Coq in the classroom. Perhaps more importantly for this application though, a bug in a proof checker could disrupt students' learning of a mental model for how proofs are checked. Billingsley and Robinson [5] noticed that their proof checker had a bug that would falsely mark some proofs as wrong, but "students could not tell that this was due to a bug and assumed their proofs were wrong".

Design Criterion 9 (Robustness). *Classroom proof assistants should minimize distractions caused by bugs, and in particular ensure that the computational model is correct to prevent students learning an incorrect computational model.*

Experiment 9.1. *We can ask the user to rate their agreement with the statement "bugs in the proof tool hindered my ability to complete assignments" to check the robustness of the tool.*

2.4 Context of use

Proof assistants designed for research can be difficult to set up, especially in a classroom environment. Historically, proof assistants user interfaces are often emacs-based, and require tricky combinations of keyboard shortcuts. Teachers wanting to teach math courses do not want to have to teach their students emacs, especially if the course is not part of a computer science course.

This setup barrier hampered early attempts [35] to introduced proof assistants into the classroom, but now Proofweb [21], JsCoq [13], ProofBuddy [22], and the Natural Number Game [4], among others, all provide web interfaces for the prover they are based off of to make installation and setup easier.

Design Criterion 10 (Easy setup). *Installation and setup of the tool should be as straightforward as possible, so that students and instructors' time are not wasted on setup. A web interface which requires no installation is ideal.*

While these tools provide the ability for students to work independently, it is important too that they provide support for teachers using these tools in their class. This includes the ability for teachers to set their own exercises, track students' progress, and present ideas using the tool.

Design Criterion 11 (Classroom integration). *To simplify the job of course staff needing to record grades for proof problems, a classroom proof assistant should provide easy-to-use automatic graders that can integrate with software used to run classes and distribute grades.*

3 LEARNING OBJECTIVES AND SCAFFOLDING VS SUPPORT

As well as ensuring that a classroom proof assistant meets our design criteria, we also need to make sure that our proof assistant's design is appropriate to its learning objectives. In particular, whether a skill is scaffolded or supported (section 2.2) will depend on the learning objectives for that skill.

There are three broad categories of skill:

(1) The learner does not have this skill, and does not need to learn the skill. This skill may be an idiosyncrasy of computer proof that you wouldn't otherwise teach students, or it may be inconvenient to introduce too early in the course.
These skills should be *supported* with automated and elided steps. If this skill were scaffolded, it could lead to distractions (section 2.3.1), or it could create a learning curve that is too steep (criterion 7).
(2) This skill is one of the learning objectives of the course. The student doesn't have the skill at the start of the course, so will need help, but the student should be able to use the skill without help by the end of the course.
These skills should be *scaffolded*. If one of this skills is not transferring to pen-and-paper **??**, it is an indication that this skill is supported, not scaffolded.
(3) The learner already has this skill, and does not need to practice using it. It may be a tedious skill, or it could distract the user from skills in category (2) that they are trying to learn.
These skills should be also be *supported*.

When designing a classroom proof assistant, therefore, it is important to first consider carefully what the learning objectives of the assistant are. These objectives may change as the course continues, for example, an instructor may want to initially elide some skill as in (1), then introduce it to the students as in (2), and once the students are comfortable with it, provide automation as in (3) so that they do not waste further time on the skill.

Once we have the learning objectives, it becomes clearer which tools should be scaffolded, and which should be supported. Scaffolding can be achieved by requiring engagement (criterion 4), using notation close to the notation expected to be used without the tool (criterion 5), and ensuring users are learning all the content required to produce the proof (criterion 6). Support can be achieved by automating and eliding (criterion 8) proof steps that the user does not need to learn.

We will demonstrate this process by considering the design of our work-in-progress classroom proof assistant, *the Hazel Prover*.

4 THE HAZEL PROVER

4.1 Context for the Hazel Prover

We are working on a classroom proof assistant for Discrete Mathematics (EECS 203) at the University of Michigan. EECS 203 is a lower-level undergraduate computer science course required for students wishing to major in computer science that covers mathematical foundations of computer science. This is paired with an introduction to elementary proof techniques including mathematical and structural induction.

We plan to begin by focusing on the part of the course that teaches functions and recurrences (defined equationally) on numbers, sequences, trees, and graphs. We want students taking our course to become comfortable with recursive definitions and inductive reasoning about properties of recursive definitions.

Our proof assistant is built on Hazel [33], a web-based live functional programming environment that has been deployed into the classroom at the Universtiy of Michigan. The Hazel Editor introduces *gradual structure editing*, an approach to keyboard-driven structure editing that eliminates syntax errors entirely, instead automatically correcting the entered syntax by inserting holes and tracking delimiter matching obligations in a visual "backpack", while otherwise supporting standard keyboard-driven editing affordances. Hazel programs can either be evaluated using the live evaluator or using a single-step evaluator (similar to the evaluator in DrRacket [12]), in which users can view an evaluation trace.

We are now looking to extend this single-step evaluator with symbolic reasoning so that it can be used to prove that two expressions always evaluate to the same value, even when these expressions are quantified over variables. Some of this reasoning will involve proof by induction, either over natural numbers, over lists, or over Hazel's custom recursive data types.

4.2 Induction

In the Hazel Prover, we want to allow students to produce proofs by induction over natural numbers, lists, or Hazel's abstract data types. Students learning how to reason inductively is an explicit learning goal of the course and thus induction fits into category (2) from section 3. We therefore want to *scaffold* induction.

4.2.1 Supporting Induction. In a typical tactic-based proof language, by starting an induction over some value, each of the cases are automatically written out as goals, with induction hypotheses also generated automatically. This can save a lot of time writing out long hypotheses, and also helps users by explicitly telling them at all times what exactly needs to be proved so they don't need to keep track.

4.2.2 Scaffolding Induction. We can let the user themselves write out the necessary cases for the induction (as done by Wemmenhove et al. [43]). This explicit signposting for each case also makes the structure of the computer proof match written proofs more closely to aid transfer. In order to guide the user to creating a complete set of cases, the editor can provide feedback for whether cases are missing or redundant. To meet criterion 1, these errors should be well explained, we could also provide examples of expressions that are not matched to give the user a hint for what cases to add. We should also design the interface for induction carefully so that it resembles written proofs and transfers well to pen-and-paper.

4.3 Calculation and Reduction Steps

There are several steps in a proof that are the same as evaluation in a functional language, for example, reductions of arithmetic ($1 + 1 \longrightarrow 2$) or unfolding of function definitions ($\texttt{double}(x) \longrightarrow x \times 2$). This course is not primarily concerned with teaching small-step reduction semantics, and small-step reduction therefore fits into category (1). We likely want to *support* and *elide* reduction steps.

4.3.1 As a support. A proof assistant could automatically perform any available computation steps, reducing an expression as much as possible. Many proof languages feature a similar normalization-by-evaluation strategy, where proof goals are automatically evaluated as far as they can be evaluated before the next step of a proof. Automatic normalization could confuse students who are not expecting it, so instead we will highlight expressions that can be reduced, and allow users to select and fully reduce these expressions.

4.3.2 As a scaffold. If we wanted to specifically teach small-step programming language semantics, we could ask students to write out what the next step of evaluation would be, instead of letting the computer calculate it.

4.4 Axiomatic Rewrites

Proof steps with symbolic terms cannot always be evaluated, such as using the commutativity of addition ($x + y \longrightarrow y + x$). The verbosity of large numbers of axiomatic rewrites can serve as a distraction from the overall structure of a proof. For example, the structure of an induction can get drowned out by large numbers of rewrites that need to be found in between induction cases. Thus we want to provide our users with an SMT-based automated support for axiomatic rewrites.

4.4.1 As a support. SMT solvers, or other equational rewrite proof generators [39] can be used to try to automatically verify rewrites steps a user wants to take. It is important not to be too permissive with this, for example, we do not want students to be able to solve exercises complete exercises by just rewriting them to true.

4.4.2 As a scaffold. Tools for teaching logic often provide users with a way to explore the possibly axioms they can use. In order to ensure that students understand how scaffolds have been applied, some tools also require students to write out the result of applying an axiom.

ACKNOWLEDGMENTS

The authors would like to acknowledge Andrew Blinn, Jiawei Chen, Haoxiang Fei, Jean-Baptiste Jeannin, Nishant Kheterpal, and Thomas Porter for their contributions to the Hazel Prover. The authors would also like to thank the anonymous referees at HATRA 2024 for their feedback on this paper. This work is funded in part by the National Science Foundation under Grant No. 2422028

REFERENCES

[1] Oluwatobi Alabi, Anh Vu, and Peter-Michael Osera. 2023. Snowflake: Supporting Programming and Proofs. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 2, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023*, Maureen Doyle, Ben Stephenson, Brian Dorn, Leen-Kiat Soh, and Lina Battestilli (Eds.). ACM, 1398. https://doi.org/10.1145/3545947.3576342

[2] Jonathan Aldrich, Robert J. Simmons, and Key Shin. 2008. SASyLF: an educational proof assistant for language theory. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*. ACM, Victoria BC Canada, 31–40. https://doi.org/10.1145/1411260.1411266

[3] Jeremy Avigad. 2019. Learning Logic and Proof with an Interactive Theorem Prover. In *Proof Technology in Mathematics Research and Teaching*, Gila Hanna, David A. Reid, and Michael de Villiers (Eds.). Springer International Publishing, Cham, 277–290. https://doi.org/10.1007/978-3-030-28483-1_13

[4] Alexander Bentkamp, Jon Eugster, Kevin Buzzard, Mohammad Pedramfar, and Patrick Massot. [n. d.]. Natural Number Game. https://adam.math.hhu.de/#/g/leanprover-community/nng4

[5] William Billingsley and Peter Robinson. 2007. Student Proof Exercises Using MathsTiles and Isabelle/HOL in an Intelligent Book. *Journal of Automated Reasoning* 39, 2 (Aug. 2007), 181–218. https://doi.org/10.1007/s10817-007-9072-3

[6] Sebastian Böhne and Christoph Kreitz. 2018. Learning how to Prove: From the Coq Proof Assistant to Textbook Style. *Electronic Proceedings in Theoretical Computer Science* 267 (March 2018), 1–18. https://doi.org/10.4204/EPTCS.267.1 arXiv:1803.01466 [cs].

[7] Merlin Carl. 2020. Number Theory and Axiomatic Geometry in the Diproche System. *Electronic Proceedings in Theoretical Computer Science* 328 (Oct. 2020), 56–78. https://doi.org/10.4204/EPTCS.328.4 arXiv:2006.01794 [math].

[8] David M. Cerna, Rafael P.D. Kiesel, and Alexandra Dzhiganskaya. 2020. A Mobile Application for Self-Guided Study of Formal Reasoning. *Electronic Proceedings in Theoretical Computer Science* 313 (Feb. 2020), 35–53. https://doi.org/10.4204/EPTCS.313.3

[9] The Coq Development Team. 2024. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.11551307

[10] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer, 378–388.

[11] David Delahaye, Mathieu Jaume, and Virgile Prevosto. 2005. Coq, un outil pour l'enseignement. Une expérience avec les étudiants du DESS Développement de logiciels srs. *Tech. Sci. Informatiques* 24, 9 (2005), 1139–1160. https://doi.org/10.3166/TSI.24.1139-1160

[12] Robert Bruce Findler. 2014. DrRacket: The Racket Programming Environment. *Racket Language Documentation* (2014).

[13] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. 2017. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science* 239 (Jan. 2017), 15–27. https://doi.org/10.4204/EPTCS.239.2

[14] Olivier Gasquet, François Schwarzentruber, and Martin Strecker. 2011. Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students. In *Tools for Teaching Logic*, Patrick Blackburn, Hans van Ditmarsch, María Manzano, and Fernando Soler-Toscano (Eds.). Springer, Berlin, Heidelberg, 85–92. https://doi.org/10.1007/978-3-642-21350-2_11

[15] Gaetano Geck, Artur Ljulin, Sebastian Peter, Jonas Schmidt, Fabian Vehlken, and Thomas Zeume. 2018. Introduction to Iltis: An Interactive, Web-Based System for Teaching Logic. In *Proceedings of the 23rd Annual ACM Conference*

on Innovation and Technology in Computer Science Education. 141–146. https://doi.org/10.1145/3197091.3197095 arXiv:1804.03579 [cs].

[16] Michael Greenberg and Joseph C Osborn. 2019. Teaching discrete mathematics to early undergraduates with software foundations.

[17] Mark Guzdial. 1994. Software-Realized Scaffolding to Facilitate Programming for Science Learning. *Interact. Learn. Environ.* 4, 1 (1994), 1–44. https://doi.org/10.1080/1049482940040101

[18] Maxim Hendriks, Cezary Kaliszyk, F van Raamsdonk, and Freek Wiedijk. 2010. Teaching logic using a state-of-the-art proof assistant. *Acta Didactica Napocensia* (2010).

[19] Martin Henz and Aquinas Hobor. 2011. Teaching Experience: Logic and Formal Methods with Coq. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, Berlin, Heidelberg, 199–215. https://doi.org/10.1007/978-3-642-25379-9_16

[20] Frederik Krogsdal Jacobsen and Jørgen Villadsen. 2023. On Exams with the Isabelle Proof Assistant. *Electronic Proceedings in Theoretical Computer Science* 375 (March 2023), 63–76. https://doi.org/10.4204/EPTCS.375.6

[21] CS Kaliszyk, F van Raamsdonk, Freek Wiedijk, Hanno Wupper, Maxim Hendriks, and Roel de Vrijer. 2008. Deduction using the ProofWeb system. Nijmegen: ICIS.

[22] Nadine Karsten, Frederik Krogsdal Jacobsen, Kim Jana Eiken, Uwe Nestmann, and Jørgen Villadsen. 2023. ProofBuddy: A Proof Assistant for Learning and Monitoring. *Electronic Proceedings in Theoretical Computer Science* 382 (Aug. 2023), 1–21. https://doi.org/10.4204/EPTCS.382.1

[23] Marie Kerjean, Frédéric Le Roux, Patrick Massot, Micaela Mayero, Zoé Mesnil, Simon Modeste, Julien Narboux, and Pierre Rousselin. 2022. Utilisation des assistants de preuves pourl'enseignement en L1. *Gazette des Mathématiciens* 174 (Oct. 2022). https://hal.science/hal-03979238 Publisher: Société Mathématique de France.

[24] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (sep 2018), 43 pages. https://doi.org/10.1145/3231711

[25] Avraham N Kluger and Angelo DeNisi. 1996. The effects of feedback interventions on performance: a historical review, a meta-analysis, and a preliminary feedback intervention theory. *Psychological bulletin* 119, 2 (1996), 254.

[26] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne, and Christoph Kreitz. 2017. Theorem Provers as a Learning Tool in Theory of Computation. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, Tacoma Washington USA, 83–92. https://doi.org/10.1145/3105726.3106184

[27] Josje Lodder, Bastiaan Heeren, and Johan Jeuring. 2019. A comparison of elaborated and restricted feedback in LogEx, a tool for teaching rewriting logical formulae. *J. Comput. Assist. Learn.* 35, 5 (2019), 620–632. https://doi.org/10.1111/JCAL.12365

[28] Josje Lodder, Bastiaan Heeren, and Johan Jeuring. 2020. Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction. *Electronic Proceedings in Theoretical Computer Science* 313 (Feb. 2020), 17–34. https://doi.org/10.4204/EPTCS.313.2

[29] Benjamın Machın and Luis Sierra. 2011. Yoda: a simple tool for natural deduction. In *Proceedings of the Third International Congress on Tools for Teaching Logic (TICTTL'11)*, Vol. 6680.

[30] Tobias Nipkow. 2012. Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer, Berlin, Heidelberg, 24–38. https://doi.org/10.1007/978-3-642-27940-9_3

[31] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.

[32] Liam O'Connor and Rayhana Amjad. 2022. Holbert: Reading, Writing, Proving and Learning in the Browser. http://arxiv.org/abs/2210.11411 arXiv:2210.11411 [cs].

[33] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL (2019), 14:1–14:32. https://doi.org/10.1145/3290327

[34] Benjamin C Pierce. 2009. Lambda, the ultimate TA. *ACM Sigplan Notices* 44, 9 (2009), 121–122.

[35] Henry L. Roediger and Mary A. Pyc. 2012. Inexpensive techniques to improve education: Applying cognitive psychology to enhance educational practice. *Journal of Applied Research in Memory and Cognition* 1, 4 (2012), 242–248. https://doi.org/10.1016/j.jarmac.2012.09.002

[36] Benoit Rognier and Guillaume Duhamel. 2016. Présentation de la plateforme edukera. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, Julien Signoles (Ed.). Saint-Malo, France. https://hal.science/hal-01333606

[37] Elaine Seymour and Nancy M Hewitt. 1997. *Talking about leaving*. Vol. 34. Westview Press, Boulder, CO.

[38] Nikki Sigurdson and Andrew Petersen. 2019. A Survey-based Exploration of Computer Science Student Perspectives on Mathematics. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, Elizabeth K. Hawthorne, Manuel A. Pérez-Quiñones, Sarah

Heckman, and Jian Zhang (Eds.). ACM, 1032–1038. https://doi.org/10.1145/3287324.3287416

[39] Nicholas Smallbone. 2021. Twee: An Equational Theorem Prover.. In *CADE*. 602–613.

[40] Korbinian Staudacher, Sebastian Mader, and François Bry. 2019. Automated Scaffolding and Feedback for Proof Construction: A Case Study. In *Proceedings of the 18th European Conference on e-Learning*. ACPI, 64. https://doi.org/10.34190/EEL.19.114

[41] Gabriel J Stylianides, Andreas J Stylianides, and Keith Weber. 2017. Research on the teaching and learning of proof: Taking stock and moving forward. *Compendium for research in mathematics education* (2017), 237–266.

[42] Athina Thoma and Paola Iannone. 2022. Learning about Proof with the Theorem Prover LEAN: the Abundant Numbers Task. *International Journal of Research in Undergraduate Mathematics Education* 8, 1 (April 2022), 64–93. https://doi.org/10.1007/s40753-021-00140-1

[43] Jelle Wemmenhove, Dick Arends, Thijs Beurskens, Maitreyee Bhaid, Sean McCarren, Jan Moraal, Diego Rivera Garrido, David Tuin, Malcolm Vassallo, Pieter Wils, and Jim Portegies. 2024. Waterproof: Educational Software for Learning How to Write Mathematical Proofs. *Electronic Proceedings in Theoretical Computer Science* 400 (April 2024), 96–119. https://doi.org/10.4204/EPTCS.400.7