Is truth futureproof?

On the possible futures of mechanized proofs

CHRIS MARTENS, EMMA TOSCH, ELAN SEMENOVA, and CYNTHIA LI, Northeastern University, USA

Interactive theorem provers play increasingly important roles in the programming languages and mathematics communities, resulting in a large body of mechanized proofs that bear witness to mathematical knowledge. The ideal promise of accumulating these artifacts is that they allow us to preserve that knowledge across time into future generations and across intellectual silos. However, by virtue of being software, mechanized proofs are at risk of breaking as they age, hindering reproduction (e.g. of the proof's correctness), inspection, and reuse. How can we future-proof mechanized proofs?

In this paper, we posit two archetypal motivations for mechanization, *convincing* and *explaining*, which motivate overlapping but distinct ideals and practices that sometimes conflict or give rise to different priorities. We survey current practices in mechanized proof communities that relate to the longevity of mechanized proofs, indicating where they align or misalign with these ideals. We conclude with a set of discussion questions for the community.

ACM Reference Format:
Chris Martens, Emma Tosch, Elan Semenova, and Cynthia Li. 2026. Is truth futureproof? : On the possible futures of mechanized proofs. In *Proceedings of the 15th PLATEAU Workshop on Programming Languages and Human-Computer Interaction (PLATEAU '25)*. 12 pages. https://doi.org/10.1184/R1/31825492

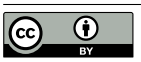
1 Introduction

> "We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem—and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs."
> — De Millo, Lipton, and Perlis, "Social Processes and Proofs of Theorems and Programs." [9]

The project of mechanizing mathematics is more active now than ever in history: with the help of interactive theorem proving (ITP) software, formal systems can be defined, specified, and proven correct to a high standard of trust, accelerating the pace of discovery while keeping human error in check. ITPs have enabled massive advances in mathematics that would not have been possible without them, including famous proofs of previously unsolved conjectures like the Four Color Theorem [13] and Kepler Conjecture [14]. In computing, the ability to formally verify properties of programming languages [3, 15], compilers [10, 24, 25], operating systems [19], and security protocols [6, 27, 35], lends a higher degree of trust that these systems adhere to mathematically-definable criteria.

Whenever a new mathematical result is proved and published, it changes the status of human knowledge. Some question that was once unanswered now has an answer, and anyone with sufficient fluency in the vocabulary and methods of the field can learn it. Research ethics demands that we also strive for reproducibility: publishing sufficient information that another expert with

Authors' Contact Information: Chris Martens; Emma Tosch; Elan Semenova; Cynthia Li, Northeastern University, Boston, MA, USA.



PLATEAU '25, Boston, MA

https://doi.org/10.1184/R1/31825492

the right resources could follow the same steps and reach the same conclusions. Reproducibility is part of a philosophy of science that sees knowledge as *growing monotonically*, i.e., an expectation that we do not *lose* scientific and mathematical results once someone has written them down (the notion of a "lost theorem" arising to mark lamentable counterexamples). That is, mathematical truth should be "future-proof".

Proof mechanization presents opportunities for the durability of mathematical knowledge. On the one hand, they cannot elide details or use obscure notation whose meaning is known only by the author: to pass the proof checker, proofs must be expressed in the syntax of that tool and leave no gaps in the steps of reasoning. In this case, a novitiate in the techniques of the proof can get the high-level idea of it by reading the paper, but refer to the mechanization if they are puzzled by elided steps or terse definitions.

On the other hand, unfortunately, mechanized proofs (and the ITPs used to write and check them) are software. Software in general comes with well-documented challenges for longevity, broadly characterized as "bit-rot," which include changes to the language or type checker underlying the ITP, updates or renamed identifiers in libraries or other dependencies, and changes to build tools. If a mechanized proof associated with some famous prior result from five or ten years ago is downloaded today, but no longer passes the proof-checker, what does that mean as to the status of mathematical knowledge it represents? This question isn't just hypothetical; actively-used ITPs also tend to be actively developed, and indeed we find that several proofs from 5-10 years ago no longer work "out of the box."

The primary purpose of this paper is to generate discussion among the community. Our preliminary conversations and investigations reveal that two sets of values, *increasing confidence in results* (**convincing**), and *communicating formal ideas* (**explaining**), lead to distinct perspectives and practices that shape the creation, review, archival, and reuse of mechanized proofs. Thus the question of *what we actually want* for the future of mechanized proofs is not yet known. We then document practices around creating, sharing, organizing, and maintaining proofs. We attempt to identify where these practices help or hinder the cause of future-proofing mechanized mathematics, underscoring where the most significant hindrances give us cause for concern. Finally, we close with concrete prompts for discussion with the PLATEAU community, whom we hope to engage in light of its members' distinctive intersection of fluencies in both the mathematical and social aspects of formal proof.

2 Background

What we have been calling a "mechanized proof" actually refers to more than just a theorem statement and its proof: the mathematician must communicate all relevant definitions, functions, and lemmas that build up to one or more "top level" results. All of this information must be expressed in the computer language supported by an ITP in order for that software to assist with interactive proof development and proof-checking. *Definitions* often comprise the majority of codebases, especially in Programming Languages research, where definitions are used to represent so-called *object languages* (as opposed to the meta-language of the ITP itself) about which theorems will

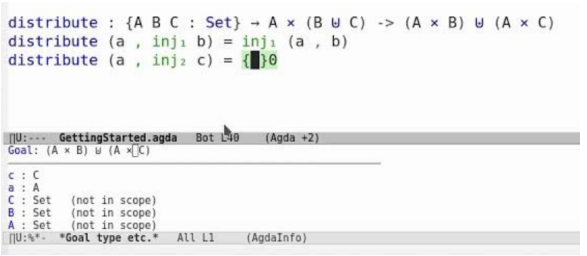


Fig. 1. An in-progress proof of distributivity of conjunction over disjunction in Agda. The in-progress proof is represented as a proof term, a piece of code whose *type* corresponds to the theorem statement, with a *typed hole* (the `{_}0` syntax) representing the part of the proof that remains to be written. The *proof state* beneath the code shows the goal and in-scope variables corresponding to current assumptions.

be proved. With these tools, the human mechanizer writes code to formulate definitions, state theorems about those definitions, and prove each theorem step by step with interactive feedback and automation from the software.

Actively-used and supported ITPs today include Agda, Lean, Rocq, Isabelle, Idris, HOL Lite, and many, many more. These tools vary in many respects, such as the mathematical foundation underlying their language's type theory, their support for different language features, and their workflows for developing and checking proofs. One notable place of variation of interest to us from the proof maintenance perspective is the final form of the proof itself, as recorded in the source file. There are two common design choices: **proof as tactic script** and **proof as proof term**. We briefly define these terms for the unfamiliar.

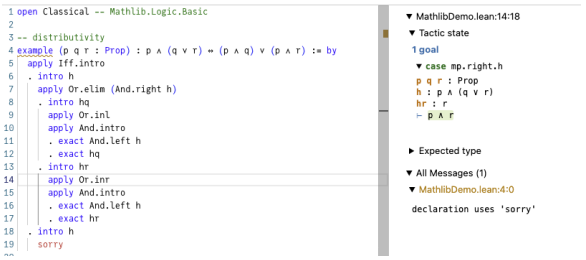


Fig. 2. An in-progress proof of distributivity of conjunction over disjunction in Lean (viewed in Lean's web interface). The in-progress proof is represented as a partial tactic script, a nested series of commands that apply reasoning principles to transform the proof state, displayed to the right.

Tactic Scripts. In a system like Rocq, for example, it is standard to develop a proof with *tactics*: imperative commands that modify a *proof state*, which tracks the assumptions and proof obligations at each step. A tactic is (in principle) analogous to saying something like "by induction [on some structure]" or "by linear arithmetic" on paper; you appeal to a reasoning principle that is justified by (but not necessarily directly a reference to) the language's logical principles. See figure 2 for an example of a tactic script in development.

Proof Terms. Most proof assistants, including those with tactic support, exploit the propositions-as-types correspondence in which programs in the language itself can be read as a proof of the theorem. For example, the lambda term $\lambda p.\ \mathsf{case}\ \pi_2\ p\ \mathsf{of}\ \mathsf{inl}\ y \Rightarrow (\pi_1\ p, y) \mid \mathsf{inr}\ z \Rightarrow (\pi_1\ p, z)$ can be read as a proof of the first direction of the running distributivity example, interpreting disjunction as a sum type, conjunction as a product type, and implication as function type. In this setting, inferences in the proof are justified by the introduction and elimination forms of the connective, as witnessed by the corresponding program passing the typechecker. Proofs by induction, for example, are inductive definitions over data corresponding to the inductive subject. See figure 1 for an example of a proof term in development.

2.1 History of the uptake of ITPs

Back in 2005, some members of the Programming Languages research community went on a mission to get more people to mechanize their proofs. The POPLMark Challenge [3] was a coordinated effort to build a set of benchmarks on which to compare proof assistants across a number of criteria, especially expressivity for different kinds of proofs of interest to the community. It represents a key example of prior work on establishing cross-proof-assistant knowledge, coordinating users of multiple proof assistants and from multiple research lineages. The records of this effort are still, as of December 15, 2025, available online at https://www.seas.upenn.edu/~plclub/poplmark/. The website also hosts some (compressed) files implementing the solutions themselves, but in some cases only links to the authors' websites, some of which no longer resolve. The POPLMark Reloaded [1] project attempted to supplement the original POPLMark problems with a wider scope of proof techniques considered necessary for practical work in programming language theory, especially logical relations proofs; however, we cannot find records of the results of this initiative.

To the extent that POPLMark's goal was to drive adoption, it was wildly successful. We are now in a situation where a huge number of mechanized proof developments are hosted online for public consumption, and referenced within the mathematical literature as *evidence for mathematical claims*. Each ITP has its own user community and proof ecosystem, including libraries (standard and contributed), versioning and dependency tracking mechanisms, documentation, forums, and chat servers. Mechanization developments exist in proliferation across formal artifact repositories, ITP library distributions, independent code repositories, and author websites.

There are currently two landmark studies of multiple proof assistants and their user communities: first, Ringer et al. [29] survey the *proof engineering* practices of proof assistant users in 2018, resulting in a 183-page manuscript spanning foundations, automation facilities, organization and scalability, user interfaces, tooling, and proof ecosystems. Second, Shi et al. [33] describe an observation study they carried out with users of different proof assistants carrying out specific mechanization tasks that the researchers gave them, and documented practices as directly observed. The practices documented in these studies articulate the goals and potential, realized and unrealized, of proof mechanization, and we draw from their findings for community consensus around common and best practices.

3 Why do we write mechanized proofs in the first place?

Through published academic papers, first-hand experience, and innumerable hallway conversations, we have suggest the following major reasons people write mechanized proofs:

R1 Assisting with high proof complexity. Historically, mechanization was motivated by a need to improve confidence and scalability for people *writing* the proofs or *reviewing* to assess correctness of novel results [29]. There are two major sources of complexity, distinguished by how theorems are used, that motivate the use of ITPs:

- **Proven theorems as units of knowledge (mathematics).** On-paper proofs can be difficult to check, and human reviewers are not flawless proof checkers. *A priori* trust in the prover's kernel and agreement that the theorem faithfully represents its natural language ought to increase trust in correctness [2].
- **Proven theorems as software validation (computing).** Proofs are also written as part of the software development process for the sake of increasing trust in the safety of the associated code, according to some formal soundness principles. Proofs may also be generated or otherwise integrated into compiler pipelines, as in certified and certifying compilers [24, 26].

R2 Communicating formal ideas. Mathematical language can be imprecise, and proof assistant languages create more precision by enforcing type correctness in definitions, functions, and theorems. When multiple people know the same type system well enough to write type-checking programs in it, it can be used to transmit ideas between one another in a precise way. Examples of ITP users emphasizing their efforts towards human readability can be found in Shi et al.'s recent user study [33].

3.1 Convincing and Explaining

For the purposes of shorthand reference to these perspectives and the concerns they encompass, we define two archetypal motivations, **Convincing** and **Explaining**. These should be understood not as opposite or conflicting attitudes, but rather as mindsets that any given person can adopt in different measures at different times, and which can motivate different decisions and perspectives on proof mechanization.

3.1.1 Convincing. The activity of *convincing* is primarily concerned with *increasing confidence in results* and *increasing trust in formally verified software*, the first two desiderata mentioned

previously. The convincing mindset often arises in response to heightened scale and complexity (**R1**). It is reflected in some corners of the mathematics community, where mechanization has been heralded as the "new standard of rigor" [2]; Avigad and Harrison assert, "Computational proof assistants make it possible to check the correctness of [proofs], thereby increasing the reliability of mathematical claims." Additionally, the software verification community sometimes touts mechanization's ability to "improve actual and perceived reliability," observing that "it can make explicit which parts of the system are trusted, and further decrease the burden of trust as more of the system is verified" [29].

For the purpose of convincing, it is very important to know exactly what the proof language is, how theorem statements connect to the results stated in papers, what constitutes the trusted code base (TCB) of the proof assistant, and what "escape hatches" one must check for in the development to ensure a positive result is meaningful. The activity of mechanization serves as a guardrail for getting definitions right and confirming (or refuting) one's beliefs about certain hypothesized properties of an object system. Usually, the proof of such a property is meant to carry positive valence, like "safety" or "soundness" of an object system, where the merit of that system depends on this property holding of it. From the convincing perspective, neither the process of proof nor the resulting proof artifact matter quite as much as the proof assistant's confirmation of correctness. The cause of convincing it thus well-served by tactic-based interaction and powerful automation support.

3.1.2 Explaining. The *Explaining* mindset is concerned with *communicating formal ideas* (**R2**). Explaining motivates the use of ITPs to articulate problem domains as clearly as possible with an aim to evolve your development towards a legible, well-structured web of definitions, theorem statements, lemmas, and proofs. The goal of explanation leaves us unsatisfied with unnecessarily verbose and complicated proofs, leading us to rewrite definitions that better characterize the problem and enable simpler forms of reasoning. Explanation motivates commenting and documenting not just the statements of theorems and lemmas, but design decisions made in definitions, individual steps of proofs, and the overall structure and techniques used in the full development.

The activity of explanation benefits less from automation, since human comprehension of the proving process is just as important as seeing that the software accepts the proof. In light of the goal of mechanization serving as communicative artifacts, we must also consider the *reader* of mechanized proofs ("when one starts wondering *why* a statement holds, the readability of the proof matters" [28]). This is also important in the context of *reuse* of components of a previous development: the process of assessing which components can be reused and how is a human-mediated activity involving end-user comprehension. Mechanized developments that support understanding *why* a result holds, or *when and how* to use a certain proof technique, must be well-organized and readable in order to clearly communicate the author's intent.

4 Current Practices

In light of both *convincing* and *explaining* as motivations, we can refine our notion of what it means for mechanized developments to be "future-proof". We desire the following capabilities:

- **C1** Storing proofs artifacts using long-term, trustworthy digital storage;
- **C2** Supporting search, browse, filter, and retrieve archived proofs as archives grow;
- **C3** Running proofs, i.e. asking the ITP software to verify their correctness;
- **C4** Reading proofs, assuming sufficient literacy in the proof language;
- **C5** Adapting old proofs to new ones, e.g. updating them to new ITP versions, translating them to different ITPs, separating out modular components to reuse in distinct developments.

A number of initiatives and activities already address many of these desiderata. These include:

- Archiving proofs: artifact evaluation and corresponding supplementary material for academic papers, as well as ITP-specific proof archives (§4.1, §4.2)
- Search functionality over archives: ITP-specific tools; artifact metadata on archives and search on the ACM Digital Library (ACMDL) (§4.2),
- Running/reproducing proof-checking: containerization of artifacts; version update utilities; forward-compatibility (§4.2),
- Support for reading and interacting with completed proofs: literate programming; hosted interactive proofs (§4.3),
- Comparing/adapting proofs across ITPs: the POPLmark challenge; automated proof adaptation and repair (§4.4, §4.5)

4.1 Artifact Evaluation and Artifact Repositories

As soon as mechanized metatheory started taking a foothold in formal research, namely in the programming languages community, researchers began tackling the problem of incorporating the mechanized proof into the publication process: certainly if a paper's correctness depends on that proof, then the proof needs to be included alongside the paper as part of the published result. As a result, SIGPLAN's flagship conferences have constituent processes including the publication of *supplementary material*, which can include mechanizations, and *artifact evaluation* (AE), a supplement to peer review in which software and proof artifacts are evaluated for their alignment with claims in the paper and confirmation that they operate as described (e.g., that the reviewer can "run" a proof in the ITP and independently confirm that it passes the checker.) Evaluated artifacts and supplementary materials are then hosted on ACM servers alongside paper publications, constituting one form of *proof repository* that we consider.

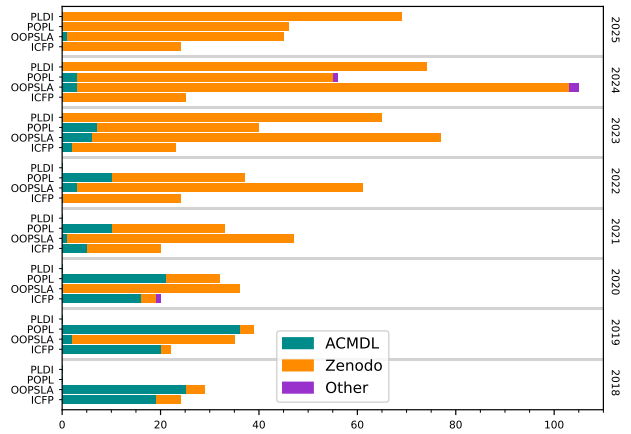


Fig. 3. A histogram of hosting platforms for 1108 papers having artifacts listed on their ACMDL pages from the 2018–2025 proceddings of PLDI, POPL, ICFP, and OOPSLA, with the exception of POPL 2018 and PLDI prior to 2023. There were a total of 914 artifacts hosted on Zenodo, 190 on the ACMDL, and 4 other platforms.

The motivation for the first AE at POPL was to address "the insufficient respect paid to the artifacts that back papers...especially...[in]...areas that are so centered on software, models, and specifications," while also imposing a social cost on undesirable behavior ranging from "mere sloppiness to, in extreme cases, dishonesty," rewarding those, "who take the trouble to vigorously implement and test their ideas" [22]. Over time, AE has come to be synonymous with promoting repeatable, replicable, or reproducible (rep*) research [23], aligning with the *convincing* mindset. More recently, retrospectives on AE have recast its core value as promoting *reuse*. Conclusions about the role that AE plays in fostering reuse vary (e.g., Winter et al. [34] vs. Baldassarre et al. [4]) and will not likely be known for some time. These findings also certainly affect the *specific* artifacts that are mechanized proofs, but little research has been done to date on their unique challenges.

Limitations and potential. The artifact evaluation process—an ACM-sanctioned review activity—includes practices that overlap with our desiderata for future-proofing mechanized metatheory: multiple non-authors read through documentation and proof scripts, executing proof scripts on different physical machine, possibly using different software versions (**C4**, **C3**). Artifact evaluation additionally entails curation: only artifacts whose associated research papers have successfully passed peer review are eligible to be evaluated.

Unfortunately, evaluation is one-shot (i.e., cannot satisfy **C5**), occurring only during the peer review process for the associated paper. We find no mention of repetition, replication, or reproducibility outside sections devoted to artifact evaluation in the CfPs of the four major SIGPLAN conferences from 2018–2025. There are, however, several activities where such work might be within scope:

- The OOPSLA CfPs from 2018 and 2022–2025 all mention "reuse of software systems."
- The ICFP CfPs from 2018–2025 all include "experience report" as a category of publication.
- All POPL CfPs from 2018–2025—with the exception of 2024—mention experience reports in their description of relevant types of work. Unlike the ICFP calls, they do not provide additional guidance, nor details.

We performed a manual cursory investigation of paper titles to ascertain whether there was any substantial rep* work being published, specifically with respect to mechanized proofs, formal verification, or related domains. Despite OOPSLA's explicit call for work that examines reuse, we found only one published paper whose title and abstract might qualify [18]. We found at least one Experience Report in each of the ICFP programs we examined, with the exception of 2020-2021; none appeared explicitly as rep* papers, although at least three could be recast as such [7, 16, 31]. All PLDI CfPs from the 2018–2020 period emphasized novelty and empiricism above all else ("Novel system designs, thorough empirical work, well-motivated theoretical results, and new application areas are all in scope for PLDI."); despite this, we did find one replication study at PLDI [5].

Another limitation of artifact evaluation as it is currently practiced is that it is separate from artifact storage; in fact, the ACM explicitly states:

> We do not mandate the use of specific repositories. Publisher repositories (such as the ACMDL), institutional repositories, or open commercial repositories (e.g., figshare or Dryad) are acceptable. In all cases, repositories used to archive data should have a declared plan to enable permanent accessibility. Personal web pages are not acceptable for this purpose. [11]

While the ACMDL has long supported storage of supplementary materials (**C1**), explicit metadata and indexing of artifact began more recently. Despite increasingly archival-quality metadata and storage for artifacts, as seen in Fig. 3, authors increasingly publish their artifacts to the general science artifact archive Zenodo over the specialized computing archive of the ACMDL.

4.2 ITP-specific proof repositories and search features

Nearly every ITP has a standard library and other mechanisms for supporting an "ecosystem" of contributed proofs, definitions, and developments. These repositories also typically support some kind of proof or theorem retrieval through search (in the "search engine" sense, not the "proof search" sense) via text-based input (**C2**). For example, Rocq, Lean, and Agda all support substring search over theorem names, structured search over theorem shapes, and premise and goal search, from within the ITP itself.

The Isabelle Archive of Formal Proofs (AFP) [20] (https://www.isa-afp.org/) — an archive of author-submitted, peer-reviewed proofs written with the Isabelle proof assistant — is a noteworthy

example. Its approach to knowledge archiving, searchability, and readability set a high standard maintained by clear criteria enforced by peer review and forwards compatibility (**C1**–**C5**).

The Mathlib Initiative (https://mathlib-initiative.org/) and corresponding NSF-supported Institute for Computer-Aided Reasoning in Mathematics seeks to advance computer verification of mathematical knowledge, promising "unprecedented confidence in complex proofs" (via their About page). Mathlib takes the form of a set of libraries in the Lean proof assistant. Its primary goal is to grow in scope and breadth of community involvement. The Mathlib contributor documentation is itself a bit lean: unlike AFP, it does not mention archival practices, nor forward compatibility. Submissions take the form of pull requests on Github, a for-profit private company (failing **C1**). End-users can search using the Mathlib-supported Loogle search engine, as well as third-party software such as `moogle.ai` (**C2**).

Search does not have a strong association with either convincing or explaining. However, the archive and repository practices we have observed align more with the desiderata for convincing than for explanation.

Limitations and potential. AFP provides an exemplary model of how to support future-proofing proven results. Meanwhile, the Mathlib Initiative has seen substantial community uptake and financial investment, suggesting a growing need for our complementary infrastructure. The main limitation of existing archives and archive initiatives is that they are siloed by software: we are unaware of any archives that support multiple proof assistant frameworks and therefore none that support translation between proof scripting languages (**C5**). Furthermore, initiatives driven and funded by private organizations may themselves be privatized or scrapped, making reliance on single ITP a fragile state for the longevity of mechanized mathematical knowledge.

4.3 Interactive proof replay and inspection tools

The process of mechanizing any nontrivial proof is an interactive dialogue between human and proof checker: the human states a theorem, proposes the first step, and studies the proof state produced by their interlocutor to make the next move. But the tactic script that results from this process is only half of the conversation, just the moves used by the human. As Pit-Claudel puts it, "the written part of a proof script records only the steps taken, not the states that they led to." [28] To rectify this situation, literate programming tools like CoqDoc have been developed alongside a "best practice" to manually copy-and-paste proof states in comments interleaved within proof scripts. Pit-Claudel's project Alectryon attempts to improve this state of affairs, using a wrapper around Rocq's APIs to record proof states and generate an interactive HTML document through which readers can explore proofs without running the prover locally [28]. The Lean prover's current "Lean 4 web" interface (https://live.lean-lang.org/) supports an apparently similar mode of interaction (albeit with less customizability on behalf of the proof author). Other promising approaches attempt to lift the "interactive proof editing" process to the realm of programs so that this process is recorded in the source code. For instance, Joomy Korkut's masters thesis devised a notion of "edit time tactics" for Idris that surface users' proof editing interactions as Idris metaprograms [21]. Shi et al.'s notion of proof *deautomation*, which they demonstrate as a process for reconstructing the underlying steps of Rocq proofs from automation tactics [32], could also play a role in unfolding proof details for readers. The motivation for all of these methods aligns with the explanation mindset.

Limitations and potential. Proof replay tools go a long way for assisting with reader understanding, so it seems natural to want to support them as part of a proof archival process. Although we are unaware of archival formats and practices for this form of interactivity that we expect to work "out of the box," we might find inspiration from another field interested in the archival of interactive artifacts: videogames. Scholars of games are interested in referring to certain moments or game

design choices from within a game, and thus have investigated expanding citation practices. The branching nature of games means textual description alone can be insufficient to describe a moment of gameplay and replicate it, depending on the story events and skill needed to get a player to a particular state. The Game and Interactive Software Scholarship Toolkit (GISST) and playable quotes both attempt to emulate executable states and input recordings to allow a reader to interact with a game inline in their browser [12, 17]. While they target legacy platforms and game systems like the Game Boy, these projects consider interaction as relevant to citation and replication, and are meaningful for software preservation efforts.

However, shifting technology stacks (especially for the web) may impact the backwards compatibility of emulation software, which then merely shifts the preservation problem to the emulation software itself. These projects also lack a central, stable repository, and any citation to a specific emulated instance of a game snippet relies on the continued existence of a server that hosts the software. Any sort of archival of proof replay support will have to reckon with the same issues of software preservation.

4.4 Cross-ITP comparisons and interoperability

The POPLMark challenges provide a good reference point for comparing proof assistants, and contribute a starting point set of benchmarks for their comparison. During an overlapping time span, the Logosphere project (https://kwarc.info/projects/logosphere/) sought to establish interoperability mechanisms between different proof assistants (in particular, a shared intermediate representation format based on the LF logical framework; **C5**). Inter-ITP communication as not been an active research area for some time.

Limitations and potential. POPLMark's main limitation, from our perspective as to its benefits, is that it only happened twice. While POPLMark has achieved its original purpose of driving adoption of the practice of proof mechanization, we see a great deal of potential in the idea of continuing to develop repositories of proofs that solve benchmark problems in multiple different ITPs, posed as periodic challenges to the community. Competitions could include new problems representative of the most active research challenges in their time, and the resulting proof repositories could document the differences and capabilities of ITPs as they evolve.

From personal communication regarding the Logosphere project, we learned that ultimately, while some promising ideas were developed for modularity boundaries across mechanizations, the theories of different ITPs were too distinct to translate into a single, unifying framework and achieve critical-mass support from the community. In light of the downfall of the Logosphere project, we believe it is worthwhile to carry forward the project of cross-framework knowledge sharing focusing on the *human* practice of search and component reuse, rather than on purely mechanistic interoperability (**C4**, **C5**). Thus, we see prior interoperability attempts as aligning with the convincing mindset and believe its goals would be better served by additionally considering explanatory motivations.

4.5 Proof maintenance and repair

In anticipation of the breakage of proofs with evolving proof ecosystems, both near-term and far-term approaches have been proposed. These tools are primarily concerned with advancing a key *convincing* goal: ensuring that proofs continue to be validated by a proof checker as one or both of them evolves.

Version managers are tools for letting developers manage multiple installations (of multiple versions) of software. Tools for releasing projects can interoperate with these. For example, Lean has a version manager called elan (https://github.com/leanprover/elan) that interoperates with

`lean-toolchain` files associating projects to their versions and dependencies. *Automated proof repair* is a more nascent idea in research whose goal is to search for and suggest changes to broken proofs that, if applied, will cause the theorem to go through. Ringer et al. have made strides in this area, starting with a tool that, based on a history of changes to specifications and proofs, automatically generate "patches" for components of proofs that have yet to be updated to reflect those changes [30]. Finally, there is some preliminary work on "regression proof selection" (analogous to regression testing) for the Rocq prover [8], adapting continuous integration practices from the software industry in a similar effort to keep code working as its ecosystem evolves.

Limitations and potential. Version management and continuous integration practices seem very promising to explore from the perspective of keeping proofs "working", or at least surfacing the places where they break to developers who can manually intervene. Automated proof repair is still in the basic research stage and limited to restricted forms of structural analogy across proofs, but also seems promising in the mong term, especially if supported by an interactive process. From the perspective of readability and communication, however, we worry that applying these techniques with too much automation could alter the original context and reasoning approach beyond recognition and readability. To this end, we see potential for *manual* proof adaptation and repair guided by human-readable documentation.

5 Conclusion and Discussion Questions

We have attempted to synthesize a set of criteria, and practices supporting those criteria, for the maintenance of mathematical knowledge embodied by mechanized proofs, drawn widely from across the research communities who generate and rely on this knowledge. We close with a set of questions for discussion.

(1) Have you ever attempted to run someone else's mechanized proof from several years ago? What was your goal, what steps did you take, and did you ultimately succeed or fail?
(2) When and why do *you* mechanize, or encourage others to mechanize, a mathematical development? Do you find yourself more motivated by one of the Convincing or Explaining mindsets?
(3) What knowledge artifacts should we attempt to preserve in order to support diverse motivations and social processes of proof?
(4) Proving a theorem in multiple ITPs has a number of benefits, including cross-comparison of tools and resilience against the loss of knowledge. How can we incentivize this kind of effort and share knowledge across different ITP communities?
(5) The stakeholders of preserving mechanized knowledge include ITP maintainers, ITP users, and professional communities of researchers. Who should bear what responsibility for the stewardship of proof archives?

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019), e19.
[2] Jeremy Avigad and John Harrison. 2014. Formally verified mathematics. *Commun. ACM* 57, 4 (2014), 66–75.
[3] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 50–65.
[4] Maria Teresa Baldassarre, Neil Ernst, Ben Hermann, Tim Menzies, and Rahul Yedida. 2023. (Re) Use of Research Results (Is Rampant). *Commun. ACM* 66, 2 (2023), 75–81.

[5] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. “Synthesizing input grammars”: A replication study. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 260–268.
[6] Bruno Blanchet. 2008. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing* 5, 4 (2008), 193–207.
[7] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, set, verify! applying hs-to-coq to real-world Haskell code (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–16.
[8] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. iCoq: Regression proof selection for large-scale verification projects. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 171–182.
[9] Richard A De Millo, Richard J Lipton, and Alan J Perlis. 1979. Social processes and proofs of theorems and programs. *Commun. ACM* 22, 5 (1979), 271–280.
[10] Axel Dold, Thilo Gaul, and Wolf Zimmermann. 1998. Mechanized verification of compiler backends. *Proc. STTT* 98 (1998).
[11] Association for Computing Machinery. 2020. Artifact Review and Badging — Current. https://www.acm.org/publications/policies/artifact-review-and-badging-current
[12] Joël Franušić, Kathleen Tuite, and Adam Smith. 2023. Playable Quotes for Game Boy Games. In *Proceedings of the 18th International Conference on the Foundations of Digital Games* (Lisbon, Portugal) *(FDG ’23)*. Association for Computing Machinery, New York, NY, USA, Article 14, 11 pages. doi:10.1145/3582437.3582479
[13] Georges Gonthier et al. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393.
[14] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. 2017. A formal proof of the Kepler conjecture. In *Forum of mathematics, Pi*, Vol. 5. Cambridge University Press, e2.
[15] Robert Harper and Daniel R Licata. 2007. Mechanizing metatheory in a logical framework. *Journal of functional programming* 17, 4-5 (2007), 613–673.
[16] Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random testing of a higher-order blockchain language (experience report). *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 886–901.
[17] Eric Kaltman and Joseph C Osborn. 2024. The Potential of Cited Executable State for Software Object Access, Validation, and Research: A Preview of the GISST System. In *iPRES 2024*.
[18] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.
[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.
[20] G Klein, T Nipkow, L Paulson, and R Thiemann. [n. d.]. Isabelle Archive of Formal Proofs. https://isa-afp.org. https://isa-afp.org
[21] Joomy Korkut. 2018. *Edit-time tactics in idris*. Ph. D. Dissertation. Wesleyan University.
[22] Shriram Krishnamurthi. 2013. Artifact evaluation for software conferences. *ACM SIGSOFT Software Engineering Notes* 38, 3 (2013), 7–10.
[23] Shriram Krishnamurthi and Jan Vitek. 2015. The real software crisis: Repeatability as a core value. *Commun. ACM* 58, 3 (2015), 34–36.
[24] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 42–54.
[25] Robin Milner and Richard Weyhrauch. 1972. Proving compiler correctness in a mechanized logic. *Machine intelligence* 7, 3 (1972), 51–70.
[26] George C Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices* 33, 5 (1998), 333–344.
[27] Lawrence C Paulson. 1997. *Mechanized proofs of security protocols: Needham-Schroeder with public keys*. Technical Report. University of Cambridge, Computer Laboratory.
[28] Clément Pit-Claudel. 2020. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 155–174.
[29] Talia Ringer, Karl Palmskog, Ilya Sergey, Gligoric Milos, and Zachary Tatlock. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages* 5, 2-3 (2019), 102–281.
[30] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 115–129.
[31] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proceedings of the ACM on Programming*

Languages 7, ICFP (2023), 878–894.

[32] Jessica Shi, Cassia Torczon, Harrison Goldstein, Andrew Head, and Benjamin C Pierce. 2025. Designing Proof Deautomation for Rocq. In *Proceedings of the 15th PLATEAU Workshop on Programming Languages and Human-Computer Interaction (PLATEAU'25). ACM.*

[33] Jessica Shi, Cassia Torczon, Harrison Goldstein, Benjamin C Pierce, and Andrew Head. 2025. QED in Context: An Observation Study of Proof Assistant Users. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 337–363.

[34] Stefan Winter, Christopher S Timperley, Ben Hermann, Jürgen Cito, Jonathan Bell, Michael Hilton, and Dirk Beyer. 2022. A retrospective study of one decade of artifact evaluations. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 145–156.

[35] Fuyuan Zhang, Limin Jia, Cristina Basescu, Tiffany Hyun-Jin Kim, Yih-Chun Hu, and Adrian Perrig. 2014. Mechanized network origin and path authenticity proofs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 346–357.