A Data-Centered User Study for Proof Assistant Tools

Hanneli C. A. Tavante
McGill University
hanneli.andreazzitavante@mail.mcgill.ca

Abstract

Proof assistants gained momentum in the past decade. The popularization of functional programming and the rise of industry cases using formal verification contributed to the popularity growth of Coq, Lean, Agda, and other related projects. However, the existing tools for writing proofs in most of these assistants are limited. There is almost no record of usability and interface studies for tools targeting the development of proofs. After presenting a survey on the plug-ins and standalone working interfaces for Coq and Lean, this work introduces a proposal for a data-centered user study to improve the existing development environments for proof assistants. We will also explore a few features that would decrease the entry-bar for Coq, potentially helping students and even experienced users.

1. Introduction

When users attempt to learn a new programming language, they will have multiple tools available to support their learning process. For example, plugins for syntax highlighting, code refactoring, and style checkers are some helpful assets that can be installed in multiple Integrated Development Environments (IDEs) or text editors.

In the case of proof assistants, more specifically Coq, the tooling options are far more limited. Following the official documentation, located at https://coq.inria.fr/, it is possible to find two main standalone projects (namely IDEs) and a few other alternative packages and plugins for other text editors.

For standalone interfaces, the first official recommendation is CoqIDE. As the name suggests, it is an IDE specific for Coq, and Fig. 1 illustrates how it looks. The second alternative is jsCoq (Gallego Arias, Pin, & Jouvelot, 2017), an online environment where the user can use Coq without installing any dependencies in their local machine. Fig. 2 shows the basic interface of jsCoq.

For complete beginners, like students with only some programming background in an object oriented language, CoqIDE might not have the most intuitive user interface. It might require a new mindset in order to get used to the interactive proving style and all the existing views. The top buttons, being a large set of arrows (visually), might confuse the user and increase the entry bar for newcomers.

jsCoq, on the other hand, provides a tutorial on its main page, so a beginner would be able to obtain an automated guidance for using the existing buttons and shortcuts. However, there are multiple usability issues (the placement of the buttons, the color scheme, window resizing, and the overall responsiveness of the layout are just some examples to mention).

Moving to the exiting plugins, vscoq ¹ provides an interface for Coq in the Visual Studio Code IDE ² (VSCode). When compared to CoqIDE, vscoq offers the same functionalities. But it lacks many interactive and liveness features supported in other programming languages plugins designed for VSCode. To list a few examples, smart navigation between proofs and views, refactoring/extraction suggestions, and robust error indicators are some of the missing features in the plugin.

One of the most popular plugin options is ProofGeneral (Aspinall, 2000), an extension for Emacs that provides users with an interactive proof environment for several proof assistants, including Coq. It offers multiple views, shortcuts, and editing facilities, including sophisticated navigation paths. It is

¹https://github.com/coq-community/vscoq
²https://code.visualstudio.com/
certainly a rich tool, but like multiple other niche open source tools, there is room for improvement in its interface.

There are other plugins and IDEs, such as Coqtail for Vim \(^3\), a Jupyter Notebook style interface \(^4\), CoqPIE (Roe & Smith, 2016) and Company-Coq (Pit-Claudel & Courtieu, 2016). All these options for the end-user come with their own interface issues.

2. An Investigation for a better UI/UX

As we saw in Sec. 1, most of the presented tools for Coq come with some sort of usability and interface limitations. There may be multiple reasons to find the same issue across different tools for proof assistants.

The first and more direct reason may be related to the lack of UI/UX experts in the research groups or teams developing these environments or plugins.

Another issue may be related to the target audience: are those interface issues specific to Coq? One could ask if other proof assistants have better tools.

To answer the previous question, it is worth investigating the environment for the Lean Theorem Prover. There is a sophisticated, fully online tutorial, which is suitable for complete beginners or people with previous experience in writing proofs, but new to Lean. Fig. 3 shows an overview of the tutorial environment.

Lean’s online tutorial interface comes with a handy menu of tactics, helpers, and easy access to its documentation. These interface elements provide the user (which we assume to be a newcomer) with a friendly environment, lowering the entry bar to the proof assistant.

It is also worth mentioning the Lean plugin for VSCode has rich integration with the IDE, but possibly because both projects belong to Microsoft. The support for Lean in other text editors is arguably limited.

Instead of adopting the existing interfaces of the Lean proof assistant as a baseline for the Coq tools, it is possible to look at another direction: how are people using these tool, and, more importantly, how are they writing their proofs?

From previous experience in the software development community, we know that large datasets can be the key to reveal existing problems in interfaces. Analyzing data is also a valid approach to enhance the overall user experience of a given tool.

An example of this heavily data-centered approach can be found for the Eclipse IDE (Murphy, Kersten, & Findlater, 2006), and also for other Java development environments (such as BlueJ with the Blackbox

\(^3\)https://github.com/whonore/Coqtail
\(^4\)https://github.com/EugeneLoy/coq_jupyter
project (Brown, AlTadmri, Sentance, & Kölling, 2018)). In these studies, there is a general idea of reconstructing the user steps by logging and persisting their full sequence of steps and actions when using the IDE.

However, up to this point in time, there is no record of any large data-centered user experience study that aims to understand how people are writing their proofs in Coq.

Not even other proof assistants have attempted this data approach. The closest the research community gets is the Archive of Formal Proofs (Blanchette, Haslbeck, Matichuk, & Nipkow, 2015) for the Isabelle proof assistant. It is a collection of publications, including the proofs themselves, but not a dedicated database of Isabelle proofs.

There has been prior work aiming a better understanding of how users write proofs. For Coq, there have been attempts with smaller groups of users (as in (Knobelsdorf, Frede, Böhne, & Kreitz, 2017), targeting students, and (Ringer, Sanchez-Stern, Grossman, & Lerner, 2020), targeting experienced users). Still, the final dataset for analysis is relatively small.

Using the idea of generating large datasets, one concrete path to efficiently trace an accurate set of issues in the user experience would be generating snapshots upon certain events in the tools for the Coq proof assistant. For example, pressing a button or activating a shortcut would generate a new entry on the dataset. These entries would capture the existing status of the proof (a text), the current output of the assistant, a cursor to indicate what has already been evaluated, and a timestamp. Together, these events would allow a full timeline reconstruction of the user’s steps. In online environments such as jsCoq, it would be possible to enable this kind of data collection by using a simple architecture of asynchronous HTTP requests. Upon every click, the data to be collected would be sent to a remote web server, which then would persist a record in a database. A similar approach has been described for the LearnOCaml platform (Canou, Cosmo, & Henry, 2017) at (Ceci, Tavante, Pientka, & Si, 2021).

Apart from this data-centered user study, better interfaces could come from more robust integrations with other editors and IDEs. This need could be the basis for creating a specific LSP-like (Language Server Protocol) for Coq. With a better integration protocol, standards could be established for providing text editors and IDEs with an appropriate experience for the users.

3. A Working Plan and Future Work
In Sec. 2, we visited some of the possible reasons for the poor UI/UX for proof assistants. In order to improve these tools, it would be possible to consider the following working plan, which tackles three main aspects:

1. Fix basic interface components - this topic targets mostly online tools; namely, jsCoq. Basic interface components refer to aspects such as buttons size and position, color scheme, easy access to documentation, proper highlighting, hiding unnecessary information from the user, responsive layouts, and, very importantly, considering accessibility options. Online tools, like jsCoq, are part of a web environment, and hence, we can apply prior well-known usability principles to it.

2. In parallel, it would be possible to start a draft of a LSP-like protocol for Coq. There is currently an ongoing discussion in the community. One key point to consider is that writing a proof might require different support than the settings we have for writing code for web development, for example. But at this time, we don’t know how such protocol should look like.

3. Lastly, also in parallel, it would be important to support a verified live programming environment. Projects like Hazel (Omar, Voysey, Chugh, & Hammer, 2019) use typed holes to provide live typing feedback for the user. The project uses a simplified functional programming language. It

---

5https://microsoft.github.io/language-server-protocol/
6https://github.com/ejgallego/coq-serapi/issues/26
would be interesting to extend its metatheory to formalize other languages like Coq, ensuring, for example, that all the provided hints are also formally verified.

The three base steps previously described could then be followed by the data-centered user study (described in Sec. 2). Combined with a proposal of the LSP-like protocol, it could be the key to achieving a more stable UI for Coq tools. This UI would be suitable for both beginners and advanced users.

An enhanced metatheoretical formalization, combined with the data analysis results, could lead to a concrete classification of proof builds and profiles. Ideally, the end user would benefit from automated hints adapted to their experience level (novices or experienced users, for example).

Fig. 4 summarizes a potential roadmap for achieving a better user experience when using proof assistants.

This paper aims to show that it is possible to build better tools for proof assistants. The enhancements could lower the entry bar for newcomers, and also better serve experience users. Moreover, there seems to be a whole new set of research topics involved: online proof environments, accessibility components (for web and offline tools), hint generation, standards/protocols for proof environments, proof build classification, etc. Whereas the roadmap previously mentioned is not fully completed, it describes initial concrete steps towards better tools for the Coq proof assistant.

The data analysis of the described dataset could be the groundwork for subsequent studies in the education field (how students learn to prove properties in proof assistants; how to design effective programming walkthroughs (Bell et al., 1994) for Coq), usability field, and even bring benefits for the machine learning domain, possibly complementing projects such as CoqGym (Yang & Deng, 2019)\footnote{https://github.com/princeton-vl/CoqGym}.

4. Acknowledgments
The author would like to thank the PPIG members for encouraging the publication of these ideas.

5. References


Roe, K., & Smith, S. F. (2016). Coqpie: An IDE aimed at improving proof development pro-