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.

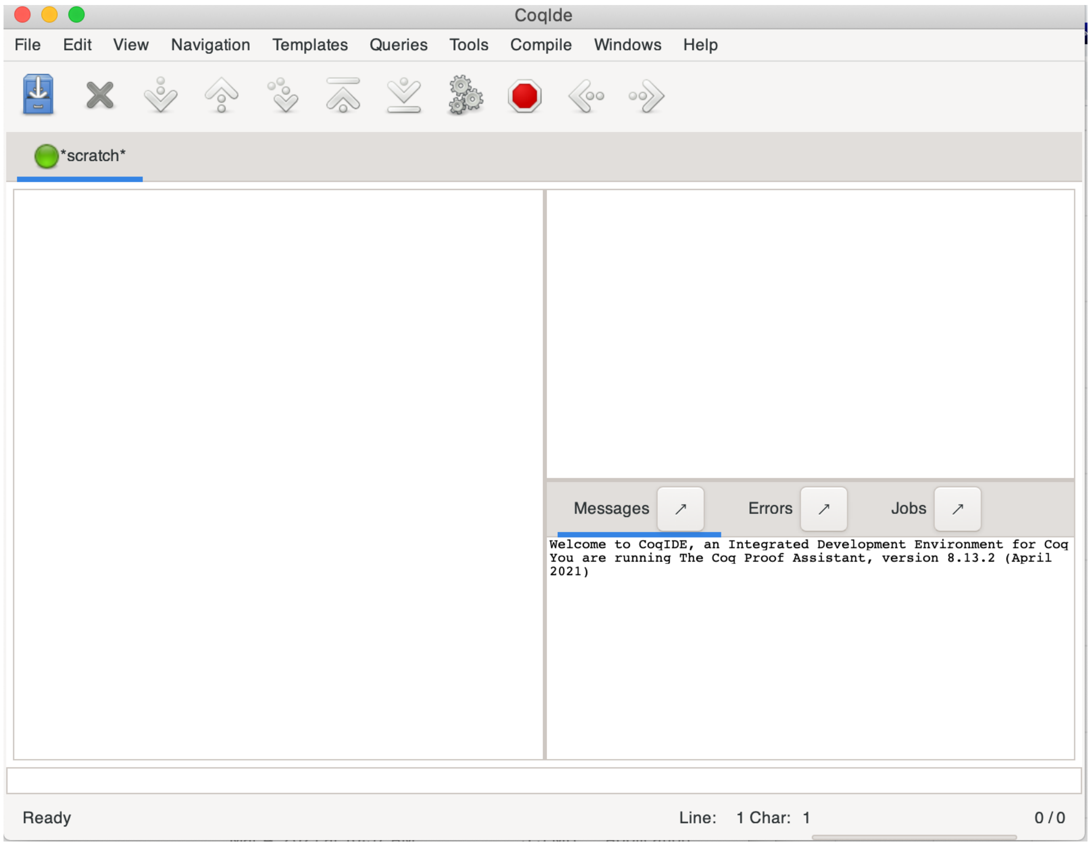
For complete beginners, like students with only some programming background in an object oriented language, CoqIDE might not have the most intuitive user interface. It might require a new mindset in order to get used to the interactive proving style and all the existing views. The top buttons, being 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.

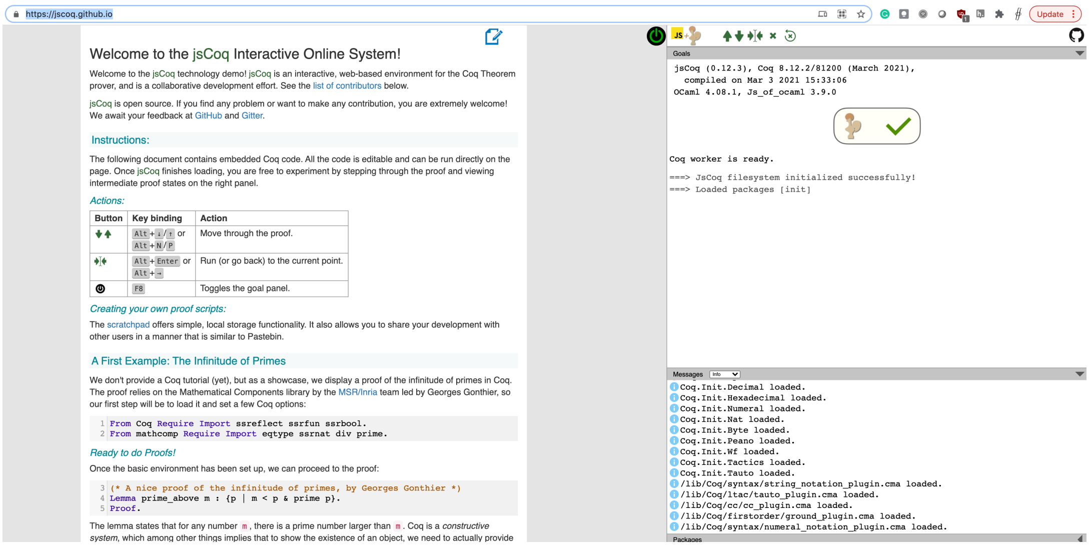
One of the most popular plugin options is ProofGeneral (Aspinall, 2000), an extension for Emacs that providers users with an interactive proof environment for several proof assistants, including Coq. It offers multiple views, shortcuts, and editing facilities, including sophisticated navigation paths.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

[1] https://github.com/coq-community/vscoq
[2] https://code.visualstudio.com/



```
Welcome to CoqIDE, an Integrated Development Environment for Coq
You are running The Coq Proof Assistant, version 8.13.2 (April
2021)
```

Figure 1 – CoqIDE



Welcome to the jsCoq Interactive Online System!

Welcome to the jsCoq technology demo! jsCoq is an interactive, web-based environment for the Coq Theorem prover, and is a collaborative development effort. See the list of contributors below.

jsCoq is open source. If you find any problem or want to make any contribution, you are extremely welcome! We await your feedback at GitHub and Gitter.

Instructions:

The following document contains embedded Coq code. All the code is editable and can be run directly on the page. Once jsCoq finishes loading, you are free to experiment by stepping through the proof and viewing intermediate proof states on the right panel.

Actions:

Button	Key binding	Action
⬇⬆	Alt+↓/↑ or Alt+N/P	Move through the proof.
➧|⬅	Alt+Enter or Alt+→	Run (or go back) to the current point.
⏻	F8	Toggles the goal panel.

Creating your own proof scripts:

The scratchpad offers simple, local storage functionality. It also allows you to share your development with other users in a manner that is similar to Pastebin.

A First Example: The Infinitude of Primes

We don't provide a Coq tutorial (yet), but as a showcase, we display a proof of the infinitude of primes in Coq. The proof relies on the Mathematical Components library by the MSR/Inria team led by Georges Gonthier, so our first step will be to load it and set a few Coq options:

```
From Coq Require Import ssreflect ssrfun ssrbool.
From mathcomp Require Import eqtype ssrnat div prime.
```

Ready to do Proofs!

Once the basic environment has been set up, we can proceed to the proof:

```
(* A nice proof of the infinitude of primes, by Georges Gonthier *)
Lemma prime_above m : {p | m < p & prime p}.
Proof.
```

The lemma states that for any number m, there is a prime number larger than m. Coq is a *constructive system*, which among other things implies that to show the existence of an object, we need to actually provide

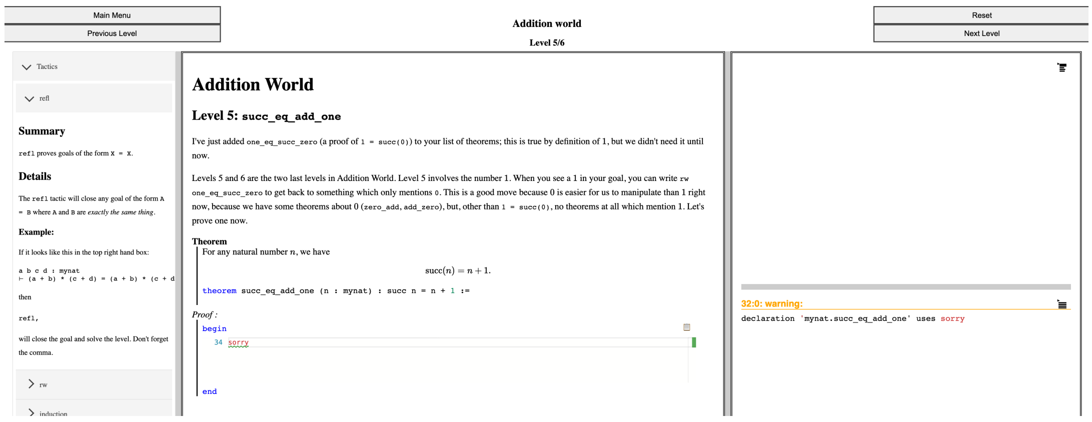
```
jsCoq (0.12.3), Coq 8.12.2/81200 (March 2021),
  compiled on Mar 3 2021 15:33:06
OCaml 4.08.1, Js_of_ocaml 3.9.0

Coq worker is ready.

===> JsCoq filesystem initialized successfully!
===> Loaded packages [init]
```

```
Coq.Init.Decimal loaded.
Coq.Init.Hexadecimal loaded.
Coq.Init.Numeral loaded.
Coq.Init.Nat loaded.
Coq.Init.Byte loaded.
Coq.Init.Peano loaded.
Coq.Init.Wf loaded.
Coq.Init.Tactics loaded.
Coq.Init.Tauto loaded.
/lib/Coq/syntax/string_notation_plugin.cma loaded.
/lib/Coq/ltac/tauto_plugin.cma loaded.
/lib/Coq/cc/cc_plugin.cma loaded.
/lib/Coq/firstorder/ground_plugin.cma loaded.
/lib/Coq/syntax/numeral_notation_plugin.cma loaded.
```

Figure 2 – jsCoq



Summary

`refl` proves goals of the form `X = X`.

Details

The `refl` tactic will close any goal of the form `A = B` where `A` and `B` are *exactly the same thing*.

Example:

If it looks like this in the top right hand box:

```
a b c d : mynat
⊢ (a + b) * (c + d) = (a + b) * (c + d
```

then

```
refl,
```

will close the goal and solve the level. Don't forget the comma.

Addition World

Level 5: `succ_eq_add_one`

I've just added `one_eq_succ_zero` (a proof of `1 = succ(0)`) to your list of theorems; this is true by definition of 1, but we didn't need it until now.

Levels 5 and 6 are the two last levels in Addition World. Level 5 involves the number 1. When you see a 1 in your goal, you can write `rw one_eq_succ_zero` to get back to something which only mentions `0`. This is a good move because 0 is easier for us to manipulate than 1 right now, because we have some theorems about 0 (`zero_add`, `add_zero`), but, other than `1 = succ(0)`, no theorems at all which mention 1. Let's prove one now.

Theorem
For any natural number n, we have

$$\mathrm{succ}(n) = n + 1.$$

```
theorem succ_eq_add_one (n : mynat) : succ n = n + 1 :=
```

Proof :

```
begin
  sorry


end
```

```
32:0: warning:
declaration 'mynat.succ_eq_add_one' uses sorry
```

Figure 3 – Lean Theorem Prover introductory tutorial

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 [5]) 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 [6]. 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

[5]https://microsoft.github.io/language-server-protocol/
[6]https://github.com/ejgallego/coq-serapi/issues/26

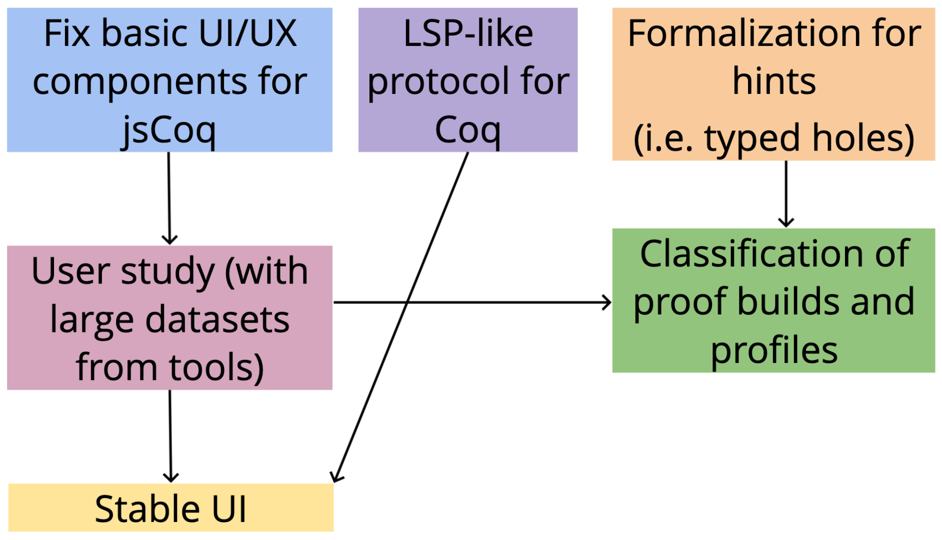

Fix basic UI/UX components for jsCoq
LSP-like protocol for Coq
Formalization for hints (i.e. typed holes)
User study (with large datasets from tools)
Classification of proof builds and profiles
Stable UI

Figure 4 – Working steps for building better tools for Coq.

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)[7].

4. Acknowledgments

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

5. References

Aspinall, D. (2000). Proof general: A generic tool for proof development. In S. Graf & M. I. Schwartzbach (Eds.), *Tools and algorithms for construction and analysis of systems, 6th*

[7]https://github.com/princeton-vl/CoqGym

international conference, TACAS 2000, held as part of the european joint conferences on the theory and practice of software, ETAPS 2000, berlin, germany, march 25 - april 2, 2000, proceedings (Vol. 1785, pp. 38–42). Springer. Retrieved from `https://doi.org/10.1007/3-540-46419-0\_3` doi: 10.1007/3-540-46419-0_3

Bell, B., Citrin, W., Lewis, C. H., Rieman, J., Weaver, R. P., Wilde, N., & Zorn, B. G. (1994). Using the programming walkthrough to aid in programming language design. *Softw. Pract. Exp.*, *24*(1), 1–25. Retrieved from `https://doi.org/10.1002/spe.4380240102` doi: 10.1002/spe.4380240102

Blanchette, J. C., Haslbeck, M. W., Matichuk, D., & Nipkow, T. (2015). Mining the archive of formal proofs. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, & V. Sorge (Eds.), *Intelligent computer mathematics - international conference, CICM 2015, washington, dc, usa, july 13-17, 2015, proceedings* (Vol. 9150, pp. 3–17). Springer. Retrieved from `https://doi.org/10.1007/978-3-319-20615-8\_1` doi: 10.1007/978-3-319-20615-8_1

Brown, N. C. C., AlTadmri, A., Sentance, S., & Kölling, M. (2018). Blackbox, five years on: An evaluation of a large-scale programming data collection project. In L. Malmi, A. Korhonen, R. McCartney, & A. Petersen (Eds.), *Proceedings of the 2018 ACM conference on international computing education research, ICER 2018, espoo, finland, august 13-15, 2018* (pp. 196–204). ACM. Retrieved from `https://doi.org/10.1145/3230977.3230991` doi: 10.1145/3230977.3230991

Canou, B., Cosmo, R. D., & Henry, G. (2017). Scaling up functional programming education: under the hood of the ocaml MOOC. *Proc. ACM Program. Lang.*, *1*(ICFP), 4:1–4:25. Retrieved from `https://doi.org/10.1145/3110248` doi: 10.1145/3110248

Ceci, A., Tavante, H. C. A., Pientka, B., & Si, X. (2021). Data collection for the learn-ocaml programming platform: Modelling how students develop typed functional programs. In M. Sherriff, L. D. Merkle, P. A. Cutter, A. E. Monge, & J. Sheard (Eds.), *SIGCSE '21: The 52nd ACM technical symposium on computer science education, virtual event, usa, march 13-20, 2021* (p. 1341). ACM. Retrieved from `https://doi.org/10.1145/3408877.3439579` doi: 10.1145/3408877.3439579

Gallego Arias, E. J., Pin, B., & Jouvelot, P. (2017). jsCoq: Towards hybrid theorem proving interfaces. In S. Autexier & P. Quaresma (Eds.), *Proceedings of the 12th Workshop on user interfaces for theorem provers, coimbra, portugal, 2nd july 2016* (Vol. 239, p. 15-27). Open Publishing Association. doi: 10.4204/EPTCS.239.2

Knobelsdorf, M., Frede, C., Böhne, S., & Kreitz, C. (2017). Theorem provers as a learning tool in theory of computation. In J. Tenenberg, D. Chinn, J. Sheard, & L. Malmi (Eds.), *Proceedings of the 2017 ACM conference on international computing education research, ICER 2017, tacoma, wa, usa, august 18-20, 2017* (pp. 83–92). ACM. Retrieved from `https://doi.org/10.1145/3105726.3106184` doi: 10.1145/3105726.3106184

Murphy, G., Kersten, M., & Findlater, L. (2006). How are java software developers using the elipse ide? *IEEE Software*, *23*(4), 76-83. doi: 10.1109/MS.2006.105

Omar, C., Voysey, I., Chugh, R., & Hammer, M. A. (2019). Live functional programming with typed holes. *Proc. ACM Program. Lang.*, *3*(POPL), 14:1–14:32. Retrieved from `https://doi.org/10.1145/3290327` doi: 10.1145/3290327

Pit-Claudel, C., & Courtieu, P. (2016, January). Company-coq: Taking proof general one step closer to a real ide. In *Coqpl'16: The second international workshop on coq for pl.* Retrieved from `http://hdl.handle.net/1721.1/101149` doi: 10.5281/zenodo.44331

Ringer, T., Sanchez-Stern, A., Grossman, D., & Lerner, S. (2020). Replica: REPL instrumentation for coq analysis. In J. Blanchette & C. Hritcu (Eds.), *Proceedings of the 9th ACM SIGPLAN international conference on certified programs and proofs, CPP 2020, new orleans, la, usa, january 20-21, 2020* (pp. 99–113). ACM. Retrieved from `https://doi.org/10.1145/3372885.3373823` doi: 10.1145/3372885.3373823

Roe, K., & Smith, S. F. (2016). Coqpie: An IDE aimed at improving proof development productivity - (rough diamond). In J. C. Blanchette & S. Merz (Eds.), *Interactive theorem proving - 7th international conference, ITP 2016, nancy, france, august 22-25, 2016, proceedings* (Vol. 9807, pp. 491–499). Springer. Retrieved from `https://doi.org/10.1007/978-3-319-43144-4\_32` doi: 10.1007/978-3-319-43144-4_32

Yang, K., & Deng, J. (2019). Learning to prove theorems via interacting with proof assistants. In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th international conference on machine learning, ICML 2019, 9-15 june 2019, long beach, california, USA* (Vol. 97, pp. 6984–6994). PMLR. Retrieved from `http://proceedings.mlr.press/v97/yang19a.html`