A Survey of Development Tools for Proof Assistants

Hjalte Sorgenfrei Mac Dalland *hjda@itu.dk*, Jakob Israelsen *jais@itu.dk*, Simon Green Kristensen *sigk@itu.dk*

Abstract

Compared to general-purpose programming, the field of proof engineering is a relatively niche field, with untapped potential for better tools and infrastructure. This paper explores possible features that could improve the user experience of proof engineering, the mechanization of proofs using computers. The inspiration for this paper was to improve the Visual Studio Code extension *VSCoq*, in order to make the experience of using Coq in Visual Studio Code better.

These feature ideas are gathered in two ways. The first method is a manual exploration of different IDEs and languages for mechanizing proofs, to seek inspiration for features that could be added to VSCoq. The second method consists of conducting interviews with four proof engineers, asking about aspects of their tools that have helped or frustrated them.

Some of the features that result from the exploration and interviews include automatic proof synthesis, counterexample generation, automatic formatting, and better discoverability.

We conclude that there are many novel and interesting feature ideas, found through investigating IDEs and interviewing users. Some were found that are also novel concepts for programming, such as the automatic naming of methods, which could also be applied to naming lemmas when mechanizing proofs.

I. Introduction

VERIFYING a proof's soundness can be as difficult as creating the initial proof. To increase confidence in the soundness of a proof, software was created to mechanize proofs and verify them by way of a small trusted kernel. One kind of proof that is particularly difficult is formal verification of software, where adherence to particular formal specifications is proven.

Around these trusted kernels, tools were developed to help proof engineers create their proofs. These tools include specification languages, Integrated Development Environments (IDEs), and documentation generation to create HTML or LaTeX from mechanized proofs.

While sharing many similarities with general-purpose programming, the relative niche of mechanizing proofs means that less time has been put into surrounding infrastructure, such as IDEs. This sometimes leads to a lacklustre user experience for proof engineers, as many features from modern IDEs that have come to be expected as standard are missing.

We, therefore, believe that there is a large untapped potential in IDEs for features supporting the mechanization of proofs. Furthermore, despite its relatively small size, the field of proof engineering has a multitude of languages and editors, with a fair variety of features. Each of these editors were created to cover potentially quite different needs, as each language has different requirements, possibilities, and target users. It may therefore be prudent to take inspiration from what has been done before, as the features of one language or editor might work well in another.

This paper aims to describe different features beneficial for specifically mechanizing proofs, with the explicit purpose of trying to improve VSCoq [1]. VSCoq is an extension for *Visual Studio Code* (VSCode) [2] created to support the creation of proofs with Coq, a popular proof management system.

In this paper, we start by describing our two methods of exploring possible feature ideas; investigating existing IDEs and languages, and interviewing users about their experiences. We then explore different features and tools that can enhance the VSCoq extension, such as improved error highlighting, methods for strengthening proof readability, and support for features like automatic theorem proving and counterexample generation. By examining these areas, we hope to provide a potential path of how to improve the VSCoq extension and enhance the experience for proof engineers using Coq in VSCode.

December 15, 2022

II. Methods

We have investigated possible features to support mechanizing proofs using two different methods. The first was to try out a selection of the most popular languages and proof assistants we could find, testing out various IDEs for them, and looking through their documentation, to find useful features already implemented in other languages. The second method was interviewing users of some of these tools, asking about what features they like or dislike in the languages they use. We gathered a list of potentially useful features from these two methods, which we present in section III.

A. Language and IDE Evaluation

We have investigated the following languages for their IDE features:

- Coq [3], an interactive theorem prover with a dependently typed specification language, Gallina
- Lean [4], a functional programming language with an interactive theorem prover
- Isabelle [5], a proof assistant that supports proof automation by using tools such as satisfiability modulo theories (SMT) solvers
- Agda [6], a dependently typed functional language and proof assistant
- Idris [7], a dependently typed programming language
- Dafny [8], a verification-aware imperative programming language

In each of these languages, we followed their tutorial and checked out a large project to see how the IDE features worked for both a beginner and when working on a complex project, as well as read through their documentation.

B. Interviews

We interviewed 4 users about their experience with proof mechanization, and what languages they used:

- Interviewee A: A researcher with 12 years of experience using Coq, as well as some experience with Dafny and Hol4.
- Interviewee B: A college student with about 2 years of experience using Lean and Coq, aswell as maintainer of a open source Coq project.
- Interviewee C: A researcher also working to develop Coq, with more than 7 years of experience using Coq, as well as a host of other languages, including Agda, Isabelle, and Lean
- Interviewee D: A researcher with a few years of experience of dabbling with Coq, as well as some experience using Spark and Agda.

We conducted semi-structured interviews, asking participants about their experience with using proof assistants, what features of the assistants they liked and disliked, and what IDEs they used for their respective languages and why. Select quotes from the interviews can be found in appendix A.

The interviewees consisted of three researchers and one college student who mainly use Coq for formal verification, but also have experience using other technologies like Lean, Hol4, and Dafny. Most of the interviewees were found by reaching out through the Zulipchat for Coq [9], which is a forum used by members of the Coq Community. The interviewees had varying amounts of experience with Coq, ranging from having used it sparingly for a few projects to a researcher with 12 years of experience with Coq.

III. Features found in languages and IDEs

This section describes the features found in our investigation of IDEs. It details some characteristics inherent to development tools like installation and responsiveness considerations, before moving on to unique automated features like proof checking, counterexamples, and proof synthesis.

A. Installation

A small but cool feature in Dafny is that the installation of Dafny and all required dependencies was done automatically when installing the VS Code extension for Dafny. It's especially useful in a teaching environment, where students can be told to install a plugin in their IDE and immediately start creating proofs. Compared to potential frustrations that can stem from a complicated setup, having the first impression be that it just works effortlessly, increases the chance that people will stick with the language.

While Coq is not difficult to install, having both a snap package and binary installers [10], it is not as smooth as Dafny for first-time users. This does track with the other tools and languages examined, as they all provide binaries, and most are installable via package managers.

B. Responsiveness

When moving around a Coq file in the editor, the user needs to *step through* proofs in the file, meaning that Coq needs to check whether each application of a tactic is valid. Upon reaching QED, the proof is checked by the kernel, which can be slow. While the editor can step through files in an acceptable amount of time, these checks do slow down interaction when the user needs to step to a certain part of their proof, instead of just placing the cursor at their desired editing location.

Languages like Lean, Isabelle, and Dafny are, in contrast, more responsive to interact with, in terms of time from editing a proof to the user knowing if the proof failed. In Lean, moving around in a file and looking at the info-view, the context showing proof status is updated almost instantly to match the cursor position, regardless of how the document location was reached. Isabelle checks proof asynchronously and automatically, meaning that the user does not have to tell the IDE to check if the step is valid. Features like these, potentially give the proof engineer a more seamless workflow, reducing frustration and increasing productivity and satisfaction.

Continuous checking of proofs: Dafny adopts an interaction model where the user does not input commands; they write their code and are automatically returned the results from the compiler and the automated theorem prover. Dafny's VSCode extension continuously checks that proofs are valid, which includes pre- and post-conditions of methods and functions. When an edit is done to a method or function, Dafny checks then automatically checks that all proofs depending on it are also still valid. This creates a highly interactive experience for the proof engineer, as they get quick feedback to show if their changes are correct, instead of manually initiating proof checking as is the case in VSCoq.

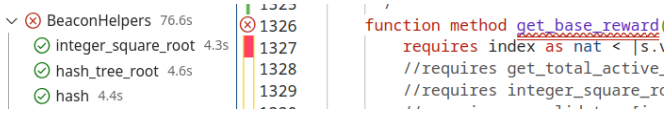

Fig. 1. Test view in VSCode showing on the left that the functions `integer_square_root`, `hash_tree_root` and `hash` succeeds, but that the file `BeaconHelper` has a function with an error. The error is shown on the right with `get_base_reward` reporting an error next to the line number and under the function name with a red squiggly.

The unit test view from VSCode, shown in figure 1, is used to give a clear overview of all the proofs and their result; if they are waiting, running, correct, or have failed.

When working across multiple files it has the limitation of only checking the current working file. But it should be possible to check all files, though it may present some performance issues if a lot of proofs suddenly have to be checked.

Dafny has both functions and methods where methods only guarantee the type of the function and its' post-condition and input has to satisfy the pre-condition. Changing a method implementation therefore only necessitates checking the method again and not anything that depends on it. Functions do not

require pre- and post-condition as the implementation of the function is checked when constructing proofs. This also means that when a function implementation is changed, a lot of proofs potentially have to be checked. This was obvious when making changes to ETH2.0 [11], a large project implementing the Ethereum protocol in Dafny, where it could take upwards of half a minute to run all proofs in a file due to everything being functions and not methods.

Continuously checking proofs in Coq would also be convenient, as some proofs can be slow to run. When editing a file with many proofs this could mean that it takes minutes to reach the bottom. If a definition changes at the top of the file and breaks a proof at the bottom of the file, it can be a while before the proof engineer knows, which can be a frustrating experience. Coq's asynchronous mode, which is used in CoqIDE, lessens the frustration by speeding up reevaluation, but proof checking still has to be started manually by the proof engineer [12]. In contrast, Dafny's continuous checking means that the proof engineer is informed about errors earlier and without interrupting the development flow.

C. Case code generation

When splitting on possibilities for an expression in pattern matching, being required to manually write all cases is cumbersome. This is why the programming languages Agda and Idris make it possible to automatically generate all inductive cases for a parameter with a single command when implementing a function. With Agda, the command has to be executed from a hole in the goal of a function or definition and inputting what to split on. In Idris, the split case command is executed by placing the cursor on the parameter to be split, which can be faster than Agda's method, as it does not require typing a name. Upon adding more cases for a type it is also possible to add these new cases to pattern matches, by splitting the functions again.

In figures 2 and 3, the before and after of performing case split in Idris on the argument `xs` can be seen. By placing the cursor on the `xs` argument in figure 2 and running the editor command `idris.case-split` (via `ctrl+alt+c`), the line is duplicated, with `xs` replaced with the inductive constructors `nil` and `_::_`.

```
reverse : List a -> List a
reverse xs = ?rhs
```

Fig. 2. Idris code showing the function `reverse` before performing a case split on `xs`.

```
reverse : List a -> List a
reverse [] = ?rhs_1
reverse (x :: xs) = ?rhs_2
```

Fig. 3. Idris code showing the result of a case split on the argument list `xs`.

Generating cases in a match statement in Coq functions would be similarly useful, as to avoid having to write out all cases manually.

D. Interactive Visualisation

Most scientific articles are static documents, and articles with a lot of math involved can be difficult to understand due to the burden of managing terse syntax and having to update a mental model of the state of proofs, etc. While static visualizations of proof states are used in articles, dynamic visualizations might be able to better convey proofs. Lean Infoview [13] can provide these dynamic visualizations in the case where the implementation of mathematical proofs is provided along with the article.

Lean Infoview is a project which allows Lean and the Lean VSCode extension [14] through the LSP to define user-defined custom widgets for visualizing details directly from within Lean projects. This is utilized to great effect in the Rubik's cube example [15], where the state of a Rubik's cube is rendered in 3D in the Infoview panel, which could be useful if trying to e.g. prove a solver algorithm to be correct, optimal, etc.

However, not all proof assistant users have web development skills, so creating these Infoview extensions requires either expertise in both proof assistants and web technologies or the commission of specialized proof engineers. One could imagine that, with time, a solid ecosystem with libraries of visualizations could become available for general data structures, but that remains to be seen.

Some effort has gone into interactive visualizations in Coq, such as with Prooftree [16]. Prooftree is an extension for Proof General [17] which visualizes proofs as trees. Branches correspond to sub-goals, with nodes indicating the tactics applied and the proof contexts they produce, splitting off new branches when tactics produce new sub-goals. Branches of accepted goals show up in green, admitted sub-goals are shown in red, and the current context is marked in blue. The tree can be used to easily overview the status of proofs and allows quick navigation throughout, which is especially helpful if the proof is hard to read due to a lack of formatting or bullet points (see section IV-A).

E. Automatic proof synthesis

A useful feature is for the editor to automatically suggest how to prove the lemma the user is working on. This already exists in some form as the *Hammer* tactics from Coq [18] and Isabelle [19], as well as other tools mentioned in section V. In Isabelle, the Hammer tactic works by feeding the goal and context into an external solver such as Z3 [20] or Vampire [21], and then trying to transform the proof back into a series of tactics. Transforming the proof back into tactics uses the prover Metis, and may fail if it is too complicated for Metis, in which case it runs forever or gives an error [22].

For general-purpose programming, there exists the similar concept of program synthesis. Program synthesis does however have the limitation that the solution will only be as good as the test suite for the program, and potentially erroneous solutions may be synthesized if the test cases are not covering. This is not a problem in Coq, as the synthesized proof only needs to satisfy the corresponding lemma. Coq's trusted kernel checks

that the proof is valid; if the proof is conclusive, then it is correct [23].

Idris is capable of code generation based on type information, via *Proof Search* [24]. By simple brute force, it attempts to construct a value using existing constructors, naively matching the expected type recursively on parameters. When parameters are reached which cannot be simply constructed from the local context, Idris instead inserts a hole. While this may produce values of the correct type, it does not produce values that necessarily conform to the model or implementation the developer wants.

For example, trying to fill the holes `?rhs_1` and `?rhs_2` in figure 3 just fills with the argument which conforms in type, as can be seen in figure 4. As the `List` type doesn't depend on the length of the list, the list tail suffices to fill the hole. If done again with a length-dependent list type like `Vect`, Idris fills the hole with a concatenation of `x` and `xs`, again conforming to the type rather than the intention.[1]

```
reverse : List a -> List a
reverse [] = []
reverse (x :: xs) = xs
```

Fig. 4. Idris code showing the result of filling holes in the `reverse` function

One interviewee (see A-B) suggested that most of the time when writing Coq proofs, the user is thinking, rather than writing. While the user is thinking, generally the IDE is not doing anything, except waiting for input. This idle CPU time could be spent trying to synthesize proof for the current lemma.

In section V-B we mention related work that implements proof synthesis for Coq. If implemented in VSCoq, then from a proof engineer's point of view, the proof synthesis could just run in the background. Then if a solution is found, a prompt would be shown.

F. Counterexamples

When attempting to prove lemmas, one might have based it on wrong assumptions, making it unprovable. In these cases, it can be practical to come up with a counterexample, in order to not waste time on proving something that is unprovable. This can be especially useful, as some lemmas are easier to disprove than prove. They usually have to hold true for every conceivable value, hence any single value could potentially disprove them. Generating these counterexamples without the input of the user can therefore help avoid wasting effort.

Isabelle has this feature as *Nitpick* [25], which can take a lemma and via external theorem provers such as Z3 [20] check the lemma's soundness. If the lemma is not sound, it will send a negation of it to the theorem prover, which is used to generate a counterexample. This counterexample can then inform the user that Isabelle can't prove the lemma, along with an example of how it fails. The proof engineer can then use this counterexample to guide debugging the lemma and finding out where their assumptions fail.

[1]Admittedly, reversing a vector requires explicitly using a proof of commutativity for addition in Idris, which is beyond just constructing an expression.

Counterexamples are also a feature of Dafny, where if a pre-/postcondition or assertion fails, Dafny can generate a counterexample. The editor then annotates variables with the values generated by the counterexample, changing the annotation on each line of the function as the variables change due to instructions.

These counterexamples are generated by the underlying theorem prover and can be switched on and off by a single keyboard command [26]. As such the counterexamples can be shown to increase knowledge of why a proof fails, or disabled to decrease visual noise. By making it quick and easy to enable or disable counterexamples, it decreases the mental load of using them, thereby allowing the proof engineer to focus more on writing code. In comparison, Isabelle necessities running a different command, which means that the proof engineer has one more thing to think about, even if it is a small thing.

IV. Features suggested in interviews

Through interviews with members of the proof engineering community, we found ideas for features, based on upon their frustrations and wants. While it is harder to say how feasible the features would be to implement for Coq, we have speculated on routes to implement them based on related projects.

A. Proof readability

One of the problems with reading Coq proofs is that the formatting is often different for each project. Running automatic formatting and changing the code to a format more familiar to the proof engineer should make it easier for them to understand. Some lemmas might be helped by being split out over multiple lines, which should be possible to do in an auto formatter.

Another possibility is to automatically indent and use the appropriate bullets [2] for cases. For example, if a proof at one point branches into multiple cases, like with an induction proof, it can create multiple goals. In Coq, it is valid to solve all goals without adding structure with brackets or bullets, which hides from the reader that multiple goals were involved. [3] An example can be seen in figures 5 and 6, where two otherwise identical proofs of $n+0=n$ are formatted without and with bullets indicating sub-goals.

Another problem is that if a proof has errors, you have to evaluate the specific line to see its errors. This is a worse developer experience compared to the asynchronous experience offered by common editors, where errors are underlined in red. Red squiggly lines make it possible to see multiple errors at the same time and hover over them to see their cause. This is the case in Dafny, where failing conditions are underlined to clearly show what obligations the proof engineer still has to prove [28].

[2]Bullet is Coq's name for symbols used to indicate lists. There are three allowed symbols: `-+*`.

[3]For compound tactics [27], tactics delimited with `;` as in `split; auto`, it still might make sense to keep everything on one line, as the compound tactic might only leave a single goal unsolved.

```
Theorem add_0' : forall n:nat, n + 0 = n.
Proof.
  intros n. induction n.
  reflexivity. simpl.
  rewrite -> IHn. reflexivity.
Qed.
```

Fig. 5. Aside from knowing what the tactic does, it is not legible that the first sub-goal, introduced by induction on n, is concluded by `reflexivity`.

```
Theorem add_0 : forall n:nat, n + 0 = n.
Proof.
  intros n. induction n.
  - reflexivity.
  - simpl. rewrite -> IHn. reflexivity.
Qed.
```

Fig. 6. In this example, bullet points are used to illustrate which case is solved.

B. Discoverability

When writing proofs in Coq with a codebase that a proof engineer is not intimately familiar with, it can be a problem to discover what possibilities they have to further the state of the proof. This was both a problem we encountered ourselves when learning Coq and a problem reported by interviewee A-A. [4]

A common solution to this problem is intelligent code completion such as IntelliSense [29], where the IDE suggests the best matches for functions and parameters based upon type information, surrounding lines and increasingly also AI [30]. These matches are then narrowed down as text input is provided, acting like a search.

While VSCoq and CoqIDE do provide suggestions as you type, those suggestions anecdotally are not that relevant. We presume that it performs a simple text search through tokens used in open documents, enhancing the suggestions with a curated list of often-used keywords and tactic names.

In order to provide relevant suggestions, it would be necessary to implement some heuristics for both what type of text token would be relevant, and from there which token best matches the input so far. Token type would likely have to be inferred from what context the text cursor resides in, whether that is a function implementation, a logical proposition, or the body of a proof.

One approach to finding usable lemmas would be to use the command *Search* [31]. Search works by allowing the users to input filters that are used on goals and the global context to find objects that satisfy these conditions. An example of this from the Coq documentation is `Search "_assoc".`, _ meaning wildcard, which would return all identifiers ending in ”assoc”; or `Search (_ ?n ?m = _ ?m ?n).` which finds commutativity lemmas. By transforming the current goal or part of the goal into a search query, Search could be used to generate suggestions for an intelligent auto-complete. One way suggestions could be found, would be the method that the `auto` tactic uses. By recursively considering the outermost or first construct of the current goal, it tries to apply tactics and lemmas which match these constructs [32].

[4]Part of the problem may be that the interviewee did not use the Search command in Coq. Through the Search command, it's possible to search for proofs by their form which greatly speeds up finding them, if a bit hampered by a cumbersome syntax at times.

C. Automatic Naming

Giving appropriate names to lemmas and theorems is important for the readability of code, especially if the reader is unfamiliar with the code in question. Good naming can be difficult and time-consuming, and it is made harder by different conventions for naming. Using machine learning to give good names is a possible way to make this easier. Besides just good naming, if this machine learning is based upon proofs made by the community, this would hopefully also nudge the community towards similar naming conventions, making it easier for users to read each other's proofs. This is also a fairly unexplored territory in general programming. To the best of our knowledge, no IDEs or editors have this functionality, although some research has been made into the matter of automatically naming methods in Java [33].

Such a project was created by Pengyu Nie et al. in 2020. *Roosterize* [34] uses deep learning to suggest lemma names in Coq and uses a model pre-trained on the MathComp [35] project. It works by learning the naming scheme used in the user's project and suggesting lemma names that may be improved and a suggestion with an improvement.

Roosterize does not work with newer versions of Coq after 8.10.2, and its installation process is not as simple as just installing the plugin in VSCode. Integrating it into VSCoq and making its installation process automatic would make it easier for potential users to adopt the tool.

The inverse idea was also briefly mentioned. Using machine learning, it might be possible to give a name for a lemma and have the lemma constructed automatically. This would be more likely to work for simple lemmas proving associativity, commutativity, idempotency, and the like.

V. Related Work

Earlier work has been done on improving the developer experience when working with Coq. One such project is *Coqoon* [36] by Faithful et al. which improves the management of larger Coq codebases. Without Coqoon, project management is left to the proof engineers and is usually done with Makefiles and shell scripts. Instead, Coqoon utilizes the Eclipse editor to construct dependencies between files in the project and automatically check changes. This also allows for using Coq's quick compilation mode to check the entire project in parallel, greatly speeding up proof checking [36].

Coqoon also enables OCaml [37] support in the same editor, which is useful when working with Coq projects that rely on OCaml plugins such as the *SSReflect* [38] or *LTac* Coq core plugins. This was achieved with the Eclipse plugin *OcaIDE*, which has a VSCode counterpart in *OCaml Platform*. It should therefore be possible to reach a similar level of OCaml integration in VSCode compared to Coqoon.

However, as of 2022 Coqoon is sadly unsupported and does not appear to work with versions of Coq later than 8.5. The API Coqoon relies on for communication with Coq, PIDE, was implemented in the PIDEtop project which is also left unsupported [39]. This means Coqoon can only be used with older versions of Coq, making it less desirable to active developers.

Coqoon is also implemented in Eclipse, which has since then fallen out of favour with many developers, being replaced by VS Code. [5]

A. Formatting via Machine Learning

A project by Pengyu Nie et al. seeks to format code by using neural networks [42]. It works by learning the conventions applied elsewhere in a project and formatting the code accordingly. This is similar to what we suggest in section IV-A, but where we suggest a method to make a foreign proof more readable, Pengyu Nie et al.'s method makes a project overall more readable by making the formatting similar in all proofs.

B. Proof synthesis via Machine Learning

Proof synthesis is the concept of proofs being generated by a machine, which means the user will not have to write the proof themselves. Even partial proof synthesis is useful, solving some sub-goals of a proof, possibly allowing users to save time on creating proofs and leading to faster development.

One of the approaches for proof synthesis is to use machine learning to improve solvers. *CoqGym*, a learning environment for Coq, has a dataset containing human-written proofs extracted from community projects developed with Coq and was used to train the deep learning-based model ASTactic [43]. With ASTactic it was possible to prove theorems that were not previously proven by automatic tools. A subset of CoqGym's dataset is also used as a benchmark to evaluate further work in automatically generating proofs.

TacTok [44] by First et al. is one such project which uses semantic aware search with machine learning to generate proofs. It works by evaluating the partial proof state generated when executing partial proof scripts. The search by TacTok is biased towards proof steps that are likely to work, ranked by the frequency they occur in successful proofs. The frequency of successful proof steps was learned by doing supervised learning on CoqGym.

Passport [45] is a project by Sanchez-Stern et al. which enriches TacTok by adding identifier information to the proof search. The project encodes identifiers in three different ways. These identifiers are then used to enrich TacTok to inform the search and expand the number of proofs it can synthesize.

Combined with Roosterize, Passport could potentially increase the likelihood of creating a successful proof if Roosterize found a better name.

[5]Back in 2016 Eclipse was the most popular easily extendable IDE with 22.7% of developers using it as per Stack Overflow's yearly survey [40]. In 2022, Eclipse has fallen to 12.6% and VSCode is used by 74.5% of developers [41].

Diversity-Driven Automated Formal Verification (DIVA) [23] is another project by First et al. which differentiates itself from TacTok by using a diverse set of models to allow for better variability. Because Coq is nearly infallible if any of these diverse models yield a successful proof, then this approach is good enough. The rationale in DIVA is therefore that increasing the number of diverse models yields better results, as more possible proofs are explored increasing the chance of finding a correct one. On the CoqGym benchmark, DIVA achieves the best result to date, being able to prove 33.8% of the theorems when complemented by CoqHammer [18].

VI. Conclusions

While proof mechanization is relatively niche, our survey of IDE features has yielded a lot of results. These range from features that help onboard new users and improve code understanding, to simplifying and speeding up the process of mechanizing proofs, showing that there are many ways of improving the developer experience. Some of these features are novel even to general-purpose programming such as the naming of lemmas.

Most of all, this survey shows some development effort in the scene, even in terms of bringing innovation to Coq. While most of the other tools and languages do some things better than VSCoq and Coq, VSCoq is ripe for major improvements to frog leap its competitors.

A. Further Work

Many of the features listed could make for interesting topics of research, not only to implement as described here but also for experimenting with how the user would interact with them. Specifically, Proof Synthesis is already a field where much research has been conducted, but it would be interesting to see if there is progress to be made by combining the usage of Roosterize and Passport to increase the chance of finding a proof. Interactive proof visualization is another rich topic, as there are many types of proofs that would benefit greatly from having a visual example or visual proof to help explain the logic to a reader. However, the use case for any particular type of visualization might be narrow, and it could be difficult to create widely applicable forms of visualization.

References

[1] Maxime Dénès. (2022) VSCoq. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=maximedenes.vscoq
[2] M. Corporation. (2022) Visual Studio Code. [Online]. Available: https://code.visualstudio.com/
[3] Coq maintainers at INRIA-Rocquencourt, "Coq." [Online]. Available: https://coq.inria.fr
[4] Microsoft Research. (2022) Lean theorem prover. [Online]. Available: https://leanprover.github.io/
[5] University of Cambridge and Technische Universität München et al. (2022) Isabelle. [Online]. Available: https://isabelle.in.tum.de/
[6] The Agda Wiki. [Online]. Available: https://wiki.portal.chalmers.se/agda/pmwiki.php
[7] Edwin Brady. (2022) Idris: A language for type-driven development. [Online]. Available: https://www.idris-lang.org/
[8] Microsoft Research. (2022) Dafny. [Online]. Available: https://dafny.org/
[9] (2022) Recent conversations - Coq - Zulip. [Online]. Available: https://coq.zulipchat.com/

[10] Coq maintainers at INRIA-Rocquencourt, "Install Coq." [Online]. Available: https://coq.inria.fr/download
[11] F. Cassez, J. Fuller, and A. Asgaonkar, "Formal verification of the ethereum 2.0 beacon chain," *CoRR*, vol. abs/2110.12909, 2021. [Online]. Available: https://arxiv.org/abs/2110.12909
[12] E. Tassi. (2022) Asynchronous and Parallel Proof Processing. [Online]. Available: https://coq.inria.fr/refman/addendum/parallel-proof-processing.html
[13] G. Ebner, C. Lovett, and W. Nawrocki. Lean 4 Infoview. [Online]. Available: https://www.npmjs.com/package/@leanprover/infoview
[14] leanprover. Lean 4 VSCode Extension. [Online]. Available: https://github.com/leanprover/vscode-lean4
[15] G. Ebner, C. Lovett, and W. Nawrocki. lean-4-samples: Rubik's cube visualiser. [Online]. Available: https://github.com/leanprover/lean4-samples/tree/main/RubiksCube
[16] H. Tews. (2017) Proof tree visualization for Proof General. [Online]. Available: http://askra.de/software/prooftree/
[17] The PG dev team. (2022) Proof General - A generic Emacs interface for proof assistants. [Online]. Available: https://proofgeneral.github.io/
[18] Ł. Czajka and C. Kaliszyk, "Hammer for coq: Automation for dependent type theory," *Journal of automated reasoning*, vol. 61, no. 1, pp. 423–453, 2018.
[19] University of Cambridge and Technische Universität München et al. (2022) Sledgehammer. [Online]. Available: https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html
[20] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.
[21] L. Kovács and A. Voronkov, "First-Order theorem proving and vampire," in *Computer Aided Verification*, ser. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35.
[22] J. C. Blanchette and L. C. Paulson. (2022) Hammering away. a user's guide to sledgehammer for isabelle/hol. [Online]. Available: http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/sledgehammer.pdf
[23] E. First and Y. Brun, "Diversity-driven automated formal verification," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 749–761. [Online]. Available: https://doi.org/10.1145/3510003.3510138
[24] I. Maintainers. (2022) Pattern Matching Proofs. [Online]. Available: https://docs.idris-lang.org/en/latest/proofs/patterns.html
[25] J. Blanchette and T. Nipkow, "Nitpick: A counterexample generator for higher-order logic based on a relational model finder," in *International Conference on Interactive Theorem Proving*, vol. 6172, 07 2010, pp. 131–146.
[26] A. Chakarov, A. Fedchin, Z. Rakamarić, and N. Rungta, "Better counterexamples for dafny," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 404–411. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-99524-9_23
[27] Coq maintainers. Basic notions and conventions. [Online]. Available: https://coq.inria.fr/distrib/current/refman/language/core/basic.html#term-sentence
[28] M. Research, "Dafny documentation," 2022. [Online]. Available: https://dafny.org/latest/DafnyRef/DafnyRef#sec-verification-debugging
[29] "Intellisense," 2022. [Online]. Available: https://code.visualstudio.com/docs/editor/intellisense
[30] C. Caldwell, "Ai-assisted intellisense for your team's codebase," 2019. [Online]. Available: https://devblogs.microsoft.com/visualstudio/ai-assisted-intellisense-for-your-teams-codebase/
[31] Coq maintainers. Commands - search. [Online]. Available: https://coq.inria.fr/refman/proof-engine/vernacular-commands.html#coq:cmd.Search
[32] T. Zimmermann, J. Fehrle, and P.-M. Pédrot. (2022) Programmable proof search. [Online]. Available: https://coq.inria.fr/refman/proofs/automatic-tactics/auto.html
[33] J. G. Shusi Yu, Ruichang Zhang, "Properly and Automatically Naming Java Methods: A Machine Learning Based Approach," in *International Conference on Advanced Data Mining and Applications*, Shanghai, China, 2012. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-35527-1_20
[34] P. Nie, K. Palmskog, J. J. Li, and M. Gligoric, "Roosterize: Suggesting lemma names for coq verification projects using deep learning," *CoRR*, vol. abs/2103.01346, 2021. [Online]. Available: https://arxiv.org/abs/2103.01346
[35] Mathematical Components Team and Inria-Microsoft Research Joint Center. Mathematical Components. [Online]. Available: https://math-comp.github.io/
[36] A. Faithfull, J. Bengtson, E. Tassi, and C. Tankink, "Coqoon An IDE for interactive proof development in Coq," in *TACAS*, Eindhoven, Netherlands, Apr. 2016. [Online]. Available: https://hal.inria.fr/hal-01242295
[37] Inria. (2022) OCaml. [Online]. Available: https://ocaml.org/
[38] G. Gonthier, A. Mahboubi, and E. Tassi, "A Small Scale Reflection Extension for the Coq system," Inria Saclay Ile de France, Research Report, 2008. [Online]. Available: https://hal.inria.fr/inria-00258384
[39] Coq/PIDE. PIDEtop. [Online]. Available: https://bitbucket.org/coqpide/pidetop/src/PIDEtop/
[40] Stack Overflow. (2016) Stack Overflow Developer Survey 2016. [Online]. Available: https://insights.stackoverflow.com/survey/2016
[41] ——. (2022) Stack Overflow Developer Survey 2022. [Online]. Available: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment
[42] P. Nie, K. Palmskog, J. J. Li, and M. Gligoric, "Learning to format coq code using language models," *CoRR*, vol. abs/2006.16743, 2020. [Online]. Available: https://arxiv.org/abs/2006.16743
[43] K. Yang and J. Deng, "Learning to prove theorems via interacting with proof assistants," *CoRR*, vol. abs/1905.09381, 2019. [Online]. Available: http://arxiv.org/abs/1905.09381
[44] E. First, Y. Brun, and A. Guha, "Tactok: Semantics-aware proof synthesis," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428299
[45] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer, "Passport: Improving automated formal verification using identifiers," 2022. [Online]. Available: https://arxiv.org/abs/2204.10370

APPENDIX A
QUOTES FROM INTERVIEWS

A. Suggestions

I have mostly used Proof General, and sometimes I have a proof obligation where it is not so obvious how to continue. Have I maybe unfolded a definition I shouldn't, and should I do some kind of rollback, or can I continue? I know that when you write in a general language, for example, VS Code, you have IntelliSense where you can write object dot, and then you get method names. This is less the case in Coq, for example, if you are in a proof, then what can you do from here?

B. Proof synthesis

I wonder how difficult it would be to prove something in Coq if you have a finite amount of primitives if you can make a proof searcher that tries to prove from the smallest proof and outwards and it can run in parallel with you proving things. Maybe some of the proofs can be cached, so if you are making an induction proof, and you have a subproof, this proof searcher can tell you how to make the proof when you get there. I imagine that most of the time in Coq, you are just staring at a screen and thinking about the proof, and that is a lot of processor time that could be spent better.