# HenBlocks: Structured Editing for Coq

BERNARD BOEY* and MICHAEL D. ADAMS†, Yale-NUS College, Singapore

When writing a computer program, we often use unit tests to check correctness. However, unit testing is insufficient to detect all errors, resulting in the potential for software bugs. A better way to demonstrate correctness is to use proofs to guarantee that our program is bug-free. One popular way of writing proofs is using the Coq Proof Assistant. However, there are a number of pain points in using it, which affects beginners most.

Structured editors, which allow the user to manipulate structured blocks corresponding to the abstract syntax of a program, have been used to make programming more accessible to beginners. However, they have not been applied to proving thus far.

The objective of this capstone is to build an interactive graphical user interface for the structured editing of Coq proofs. In this thesis, we present HenBlocks (available at https://henblocks.github.io), a web-based fully-fledged structured editor that allows users to write Coq proofs by manipulating blocks. We conclude that structured editing is a promising approach to proof writing that warrants more exploration, development, and testing.

Additional Key Words and Phrases: structured editor, visual programming, theorem prover, proof assistant, program verification, user interfaces

## 1 BACKGROUND

Readers are assumed to have a computer science background with good knowledge in functional programming and some knowledge in proving.

Unit testing involves writing a set of test cases for a corresponding portion of code that we wish to test. However, unit testing is not exhaustive. Although programmers try to test all possible code paths (i.e. maximising test coverage), it is infeasible to enumerate and test all cases. Thus, passing a unit test does not guarantee that a function is correct. As explained by Edsger W. Dijkstra, "testing shows the presence, not the absence of bugs" [2].

To combat the aforementioned problems with unit testing, we can use formal verification. This is done by first creating a formal specification of what we want our program to do. Then, by writing a proof that demonstrates that our program fulfills the specification, we can guarantee the correctness of our program. In order for a proof to be able to verify a program, the proof itself needs to be formally verified. This is often done via a proof assistant that mechanically checks the proof, and oftentimes provides additional features such as automatic theorem proving procedures.

Coq is an interactive theorem prover that allows users to write proofs interactively and check the state of the proof at each step. Upon completing a proof, Coq allows the user to extract a certified program in OCaml. Users usually interface with an Integrated Development Environment (IDE) such as CoqIDE (ships with Coq), Emacs, VSCode, or Vim.

Coq uses the specification language Gallina (which also doubles as the progamming language of Coq). The syntax of Gallina is based on OCaml. However, unlike OCaml, Gallina is a dependently typed programming language, has no imperative features, and is pure (i.e. there are no side effects). As a dependently typed[1] programming language, Gallina can express universal (i.e. $\forall$) and existential quantification (i.e. $\exists$).

*ACM Student Member Number: 9117317

†Research advisor

[1]Dependent types are types whose definition depends on a value

Authors' address: Bernard Boey, bernard@u.yale-nus.edu.sg; Michael D. Adams, michael.d.adams@yale-nus.edu.sg, Yale-NUS College, Singapore.

While the user must write specifications and programs in Gallina, they can choose to write proofs either using tactics (via the Ltac language) or directly providing proof terms. Tactics are the more common option (especially for beginners), as they abstract away complexity and allow us to build proof terms incrementally. Using proof terms "requires more expertise and is usually tedious" [8].

Coq has been used for a variety of applications, including building a formally verified C compiler, CompCert, which has been demonstrated to be more reliable than other common C compilers that have not been formally verified [11]. Coq has also been used to prove mathematical theorems such as the four colour theorem [7].

*Structured editing* is manipulation of underlying text content in a syntax-directed manner. Instead of the user making low-level edits by directly modifying text, the editor helps them make higher-level edits that require awareness of the syntax of the content. On one hand, we have text editors with some structured editing support (e.g. IDEs such as IntelliJ IDEA, Emacs, and VS Code). On the other hand, we have fully-fledged structured editors (e.g. Scratch, Hazel). We focus on fully-fledged structured editors, where it is usually not possible to have incorrect syntax, because the editor generates output syntax from higher level representations.

## 2 MOTIVATION

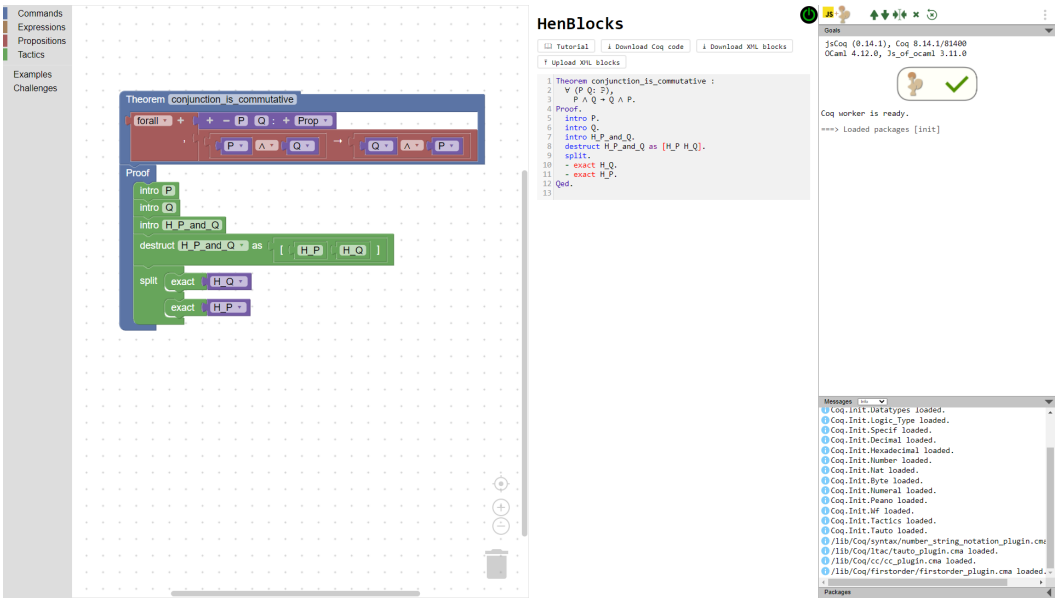
There are a number of pain points in using Coq. First, the **type system is complex and difficult to understand**, such as the use of dependently typed programming. Such complexities contribute to the difficulty in making "proper mental models for what happens 'behind the scenes' when [interacting] with a proof assistant" [8]. Second, there is **difficulty in learning new specification and tactic languages** (i.e. Gallina, Ltac). While seemingly similar to their functional programming counterparts (i.e. OCaml), such languages have different rules and a tremendous amount of new vocabulary. For example, Coq tactics "have unstructured names and are therefore hard to remember" [3]. Third, there is **friction in the user experience** (e.g. incomprehensible syntax error messages). Based on our research, existing interfaces for Coq or theorem proving (e.g. Prooftree [10], Proof-by-pointing [1], Actema [4], PeaCoq [8], Chick [8]) do not sufficiently simplify the learning process. Additionally, fully-fledged structured editing has not been applied to Coq thus far.

## 3 SOLUTION

We present HenBlocks, a web-based fully-fledged structured editor for Coq built using the Blockly library [9] and jsCoq [6]. The primary target audience for HenBlocks is undergraduate students who have some experience with functional programming but with little or no experience in proving. The intended use case is for such students to learn, discover, and practise proving with HenBlocks, and eventually transition to writing textual proofs via a text editor such as CoqIDE or Emacs. HenBlocks is freely accessible at https://henblocks.github.io. The source code can be found at https://github.com/henblocks/henblocks.github.io. The user interface is divided into four sections from left to right: 1) Toolbox (expandable panel containing all types of blocks that can be used), 2) Workspace (where the user rearranges and modifies blocks), 3) Code (generated Coq code from the blocks), and 4) Goals.

## 4 DESIGN AND IMPLEMENTATION

HenBlocks provides a number of structured editing features. First, we have **variable dropdowns**, which allow the user to select an identifier (e.g. theorem name, variable name, constructor, or hypothesis), that is guaranteed to be in scope, from a pre-populated dropdown list. Second, whenever the user modifies the name of an identifier, all subsequent references are **automatically renamed**.




Third, when the user selects a specific intro pattern (e.g. for a `destruct` tactic), HenBlocks **automatically creates slots** for the correct number of **subgoals**. Fourth, when the user selects a constructor from the dropdown list, HenBlocks **automatically creates slots** for the correct number of **arguments**.

## 5 DISCUSSION

The main limitation of HenBlocks is the potential for visual clutter. Additionally, dragging and dropping is slower than typing, and only a limited number of constructs/tactics are supported. However, this limitation is somewhat mitigated by the fact that HenBlocks is intended for beginner users and that users should have an "exit strategy" for transitioning to text editors [5]. The most pressing future work involves rigorous user testing of HenBlocks to evaluate its effectiveness (e.g. via A/B testing). Additionally, we need to develop HenBlock further to support more tactics and constructs, and provide more structured editing features. Lastly, there are user experience improvements that can be made.

In conclusion, we have made a novel contribution by applying fully-fledged structured editing to proof writing. We have also developed advanced structured editing features by providing scoped variable dropdown selection, automatic renaming, automatic slots for subgoals, and automatic slots for constructor arguments. Fully-fledged structured editing is a promising approach to proof writing that warrants more exploration, development, and testing.

## ACKNOWLEDGMENTS

This project would not have been possible without the supervision of Michael D. Adams and his immense guidance and support. Additionally, this project has benefited from the tremendous assistance of jsCoq co-author Emilio Jesús Gallego Arias, who provided technical advice and helped experiment with new features. Part of the work published here is derived from a capstone project submitted towards a BA/BSc from, and financially supported by, Yale-NUS College, and it is published here with prior approval from the College.

## REFERENCES


[1] Yves Bertot, Gilles Kahn, and Laurent Théry. 1994. Proof by pointing. In *Theoretical Aspects of Computer Software*, Masami Hagiya and John C. Mitchell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 141–160.

[2] John Buxton and Brian Randell. 1970. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO.*

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

[4] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. A Drag-and-Drop Proof Tactic. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3497775.3503692

[5] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 49–50. https://doi.org/10.1109/BLOCKS.2015.7369000

[6] Emilio Jesús Gallego Arias and Shachar Itzhaky. 2022. *jsCoq*. Retrieved June 8, 2022 from https://coq.vercel.app/

[7] Georges Gonthier. 2008. Formal Proof—The Four- Color Theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393.

[8] Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs.* Ph. D. Dissertation. University of California San Diego.

[9] Blockly Developer Team. 2022. *Blockly*. Retrieved June 8, 2022 from https://developers.google.com/blockly

[10] Hendrik Tews. 2017. *Proof tree visualization for Proof General*. Retrieved June 8, 2022 from https://askra.de/software/prooftree/

[11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532