HenBlocks: Structured Editing for Coq

Bernard Boey and Michael D. Adams

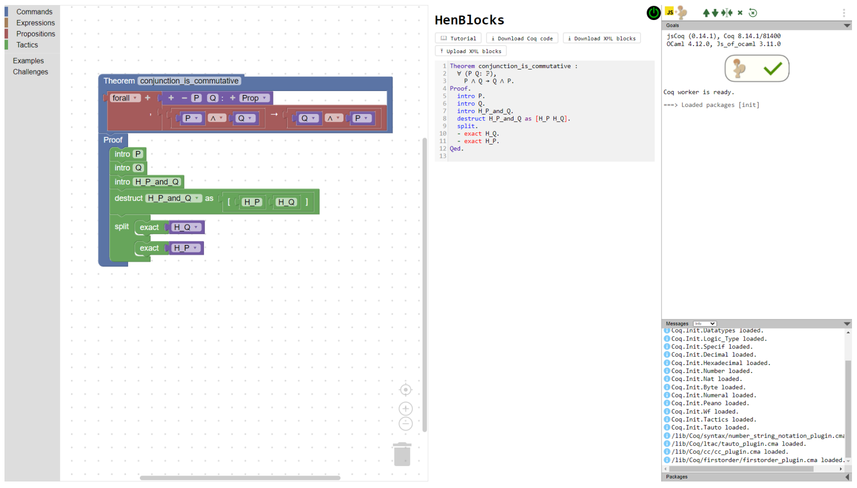
Yale-NUS College, Singapore

Abstract

There are a number of pain points in using the Coq Proof Assistant, 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. 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.

Background. Readers are assumed to be familiar with theorem proving and the Coq Proof Assistant. *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. Integrated Development Environments (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.

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" [6]. 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" [2]. 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 [8], Proof-by-pointing [1], Actema [3], PeaCoq [6], Chick [6]) do not sufficiently simplify the learning process. Additionally, fully-fledged structured editing has not been applied to Coq thus far.



```
Theorem conjunction_is_commutative :
  ∀ (P Q: ℙ),
    P ∧ Q → Q ∧ P.
Proof.
  intro P.
  intro Q.
  intro H_P_and_Q.
  destruct H_P_and_Q as [H_P H_Q].
  split.
  - exact H_Q.
  - exact H_P.
Qed.
```

```
jsCoq (0.14.1), Coq 8.14.1/81400
OCaml 4.12.0, Js_of_ocaml 3.11.0

Coq worker is ready.

===> Loaded packages [init]
```

```
Coq.Init.Datatypes loaded.
Coq.Init.Logic_Type loaded.
Coq.Init.Specif loaded.
Coq.Init.Decimal loaded.
Coq.Init.Hexadecimal loaded.
Coq.Init.Number loaded.
Coq.Init.Nat loaded.
Coq.Init.Byte loaded.
Coq.Init.Numeral loaded.
Coq.Init.Peano loaded.
Coq.Init.Wf loaded.
Coq.Init.Tactics loaded.
Coq.Init.Tauto loaded.
/lib/Coq/syntax/number_string_notation_plugin.cma
/lib/Coq/ltac/tauto_plugin.cma loaded.
/lib/Coq/cc/cc_plugin.cma loaded.
/lib/Coq/firstorder/firstorder_plugin.cma loaded.
```

Solution. We present HenBlocks, a web-based fully-fledged structured editor for Coq built using the Blockly library [7] and jsCoq [5]. 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` as a static web app. The source code (primarily JavaScript) 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.

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**.

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 [4]. 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 HenBlocks 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.

Acknowledgements. 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 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. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 141–160, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[2] Sebastian Böhne and Christoph Kreitz. Learning how to prove: From the coq proof assistant to textbook style. *Electronic Proceedings in Theoretical Computer Science*, 267:1–18, mar 2018.

[3] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. A drag-and-drop proof tactic. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 197–209, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Neil Fraser. Ten things we've learned from blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 49–50, 2015.

[5] Emilio Jesús Gallego Arias and Shachar Itzhaky. jscoq. `https://coq.vercel.app/`.

[6] Valentin Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. PhD thesis, University of California San Diego, 2018.

[7] Blockly Developer Team. Blockly. `https://developers.google.com/blockly`.

[8] Hendrik Tews. Proof tree visualization for proof general. `https://askra.de/software/prooftree/`.