



HenBlocks: Structured Editing for Coq

Bernard Boey Khai Chen

**Capstone Final Report for BSc (Honours) in
Mathematical, Computational and Statistical Sciences**

Supervised by: Michael D. Adams

AY2021/2022

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

HenBlocks: Structured Editing for Coq

by Bernard BOEY

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.

Keywords: Structured editor, visual programming, theorem prover, proof assistant, program verification, user interfaces.

Contents

Abstract	i
1 Background	1
2 Motivation	11
3 Methods	15
4 Solution: HenBlocks	19
5 Design and Implementation	24
6 Discussion	33
Bibliography (Main)	37
Bibliography (Supplementary)	38
Acknowledgements	41
A HenBlocks Usage	42

Chapter 1

Background

1.1 Introduction

When writing a computer program, we often use unit tests to check it's correctness. However, unit testing is non-exhaustive, and it is possible to write a program that passes our unit tests but fails on some edge cases. This is a common source of bugs when deploying programs in the real world.

A better way to demonstrate correctness is to use proofs. By writing a valid specification for what we want our program to do, writing the program, and then proving that our program fulfills the specification, we can guarantee that our program is bug-free.

One popular way of writing proofs is using the Coq Proof Assistant, also known as Coq. However, there are a number of pain points in using it (and proof assistants in general), which especially affects beginners. These include the complexity of understanding the type system, the difficulty in learning new specification and tactic languages, and the friction of the user experiences. The pain points have contributed in part to the lack of mainstream adoption of proof assistants.

Readers of this thesis are assumed to have a computer science background with good knowledge in functional programming (e.g. algebraic data types, pattern matching) and some knowledge in proving.

1.2 Objective

The aim of this project is to explore the use of structured editing in writing Coq proofs, and evaluate whether it can help alleviate the aforementioned pain points and be a good alternative to text editors. This should be accomplished by building an interactive graphical user interface (GUI).

1.3 Unit Testing

Unit testing involves writing a set of test cases for a corresponding portion of code that we wish to test.

In the functional programming paradigm, functions are usually pure, and they take in inputs and produce corresponding outputs. They do not mutate any global state and have no side effects. Hence, for the same input, we can always expect the same output no matter when and how many times we apply the function.

Typically, the programmer generates a set of inputs and expected outputs, ensuring that edge cases are included. Then, the function is applied to each input, and the actual output is compared with the expected output. If there are no differences, we can describe the function as having passed the unit test, and we can deem it to be “correct”.

In the imperative programming paradigm, not all functions are pure, and functions and methods are typically written to change some non-local state. The concept of unit testing still applies, but instead of checking that the output corresponds to what we expect, we might have to verify properties of the program state (this applies especially for functions that don’t return anything, also known as void functions).

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” [5, p. 16].

The potential problems arising from bugs can range from harmless to disastrous. For example, service outages of online platforms can happen due to undiscovered bugs being deployed, causing inconvenience to users. In the worst case, bugs in critical software applications like flight control systems can result in loss of life.

1.4 Theorem Proving

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.

This method only works if our specification is valid - that it indeed describes what we want it to. One way of showing validity is to prove that the specification is non-vacuous (that it is non-contradictory and there is at least one function that satisfies it) and unambiguous (that it specifies at most one function).

Proofs can be written in forward reasoning or in backward reasoning. In forward reasoning, we start from the assumptions that we have and build upon them to reach the end goal that we want to prove. In backward reasoning, we start from the end goal we want to prove and transform it such that

the transformed goal corresponds with one of our assumptions. In some cases, we have to split the goal into constituent parts, thus generating sub-goals for each part. We then have to prove each sub-goal before moving on to the next one.

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.

Note that we have the additional problem of ensuring that the proof assistant itself is correct. In practice, this is not a big issue as the implementation of the proof assistants are heavily scrutinized by researchers and code that have been approved form part of the trusted code base. There is also progress in verifying the proof assistants using the proof assistants themselves.

1.4.1 Coq Proof Assistant

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 programming 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).

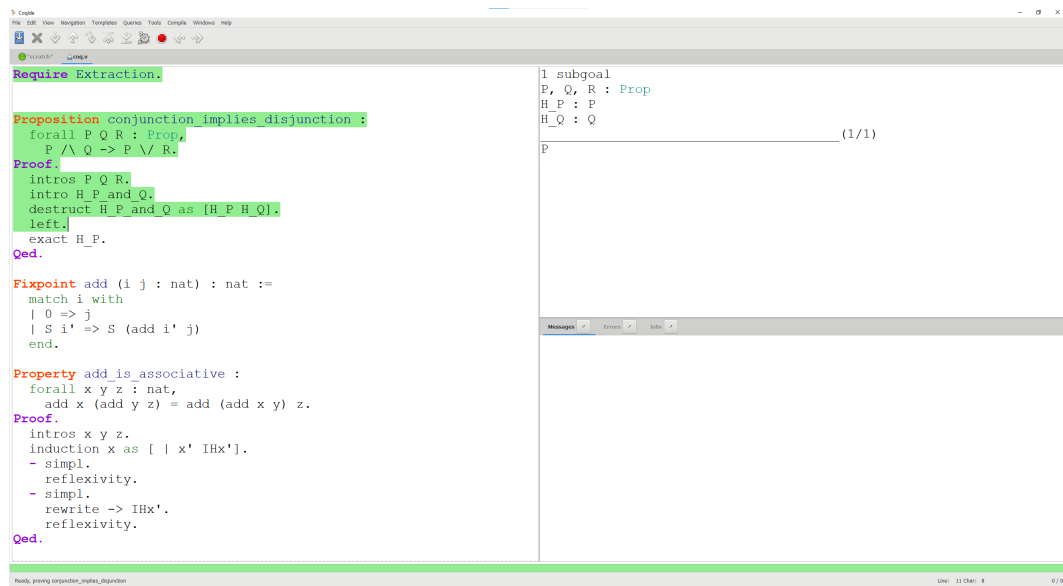


FIGURE 1.1: Screenshot of CoqIDE

As a dependently typed¹ programming language, Gallina can express universal (i.e. \forall) and existential quantification (i.e. \exists).

Coq also has the Vernacular language, which provides top-level commands (e.g. the `Definition` command which defines a function). All commands start with a capital letter and end with a dot (e.g. `Check 0.`, `Qed.`).

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" [18].

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

¹Dependent types are types whose definition depends on a value

verified [21]. Coq has also been used to prove mathematical theorems such as the four colour theorem [13].

1.5 Structured Editing

In its most basic form, structured editing is ubiquitous. When we use a keyboard shortcut or press a button to make text bold (e.g. in a document editor, email client, or messaging application), we are employing structured editing techniques. Behind the scenes, the software adds corresponding code to display that portion of text as bold (e.g. if the software uses HTML, it would add the `` tag before the portion of text and `` after).

Put more formally, structured editing is manipulation of underlying text content in a syntax-directed manner. This means that instead of the user making low-level edits by directly modifying text (e.g. adding/removing characters), the editor helps them make higher-level edits that require awareness of the syntax of the content. Such underlying text content can be of a programming language (e.g. OCaml), marked up text (e.g. HTML), etc.

Editors implement varying degrees of structured editing. On one end of the spectrum, we have basic text editors that have no structured editing support, such as Windows Notepad which has minimal functionality (no support for making a portion of the text bold etc.). At this level, users have to make edits character by character (or by copying and pasting).

In the middle, we have text editors with some structured editing support. Examples include the Integrated Development Environments (IDEs) made by JetBrains (e.g. IntelliJ IDEA, PyCharm, WebStorm). They provide functionality to rename a variable/function and all of its usages (note that this is not a mere find-and-replace operation), as well as code completion

(e.g. listing out variables/functions that were defined elsewhere in the file). At this level, users can still make edits character by character, but can also employ structured editing functionality which triggers a high-level modification. However, it is usually the case that such functionality requires the text to be syntactically correct and complete. This is because the editor has to parse the text in order to be aware of its structure and abstract syntax tree, and such parsing cannot be completed in the event of a syntax error.

On the other end of the spectrum, we have fully-fledged structured editors. These include visual editors, also known as WYSIWYG (What You See Is What You Get) editors, such as website editors (e.g. Wix, Weebly, Squarespace) and slide presentation software (e.g. Microsoft PowerPoint, Google Slides). At this level, users cannot directly edit the underlying text (e.g. HTML for website editor, XML for slide presentation software). Instead, they manipulate higher level representations such as blocks, text boxes, shapes, etc. The software then translates modifications of the representations into modifications of the underlying text.

With such editors, it is usually not possible to have incorrect syntax. This is because the editor generates the output syntax from the higher level representations. Assuming that the generative rules underlying the representations adheres to the grammar rules of the underlying text, any action taken by the user would correspond to a legal modification of the syntax.

To provide a more concrete illustration about structured editors, let us explore two examples: Scratch and Hazel.

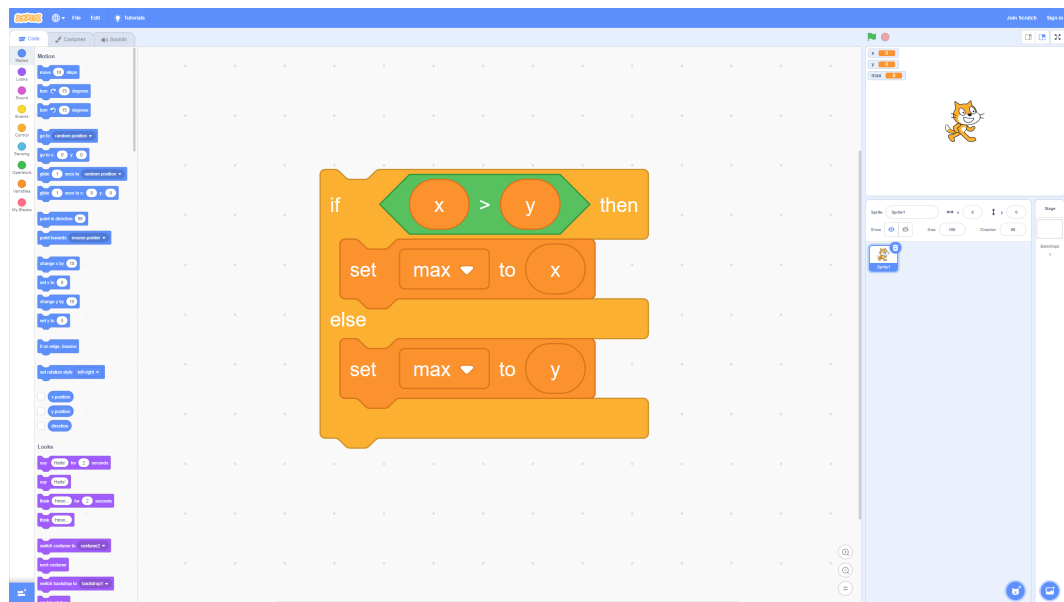


FIGURE 1.2: Screenshot of the Scratch editor

1.5.1 Scratch

Scratch is a visual programming language editor primarily designed to teach programming to children. Scratch is considered a fully-fledged structured editor as users cannot directly edit the source code. Users are presented with a series of blocks (in the block palette on the left of the screen) that correspond to programming constructs (e.g. conditional statement, variable assignment). These blocks are draggable into the coding area, where they can be arranged to form scripts.

Some blocks have holes for the user to type in values (e.g. strings and integers) or slot in a compatible block that reports a value (e.g. variable). Other blocks have spaces for blocks to be nested (e.g. repeat block which corresponds to a for loop) and connectors to chain blocks together, similar to a jigsaw puzzle. Through the usage of these blocks, Scratch is Turing-complete, which means that any computation possible in another general-purpose programming language is also possible in Scratch.

The blocks ensure that users do not have to worry about syntax (e.g. a missing semicolon, or the syntax of a for loop). Additionally, by differentiating block types and imposing rules on what type of blocks can be used, it is not possible to write code that has incorrect syntax, allowing the user to focus on expressing what they want the code to accomplish. By listing the available blocks in the block palette, Scratch also enables vocabulary discovery, reminding the user of new features that they can explore.

Scratch was released in 2007 and is still actively maintained. It sees over 100 million website visits each month. It is also currently being used in the first week of CS50, Harvard’s famed introductory computer science course, as a gentle introduction to programming. Research has shown that learning with Scratch improves students’ logical thinking and problem solving [16].

1.5.2 Hazel

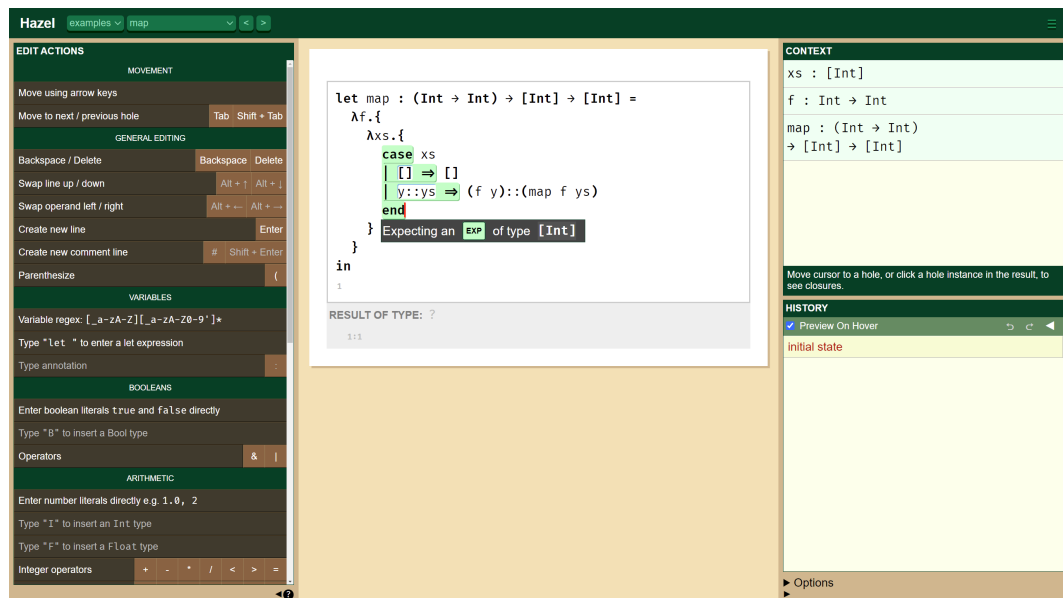


FIGURE 1.3: Screenshot of the Hazel editor

Hazel is an online functional programming editor intended as an exploration of the semantics of incomplete programs. It is also being used as a teaching tool for functional programming. On the left sidebar, users are presented with a series of edit actions that can be taken, which include navigation (e.g. moving by arrow keys), general editing (e.g. creating a new line), and inserting programming constructs (e.g. function). The actions are taken via clicking or keyboard shortcuts. With keyboard shortcuts, the experience is more seamless and similar to that of a regular text editor.

Hazel is also considered a fully-fledged structured editor as users cannot directly edit the source code (although it might seem possible). For example, users cannot edit the individual characters of a "case" expression; They can only type in designated places or delete the whole expression.

Initially, there is a single hole in the editor, which can be filled via edit actions. When inserting a programming construct, a "block" takes the place of the initial hole, and it may have additional holes to be filled. For example, a lambda function has a hole for the input parameter and a hole for the function body. Such holes can be further nested and filled via edit actions. There are rules on what edit actions can be taken depending on what the hole requires. Hazel also provides useful information such as the required type to fill a particular hole, and the available variables that are in scope.

In comparison to Scratch, Hazel can be considered a less restrictive structured editor. Semantic errors are more common, such as referencing variables that have not yet been created. Such errors are shown to the user in real-time. Unlike text editors with structured editing support, Hazel is able to identify and display such errors even if the program is incomplete, because of the well-formed structure that it enforces.

Chapter 2

Motivation

2.1 Pain Points of Coq

There are a number of pain points in using Coq and other proof assistants. First, the **type system is complex and difficult to understand**. Proof assistants require dependently typed programming, which beginners do not typically encounter in typical functional programming. Additionally, users have to explicitly provide a type parameter for polymorphic functions/quantifiers. Such complexities contribute to the difficulty in making "proper mental models for what happens 'behind the scenes' when [interacting] with a proof assistant" [18].

Second, there is **difficulty in learning new specification and tactic languages**. While seemingly similar to their functional programming counterparts, such languages have different rules and a tremendous amount of new vocabulary. When some researchers used Coq to teach textbook proofs, they defined their own tactics instead of using built-in tactics, because Coq tactics "have unstructured names and are therefore hard to remember" [6]. For example, it is not immediately clear that the `split` tactic is used to generate two subgoals corresponding to a logical conjunction in the goals.

Additionally, there are multiple ways of accomplishing the same thing. For example, to define a constructor of an inductive type, one can specify

just the types of the parameters (and the constructor) using annotations (e.g. `| S : nat -> nat`) or specify the names and types of the parameters (e.g. `| S (n : nat)`). There is thus no single obvious way to follow, especially since different courses and resources use different conventions.

Some tactics (e.g. `destruct`) sometimes create subgoals (e.g. in the case of a disjunction) and sometimes do not (e.g. in the case of a conjunction). This can be confusing for the user. Some tactics also accomplish more things than expected (e.g. the `reflexivity` tactic does simplification as well).

Third, there is **friction in the user experience**. One example is that syntax errors are hard to understand. For example, failure to include a period at the end of a tactic or command (a very common mistake among beginners) could result in one of the following messages: 1) "Syntax error: illegal begin of vernac. The reference TACTIC_NAME was not found in the current environment.", 2) "Syntax error: [`ltac_use_default`] expected after [`tactic`] (in [`tactic_command`]).", 3) "Syntax error: `'.'` expected after [`command`] (in [`vernac_aux`]).". Only the last message is useful to the user.

2.2 Existing Approaches

Fully-fledged structured editing has not been applied to Coq thus far. Additionally, little research has been done on alternative interfaces to Coq. Here are some of the limited examples:

1. *Prooftree* is a program that displays a visualisation of the proof tree for Coq proofs. This "helps against getting lost between different subgoals" [20]. The most recent version was released in 2017 and the second-most recent version is from 2013.

1. *Proof-by-pointing* is a principle developed in 1994 whereby the user can point and click on certain sub-expressions of a goal to perform transformations towards solving it [4]. For example, if the user selects a sub-expression of a conjunction in the hypotheses, the conjunction is broken up into the two hypotheses corresponding to the two conjuncts. If the user selects a sub-expression of a conjunction in the goals, then two subgoals are generated corresponding to the two cases. Although seemingly simple, this technique is effective because all the basic logical connectives (i.e. conjunction, disjunction, implication, negation, universal quantification, and existential quantification) are accounted for. This interface was developed for Coq in the form of CtCoq [3], which was subsequently replaced by PCoq [2], which has not been maintained since 2003.

2. *Actema* is an interface that allows users to construct proofs via drag-and-drop actions, building on the ideas of proof-by-pointing [7]. For example, given two hypotheses $\forall x, P(x) \implies Q(x)$ and $P(x)$, the user can drag the latter to overlap with the antecedent of the former and drop it. Since both subexpressions correspond exactly ($P(x)$), the implication rule is applied and a new hypotheses $Q(x)$ is created. Actema was released in 2022 as a standalone online prototype.

3. *PeaCoq* provides 3 main functionalities: 1) a visualisation of the proof tree, 2) a visualisation of the difference in proof state before and after a tactic is used (in the form of a side by side comparison with highlighting), and 3) automatic tactic exploration and suggestion to the user. This solves 3 problems that beginners face: 1) getting lost in the proof tree structure, 2) difficulty in identifying "effects of a tactic on the proof context", and 3) difficulty in identifying relevant tactics that can be used for a particular goal

[18]. PeaCoq was released in 2015 as a standalone web interface similar to CoqIDE and was most recently updated in 2018.

4. *Chick* is developed by the same creator as PeaCoq. It is a programming language with syntax similar to Gallina, designed to be able to allow automated complex refactoring with changes being propagated [18]. For example, if the user were to rename a variable or function definition, subsequent usages would be automatically renamed. The same applies for re-ordering of parameters to functions.

While the above tools do provide benefits, they have limitations as well. Firstly, the proof tree visualisations in ProofTree and PeaCoq are helpful in the case of complicated proofs and nested subgoals, but such situations are rare for beginners. For beginner proofs, subgoals are nested usually at most a few levels deep, and new subgoals are typically created two at a time. Hence, this is a feature which might be more useful for advanced users. The same can be said for the complex refactoring provided by Chick, as evidenced by the creator's intention of it as a tool for experts.

Additionally, some of the tools (Proof-by-pointing, Actema, Chick) are either unrelated to proving in Coq or uses a different system. Thus, while the user may learn and better understand proving concepts, they would not gain familiarity with writing proofs in Coq.

Lastly, some of these tools are old and no longer maintained (ProofTree, and PCoq which implements proof-by-pointing).

Chapter 3

Methods

3.1 Structured Editor

We first have to discuss how we want our structured editor to be designed. As explained earlier, we can build a fully-fledged structured editor or a text editor with structured editing support. Based on the prior examples mentioned, it seems that text editors are better suited to advanced users who are already familiar with the language, and can make full use of advanced structured editing features (e.g. extract method refactoring). Fully fledged structured editors are better suited to beginners who more easily trip up and get frustrated by syntax errors. Additionally, looking at the existing approaches to Coq interfaces, it is clear that although text editors with structured editing support have been attempted with limited success, fully-fledged structured editing for Coq has not been attempted thus far. Hence, creating a fully fledged structured editor would be a novel contribution.

Next, two obvious choices came up: We can implement our structured editor as 1) a desktop app, or 2) an online web app. The traditional way of writing proofs involves desktop IDEs (e.g. Emacs) which requires extensive set up such as the installation of Coq and IDE plugins to support Coq, which is not well-supported on certain platforms (e.g. Windows). This may turn away new users who just want to get a taste of it before diving deeper.

In contrast, an online web app guarantees minimal set up and allows the user to quickly get started. For example, jsCoq provides an online scratchpad to write and evaluate Coq proofs on the web. There are also multiple examples of structured editors implemented as web apps (e.g. Scratch, Hazel). An added benefit of writing a web app is that it can be exported as a desktop app using a framework such as Electron. For example, VSCode is primarily used as a desktop app but is built as a web app using TypeScript. With these considerations in mind, we decided to implement a web-based fully-fledged structured editor, especially since it is intended as a tool for beginners to learn to write proofs.

3.2 Coq APIs

The first step of building the structured editor involved figuring out how to connect the Coq backend to our editor frontend. Coq ships as a set of command-line tools: `coqtop` (the Coq toplevel that provides an interactive mode), `coqc` (the Coq compiler that provides batch compilation), and `coqchk` (the Coq checker that validates compiled libraries).

However, users normally interface with Coq through an IDE (e.g. CoqIDE, VSCode, Emacs). These IDEs connect to `coqtop` to provide an interactive proving mode, whereby the user can develop proofs step by step, evaluate incomplete proofs, and view the goals that need to be proved.

For an IDE to connect to Coq, it has to make use of a backend Application Programming Interface (API). Information on available APIs are sparse and not well documented. Nevertheless, there are 3 main APIs: 1) OCaml API, 2) XML Protocol, 3) SerAPI.

The **OCaml API** is used by Coq plugins. However, it is “restricted to Ocaml-friendly environments” [11]. Additionally, the API is constantly changing, is “difficult to master”, and there is “no official documentation . . . other than looking at the source code” [10].

Most of the IDEs listed above make use of Coq’s **XML protocols**. They interface with coqtop by sending and receiving XML messages. However, the “amount of serialization boilerplate [is] high” [11], as we would need to write code in our chosen language to parse and serialize the XML messages.

Recognising the disadvantages of the above two options, a pair of researchers built a new API to overcome them. **SerAPI** provides an interface (sertop) for interactive proofs, similar to coqtop. Instead of XML, the API uses S-expressions. It also provides a serialization library (serlib) [11]. While the creators warn that the API is experimental in nature, it has proven to be the most modern and well-maintained out of the three. SerAPI is used by jsCoq, which is a JavaScript port of Coq that can be run in a browser. jsCoq extends SerAPI by providing a JavaScript interface, providing access to low-level serialized Coq data structures [12].

Based on our evaluation, SerAPI was the most suitable. Additionally, since jsCoq provides a JavaScript interface, we can write our structured editor in JavaScript (the primary language for developing web apps) and easily integrate with jsCoq.

3.3 Blockly Library

Blockly is a JavaScript library released in 2012 for creating visual programming editors. Similar to Scratch (in fact, Scratch 3.0 is built using Blockly), it uses “interlocking, graphical blocks” to represent programming constructs

(e.g. conditional statements) [14]. Blockly provides commonly used blocks (e.g. for loop), but allows developers to define their own blocks as well. Blockly also comes with the ability to export built-in blocks to JavaScript, Python, PHP, Lua, or Dart code, and allows developers to extend it with code generators for other languages. Additionally, Blockly provides a user interface consisting of a toolbox (containing the types of blocks that can be used) and a workspace (where users rearrange the blocks they are using).

Blockly suits our purposes because we do not have to write graphical components from scratch. Instead, we can build on the block-based framework it provides, and focus our time on designing the representation of Gallina and Ltac constructs as well as implementing the structured editing features. Blockly is also a well-supported library in active development, used by hundreds of projects. Finally, with the success that block-based programming has achieved (through Scratch), it is worthwhile exploring this approach for writing proofs as well.

Chapter 4

Solution: HenBlocks

4.1 Introduction

HenBlocks is a web-based fully-fledged structured editor for Coq that I built using the Blockly library and jsCoq. It can be considered a visual programming language editor due to the use of graphical blocks to represent programming and proving constructs. 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 in the Coq Proof Assistant via a text editor such as CoqIDE or Emacs.

HenBlocks is freely accessible on the internet at <https://henblocks.github.io> (readers of this thesis are strongly encouraged to test it out). Users can dive straight in to building proofs with HenBlocks, or follow the step-by-step tutorial. The source code can be found at <https://github.com/henblocks/henblocks.github.io>.

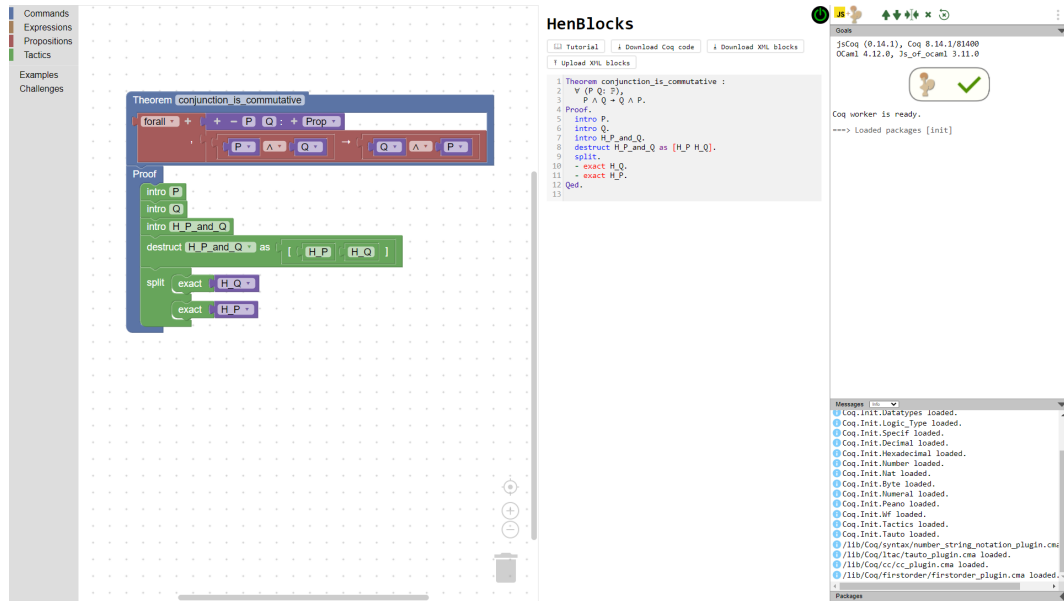


FIGURE 4.1: Screenshot of HenBlocks

4.2 Interface

The user interface is divided into four sections from left to right: 1) Toolbox, 2) Workspace, 3) Code, and 4) Goals.

1) **Toolbox**: The leftmost section is an expandable panel that contains all the types of blocks that can be used. Upon choosing a block, the user can drag it to the workspace. The toolbox is organised into categories according to what kind of constructs the blocks represent: a) Commands, b) Expressions, c) Propositions, and d) Tactics. There are also 2 additional categories: Examples, which list out a few example theorems and accompanying proofs, as well as Challenges, which list out a few sample theorems without proofs, for users to attempt to prove. Tactics are further organised into subcategories based on their function: d1) Managing context, d2) Solving goals, d3) Transforming goals/hypotheses, d4) Transforming conjunctions/disjunctions in goals, and d5) Transforming conjunctions/disjunctions/inductive types in hypotheses.

2) **Workspace:** The second section is the workspace, which contains all the blocks that the user is using. This is where the user will spend most of their time rearranging and modifying blocks in order to write proofs. Upon choosing a block, the user can drag the block to this workspace and place them anywhere they want. They can reposition the block, connect it to other blocks, or delete it.

3) **Code:** The third section contains the generated Coq code. Whenever a change is made in the workspace, the corresponding Coq code is generated and displayed in this section. This usually happens instantaneously. There is a button to download the current generated code as a Coq (.v) file, if the user wishes to open it in another editor (e.g. CoqIDE/Emacs). There are also 2 buttons to download and upload the XML representation of the blocks. The generation and parsing of XML representation is handled by the Blockly library, but the buttons and associated functionality are created from scratch. Providing such buttons allows the user to share their HenBlocks creations with other people.

4) **Goals:** The rightmost section contains the intermediate state of the currently focused proof. These include the goals and hypotheses at the current state. The bottom compartment of this section also displays error messages (e.g. syntax errors) from the Coq Proof Assistant. The user can step forward and backward through the Coq code, executing commands or undoing them, by pressing the down / up arrows.

If a change is made to the blocks while a portion of the code has already been evaluated, HenBlocks takes the following approach: 1) If there are no changes made to the evaluated code (i.e. the changes to the code happen after the current cursor position), the proof state is not changed. 2) If there

are changes made to the evaluated code (i.e. the changes happen before the current cursor position), the proof state reverts to the last unmodified sentence. This is similar to the user experience in a text editor like CoqIDE.

4.3 Sample Proofs

To illustrate HenBlocks, here are some of the first few proofs and definitions from Olivier Danvy’s Functional Programming and Proving (FPP) course, an introductory Coq course at Yale-NUS College.

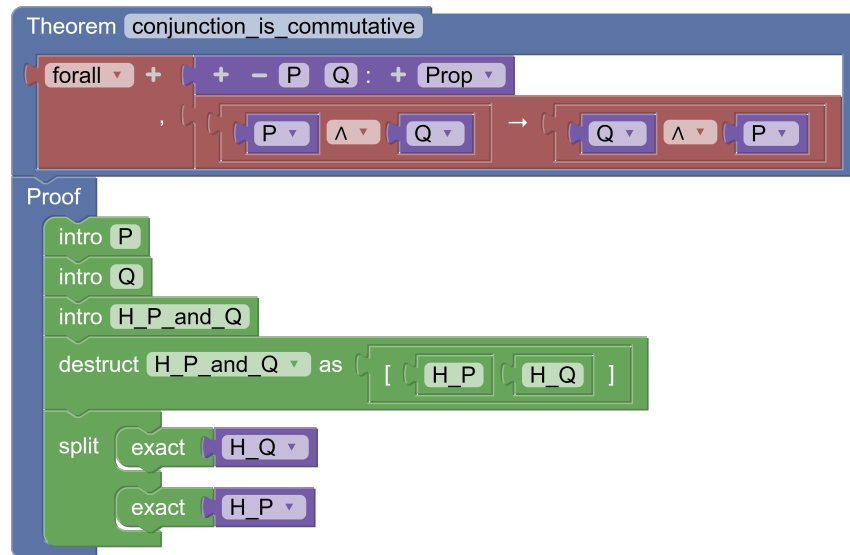


FIGURE 4.2: Proof that conjunction is commutative, in Hen-Blocks

```

Theorem conjunction_is_commutative :
  forall (P Q: Prop),
    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.

```

FIGURE 4.3: Generated Coq code from proof that conjunction is commutative

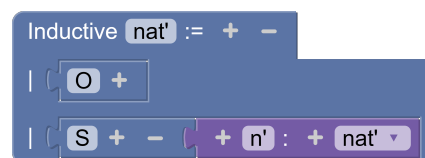


FIGURE 4.4: Inductive definition of natural numbers

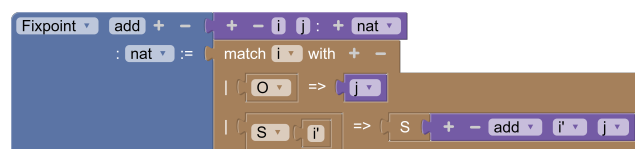


FIGURE 4.5: Recursive definition of addition function

Chapter 5

Design and Implementation

In this chapter, I go over some of the key design and implementation details of HenBlocks.

5.1 Grammar

The full grammar of Coq is extremely complex and long. Additionally, since Coq is extensible via plugins, the whole set of allowable syntax can be increased [1]. Hence, supporting the full grammar (or even a large portion) would overwhelm the user due to the sheer number of options (apart from being incredible tedious and beyond the feasibility of a capstone project). As suggested by the developers of Blockly, “you should keep the set of available blocks small to avoid frustrating your users” [17]. Furthermore, as evidenced by Danvy’s FPP course, it is possible to accomplish a large amount by relying on a small subset of tactics and syntax [9].

Apart from relying on a subset of constructs, the grammar of HenBlocks simplifies certain concepts. For example, in Coq (and in most programming languages), a binary operator typically has a production rule as such:

```
<bool> ::= true | false | <identifier> | <bool> && <bool>
```

With this production rule, it is possible to make nested constructs (e.g. `P && (Q && (R && (S && T)))`) which simplifies to `P && Q && R && S && T`.

Translating this grammar directly to HenBlocks would require at least 4 `&&` blocks to represent that expression. These blocks would be heavily nested, creating a visual mess. Thus, with HenBlocks, we use a different production rule:

```
<bool> ::= true | false | <identifier> | <bool> {&& <bool>}+
```

Here, this means that the `{&& <bool>}` component occurs one or more times. This is accomplished by creating the `&&` block with "plus" and "minus" icons so that the user can decide on the number of conjuncts present in a single block. Thus, the above example expression can be constructed using a single `&&` block, resulting in a cleaner look and simpler process for the user.

Unlike text-based language, there is no top-level/entry point. This means that it is possible to have any block placed at the "top-level" of the workspace. This is merely for convenience. When re-ordering blocks, the user may find it useful to temporarily place some blocks in the "top-level" of the workspace. This is not to say that such an arrangement is syntactically correct (e.g. a tactic must be used within the body of a proof). Thus, to prevent syntax errors, we temporarily disable any blocks that should not be at the top-level (i.e. they are greyed out, but they can still be moved) which prevents Coq code from being generated from those blocks.

5.2 Variable Dropdowns

In text editors, users can reference any identifier even if the variable was not previously defined, or is out of scope. The mistake would only be highlighted either at runtime or compile time. Some IDEs with structured editing support (e.g. JetBrains IDEs) provide variable autocompletion - upon typing a character, a dialog box pops up that indicates the variables that are

in scope. However, this is not always reliable as it may miss out on variables, especially in the case of dynamically-typed variables.

In some fully-fledged structured editors (e.g. Scratch), users are confined to only selecting variables that have been created by the user. However, such variables have no notion of scope. The code generator automatically hoists the variable declarations to the start of the code, essentially making them global variables. This is the case even for function inputs (parameters). Thus, if the user defines a function with input x , it is possible to use the value of x outside of the function without having assigned a value. While this is not a syntactic error, this is a semantic error.

In Coq, variables work similarly to other functional languages - function parameters can only be used within the function. Global variables/functions can only be used after and not before they have been defined. Similarly, quantified variables can only be used within the universal/existential quantification. In addition, as Coq has tactics which introduce named hypotheses, these hypotheses can only be used after they have been introduced. Some tactics remove hypotheses (e.g. by destructuring them), thus making them no longer referenceable.

In HenBlocks, we use an algorithm to automatically list out the variables that are in scope via a dropdown menu, solving the aforementioned problems faced by other editors. Through a change listener, the algorithm runs every time the user makes a change (e.g. a block is added, moved, edited, or removed). First, we get a list of all the blocks in the workspace from the Blockly API, ordered by how it appears (top to bottom, left to right). Then, we go through each block while maintaining a list of identifiers we encounter. We then take the next two steps (simultaneously) to handle global

(free) and local (bound) variables:

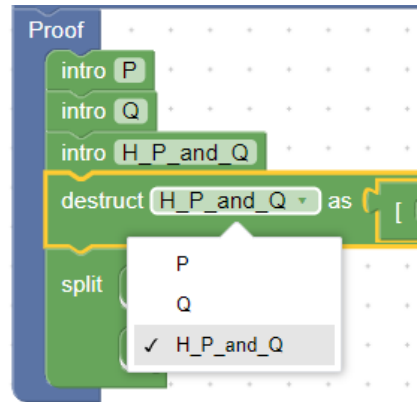


FIGURE 5.1: Variable dropdown in HenBlocks

Global Variables: When we encounter a Definition, Inductive, or Theorem block, we check if the chosen identifier has already been used (i.e. a variable name collision). If yes, we display a warning on that block. If not, we add the chosen identifier to the list of global identifiers. In addition, for the Inductive block, we do the same thing for each constructor identifier.

Local Variables: When we encounter a block containing parameters (e.g. Definition, forall), we go through the parameters defined by the user, adding them to a list of local identifiers (we also check for name collisions with global identifiers or previously defined local identifiers). Then, we traverse through the children blocks. When we encounter a variable block, we populate the dropdown menu with the list of local and global identifiers.

Certain expressions (e.g. match statements) also allow for the declaration of new identifiers. Thus, upon encountering such expressions, the new identifier will be added to the list of identifiers. As we use a depth-first traversal, new identifiers are available only after they have been defined, and in the correct scope (e.g. an identifier defined in a branch of a match statement will only be accessible in that branch).

A similar thing happens for tactics that introduces new (named) hypotheses (e.g. `intro`, `destruct`, `induction`). When we encounter a tactic that removes a hypothesis (e.g. `destruct`, `induction`, `revert`), we remove the identifier from the list of local identifiers, so they are no longer accessible after that tactic. If the user deletes a block that defines a variable (e.g. a parameter to a definition), any existing references to the variable (i.e. blocks where the user previously selected that variable from the dropdown) will display a warning, and the dropdown options will no longer display the removed identifier.

Note that in contrast to other editors / Coq itself, we implement more restrictions by disallowing (or rather, discouraging through the usage of warnings) local variables from having the same name as global variables. This is due to potential for confusion especially among beginners.

5.3 Automatic Renaming of Variables

In text editors, when the user renames a variable in the definition, they also have to rename the variable in all the places that it is referenced. This can easily introduce unintended bugs. Some IDEs with structured editing support (e.g. JetBrains IDEs) provide functionality to automatically rename all references. This has the possibility of being unreliable, so the user is asked to verify which references should be renamed if the IDE detects ambiguity.

In some fully-fledged structured editors (e.g. Scratch) where users are confined to only selecting variables that have been explicitly created, renaming of the original variable automatically renames all usages of the variable. However, in Scratch, function parameters are not renamed correctly, as they

are treated as global variables (i.e. If a function parameter is renamed, references to the parameter using the old name are untouched, because the global variable corresponding to the old name still exists. Renaming merely creates a new variable).

In HenBlocks, we use an algorithm to automatically rename variables, function/theorem names, and inductive/constructor names, while solving the above problems. Variable names are defined via text fields. When a user edits a text field, they have to click on it, placing focus to the field (it gets highlighted). Then, the user can type characters and edit in any way they wish to. Once the user is done, they either press the “enter”/“tab” key, or they click outside of the field. This terminates the text field edit. Thus, a series of events are generated which correspond to the editing of a text field. For example if a user were to change the variable “abc” to “apple”, this is what the series of events might look like (assuming the user adds/removes a single character at a time):

1. Text modification (old value: “abc”, new value: “ab”)
2. Text modification (old value: “ab”, new value: “a”)
3. Text modification (old value: “a”, new value: “ap”)
4. Text modification (old value: “ap”, new value: “app”)
5. Text modification (old value: “app”, new value: “appl”)
6. Text modification (old value: “appl”, new value: “apple”)
7. Termination of text modification

The algorithm listens for these events to implement the automatic renaming of variables. Upon detecting the first modification of a text field, the algorithm traverses through all blocks after that point and searches for

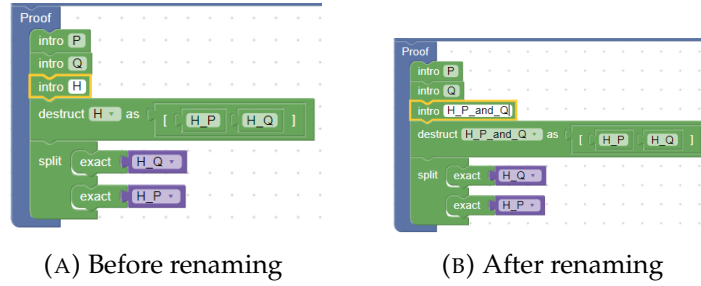


FIGURE 5.2: Automatic variable renaming in HenBlocks

variable blocks that have the old name selected. Then the old name is modified to the new name. At the same time, the block is added to a list of blocks that reference the variable currently being renamed. On subsequent modifications, we update each block in the list with the new name of the variable. Storing the blocks in a list improves the efficiency of the algorithm, as we do not need to spend more time traversing through all blocks to search for which blocks to edit.

This relies on the invariant that there are no changes to which blocks reference the variable being renamed. The invariant holds because while the user is editing the text field, other actions like modifying a block are not possible. To perform such actions, the user would have to click outside of the field, terminating the text field edit.

5.4 Automatic Slots for Subgoals

Destructuring a hypothesis (via the `destruct` tactic) creates subgoals only if the hypothesis is a disjunction or an inductive type with multiple constructors. The number of subgoals depends on how many constructors the type has. Typically, the user has to calculate this and put in the corresponding pattern, and then solve each subgoal. HenBlocks smoothens this process by

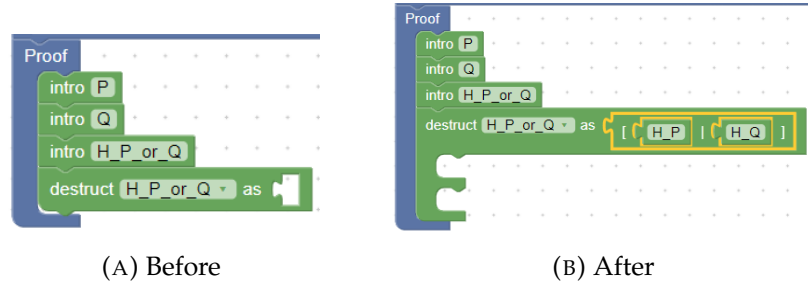


FIGURE 5.3: Automatic slots for subgoals in HenBlocks

automatically creating the correct number of slots for each subgoal, based on the pattern put in by the user. This helps the user keep track of the proof tree. Additionally, the correct hypotheses are directed to the corresponding slot (i.e. for a disjunction, only the left disjunct is available for selection in the first subgoal (via a variable dropdown), while only the right disjunct is available for selection in the second subgoal).

5.5 Automatic Slots for Constructor Arguments

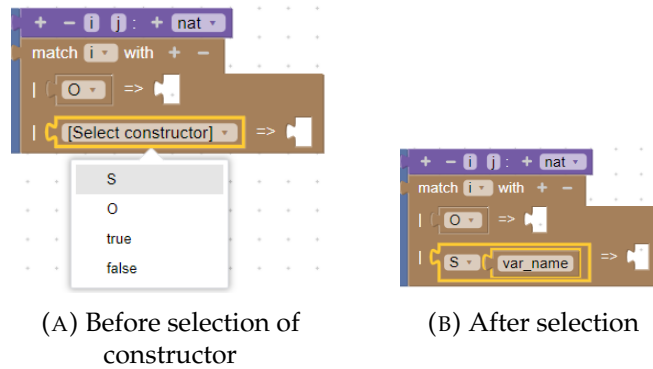


FIGURE 5.4: Automatic slots for constructor arguments in HenBlocks

Pattern matching requires all arguments of a constructor to be provided (whether via an identifier, another constructor, or an underscore). Thus, we can anticipate this and automatically generate the correct number of slots

according to the number of arguments the constructor expects. This is possible by keeping track of the arity of each constructor for each data type.

5.6 Automatic Retrieval of Hypotheses & Goals

When executing a `destruct` or `induction` tactic in Coq, we do not actually have to specify the names of the variables to be introduced (e.g. we can write `destruct n` instead of `destruct n as [| n']`). Coq would then automatically name the variables for the user, and introduce subgoals if needed. This is not always encouraged as it makes the process more implicit, and the user will have to rely on Coq to name each variable (which is not always intuitive). However, it is still a valid tactic and there are valid uses, such as letting Coq automatically compute the conjunctive/disjunctive/case analysis pattern of the variable in question. It would be useful if our editor can allow the user to employ such a tactic, and retrieve the variable names defined by Coq, as well as the number of subgoals generated. Then, we would be able to automatically put the names in the variable dropdown menus, capture the number of variables required, and generate the appropriate number of subgoal slots. This would be simpler than the user having to identify which pattern to use. Working with one of the authors of jsCoq and SerAPI, we succeeded in creating a Coq API command to speculatively execute a tactic and retrieve the names of the new variables generated for each subgoal [15]. For example, performing the command on `induction n` (where `n` is a natural number) would return `[[], [n, IHn]]`. However, there was insufficient time to integrate this feature with HenBlocks.

Chapter 6

Discussion

6.1 Findings

Through the research of this capstone project, we have found that current interfaces for Coq do not go far enough in simplifying the user experience. Most interfaces either opt for a text editing approach with some structured editing, or use a new interface unrelated to Coq. Additionally, non-Coq structured editors have drawbacks that compromise on the promise of a simple user experience. In comparison with the existing approaches, HenBlocks attempts to overcome these limitations through providing advanced structured editing features in a block-based framework. Removing syntax errors is only the first step in fully-fledged structured editing. HenBlocks tries to go further by reducing semantic errors as well. This goes a long way in alleviating the three pain points: difficulty in understanding the complex type system, difficulty in learning new specification and tactic languages, and friction in the user experience.

However, building such a fully-fledged structured editor for a complex system like Coq is a painstaking task. It can be said that the effort may not be worth it - we could spend the time learning Coq the traditional way. However, we are developing not for ourselves, but for beginners who have a fundamentally different perspective. That requires making compromises

and simplifications, even if that means not supporting all variants of a tactic. Through these compromises, we find that it provides a framework for a flatter learning curve.

6.2 Limitations

One limitation of HenBlocks is the potential for visual clutter. Blocks naturally take up more space, and compound expressions may require multiple levels of nesting. Hence, code that would normally look concise has to be stretched out when represented as blocks. Another limitation is that dragging and dropping is slower than typing. Users accustomed to coding and navigating via keyboard shortcuts may feel constrained by the need to use the mouse. Additionally, as HenBlocks currently has a limited number of constructs and tactics that are represented, users seeking to expand their vocabulary or use advanced techniques (e.g. using semicolons to apply tactics to all subgoals) have to turn to standard text editors.

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, as emphasised by the developers of Blockly [8]. In HenBlocks, the transition is aided by the similarity of blocks to actual Gallina and Ltac code, as well as the display of the generated Code code in the interface.

6.3 Future Work

Testing: First, there needs to be rigorous user testing of HenBlocks to evaluate its effectiveness. Due to the relatively small scope of a Capstone project,

a usability test of HenBlocks is not feasible. However, based on anecdotal feedback, users had a positive reaction to HenBlocks, and even found it fun to interact with.

One possibility is via A/B testing, by having two randomised groups (one group using HenBlocks and the other group using a text editor). The participants should undergo a series of tutorials and then be tested on their understanding and ability to write proofs. Another possibility is to have the participants use both HenBlocks and a text editor (instead of just one), and to complete the System Usability Scale (SUS) [19] to compare the relative usability of both systems. An additional possibility is a longitudinal study, where HenBlocks is used in the context of a course in theorem proving. Apart from user testing, there should also be more developer testing to ensure the reliability of HenBlocks.

Development: Second, there needs to be work in expanding HenBlocks by supporting more tactics (and their variants) and constructs (e.g. modules, record types). Fourth, more advanced structured editing features should be developed. For example, HenBlocks can be developed to capture the type information of each function and variable, so that we can perform static type checking and restrict which variables can be used based on the required type. Similar to PeaCoq, we can also make suggestions to the user on what tactics to use based on the current proof state. We can achieve this by either analysing the blocks, or retrieving contextual information from Coq.

User Experience: Third, there should be work put in to make an interactive tutorial where the instructions are overlaid on the interface and the user has to complete certain milestones before being allowed to proceed. Building on this, HenBlocks can be gamified by allowing proof challenges to be

created and shared among users. Upon successfully completing a proof, the interface responds accordingly. Relatedly, it would be useful to allow teachers to configure HenBlocks for their class, such as restricting which tactics can be used, and creating the questions (i.e. theorems to be proved) beforehand. Additional features to make the user experience more seamless should also be developed, such as adding an arrow to the workspace pointing to the block that is currently being evaluated by Coq, so that the user can better understand the process.

6.4 Conclusion

In conclusion, we have made a novel contribution in the world of theorem proving, proof assistants, and user interfaces, 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.

Bibliography (Main)

- [8] Neil Fraser. “Ten things we’ve learned from Blockly”. In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 2015, pp. 49–50. DOI: [10.1109/BLOCKS.2015.7369000](https://doi.org/10.1109/BLOCKS.2015.7369000).
- [9] *Functional Programming and Proving*. URL: https://delimited-continuation.github.io/YSC3236/2021-2022_Sem1/index.html.
- [11] Emilio Jesús Gallego Arias. *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq*. Tech. rep. MINES ParisTech, Oct. 2016. URL: <https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408>.
- [12] Emilio Jesús Gallego Arias and Shachar Itzhaky. *jsCoq*. URL: <https://coq.vercel.app/>.
- [14] *Introduction to Blockly*. URL: <https://developers.google.com/blockly/guides/overview>.
- [15] *[jscoq] New protocol call TacticInfo*. URL: <https://github.com/jscoq/jscoq/commit/de0c3a55a3c71980586c2696d8cac59159effe08>.
- [17] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. “Tips for creating a block language with blockly”. In: *2017 IEEE Blocks and Beyond Workshop (B B)*. 2017, pp. 21–24. DOI: [10.1109/BLOCKS.2017.8120404](https://doi.org/10.1109/BLOCKS.2017.8120404).
- [18] Valentin Robert. “Front-end tooling for building and maintaining dependently-typed functional programs”. PhD thesis. University of California San Diego, 2018.

Bibliography (Supplementary)

- [1] *Basic notions and conventions - Coq 8.15.1 documentation*. URL: <https://coq.inria.fr/refman/language/core/basic.html>.
- [2] Yves Bertot. *Pcoq : A Java-based user-interface for Coq*. URL: <http://www-sop.inria.fr/lemme/pcoq/>.
- [3] Yves Bertot. *The CtCoq User Interface*. URL: <https://www-sop.inria.fr/croap/ctcoq/ctcoq-eng.html>.
- [4] Yves Bertot, Gilles Kahn, and Laurent Théry. “Proof by pointing”. In: *Theoretical Aspects of Computer Software*. Ed. by Masami Hagiya and John C. Mitchell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 141–160. ISBN: 978-3-540-48383-0.
- [5] John Buxton and Brian Randell. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.
- [6] Sebastian Böhne and Christoph Kreitz. “Learning how to Prove: From the Coq Proof Assistant to Textbook Style”. In: *Electronic Proceedings in Theoretical Computer Science* 267 (2018), pp. 1–18. DOI: [10.4204/eptcs.267.1](https://doi.org/10.4204/eptcs.267.1). URL: <https://doi.org/10.4204/eptcs.267.1>.
- [7] 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. Philadelphia, PA, USA: Association for Computing Machinery, 2022, 197–209.

- ISBN: 9781450391825. DOI: [10.1145/3497775.3503692](https://doi.org/10.1145/3497775.3503692). URL: <https://doi.org/10.1145/3497775.3503692>.
- [10] Emilio Jesús Gallego Arias. *How to call proof assistant Coq from external software*. URL: stackoverflow.com/a/46039814.
- [13] Georges Gonthier. “Formal Proof—The Four-Color Theorem”. In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.
- [16] Jesús Moreno-León and Gregorio Robles. “Code to learn with Scratch? A systematic literature review”. In: Apr. 2016, pp. 150–156. DOI: [10.1109/EDUCON.2016.7474546](https://doi.org/10.1109/EDUCON.2016.7474546).
- [19] *System Usability Scale*. URL: <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>.
- [20] Hendrik Tews. *Proof tree visualization for Proof General*. URL: <https://askra.de/software/prooftree/>.
- [21] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: Association for Computing Machinery, 2011, 283–294. ISBN: 9781450306638. DOI: [10.1145/1993498.1993532](https://doi.org/10.1145/1993498.1993532). URL: <https://doi.org/10.1145/1993498.1993532>.

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.
2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

☒ Unrestricted access

Make the Thesis immediately available for worldwide access.

☐ Access restricted to Yale-NUS College for a limited period

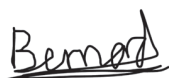
Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
_____.

After this period, the Thesis will be made available for worldwide access.

☐ Other restrictions: (please specify if any part of your thesis should be restricted)

Bernard Boey Khai Chen, Saga College

Name & Residential College of Student



Signature of Student

1 Apr 2022

Date

Michael D. Adams


Name & Signature of Supervisor

1 Apr 2022

Date

Acknowledgements

I would like to thank

- My family, for their unconditional love.
- Prof Michael Adams, for guiding me throughout this capstone project.
- Prof Olivier Danvy, for inspiring me and ultimately putting me on the path to this capstone project.
- Prof Neil Mehta, for enlightening me on what it means to be an extraordinary teacher.
- Various other professors, for their amazing mentorship and teaching.
- Various staff members, for their tremendous help and tireless work.
- Various people in this community, for their treasured friendship.
- Nikki, for being my best friend and supporter.

Appendix A

HenBlocks Usage

Here are some common ways to use HenBlocks:

Defining an algebraic data type: First, drag the Inductive block (Toolbox -> Commands) to the workspace. Specify the name of the type in the first text field. Specify the name of the first constructor in the corresponding text field. If this constructor has parameters, click on the plus icon of the constructor block, which will add a binder block. Specify the name (in the text field) and type (via the dropdown menu) of the parameter. To add more parameters of the same type, click on the plus icon of the binder block. To add more parameters of a different type, click on the plus icon of the constructor block. To add more constructors, perform the same steps as before. Control the number of constructors by clicking on the plus/minus icons of the Inductive block which will increase/decrease the number of constructor blocks.

Defining a function: First, drag the Definition block (Toolbox -> Commands) to the workspace. Specify the name of the function in the first text field. Similar to the Inductive block, specify parameters for the function via the binder blocks. Specify the return type of the function via the dropdown menu. Then, fill in the body of the function by dragging expression blocks (Toolbox -> Expressions) to the bottom right slot of the Definition block. For example, drag a match block to specify a pattern match. Additional blocks

can be added to fill in remaining slots of the currently chosen blocks.

Writing a proof: First, drag the Theorem block (Toolbox -> Commands) to the workspace. Specify the name of the theorem in the first text field. Then, drag proposition blocks (Toolbox -> Propositions) to the slot to define the theorem. For example, drag a forall block to specify a universal quantification. The parameters of a forall block can be specified via the binder blocks (similar to the Inductive and Definition block). Additional proposition (and expression) blocks can be added to fill in remaining slots of the currently chosen blocks.

Next, drag the Proof block to the workspace and connect its top with the bottom of the Theorem block. Then, drag tactic blocks (Toolbox -> Tactics) to the mouth of the Proof block to construct the proof.

A.1 Features

Here is a brief overview of some of the features of HenBlocks:

- Provided by Blockly/jsCoq:
 - Web-based, no set-up required
 - Blocks from previous session automatically restored
 - Delete all blocks with 2 clicks, and restore blocks from trash bin
- Developed from scratch:
 - Generation of syntactically-correct code (assuming all slots are filled), exportable to .v files
 - Selection of variable via dropdowns
 - Warnings for undeclared variables and duplicate variables
 - Automatic validation of variable names (reserved keywords, legal symbols)

-
- Automatic renaming of variables
 - Automatic slots for subgoals
 - Automatic slots for constructor arguments