



**Learning Support
for Writing Proofs
in Coq**

Jeremy Yew Ern

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

Supervised by: Professor Olivier Danvy

AY 2019/2020

Yale-NUS College Capstone Project

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)**

Name & Residential College of Student



Signature of Student

Date

Name & Signature of Supervisor

Date

Acknowledgements

I would like to acknowledge Prof Danvy for his generous and dedicated guidance.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Learning Support for Writing Proofs in Coq

by Jeremy YEW

I provide learning support for students in YSC3236 Functional Programming and Proving (FPP), by building an interactive, Emacs-integrated tool that checks for syntax issues in student's Coq proof submissions. The tool thus acts as a set of 'safety rails' to guide students towards developing the proper muscle memory. The Lecturer will be able to provide a grammar specification as an input to the tool.

Contents

Acknowledgements	ii
Abstract	iii
1 Introduction	1
1.1 Introduction	1
1.2 Context	2
1.2.1 Functional programming (FP)	2
1.2.2 Proving	2
1.2.3 Verifiable proofs with the Coq proof assistant	3
1.2.4 YSC3236 Functional Programming and Proving (FPP)	4
1.2.5 GNU Emacs Editor	4
1.2.6 Proof General	5
2 Building muscle memory	6
2.1 A skilled discipline	6
3 Writing proofs	7
3.1 What could go wrong?	7
3.1.1 Abuse of tactics	7
3.1.2 Misuse of tactics	8
3.1.3 Other issues	10

4	Solution	12
4.1	A grammar of grammars	12
4.2	Implementation trade-offs	13
4.3	Current progress	14
4.4	Challenges ahead	15
5	References	16
A	Appendix A	17
A.1	An example grammar	17

Chapter 1

Introduction

1.1 Introduction

The goal of this research project is to provide learning support for students enrolled in YSC3216: Functional Programming and Proving (FPP), by building a tool that checks for syntax issues in student's proof submissions.

FPP is a course in Yale-NUS College taught by Professor Olivier Danvy, under the Mathematical, Computational and Statistical Sciences major. FPP introduces students to the Coq proof assistant, which is a system for writing and verifying formal proofs.

The learning goal for the first half of the course is to build muscle memory for basic proof techniques and programming habits.

To this end, I implement a program that can act as a set of 'safety rails' to guide students towards developing the proper muscle memory. In particular, the program will enforce explicit tactic application within a subset of Coq, amongst other syntax rules. The Lecturer will be able to provide a grammar specification as an input to the tool. The program will be written as an extension of the Emacs text editor, and can therefore

be used by students interactively.

1.2 Context

1.2.1 Functional programming (FP)

Functional programming is a programming paradigm that models programs as mathematical functions. That is, a program defines a mapping of every possible input to exactly one output value. Functional programming is partly characterized by its ‘declarative’ style, in which the programmer directly expresses the desired output, derived from the input.

Students taking FPP are expected to have completed Intro to Computer Science taught in Yale-NUS, which trains them in functional programming with the language OCaml. Coq has a language of programs that is very similar to OCaml, and is in fact written in OCaml.

1.2.2 Proving

In mathematics, a proposition is a statement that either holds or does not hold; a proposition is also sometimes called a theorem or lemma.

Proofs can be defined as a logical argument about whether a proposition holds. Proofs use logical rules to demonstrate that what we are sure of (an axiom) implies the truth of something we were not sure of.

In mathematical proofs, propositions often contain equations, which are statements asserting the equality of two expressions containing variables (unknown values). In equational reasoning, we apply axioms to equations in order to incrementally transform them into something that is clearly true.

1.2.3 Verifiable proofs with the Coq proof assistant

Many proofs in mathematics or computer science are natural language proofs - that is, they are written in a natural language, like English. Even though they may use jargon and formal symbols, they may be considered informal. Since natural languages are often ambiguous, natural language proofs are susceptible to misinterpretation or misconception. Furthermore, informal proofs rely on humans to check for logical errors, but humans are fallible.

On the other hand, just as there programming languages that express a set of instructions to be executed by a computer, there are also domain-specific languages for writing formal proofs that can be automatically, or mechanically, verified by a computer.

Therefore Coq allows us to write formal, verifiable proofs in a structured logical language called Gallina, and will also automatically verify that our proofs are correct.

Proving is done as such: 1. First, we state a theorem (or lemma, proposition, etc) in the logical language of Coq. 2. Then, we solve 'subgoals' generated by Coq (sub-statements we need to prove in order to prove the theorem) by stating a sequence of 'tactics' (the method we use at each step). As we apply each tactic to the current subgoal (by executing each line of code), Coq will progressively transform the subgoal. 3. Once all our subgoals have been transformed into something that is clearly true, our proof is complete. Every proof step has been demonstrated to progress logically from each other; this process can be reproduced by any other user executing the same proof. Thus the the proof is verifiably correct.

1.2.4 YSC3236 Functional Programming and Proving (FPP)

FPP is a course in Yale-NUS College taught by Professor Olivier Danvy, under the Mathematical, Computational and Statistical Sciences major. The class is taken not only by Yale-NUS students, but also PhD and post-doctoral students from the National University of Singapore (NUS) School of Computing (SoC).

FPP introduces students to the Coq proof assistant. Through the course, students gain an appreciation for the interconnectedness of computer programs and logical proofs - which have previously been presented to them as distinct domains of knowledge. For example, they are led to realize that an explicitly written Coq proof exactly corresponds to an equivalent mathematical proof they have written in detail, by hand.

Students engage in weekly assignments consisting of rigorous, progressive exercises involving:

- writing mathematical proofs
- writing programs, and proofs about the properties of programs
- eventually, stating their own theorems and proving them

1.2.5 GNU Emacs Editor

Emacs is a family of real-time text editors which are characterized by their customizability and extensibility. GNU Emacs was written in 1984 by GNU Project founder Richard Stallman.

The user interacts with files displayed in 'buffers' - a view of a text file - via **commands** invoked by 'macros' - keystroke sequences. Feedback and status messages are displayed in a smaller buffer at the bottom of the

screen - the 'minibuffer'. The user can create and dismiss buffers, and multiple buffers can exist without all being on display.

- Emacs is customizable because users can change the behaviour of some commands via parameters, without having to redefine or modify the underlying code of the command itself. Users can also easily redefine key mappings.
- Emacs is extensible because users can write new commands as programs and bind them to new macros.
- GNU Emacs provides a language based on Lisp, Emacs Lisp, that is used to write extensions/programs run within Emacs.

GNU Emacs is used in Intro CS, Intro to Algos and Data Structures, and FPP, so students are expected to have familiarity with its interface and indeed will be required to use it, since the class uses the Proof General interface.

1.2.6 Proof General

Proof General is a powerful, configurable and generic Emacs interface for proof assistants, developed at the University of Edinburgh since 1992. It provides a common interface across various proof assistants, including Coq, and allows users to interactively edit proof scripts.

The interface presents users with three buffers (windows): one buffer in which the Coq script is to be edited, one buffer to display subgoals, and one buffer to display other responses like search results or error messages.

Chapter 2

Building muscle memory

2.1 A skilled discipline

The learning philosophy of FPP is that programming and proving is similar to training in any skilled discipline such as martial arts, cooking, or dance: beginner training should build muscle memory for basic skills and habits.

For example, if you are training to be a chef, but you don't develop proper knife skills early on, this will hurt you for the rest of your career.

Therefore, in the first half of the course, students complete rigorous, progressive exercises in order to practice specific proof techniques and programming habits. In the second half of the course, students can then rely on this muscle memory to write proofs with greater creativity and efficiency. By the end of the course, students will have independently written more proofs than they have ever written in their lives, and all of these proofs would have been verified by Coq.

Chapter 3

Writing proofs

3.1 What could go wrong?

With programming languages, there are usually many ways to write the same program. In the same way, there are many equivalent representations of a Coq proof, because Coq is flexible and allows you to take shortcuts. However, for new learners, this flexibility can be counterproductive. In the context of FPP, several issues arise.

3.1.1 Abuse of tactics

First, students may abuse tactics that have not been introduced in the course.

When students get stuck on a proof, they might Google for related solutions or search the Coq documentation for anything that will ‘solve’ the proof. They might end up using a magical tactic, for example ‘trivial’, as in the latter version of the example proof below.

```
Lemma SSSn_is_3_plus_n :  
  forall n : nat,  
    S (S (S n)) = 3 + n.
```

Proof.

```
intro n.  
rewrite <- (Nat.add_1_1 n).  
rewrite <- (plus_Sn_m 1 n).  
rewrite <- (plus_Sn_m 2 n).  
reflexivity.
```

Qed.

Lemma SSSn_is_3_plus_n :

```
forall n : nat,  
S (S (S n)) = 3 + n.
```

Proof.

```
trivial.
```

Qed.

Under the hood, the ‘trivial’ tactic uses some heuristics to automatically try various strategies to solve the current formula. However, in the first half of the course the focus is for students to understand every single proof step they write, because if students cannot explain what they are doing, they do not really understand it. Therefore, using a tactic like ‘trivial’ completely goes against the objective of the exercise.

Yet these tactics still appear in student submissions, because they might still have the bad programmer mindset of "if it works, its fine". This causes time between resubmissions to be wasted on superficial feedback.

3.1.2 Misuse of tactics

Second, even when students use tactics that have been introduced, they may misuse them by taking shortcuts.

For instance, the `rewrite` tactic is used to apply a rule to the current formula. A rewrite rule is a function that expects specific terms in the formula as arguments; Coq will rewrite the given terms. For example, the rewrite rule below accepts three arguments, `n`, `m`, `p`.

```
Check Nat.add_assoc.
```

```
# Nat.add_assoc : forall n m p : nat, n + (m + p) = n + m + p.
```

However, Coq is flexible with the number of arguments you give it. As the example proofs below demonstrate, you could give the rewrite rule three, two, one or zero of the rewrite arguments required, and Coq will simply pick the first terms in the formula that it can apply the rule to.

```
Proposition add_assoc_nested :
```

```
  forall a b c d e: nat,
    a + b + c + d + e =
    a + (b + (c + (d + e))).
```

```
Proof.
```

```
  intros a b c d e.
  rewrite -> (Nat.add_assoc a b (c + (d + e))) .
  rewrite -> (Nat.add_assoc (a + b) c (d + e)) .
  rewrite -> (Nat.add_assoc (a + b + c) d e) .
  reflexivity.
```

```
Qed.
```

```
Proposition add_assoc_nested :
```

```
  forall a b c d e: nat,
    a + b + c + d + e =
```

$a + (b + (c + (d + e)))$.

Proof.

```
intros a b c d e.  
rewrite -> (Nat.add_assoc a b )  
rewrite -> (Nat.add_assoc (a + b) ).  
rewrite -> Nat.add_assoc.  
reflexivity.
```

Qed.

However, in the first half of the course, the focus is on understanding the proof at a low level. Clearly, students need to be aware of exactly which terms they have changed at every step. Otherwise, they may get stuck in a proof because they applied a rewrite rule to the wrong term, or they might reach a solution without knowing how. Therefore, taking advantage of this shortcut goes against the spirit of the exercise.

Furthermore, this issue is not easy to check manually, especially with assignments that are hundreds of lines long.

These two issues - abuse and misuse of tactics - correspond to issues of **abstract syntax** (what language constructs are represented in the grammar) and **concrete syntax** (what structures are used to represent language constructs) respectively.

Therefore, it would be nice to have a system that can anticipate and identify both abstract and concrete syntax issues, to save both students and the Lecturer's time and help achieve the learning goals of the course.

3.1.3 Other issues

Other issues I have discussed with the Lecturer include:

- arbitrary indentation levels for proof subcases
- inconsistent naming
- breaking style conventions.

All these issues seem to persist across the progression of the module, as well as iterations of the module, despite the Lecturer explaining to the students the rationale for following provided syntactical guidelines, and repeated reminders.

The idea is for the proposed tool to cut down on the amount of *'superficial'* feedback - e.g., 'don't use this tactic, because...', or 'this is bad style, please correct it in this way', etc. - that the Lecturer must give repeatedly to individual students, and instead automatically lead students towards solutions that only require *'substantive'* feedback - e.g., ideas to pursue, possible restructuring of the proof, etc. The less superficial feedback is required, the more time the Professor can spend on providing substantive feedback. Also, students will spend less effort correcting style errors if they do so immediately.

Yet, superficial feedback is not merely incidental. Superficial feedback reflects the formal concerns of the course and helps reinforces good programming habits, which will not only assist the learning experience of students, but benefit them in future endeavors. Therefore, the tool does not simply emphasize pedantic concerns; it makes concrete the formal training prescriptions of the course.

Chapter 4

Solution

4.1 A grammar of grammars

The solution to these issues of abstract and concrete syntax is to develop a system that enforces explicit tactic application within a subset of Coq, amongst other rules. This system will be in the form of a program that takes as input a student's Coq file, as well as a grammar specification provided by the Lecturer. It will output warnings about instances where a syntax rule has been violated. (From here, I refer to 'the program' interchangeably as 'the tool' or 'the parser'.)

The program will be written as an extension of the Emacs editor, so students can execute a command within Emacs to parse the current file, and thus use the tool interactively as they construct their proofs.

The program will thus act as a set of safety rails for students to develop the right habits, in the spirit of learning for the first half of the course. As a result, students will have earlier, automated intervention on syntactical issues in their assignments, and the Lecturer can spend more time on substantive rather than superficial feedback.

However, the parser should not 'hard-code' a grammar that only enforces particular syntax rules. Instead, it should accept and parse a grammar specification that is readable and easily editable, and enforce that grammar. In other words, we need to implement a grammar of grammars. This will allow the Lecturer to modify existing rules or extend them, without having to modify the source code of the parser. This will also make it easier for other course instructors or developers to modify the parser behaviour for their own needs.

4.2 Implementation trade-offs

The first trade-off is whether to write a custom syntax parser by hand, or to use a parser generator.

A parser generator is a program that accepts a grammar specification as input, and automatically generates a parser that implements the grammar. I have identified parser generators intended to be run within Emacs, and written in Emacs Lisp (more on that below).

A parser generator would be ideal - we do not want to reinvent the wheel. Firstly, in theory, there is minimal to no programming to be done. Instead, we declare a grammar using some notation.

Secondly, relying only on the declared grammar makes the program more extensible; it is easier for the Lecturer or other developers to modify existing rules or add new rules, since they do not need to touch underlying code.

However, it will be necessary to grasp the grammar that is required as input in the first place.

A second trade-off relates to the case where we write a custom parser by hand: whether to write it using Emacs Lisp, or another language that I am more familiar with.

Of course, using a familiar language might mean that we get to a first working version faster. But an Emacs Lisp implementation has many benefits:

- Easier for other Emacs developers to build on
- Integrated with Emacs editor features
- Run within Emacs. No need to call external process, no external dependencies, no interoperability issues (e.g. making a call to create a new thread, etc).

This is why it is ideal to use a parser generator that generates Emacs Lisp code.

4.3 Current progress

I have made progress in the following areas:

- Defining a subset of Coq grammar.
- Exploring different types of parsers for different types of languages.
- How to write a parser.
- How to write Emacs Lisp programs.
- How to write an Emacs extension.

In particular, I have written a script in Emacs Lisp that registers an Emacs interactive command, which may be executed while editing a Coq proof. The command is able to take the current buffer and use the Coq shell to parse it for Coq syntax errors. This command would be eventually used to run our custom parser; the idea is to first check that the input is syntactically correct with respect to Coq's grammar. This allows us to develop our custom parser assuming that the input code is already syntactically correct, hence we may define or provide a grammar that only encompasses a subset of Coq's grammar.

Additionally, I have also started trying to use the Semantic Bovine parser generator. Semantic is a framework for writing Emacs packages, and Bovine is a built-in parser generator provided by Semantic. It accepts a BNF (Backus-Naur Formation)-like grammar specification, and generates a parser in Emacs Lisp, which is ideal. The main obstacle for now would be the slightly obscure documentation. The first step I am aiming for would be to specify a simple grammar (such as for arithmetic expressions with infix notation), and ensure that it raises an appropriate error when given a syntactically incorrect input.

4.4 Challenges ahead

The challenges ahead would be to decide on and proceed with an implementation of a program that can enforce a grammar addressing the two issues mentioned. Following which, I can then explore other rules mentioned. I will also be iterating on the tool based on usability feedback from both the Lecturer and FPP students, and the feedback will also be the measure of success for my project.

Chapter 5

References

- GNU Emacs home page <<https://www.gnu.org/software/emacs/further-information.html>>
- Proof General home page <<https://proofgeneral.github.io/>>
- Coq reference manual <<https://coq.inria.fr/distrib/current/refman/>>
- Glickstein, Bob. Writing GNU Emacs Extensions. O'reilly Media, Inc., 2010.

Appendix A

Appendix A

A.1 An example grammar

Here is an example grammar that I might use as input to either a custom parser or a parser generator. It is a subset of Coq's grammar.

Note that the grammar is incomplete, and does not yet implement any of the rules described. The example is more a demonstration of an exploratory exercise to identify the relevant grammar definitions from various separate grammar specifications in the Coq documentation ("The Gallina specification language", "Terms of the calculus of inductive construction", "The tactic language", and "Atomic tactics").

The notation "..." means any string.

```
sentence ::= assertion proof
          | Definition ... .
          | Inductive ... .
          | Fixpoint ... .
          | ... .
assertion ::= assertion_keyword ident ... .
assertion_keyword ::= Theorem
```

```

| Lemma
| Remark
| Fact
| Corollary
| Proposition
| Definition
| Example

ident ::= string
string := char | char string
char ::= a..z A..Z 0...9 | _
proof ::= Proof . tactic_invocations Qed .
        | Proof . tactic_invocations Defined .
        | Proof . tactic_invocations Admitted .
        | Proof . tactic_invocations Abort .
tactic_invocations ::= tactic_invocation | tactic_invocation tactic_
tactic_invocation ::= [- | + | >] tactic .
tactic ::= intro ident
        | intros intro_pattern_list
        | clear
        | exact rule_application
        | apply term
        | split
        | left
        | right
        | rewrite_expr
        | Compute ...

```

```
| Check ...  
| reflexivity  
rewrite_expr ::= rewrite -> rule in ...  
               | rewrite -> term args  
               | rewrite <- term args
```