

Capstone

Jeremy Yew

Capstone

Chapter 1

Context

[YSC3216: Functional Programming & Proving \(FPP\)](#)

[Functional programming \(FP\)](#)

[Proving](#)

[The Calculus of Constructions](#)

[Coq proof assistant](#)

[GNU Emacs Editor](#)

[Proof General](#)

[References](#)

Problem

[Motivating assumptions](#)

[Solution: a grammar of grammars](#)

Chapter 1

The goal of this research project is to assist the learning experience of students enrolled in YSC3216: Functional Programming and Proving (FPP). FPP is a course in Yale-NUS College that introduces students to mechanized proofs. Students exercise equational reasoning by constructing logical proofs, using the formal language provided by the Coq proof assistant (referred to as Coq).

To this end, I implement a tool to specify a subset of Coq, its libraries, and certain style rules, and to enforce their use. The tool parses text files containing Coq proofs submitted by students, and output messages about transgressions. The idea is for students to check that their code adheres to the specification before submitting.

Lastly, the tool is designed as an extension of the GNU Emacs text editor, potentially integrated in the Proof General interface. It should be easy to install, configure, and use, so that it can be useful for other courses using Coq as well.

Context

YSC3216: Functional Programming & Proving (FPP)

FPP is a Mathematical, Computational and Statistical Sciences (MCS) course taught by Professor Olivier Danvy. Through the course, students gain an appreciation for the deep relationship between computer programs and logical proofs - two domains which have hitherto been presented to them separately and independently.

Weekly assignments consist of progressive exercises involving: - writing programs, - reasoning and writing proofs about the properties of programs, - reflecting on the structure and properties of both the programs and the proofs.

Functional programming (FP)

FP is a programming paradigm that models programs as mathematical functions. That is, a program denotes a well-defined 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.

This model contrasts with the other ubiquitous paradigm, imperative programming.

Section on Turing Machine and global state

Imperative programs can therefore be understood simply as a series of explicit statements or instructions to change a program's state in order to obtain some desired output. The output thus depends on external - and thus implicit - parameters, i.e. the global state, and this program is described as 'impure'.

In contrast, a 'pure' FP program (or function, or method, or language) only has explicit parameters, and it is obvious that it will always return the same output given the same inputs.

Line about Turing completeness However, at the low level, declarative programming still compiles into imperative read-write commands that modify state; the translation of declarations into commands is safely abstracted by the compiler.

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).

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 an argument about whether a proposition holds.

Mathematical proofs use logical rules to demonstrate that what we are sure of implies the truth of something we weren't sure of. Therefore we can understand a proof as a function which takes propositions we know or assume to hold (known as "axioms") as input; the output is a theorem that has been proven true. This theorem may then also be used as an input proposition to other proofs.

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

The Calculus of Constructions

TODO.

Coq proof assistant

- Why
 - shortcomings of informal proofs
 - other proof assistants
- What
 - Proofs that can be encoded and machine-verified, in particular proofs about the properties of computer programs. Examples: satisfaction of specification, equivalence...what else?
 - "the *logical language* in which we write our axiomatizations and specifications, the *proof assistant* which allows the development of verified mathematical proofs, and the *program extractor* which synthesizes computer programs obeying their formal specifications, written as

logical assertions in the language."

- How:
 - Declare: specifications, programs, propositions. Write proofs step by step, and execute them.
 - Interactive mode. Subgoals, tactics, etc.
- What is unique: Program synthesis.
- Who: main contributors

Certified OCaml programs may be extracted from Coq proofs.

GNU Emacs Editor

Emacs is a family of real-time text editors, characterized by extensibility, customizability and self-documentation. 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 keystroke commands, with feedback and status messages 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.

GNU Emacs is used in Intro CS, Intro to Algos and Data Structures, and FPP.

Proof General

- Generic, Adaptable
- Three buffers: "The script buffer holds input, the commands to construct a proof. The goals buffer displays the current list of subgoals to be solved. The response buffer displays other output from the proof assistant. By default, only two of these three buffers are displayed. This means that the user normally only sees the output from the most recent interaction, rather than a screen full of output from the proof assistant. Proof General does not commandeer the proof assistant shell: the user still has complete access to it if necessary."
- Other features: simplified interaction, script management, multiple file scripting, a script editing mode, proof by pointing, proof-tree visualization, toolbar and menus, syntax highlighting, real symbols, functions menu, tags, and finally, adaptability."
- Coq without Proof General? CoqIDE, JSCoq. Coq in Jupyter?
 - Do the buffers look the same in CoqIDE?
- <https://proofgeneral.github.io/>
- <https://proofgeneral.github.io/doc/master/userman/Introducing-Proof-General/#Quick-start-guide>

References

- <https://coq.inria.fr/distrib/current/refman/credits.html>
- <https://www.quora.com/What-is-the-difference-between-predicate-logic-first-order-logic-second-order-logic-and-higher-order-logic>
- <https://www.gnu.org/software/emacs/further-information.html>
- https://en.wikipedia.org/wiki/Higher-order_function
- https://en.wikipedia.org/wiki/Higher-order_logic
- <https://www.gnu.org/software/emacs/emacs-paper.html>
- https://en.wikipedia.org/wiki/GNU_Emacs
- <https://www.emacswiki.org/emacs/EmacsHistory>

Problem

There are two main problems in the current learning process of FPP.

1. Coq proofs contain 'magic' tactics.
 - Students discover, via documentation or Googling, advanced tactics that have not been introduced to class, and use them in their solutions.
 - These tactics often introduce some form of automated reasoning that also hides the proof steps generated under the hood. The student is satisfied as long as "it works", without understanding the proof itself.
2. Coq proofs contain undesirable code style.
 - Students depart from the prescribed style demonstrated in class. For example:
 - Arbitrary indentation level for subcases.
 - Implicit arguments given to tactics (arguments are left blank, but inferred by Coq).
 - Bad naming.
 - These stylistic departures, while syntactically correct and accepted by the Coq interpreter, are counterproductive to FPP's learning goals. Bad style reflects disorganized thoughts, and sometimes lack of understanding. Bad style make proofs harder to read and therefore also harder to complete.

These two formal issues correspond to a lack of rules in terms of **abstract syntax** (what tactics are allowed) and **concrete syntax** (how code should be styled). Unfortunately, the power and flexibility that Coq affords, intended to make it more user-friendly, can be detrimental to beginners.

Furthermore, these issues seem to persist across iterations of the module, despite verbal and written reminders and repeated feedback.

Motivating assumptions

1. Written and verbal reminders are not efficient in training specific behaviour.
 - If reminders were enough to change behaviour, then we would not need to build a tool. We could just remind people harder. The experience of the Professor suggests otherwise.
 - Therefore, we should build a tool that automates the 'reminding'.
2. Bad style reflects disorganized thoughts, and sometimes lack of understanding.

Example.

3. Bad style make proofs harder to read and also harder to complete.

Example.

4. Neglecting to reinforce good habits allows bad habits to develop.

Section on building skills via muscle memory. Analogy of cooking: basic techniques + mis en place. Evidence from psychological literature.

- Therefore, programming and proving at the beginner level should be generally prescriptive.

Overall, these assumptions motivate the building of a tool that provides immediate prescriptive feedback on the abstract and concrete syntax of Coq code.

The idea is for the 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 Professor 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. The tool also reduces the need for the Professor to repeatedly make their case for why a student's submission is unacceptable - they can just point to the warnings generated (if not already addressed by the student, as they should be).

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

Solution: a grammar of grammars

The solution to the problems of magic tactics and bad style is a tool that introduces 'safety rails' that will mechanically guide the student towards well-formed proofs, that satisfy a specification of **abstract** and **concrete** syntax - i.e., a **grammar**.

The tool should not 'hard-code' the grammar intended by the Professor of FPP; instead, it should accept a specification that is readable and easily editable. This will allow the Professor to modify existing rules or extend them, and also allow any other course instructors to write their own specifications for use in other course, with no need to modify the rest of the code.

Therefore, at the high level, the tool will:

- accept a specification of a subset of Coq, its libraries, and certain style rules (e.g. indentation, ordering of statements, explicit argument application) in the form of a text file
- accept a text file containing Coq code
- output warnings about instances where code fails to meet the specification

However, in order to apply the specified grammar to the Coq code, the tool needs to understand how to read any given grammar. Therefore what we actually need is to specify a **grammar of grammars**.

Precedent: Racket