Capstone

Jeremy Yew

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 - proofs that can be verified automatically by a machine. Students exercise logical and equational reasoning by programming proofs in the logical 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 returns 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 functions 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.

Verifiably correct proofs with the Coq proof assistant

Most proofs in mathematics or computer science, even though they contain formal symbols and jargon, could be considered 'informal' in that they are written in natural human languages, which do not confirm to strict syntax. Natural languages are implicit and ambiguous, and thus open to misinterpretation. Furthermore, informal proofs rely on humans to check for logical errors, but humans are fallible.

Coq allows us to formalize proofs in a structured logical language, and automatically verify that our proofs are correct. In particular, we can write both specifications and programs, and prove that programs fulfill their specifications (amongst other properties). Lastly, we can extract certified programs from Coq, i.e. programs that are guaranteed to fulfill their specifications.

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.

Therefore, the Coq proof assistant has three components:

1. Language of programs and proofs
    - The logical language in which we formalize theorems, specifications, programs, and proofs, is the Calculus of Inductive Constructions (CIC).
    - In the CIC, every term has a particular 'type' (which we can think of as a kind of category) - variables, functions, proofs, and even types themselves, and terms are governed by a set of typing rules.
    - The CIC also contains a 'mini' functional programming language, with which we write our programs.
2. Proof-checker:
    - The program that runs a type-checking algorithm to determine if each proof step is logically valid, and also generates intermediate subgoals and other output.
3. Program extractor:
    - We can easily build certified functional programs by extracting them from either Coq functions or Coq proofs of specifications. Currently, OCaml, Haskell and Scheme are supported.

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.

- GNU 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.
- GNU Emacs is **extensible** because users can write new commands as programs and bind them to new macros.

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

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 edited, one buffer to display subgoals, and one buffer to display other responses like search results or error messages.

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
- https://proofgeneral.github.io/
- https://proofgeneral.github.io/doc/master/userman/Introducing-Proof-General/#Quick-start-guide
- https://hal.inria.fr/hal-01094195/document

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. Programming and proving at the beginner level should be generally prescriptive.
    - Students beginning to learn how to program and to write proofs often develop certain bad habits (such as those described in the 'Problem' section). These habits, if left unchecked, will affect their future endeavors.
        - For example, inappropriate indentation makes any code less readable, regardless of domain or language.
        - Relying on implicit logic too often also makes written proofs harder to understand.
    - Therefore, we should reinforce good habits at an early stage, in the same way that training in many sports or disciplines begin with prescriptive routine and repetition. See the next section 'Analogy: coding is like cooking'.

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 reinforces 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.

Analogy: programming is like cooking

Consider different fields of mastery and skill that we call 'disciplines' - for example, martial arts, dancing, acting, cooking, etc. Usually, they are called as such because they indeed require discipline to achieve mastery.

Discipline is required to stick to the sometimes painfully boring or mundane routines and practice exercises prescribed by experienced teachers. These exercises are often designed to build understanding as well as and muscle memory in the basics of the craft. Even in disciplines that also require some form of flexibility and creativity, students are usually only able to exercise freedom effectively when they have mastered the basics.

Similarly, both programming and proving may be considered disciplines. Cooking in particular provides a nice analogy.

In the formal training of French cooking, heavy emphasis is placed on mastering the basics: basic ingredients of french cooking, knife skills for every type of ingredient, cooking techniques for every types of dish, recipes for the canonical sauces, etc.

> Find quote about how strong knife skills should be/the kind of practice it takes. Also, mise en place: external form reflects but also affects the mind. Anthony Bourdain's story.

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**.

An example grammar

Adopted from multiple sub-grammars of Coq:

1. Vernacular of Gallina https://coq.inria.fr/distrib/current/refman/language/gallina-specification-language.html#the-vernacular
2. Terms of the Calculus of Inductive Construction: https://coq.inria.fr/distrib/current/refman/language/gallina-specification-language.html#terms
3. Tactic language: https://coq.inria.fr/distrib/current/refman/proof-engine/ltac.html#the-tactic-language
4. Atomic tactics: https://coq.inria.fr/distrib/current/refman/proof-engine/tactics.html#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_invocations
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
```

To be accounted for:

```
// How to verify number of args?
intro_pattern_list
term
bindings
Ltac
Require
Notation
Comments
occurence clauses
```

Todo

- [ ] Insert to latex, see num pages.
- [x] Check if appendix is included in page count.
    - [x] It seems only references are excluded.
- [ ] Sections to refine or add:
    - [x] Complete Coq.
    - [x] Complete Proof General.
    - [ ] Rewrite problem, motivating assumptions.
    - [ ] Add to FP.
    - [ ] Rewrite Programming and cooking.

- [ ] Add Examples of bad code.
- [x] Initial GoG BNF.
  - [ ] List missing pieces.
- [ ] Chapters 1-3.
- [ ] Try writing in emacs-lisp.