Coq

Sacha Peterson
speterson@oxy.com
Occidental College

1 Introduction

The Typma programming language project will use a proof assistant to write and prove the correctness of the representation of the semantics of Typma in the codebase. I have not done any computer science proofs before I needed to learn the basics before I start using the proof assistant Coq to prove code for the Typma codebase. In order to learn Coq, I started the coursework from the online book Pierce (2021) [1], doing the first three chapters for this paper. By working through the book, I will have the foundation to prove the correctness of my implementation of the Typma programming language.

2 Methods

The task for this tutorial was writing a variety of functions, theorems, and proofs split into three chapters with the goal of learning how to prove code in a proof assistant. Chapter one "Functional Programming in Coq" from Pierce (2021) [1], covers the basics of functional programming such as algebraic data types and pattern matching in Coq. The Coq proof assistant is based the programming language Gallina, a standard functional language, with the purpose of Coq being to prove the correctness of Gallina programs. Coq does not provide natural numbers, booleans, tuples and related operations by default and Pierce (2021) [1] had me create them from scratch to show and learn about the power of Coq's type system.

This chapter also covers three basic proof methods. The first method was simplification, this takes an expression that can be reduced with simple steps. For example:

```
1 + 3 = 4 + 0 -> 4 = 4
```

The second method was rewriting, this technique allowed me to take an expression that is too complex for a simplification but with the use of a lemma can be simplified. For example:

```
x + (y + z) -> x + y + z
```

Simplification will not do anything to change the parenthesis, instead I would define a new lemma that specifically proves this associativity of addition, and then call that lemma in the proof that needed it. The final method was case analysis with destruct: sometimes proofs have variables that can have a range of values that will change the outcome, like the commutative property of boolean expressions:

```
b and c = c and b
```

Depending on what b and c are, the expression will give a different answer making this too complex for simplification. Using the destruct case analysis, I can look at the cases where b is true and false, then nested inside of those, the cases where c is true and false. These proof tactics are the basis for proofs in Coq and are used throughout the book.

Chapter two "Proof by Induction" from Pierce (2021) [1], introduces induction as a technique to write proofs. While destruct is a useful tool for case analysis, some proofs like ones involving recursive functions or structures are too complex for the destruct technique. For these proofs, induction will be able to solve it. The induction tactic will provide an inductive hypothesis for the recursive cases. For example, the proof for a number being subtracted by itself equaling zero requires an inductive case analysis in order to over all possible natural numbers:

```
n - n = 0
```

Chapter three "Working with Structured Data" from Pierce (2021) [1], teaches how to use lists in functions and proof. Coq does not provide lists be default, so I had to define lists then apply them to the strategies from the first two chapters in proofs. Because lists are a recursive data structure, proofs involving lists frequently make use of induction.

Pierce (2021) [1] is a combination of lessons with exercises to complete. Based on the description of a function, theorem, or proof, I had to figure out what the answer was based on previous explanation and examples. In addition, each chapter imports the work from the previous chapter, so all defined functions and proofs were available to use once I completed them. Here is an example of an exercise from the book:

```
Fixpoint eqblist (l1 l2 : natlist) : bool
```

```
  (* REPLACE THIS LINE WITH ":=
      _your_definition_ ." *). Admitted.
Example test_eqblist1 :
  (eqblist nil nil = true).
 (* FILL IN HERE *) Admitted.
Example test_eqblist2 :
  eqblist [1;2;3] [1;2;3] = true.
(* FILL IN HERE *) Admitted.
Example test_eqblist3 :
  eqblist [1;2;3] [1;2;4] = false.
 (* FILL IN HERE *) Admitted.
Theorem eqblist_refl : forall l:natlist,
  true = eqblist l l.
Proof.
  (* FILL IN HERE *) Admitted.
```

This exercise first required me to write out a recursive function that evaluated if two lists were equal. Then I had to check if my function was working correctly with two test cases written as basic proofs. Finally, I had to write a theorem and prove to see if my function was correct in general. The answer is not given anywhere, so I had to use what I learned to solve the proof. I used VSCode with the Coq extension to interface with the proof assistant. I could have used another program like Emacs, but the choice of IDE does not matter.

3 Evaluation

Any proof that Coq the type-checker accepts is an evaluation of the correctness of that code. When a function is written, the Coq complier does a type check on if the function type is consistent with the declarations as well as checking if any pattern matching is missing a pattern. The functions by themselves are not fully correct until a proof is provided to show that the function is correct and every function I had to write had a theorem and proof to show the correctness of that function. Writing a proof that the Coq complier says is correct is evidence that the task has been performed correctly. This does assume that the Coq compiler is correct, but that is a safe assumption because Coq "has been under development since 1983 and that in recent years has attracted a large community of users in both research and industry" as seen in Pierce (2021) [1].

4 Discussion

I was able to complete almost every exercise in the first three chapters. There were some exercises labeled as advanced and some exercises focusing on binary numbers that I did not do because I wanted to focus on the basics of Coq. I may go back and do these advanced exercises at a later date when I want or need to learn the details they teach. I could have used a test file provided with the textbook to check my code to see if it was correct but this meant for ease grading in a class and provides no advantage for checking correctness over just compiling the files in Coq. By virtue of the Coq compiler saying that I have completed the proofs, I succeed in all of the exercises I did. Many exercise have multiple solutions and while I do not know if my proofs are written in the best way, the focus was on getting a correct answer which I was able to do. In addition to correctly doing the exercises, I learned a lot of knowledge about how to write proofs for code. Pierce (2021) [1] is a fantastic resource and I will able to incorporate proofs into the Typma programming language project as a result. I will have to complete a few more chapters to learn how to instantiate full programming language semantics in a proof, so I will continue to do the exercises for Pierce (2021) [1] to gain that knowledge.

References

[1] Pierce, Benjamin C. *Software Foundations Volume 1: Logical Foundations*. Electronic. Aug. 2021. URL: https://softwarefoundations.cis.upenn.edu/lf-current/index.html.