CPSC 490 Project Description

For my project, I plan to learn the basics of formal verification methods for software, and then using this knowledge to actually verify some software. Specifically, I will learn how to use the proof assistant Coq, which allows provides a means of translating the logic of formal mathematical proofs to something a computer can understand. I have been following the textbook *Software Foundations* by Pierce et al..

Coq is a functional language, which lends it many structural features, known as “tactics”, that naturally work with the logic of proof-writing. For example, the “destruct” tactic uses Coq's pattern-matching ability to carry out proof by casework. The “induction” tactic takes a claim for all integers as input, and splits it into two claims: a base case and and inductive hypothesis. This really is the same induction as that in pen-and-paper mathematics, just being run through a computer.

Proofs in Coq work by defining a list of goals (the statements to be proven) and an environment containing the relevant variables, quantifiers, and hypotheses. The direction of proof can proceed either “forward” or “backward” depending on context, but the idea is to eventually convert the goals into a set of simple, reflexive equalities.

Coq also allows types to be defined inductively. Perhaps the simplest example is the natural numbers, where other than the primitive “O” (which is akin to the number 0), all other natural numbers are just the successor of another one. That is, there is a construction function “S” that takes a natural number and returns a new one, so we can think of S(O) as the number 1, S(S(O)) as the number 2, and so on. The utility of such inductive types is that proofs in Coq have essentially no reliance on the built-in features. All the logic can (and should) be explicitly written by the programmer / proof-writer, so there are no black-boxes.
Theorems in Coq are objects, just like numerical values or functions. One way to think of the theorem is as a logical derivation, which takes an assertion (aka “argument”) as input, runs it through the derivation, and returns another logically valid statement. Thus, theorems provide a concise way of building logically stronger statements from simpler ones. Again, this is exactly the role theorems play when writing proofs in regular mathematics, but structured in a way so that a computer can carry it out.

All this discussion is to give a high-level overview of my introduction to Coq in the last few weeks. My current rate of progress is roughly 2-3 hours per chapter, depending on how many of the exercises I try to work on. I believe by the end of next week (2/19), I should be able to complete the chapter on Program Equivalence, which Professor Shao suggested I set as a conceptual target.

By then, I should have a good feel for the basics of Coq. Professor Shao and I have discussed finding some simple but real-world pieces of already verified software, perhaps something from his research group, that I can practice verifying with guidance from members of his group. Depending on how this goes, we will determine how ambitious my own verification project will be. Some options we have touched on include modules from his group’s CertiKOS kernel (which I am familiar with from taking CS422) or cryptographic libraries. I can provide an update when these decisions are made.