CS51 Final Project - Final Submission

<Verification of Sorting Algorithms using Coq>

Aleksandar Makelov / Ben Wetherfield / Chan Kang / Michael Fountaine

---

Narrative

The central motivation behind this project was to explore formal proofs of correctness for computer programs using the proof assistant Coq. As the algorithms used in our world become more complicated and we entrust them with more responsible tasks, it has become an issue of increasing importance to make sure that they do exactly what we want them to do. It is difficult for humans to deal with such complexity, and therefore automated proof assistants become necessary. While, in the past, people have been skeptical of the utility of such tools, nowadays they are becoming feasible and indeed necessary.

One thing we found particularly interesting was that even for relatively simple algorithms, such as sorting algorithms, proof assistants have been helpful in discovering a bug - one example is the timsort algorithm used by default in the Java and Python programming languages, which was found to be incorrect [1].

This motivated us to look into proving program correctness from the lens of sorting algorithms, and try to prove some of them are correct. Our project thus had two main goals: first, completing exercises from the textbook Software Foundations in order to get accustomed to Coq, and second, to prove correctness of a simple sorting algorithm, insertion sort, in a *fully* self-contained manner (that is, without using anything but the basic functionality provided by Coq). Although we weren't able to complete the proof, we reduced the correctness of insertion sort to several basic facts about the less-than-or-equal relation on the natural numbers.

[1] de Gouw, Stijn, et al. "OpenJDK's java. utils. Collection. sort () is broken: The good, the bad and the worst case⋆."

Links to the specifications / Video

- **Initial:** https://github.com/mfount/chicken/blob/master/specs/draft/draftspec.pdf
- **Second:** https://github.com/mfount/chicken/blob/master/specs/final/finalspec.pdf
- **Modified:** (after switching the core features of the project to exercises in Software Foundations): https://github.com/mfount/chicken/blob/master/specs/checkpoint/checkpoint.pdf
- **Video:**

Final Report

➔ How good was your original planning?

We underestimated the difficulty of proving theorems in Coq -- we expected that it would be very similar to writing formal proofs in mathematics, but there were a bunch of additional proof techniques specific to Coq. Our initial goal in this project was to prove the correctness of timsort, but we ended up changing our project to completing the first several chapters of Software Foundations.

➔ How did your milestones go?

We ended up changing the main feature of our project from verifying timsort to completing most of the exercises in Software Foundations as well as verifying insertion sort and merge sort. Unfortunately, we didn't have to time to verify merge sort.

➔ What was your experience with design, interfaces, languages, systems, testing, etc.?

It was oftentimes very frustrating when we couldn't prove a trivial theorem due to our inexperience with Coq. For example, the transitivity of 'less than or equal to' operator was rather challenging for us to tackle -- all of us put at least several hours and even after those hours we only managed to prove it assuming another basic fact about arithmetic.

➔ What surprises, pleasant or otherwise, did you encounter on the way?

The biggest frustration was almost always with the syntax and idiosyncrasy of Coq. All of us, being mathematics concentrators, are used to writing proofs in the context of mathematics, so whenever we couldn't prove something in Coq that would be trivial in mathematics, it was more frustrating than it probably would've been for other people.

In particular, one of the main idiosyncrasies of Coq (and undoubtedly other proof systems) is that it is very important not just *what* a given theorem (or definition) *means*, but *how* it is *formulated* in the syntax of Coq. There are often many wildly different ways in which a given statement can be captured in formal language, and it is crucial to pick a formulation that makes the proofs easier; for example, if we are defining several functions inductively on lists, and we want to prove relationships between these functions, our life will be much easier if we define all of them by reduction on the left.

However, we all felt that the gratification of finishing a proof in Coq was quite great. When we were able to eliminate all the goals and all that was left on the screen was "No more subgoals," it felt extremely satisfying.

➔ What choices did you make that worked out well or badly?

About two weeks before the deadline, we realized that our initial goal -- proving the correctness of timsort -- was a bit too difficult considering how much we knew at the moment (it may still be). Realizing that early on and asking professor Morrisett for advice was definitely the best choice we made. Since this project felt much more doable than initially.

➔ What would you like to do if there were more time?

In our verification of insertion sort, we struggled to prove the transitivity of the binary operator ‘less than or equal to.’ We somehow managed to prove it assuming some other near-trivial fact, but if we had more time, we would definitely try to eliminate that assumption from our verification, perhaps by some more careful application of induction. Also, we would actually attempt to write the verifier for merge sort. Now that we’ve established many basic properties of sorted lists as we worked on the proof of insertion sort, we can use them again in the proof of merge sort.

Also, if we had more time, we would try to go through more materials in Software Foundations. As we think about what the insertion sort should look like, we learned that there are a bunch of other useful materials that would’ve helped us a lot if we had known them earlier.

➔ How would you do things differently next time?

If we were to do this project again, we would try to learn Coq more thoroughly before diving into writing a verifier. If we had gone through the entire textbook first before starting to think about the verification of sorting algorithms, we probably would’ve learned about the inductive definition of ‘less than equal to’ operator for which the book has the proof of transitivity.

➔ What was each group member's contribution to the project?

Although we collaborated on everything to some degree, Chan and Ben focused on the exercises in Software Foundations and Alex wrote the verifier for insertion sort. Michael was in charge of planning, organizing and write-ups.

➔ What is the most important thing you learned from the project?

As we have mentioned earlier, there were trivial theorems in mathematics that we had hard time proving in Coq. But at the same time, using Coq to prove those trivial-looking theorems made us realize that we have regarded some theorems as trivial just based on our intuition. For example, proving the commutativity of multiplication was not trivial at all in Coq, although it seems rather trivial intuitively. Trying to prove the commutativity of multiplication in Coq forced us to use the definitions and theorems (and only the definitions and theorems), proof systems like Coq prevent us from resorting to intuition, which often fails us.