An Engineer's Self-Taught Journey with the Rocq Proof Assistant

Pierre-Emmanuel Wulfman[1]

[1]Graduate School of Mathematics, Nagoya University, Japan

Abstract

This talk shares a testimony of how a software engineer ended up learning the Rocq proof assistant without the support of an academic environment. In doing so, I aim to provide an outside perspective on the quality of the available online resources, the difficulties encountered, some highlights and shortcomings, and some proof design patterns that emerged spontaneously while proving a novel algorithm over the course of a few weekends.

1 Introduction

As a community of experts, it is often difficult to step back into the shoes of a newcomer. What is trivial for some can be inscrutable for others. In an attempt to bridge this gap, I offer my experiences as a software engineer who learned Rocq as an autodidact in his spare time.

Let me first share my background, since what I already knew before starting had a significant influence on my learning process. I am an engineer in Information Technology by training, and I have worked for several startups as a developer in various languages—notably as a compiler engineer for Tezos. So I was already familiar with writing mathematical proofs on pen and paper, and I had taught myself OCaml and some type theory when I started that position. However, I hadn't studied formal logic such as category theory or the calculus of inductive constructions.

2 Learning Rocq and Automated Theorem Proving

The first step in any learning process—so trivial that we often forget it—is to know that what you want to learn actually exists, and to be motivated enough to invest the time and effort into it. In my case, this went very smoothly. Several of my colleagues were using Coq to certify our compiler. I even tweaked some data structures, inferring their behavior from the rest of the codebase. So it was on my radar, but more as a curiosity—something that everyone said was very difficult to use. As a result, it remained at the bottom of my priority list until I had proper motivation.

That motivation came when I had to implement a red-black tree in OCaml, and I couldn't find any deletion algorithm in the literature. So I ended up implementing my own and drafting a paper on the subject. When asking for advice on how to submit it, I was told that in order to meet publishing standards, I needed to provide an automated proof of correctness—which meant I had to learn Coq.

Even then, it took me another half-year to actually get started, partly due to lack of time, and partly due to not knowing where to begin. Of course, I visited the official website, but I only found the reference manual, which at a glance looked like a formal grammar specification rather than an introduction to usage. I then asked my colleagues for advice, and they pointed me to the *Software Foundations* series.

From there, I had a clear path. My objective was not to become an expert (yet), but to prove my algorithm in order to publish it. So my roadmap was to go through the first book to get familiar with the tooling, learn the keywords to write data structures and algorithms, and then to write theorems and learn the various tactics available to prove them. After that, I practiced on Okazaki's insertion algorithm. Finally, I applied what I had learned to my own algorithm.

3 Challenges When Tackling the Problem

The initial learning process went quite smoothly. Most of the concepts were familiar to me, so the examples and exercises didn't take too long. The problems really started in phase two.

I had already implemented and tested the algorithms in OCaml. A copy-paste of the code, followed by a few modifications to fit the Rocq syntax, and I was ready to start writing theorems.

And this is where all the problems appeared at once. Should I write my properties using the `Inductive` keyword or the `Definition` one? I had defined my tree as polymorphic, so the comparator was polymorphic too—how should I model the comparison function? In the middle of a proof, I found myself stuck because I needed to use properties of order relations—how could I model this while keeping a generic comparator and a reusable structure? When should I unfold a `Fixpoint` definition, and how could I keep the goal state clean?

But the most difficult part for me was learning how to split a theorem into smaller lemmas. I began with the statement I wanted to prove, which was trivial to write:

$$\forall X : \text{Type}, v : X, t : \text{Tree } X, \text{ if } t \text{ is red-black, then insert } t\ v \text{ is also red-black.}$$

But when trying to prove it, I quickly ended up with five to seven levels of indentation and a goal that became unintelligible.

So I began proving the same property on subroutines, intending to use them in the main proof to handle complexity. In doing so, I got stuck on the main proof because applying the lemmas caused me to lose contextual information and the hypotheses I needed to prove the main theorem. That's when I came to understand what my colleagues meant when they said the hardest part is choosing the right invariants.

After two hyper-focused weekends, going back and forth between different approaches, I felt stuck in a live lock. So I stepped back and searched for Coq proofs of regular binary search trees, hoping to extrapolate some design patterns. That's how I ended up in Volume 3 of the *Software Foundations* series, which also covers Okazaki's algorithm. After analyzing and reproducing those proofs, I was finally ready to tackle the `delete` algorithm.

For the delete algorithm, I encountered the same issues—but without a book to guide me. After another two hyper-focused weekends of experimentation, I discovered the right invariant and completed the proof.

4 Discussion

Many of the difficulties I encountered could have been avoided if I had taken the time to properly learn Coq in depth. But an engineer's job is to produce, not to study; to write code that works, even if the design is terrible. So engineers seek actionable knowledge—fast—even if it's incomplete. This is especially true in startup environments, where survival is uncertain, and the short term must be prioritized. I would be surprised if startups were the target audience for the Rocq proof assistant—but even large companies and academic researchers have deadlines.

Imagine a geometer who thinks Rocq might be useful to guide their research. Should they invest a year studying a completely different field, or stick to pen and paper to avoid losing focus? What about a team of experimental physicists who want to guarantee that the instructions they send to their collider actually do what they intend?

Beyond the entry barrier, one thing I found particularly time-consuming was **refactoring**. During development—and even during execution—a codebase is alive. Regularly, you must modify your data structures. Often, you start with a certain design pattern and later realize it no longer fits the evolved project. Sometimes you need to redo the entire architecture. But doing so invalidates your proofs. One could argue that you should wait until the product is finished before proving it—but that assumes there *is* a finished product. Even after a version is released, development on the next version starts from the previous one. That's why we prioritize readability and maintainability.

In the proof I wrote—what would amount to an epsilon of a real project—refactoring often broke the proofs. For example, when working on your tenth theorem, you realize you need to reorder a function's arguments. Then you must go back to proof number two, because the naming engine assigns different names to your hypotheses.

This led me to a pattern where I would write a tactic—for instance, one that case-matches the proof state of sometimes 30 cases—inside a `repeat` block. Then I would write my proof using `;` instead of `..`

This had the effect of both "brute-forcing" the proof—applying regular patterns until they no longer match—and making the proof naming-independent. That way, I could directly return to where I left off after a refactor.

This is just a naive workaround for a much deeper problem. My superficial knowledge of Coq means I don't know what other patterns exist. But if such patterns *do* exist, it would be valuable to share them with the community—and if they don't, then this is an area that deserves more attention.