10 Years of Superlinear Slowness in Coq

Jason Gross[1,2] and Andres Erbsen[2]

[1] Machine Intelligence Research Institute, Berkeley, CA, USA
[2] MIT CSAIL, Cambridge, MA, USA
{jgross,andreser}@mit.edu

Context

In most programming languages, asymptotic performance issues can almost always be explained by reference to the algorithm being implemented. At most, the standard asymptotic performance of explicitly used operations on chosen data structures must be considered. Even the constant factors in performance bottlenecks can often be explained without reference to the implementation of the interpreter, compiler, nor underlying machine.

In 10+ years of working with Coq, we (the authors of this proposal and their colleagues) have found this pattern, which holds across multiple programming languages, to be the exception rather than the rule in Coq! This turns performant proof engineering, and especially performant proof automation engineering, from a straightforward science into an arcane form of wizardry.

By presenting in detail a sampling of examples, we propose a defense of the thesis: *Performance bottlenecks in proof automation almost always result from inefficiencies in parts of the system which are conceptually distant from the theorem being proven.* Said another way, *debugging, understanding, and fixing performance bottlenecks in automated proofs almost always requires extensive knowledge of the proof engine, and almost never requires any domain-specific knowledge of the theorem being proven.* Even worse, we know of no systematic proposal, nor even folklore among experts, of what primitives and performance characteristics are sufficient for a performant proof engine. There is no POPLMark for Proof Engines (yet).

We hope to start a discussion on the obvious corollary of this thesis: *This should not be!*

Our presentation, we hope, will serve as a call for designing (and eventually implementing) an adequate proof engine for *scalable performant modular proof automation.*

Presentation

There are three qualities of our experience with engineering proof automation that we hope to drive home to our audience:

1. Coq performance bottlenecks are often *superlinear*, sometimes even *exponential*.
2. Solving Coq performance bottlenecks is like playing whack-a-mole. The experience is *broadly predictable*—certain areas of the system are at fault more often than others—but *seemingly random in any specific instance*. Futhermore, there is (almost) always another performance issue lurking around the corner when you want to scale the same proof technique to larger applications.
3. Understanding Coq performance bottlenecks involves knowing many unrelated facts, not learning a coherent framework. Almost always, these facts feel like historical accidents. The explanation of any given bottleneck is more often than not "the thing that happened to work for small proofs has inadequate asymptotics" and not "the implementors made a deliberate tradeoff or a mistake in analysis or implementation".

Example 1: Four Thousand Millenia is Too Long!

We propose to present first the example of Section 2.2 of "Performance Engineering of Proof-Based Software Systems at Scale" [Gro21]. This is an example from Fiat Cryptography which involved generating C code to do arithmetic on very large numbers. The code generation was parameterized on the number of machine words needed to represent a single big integer. Our smallest toy example used two machine words; our largest example used 17. The smallest toy example took about 14 seconds. We were never able to compile the largest example, but based on the the compile-time performance of about a hundred smaller examples, we expect it would have taken over four thousand *millennia*!

We may also present a whirlwind tour of tactics that can do unexpected conversion, including tactics that shouldn't be doing conversion at all, tactics that have a non-obvious need to do conversion, and tactics that result in bad or duplicated conversion during `Qed`.

We hope these examples will give the audience a taste of what it's like to work in an exponential domain, as well as of the whack-a-mole quality of resolving performance bottlenecks.

Example 2: The Slowness of Evars and Contexts

We plan to present performance bottlenecks in either the example of proving well-formedness of large PHOAS trees, or in proving weakest-precondition correctness of algorithms like CHACHA20. In both of these examples, performance issues in context- and evar-management make the proof engine unusably slow. We may present our reflective solution to the problem as a demonstration of the pain of working around performance issues. We hope that this example will give the audience some understanding of what it looks like for performance bottlenecks to be simultaneously pervasive and far from the problem being solved.

Example 3: Rewriting is Hard

One of the most oft-used forms of proof automation is equational reasoning. Coq's built-in rewriting tactics are frequent performance bottlenecks in proofs involving equational issues. Often the performance of rewriting scales superlinearly in variables that should should add at most constant overhead to rewriting, such as the number of occurences of the head constant *even when there are no matches of the full pattern*.

One especially tricky problem is ensuring that `rewrite` performance scales linearly along all the axes it should. For example, `autorewrite` is superlinear in the number of rewrites because it duplicates the entire goal for every rewrite. While `rewrite_strat` fixes this particular problem, it is still superlinear in the number of binders under which rewriting occurs. Designing an adequately performant rewrite on top of a modular proof engine is an interesting challenge.

We plan to sketch out the challenges of designing such a rewriting tactic, and, time-permitting, we may present an algorithm that is capable of generating linearly-sized proof terms. Relevantly, even these linearly-sized proof terms cannot be checked by Coq in constant time, despite the fact that this should be possible in theory!

See also "Performance Bottlenecks of Proof-Producing Rewriting" in Section 4.5.1 of [Gro21].

References

[Gro21] Jason S. Gross. "Performance Engineering of Proof-Based Software Systems at Scale". PhD Thesis. Massachusetts Institute of Technology, Feb. 2021. URL: https://jasongross.github.io/papers/2021-JGross-PhD-EECS-Feb2021.pdf.