Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

Proposal for Thesis Research in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Title: Performance Engineering of Proof-Based Software Systems

Submitted by: Jason Gross
258 Prospect St, #1L
Cambridge, MA 02139

(Signature of Author)

Date of Submission: June 17, 2020

Expected Date of Completion: September—December 2020

Laboratory: Computer Science and Artificial Intelligence Laboratory

Brief Statement of the Problem:

The proposed research is a study of performance issues that come up in engineering large-scale proof-based systems in Coq. The thesis presents lessons learned about achieving acceptable performance in Coq in the course of case-studies on formalizing category theory, developing a parser synthesizer, and constructing a verified compiler for synthesizing efficient low-level cryptographic primitives. We also present a novel method of simple and fast reification, and a prototype tool for faster rewriting and customizable reduction which does not require extending Coq's trusted code base.

Massachusetts Institute of Technology
Department of Electrical Engineering
and Computer Science
Cambridge, Massachusetts 02139

Doctoral Thesis Supervision Agreement

To: Committee on Graduate Students
From: Professor Adam Chlipala

The program outlined in the proposal:

Title: Performance Engineering of Proof-Based Software Systems
Author: Jason Gross
Date: June 17, 2020

is adequate for a Doctoral thesis. I believe that appropriate readers for this thesis would be:

Reader 1: Professor Nickolai Zeldovich
Reader 2: Professor Saman Amarasinghe

Facilities and support for the research outlined in the proposal are available. I am willing to supervise the thesis and evaluate the thesis report.

Signed: _____
Associate Professor of Computer Science

Date: _____

Comments:

Massachusetts Institute of Technology
Department of Electrical Engineering
and Computer Science
Cambridge, Massachusetts 02139

Doctoral Thesis Reader Agreement

To: Committee on Graduate Students
From: Professor Nickolai Zeldovich

The program outlined in the proposal:

Title: Performance Engineering of Proof-Based Software Systems
Author: Jason Gross
Date: June 17, 2020
Supervisor: Professor Adam Chlipala
Other Reader: Professor Saman Amarasinghe

is adequate for a Doctoral thesis. I am willing to aid in guiding the research and in evaluating the thesis report as a reader.

Signed: _____
Professor of Electrical Engineering and Computer Science

Date: _____

~~Comments:~~

Massachusetts Institute of Technology
Department of Electrical Engineering
and Computer Science
Cambridge, Massachusetts 02139

Doctoral Thesis Reader Agreement

To: Committee on Graduate Students
From: Professor Saman Amarasinghe

The program outlined in the proposal:

Title: Performance Engineering of Proof-Based Software Systems
Author: Jason Gross
Date: June 17, 2020
Supervisor: Professor Adam Chlipala
Other Reader: Professor Nickolai Zeldovich

is adequate for a Doctoral thesis. I am willing to aid in guiding the research and in evaluating the thesis report as a reader.

Signed: _____
Professor of Electrical Engineering and Computer Science

Date: _____

~~Comments:~~

1 Background

Proof assistants are tools which aid the user in formulating machine-checked proofs. Users will generally build some sort of mathematical construction, which might be a representation of a program that needs verifying; state some theorem about the construction; and then prove the theorem with the aid of the tool.

Proof automation is important for verification of large software systems in Coq, as well as for proofs in large-scale mathematical libraries. Verification is human-developer-hour intensive, and automation allows engineers to construct larger proofs with less per-proof time investment. In verifying large software systems, automation is necessary for building general-purpose libraries which allow proving things about many different programs without having to write the proofs for each new program from scratch. In both software and math, automation allows code-reuse and permits factoring out common patterns and proof strategies and permits reasoning at a higher level of abstraction.

Unfortunately, as automation gets more complex or has to deal with more complex goals, the time spent compiling the proof script becomes significant, both repeatedly as the engineer updates their code and works interactively to prove a goal. The edit-compile-debug loop available when proof automation takes less than a tenth of a second is very, very different from the experience when each tactic takes multiple hours to run. Moreover, some proof automation simply stops working as the goals get large enough, failing to finish even after hundreds of hours unless carefully tuned to avoid performance issues.

Optimization in Coq is especially hard for a couple of reasons. Coq has many heuristics which are tuned for making some cases performant, but which result in puzzling differences in performance on other similar cases. In most languages, only the code needs to be optimized, and the types are just there to catch mistakes; in Coq, the code and the types must be performance-optimized simultaneously, whenever dependent types are in use. In most languages, leaky abstraction barriers cause pain only to the user; in Coq leaky abstraction barriers also incur a performance overhead every time they are used.

I propose to lay out a map of performance bottlenecks in Coq which can be used to inform future improvement and design of proof assistants. The map will divide the primary performance bottlenecks into three main categories: repeated computation around dependent types, large proof traces, and other kinds of needless duplicative work. I will describe the trade-offs which seem to be in play in each case and propose solutions drawn from extensive experience in building large-scale projects in Coq, which will be presented as case-studies. Some of the solutions will take the form design principles for users, with examples and evidence from various developments and performance tests. Other solutions take the form of prototype frameworks, tactics, or libraries which demonstrate a way to solve a particular performance bottleneck without compromising on the trade-off at play.

The primary case studies will be the formalization of category theory in Coq, a rudimentary parser-synthesizer, and the synthesis of efficient verified C code for field arithmetic used in cryptographic primitives. I will also present a novel

way of performing reification, which is a method of constructing abstract syntax trees for terms, used for taking advantage of Coq's computation in employing verified proof procedures or code transformations within the context of larger proofs. Finally, I will present a verified tool developed in the course of research for replacing proof-producing rewriting and slow custom reduction strategies with a faster reflective method which combines normalization by evaluation with pattern matching complication for reflective rewriting.

2 Timeline

- May 2012–July 2014: learn Coq; learn Ltac automation; formalize some category theory in Coq; extract lessons about what causes slowness in math libraries in Coq
- September 2014–June 2017: become acquainted with proof by refinement; work on formally verified automatically generated efficient parsers; struggle with Coq slowness in software engineering projects
- February 2016–October 2019: work on compiler pipeline for formally verified C code synthesis of modular field arithmetic for cryptographic primitives; work on reflective automation
- ≈ November 2017–February 2018: discover and codify reification by parametricity
- ≈ November 2017–October 2019: work on reflective rewriting framework for Fiat-Crypto;
- ≈ March 2019–October 2019: factor out the rewriting framework into its own tool
- October 22, 2019–November 9, 2019: performance testing and evaluation, and write-up, of rewriting framework
- November 2019–March 2020???: thesis-writing

3 The Parts in More Detail

The subsections in this section provide a brief outline of the proposed chapters in my thesis, with some context about where the information is coming from.

3.1 A map of the landscape of performance bottlenecks in Coq

The first project I did was formalizing a category theory library in Coq. This was an interesting starting point for learning Coq and for becoming acquainted with the performance landscape of Coq for a couple of reasons. Because category

theory is already on the formal end of mathematics, most of what I did was just figuring out how to translate from existing well-pinned-down formulations into Coq. At the same time, because category theory is so general, there were often multiple equivalent ways of specifying constructions, and I got to see what the effects of various choices were, while still being constrained by the mathematics. Finally, category theory is foundational in a way that stresses the expressivity of dependent type theories, especially around universes (Coq's mechanism for avoiding paradoxes involving the set of all sets that don't contain themselves and similar).

Broadly, most performance bottlenecks in Coq which are not merely an accident of history (such as an algorithmic inefficiency in Coq's implementation, or in the code written by a user) fall into one of three categories:

- Repeated computation involved in type-checking with dependent types (e.g., repeatedly type checking the same term which may show up as an argument to many constructors, just to make the types line up; repeatedly verifying that two terms are convertible)
- Generation and checking of enormous proof traces for transformations not in the trusted kernel
- Duplicative work (e.g., interpreted languages, near-identical proofs, redoing pattern-matching work)

3.2 Lessons in design of APIs of Gallina libraries

Unlike most programming languages, the design of an API in Gallina has an enormous impact on compile-time performance. I hypothesize that this is true in all dependently typed programming languages and proof assistants where arbitrary amounts of computation can happen during type checking.

The thesis will cover some broad design principles, and then talk about a couple of miscellaneous design choices that don't seem to fit into any of the design patterns we describe.

3.2.1 Abstraction barriers

The most important rule is to *not have leaky abstraction barriers*. In Coq, a leaky abstraction barrier is a case where a function that takes a term of type `T` is called on a term which has type `T'` when `T` and `T'` *are not syntactically equal*. Generally in these cases one lets the type-checker perform unification, and sweeps the details under the rug. If great care is taken, this can work, but most of the time Coq ends up spending a lot of time unifying types, when terms are big enough.

3.2.2 When and how to use dependent types painlessly

The extremes are relatively easy:

- Total separation between proofs and programs, so that programs are simply typed, works relatively well
- Pre-existing mathematics, where objects are fully bundled with proofs and never need to be separated from them, also works relatively well
- The rule of thumb in the middle: it is painful to recombine proofs and programs after separating them; it's okay to do it to define an opaque transformation that acts on proof-carrying code, but if the abstraction barrier is not perfect, enormous pain results.
- For example, consider length-indexed lists and indexing into them with elements of a finite type: problems arise only once the index is divorced from its proof of finiteness. For example, to index into the concatenation of two lists, with an index that is known to be valid for the first of the lists, the result is pain. The reason, I will explain, is that to do this indexing the index is first being considered separately from its proof of finitude, and then recombined in order to do the actual indexing.

3.2.3 Arguments vs. fields and packed records

There are two (or more) ways of designing hierarchies of mathematical objects. Roughly the distinction corresponds to different ways to answer the question "should we store information primarily in the type, or primarily in the term?" Frequently information stored in the type ends up duplicated, both in the term, and in functions manipulating this type, and in types built upon this type, resulting in quadratic blowup.

3.2.4 Proof by duality as proof by unification

One notable exception to the assertion about never having leaky abstraction barriers is when *all* of the interesting work can be offloaded into the unification engine, thereby eliminating the duplicative work of checking proofs more than needed. One example of this is *proof by reflection*, mentioned in subsection 3.4; this eliminates needless checking of proof traces. Another example is a perhaps novel contribution of the category theory library I created, proof by duality as proof by unification.

In category theory, there is a notion called "duality" which roughly means "flip all the arrows to point in the other direction." It's possible to get Coq's unification engine to do this automatically, allowing deduplication of code, often cutting build-time in half (because then the type checker runs on a generalized notion of "arrow" once, rather than on both directions of arrow separately).

3.3 Lessons in possible changes to Coq's type theory, implementation, and tooling

Coq has had many changes over the course of my PhD which have improved performance, some of which I will talk about as generalizable proof assistant

design choices that have performance implications. Notable examples of design options that have performance implications include primitive projections and universe polymorphism for code deduplication, higher inductive types / quotient types and internalized parametricity (not actually included in Coq yet) for offloading work into the metatheory, equality reflection, judgmentally irrelevant propositions, and hash consing.

Additionally, the thesis may talk about some tools that I worked on for enabling better profiling of Coq code; a profiler is one of the most useful tools an engineer can have for improving performance.

3.4 Proof by Reflection

(Note that much of this section is quoted from *Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac* [3].)

Proof by reflection [1] is an established method for employing verified proof procedures, within larger proofs. There are a number of benefits to using verified functional programs written in the proof assistant's logic, instead of tactic scripts. We can often prove that procedures always terminate without attempting fallacious proof steps, and perhaps we can even prove that a procedure gives logically complete answers, for instance telling us definitively whether a proposition is true or false. In contrast, tactic-based procedures may encounter runtime errors or loop forever. As a consequence, those procedures must output proof terms, justifying their decisions, and these terms can grow large, making for slower proving and requiring transmission of large proof terms to be checked slowly by others. A verified procedure need not generate a certificate for each invocation.

The starting point for proof by reflection is *reification*: translating a "native" term of the logic into an explicit abstract syntax tree. We may then feed that tree to verified procedures or any other functional programs in the logic. The benefits listed above are particularly appealing in domains where goals are very large. For instance, consider verification of large software systems, where we might want to reify thousands of lines of source code. Popular methods turn out to be surprisingly slow, often to the point where, counter-intuitively, the majority of proof-execution time is spent in reification – unless the proof engineer invests in writing a plugin directly in the proof assistant's metalanguage (e.g., OCaml for Coq).

Proof by reflection shows up in two or three ways in my thesis.

The zeroth way will be a recounting of existing knowledge proof by reflection, with an eye towards how it can be used for performance gains.

I will also describe how reification can be both simpler and faster than with previously-standard methods using what I've termed *reification by parametricity*. Perhaps surprisingly, we can reify terms almost entirely through reduction in the logic, with a small amount of tactic code for setup and no ML programming.

The final way reflective automation shows up is in the next section, subsection 3.5.

3.5 A Framework for Building Verified Partial Evaluators

(Note that much of this section is quoted from our draft paper *A Framework for Building Verified Partial Evaluators*.)

Partial evaluation is a classic technique for generating lean, customized code from libraries that start with more bells and whistles. It is also an attractive approach to creation of *formally verified* systems, where theorems can be proved about libraries, yielding correctness of all specializations "for free." However, it can be challenging to make library specialization both performant and trustworthy. My thesis will present a new approach (new as-of the paper this result was originally described in), prototyped in the Coq proof assistant, which supports specialization at the speed of native-code execution, without adding to the trusted code base. Our extensible engine, which combines the traditional concepts of tailored term reduction and automatic rewriting from hint databases, is also of interest to replace these ingredients in proof assistants' proof checkers and tactic engines, at the same time as it supports extraction to standalone compilers from library parameters to specialized code.

In some sense, this section will describe the central new contribution presented by my thesis, this prototype framework for rewriting and partial evaluation. The prototype framework is already used in the project that spawned it, Fiat Cryptography [2], a Coq library that generates code for big-integer modular arithmetic at the heart of elliptic-curve cryptography algorithms. (I spent a number of years of my PhD working on the Fiat Cryptography project.) This domain-specific compiler has been adopted, for instance, in the Chrome Web browser, such that about half of all HTTPS connections from browsers are now initiated using code generated (with proof) by Fiat Cryptography. However, Fiat Cryptography, prior to integration of the rewriting and partial evaluation framework, was only used successfully to build C code for the two most widely used curves (P-256 and Curve25519). Our initial method of partial evaluation timed out trying to compile code for the third most widely used curve (P-384). The rewriting and partial evaluation framework allows it to generate code for P-384, as well as much larger curves.

However, the framework is still only a *prototype*, in the sense that the performance results suggest that a framework like it could replace tactics such as `rewrite`, `autorewrite`, `rewrite_strat`, `simpl`, and `cbn` to a great performance improvement, but there is still much engineering work left to adequately replace these tactics in their full generality.

References

[1] Samuel Boutin. "Using reflection to build efficient and certified decision procedures". In: *Theoretical Aspects of Computer Software*. Ed. by Martín Abadi and Takayasu Ito. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 515–529. ISBN: 978-3-540-69530-1. DOI: 10.1007/bfb0014565.

[2] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises". In: *IEEE Security & Privacy*. San Francisco, CA, USA, May 2019. DOI: 10.1109/sp.2019.00005. URL: http://adam.chlipala.net/papers/FiatCryptoSP19/.

[3] Jason Gross, Andres Erbsen, and Adam Chlipala. "Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac". In: *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP'18)*. July 2018. DOI: 10.1007/978-3-319-94821-8_17. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2018-reification-by-parametricity-itp-camera-ready.pdf.