Towards a Proof Engine with Good Asymptotic Performance

Author: Jon Rosario

Advised by: Adam Chlipala, Andres Erbsen

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
December 2024

Abstract

Proof assistants are software systems designed to assist in the formal verification of mathematical theorems or software properties by constructing and checking logical proofs. They operate within formal systems, often leveraging type theory or first-order logic as a foundation. Proof engines are major components in proof assistants, designed to automate the construction and manipulation of proof terms within these formal systems. However, a performant implementation of a proof engine has yet to be realized. Users of popular proof engines report significant overhead in certain operations, such as rewriting, where the engine can become quadratic in complexity due to inefficiencies in handling contexts, variable substitutions, and the generation of fresh names. The proposed thesis will design and evaluate the performance of a new proof engine, based on representing proof terms as a directed acyclic graph, with the goal of achieving a proof engine that leverages subterm sharing and generates linearly-sized proofs.

1 Introduction

Validation of software is critical in ensuring their correctness and reliability, particularly in systems where errors can have significant consequences. While methods for validation such as code review and testing have been the status quo from the first computers, there has been a recent shift toward more rigourous approaches, such as formal verification, which are able to provide mathematical guarantees about a program's behavior. Proof assistants like Coq [?] and Lean [?] are at the forefront of this shift, offering powerful frameworks for formal verification using expressive logical foundations. However, the scalability of these tools is hindered by inefficiencies in their underlying proof engines, particularly in handling complex operations like equational rewriting. These inefficiencies limit their ability to verify larger and more intricate systems, creating a need for improved performance and better asymptotic behavior in proof engine design.

In particular, the verification of major projects like Fiat Cryptography, an MIT project building a perfomant elliptic-curve library, suffer from the poor performance of rewriting operations such as `setoid_rewrite` or `rewrite_strat`. The amount of time required for certain rewrite operations has been shown to grow exponentially with input size, often resulting in out-of-memory errors. These challenges underscore the urgent need for more efficient proof engine designs to support real-world applications. While all of the formal reasoning for Fiat Cryptography is done in Coq, other proof assistants suffer from similar slow-downs.

1.1 Background

1.1.1 Architecture of Proof Assistants

Most proof assistants share the same basic building blocks, including a proof-engine, type-checker, parser, and so on with a *type theory* as their foundation. Type theory defines a *type system*, which is essentially a set of rules for assigning types to terms. Type systems generalize the idea that terms in programs, like `1` or `"abc"`, have types, such as `int` or `string`. The type theory that has served as the foundation for most major theorem provers is the *Calculus of Inductive Constructions*, which is based on the earlier *Calculus of Constructions*. We will provide an detailed overview of the Calculus of Constructions in Section 1.1.2, but it is nothing but a pure type system. Within this type system, proving a theorem will correspond to showing that a particular type is *inhabited* by some term.

A *type-checker*, often referred to as the kernel, is the part of a proof assistant responsible for verifying that terms have a valid type according to the system's rules. Due to Gödel's second incompleteness theorem, we cannot verify the correctness of a proof assistant within the proof assistant itself. As a result, kernels are typically designed to be small, simple, and well-defined programs, which are trusted to be correct and bug-free.

Constructing the proof terms directly is often a cumbersome task. The *proof-engine* is the program which is responsible for building the proof term and more generally maintaining the *proof state* of an incomplete proof. Proof engines often provide access to *tactics*, which specifify how to manipulate a proof state to eventually generate a complete proof.

1.1.2 The Calculus of Constructions

The Calculus of Inductive Constructions (CoC) is an extension of the Curry-Howard Isomorphism, which establishes a deep connection between logic and programming. In this framework, the big picture is that "a proof is a program, and the formula it proves is the type for the program." This means that constructing a proof of a theorem can be viewed as writing a program, where the proof itself corresponds to the program's behavior, and the logical formula being proven corresponds to the program's type. This view forms the basis for using proof assistants like Coq and Lean, where users write and verify programs that also serve as formal proofs.

The syntax used to describe the Calculus of Constructions (CoC) is as follows. Terms can be variables (such as x or y), typed abstractions (such as $\lambda x : A, B$), or applications (such as PQ). The type of a typed abstraction is a product, written as $\Pi x : A, B$. The type of a type is a special constant called Type. Additionally, the type of all propositions is a special constant called Prop. There are some subtleties to be aware of, particularly that assuming the type of Type itself is Type is naive, as it leads to an inconsistent system, but this issue won't be addressed in this proposal.

In CoC, we also have reducible expressions (redexs). Redexs are terms that can be simplified or "reduced" to a more basic form through a series of reduction rules. These rules allow the computation of terms, much like evaluating expressions in a programming language. For example, a term like a lambda abstraction ($\lambda x : A, B$) can be reduced when applied to an argument, simplifying the term by substituting the argument for the variable x. In CoC, reduction is crucial for verifying the validity of proofs, as it ensures that the terms corresponding to the proof steps are consistent with the types being manipulated. The reduction process helps determine if a term inhabits the correct type, which is essential in constructing and verifying formal proofs.

This is an extension of the ordinary untyped lambda calculus with several desirable properties– the most important being that it is strongly normalizing. Strongly normalizing systems have the property that all sequences of reductions eventually halt. Consider the untyped term $(\lambda x, xx)$. When we apply the term to itself,

Within the Calculus of Constructions, we are concerned with ensuring that terms are well typed.

2 Related Work

2.1 Towards a Scalable Proof Engine

2.2 Bottom-Up β-Substitution: Uplinks and λ-DAGs

3 Proposed Work

Talk about Andres' design here and what I have so far...

4 Timeline

Timeline here....