Prototyping a Scalable Proof Engine

by

Jon Rosario

Bachelor of Science in Mathematics and Computer Science and Engineering

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by:	Jon Rosario Department of Electrical Engineering and Computer Science May 9, 2025
Certified by:	Adam Chlipala Arthur J. Conner (1888) Professor of Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Prototyping a Scalable Proof Engine

by

Jon Rosario

Submitted to the Department of Electrical Engineering and Computer Science on May 9, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Formal verification is an exciting development in software engineering, enabling implementations of programs to be rigorously checked against mathematical specifications. Assuming the specification is well-defined, formal verification provides guarantees of a program's correctness and freedom from bugs that are simply not possible with test-based methods. There's just one catch: the process of verifying large programs in popular theorem provers such as Coq (now known as Rocq) or Lean is painfully slow. These proof assistants rely on proof engines to construct proofs of correctness for given properties, but to our knowledge, there is no widely available proof engine that offers strong performance guarantees. Even more frustrating is the lack of consensus on what "good" performance should even mean in this context. This thesis lays the groundwork for addressing that gap by presenting a proof engine design that achieves asymptotically linear-time performance with respect to several important variables. We illustrate the design and its performance characteristics with examples from an implementation of the design and outline directions for future work.

Thesis supervisor: Adam Chlipala
Title: Arthur J. Conner (1888) Professor of Computer Science

Acknowledgments

First and foremost, I'd like to thank my thesis advisor, Professor Adam Chlipala, for his guidance and mentorship throughout this one-year journey. I'd also like to thank Andres Erbsen for volunteering his time and energy during the project and for providing the initial idea and notes that shaped the design of this proof engine. I'm grateful to Dustin Jamner for joining the project, for our discussions and the clarity he brought to them. Finally, I want to thank all the friends I've made during my time at MIT for the good memories, as well as my family back home.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Validation of software is critical in ensuring its 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 for decades, there has been a recent shift toward more rigourous approaches, such as formal verification, which is able to provide mathematical guarantees about a program's behavior. Proof assistants like Coq [1] and Lean [2] 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. 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 [3], an MIT project building a performant 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 and often results in out-of-memory errors. These challenges underline 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 slowdowns.

1.2 Summary

We present **MEngine**, a novel design and accompanying program for the generation and validation of type-checked proof terms. The defining design principle of **MEngine** is that its kernel enforces an invariant: every term (referred to as an Expression) is well-typed with respect to a (possibly given) context at creation time. Any Expression object constructed from others may thus assume those components are already valid. Internally, we represent Expressions as directed acyclic graphs (DAGs), following an approach similar to that described in [4]. This principle enables a range of optimizations, including pointer-based tactic caching, efficient substitution, and a significantly reduced memory footprint.

In Chapter 2, we provide background on the status quo, focusing on the Coq proof assistant (now known as Rocq). We then introduce the Bedrock2 project [5], a major Coq development supporting the specification, implementation, and verification of low-level programs. Bedrock2 proofs are often slow to verify, as we discuss later. In Chapter 3, we detail the kernel and proof-engine design, including the primitives and tactics implemented in our system. Chapter 4 presents performance evaluation across three benchmarks, the last of which is a real-world use case derived from Bedrock2. We conclude in Chapter 5 by reviewing our contributions, limitations, and directions for future work.

Chapter 2

Background

2.1 Proof Assistants and Coq

In this section, we give an overview of proof assistants and what they share in common.

2.1.1 Architecture of Proof Assistants

Most proof assistants, excluding various Higher-Order-Logic (HOL) variants, 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* [6, 7], which is based on the earlier *Calculus of Constructions*. We will provide an detailed overview of the Calculus of Constructions (CoC) in Section 2.1.2. Within CoC type systems, 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 valid types 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-delimited programs, which are trusted to be correct and bug-free.

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

2.1.2 The Calculus of Constructions

The Calculus of 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 $P\ Q$). 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 $\textsc{Type}$. Additionally, the type of all propositions is a special constant called $\textsc{Prop}$. There are some subtleties to be aware of, particularly that assuming the type of $\textsc{Type}$ itself is $\textsc{Type}$ is naive, as it leads to an inconsistent system, but this issue won't be addressed in this thesis.

In CoC, we also have reducible expressions (redexes). Redexes are terms that can be simplified or "reduced" to more basic forms through 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, the reduction process helps determine if a term inhabits the correct type, which is essential in constructing and verifying formal proofs.

CoC extends 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 terms $(\lambda x, x\ x)$ and y. Applying the first term to the second, we can perform a *β-reduction* to simplify the term:

$$(\lambda x, x\ x)y \xrightarrow{\beta} y\ y.$$

Since there are no longer any reducible expressions (redexes), the computation halts. However, applying the term to itself does not result in the same satisfaction. Instead, it produces an infinite loop:

$$\begin{aligned}(\lambda x, x\ x)(\lambda x, x\ x) &\xrightarrow{\beta} (\lambda x, x\ x)(\lambda x, x\ x)\\ &\cdots\\ &\xrightarrow{\beta} (\lambda x, x\ x)(\lambda x, x\ x).\end{aligned}$$

This problem is fully resolved in the strongly normalizing systems like the Calculus of Constructions.

Type-Checking in the Calculus of Constructions

In the Calculus of Constructions, the goal is to ensure that terms are well-typed. CoC defines a set of *inference rules* that specify the conclusions allowed based on a given set of hypotheses. These hypotheses typically involve *contexts* (denoted as Γ or Δ), which are collections of variable bindings, associating variables (e.g., x or y) with their corresponding types (e.g., A or B). For example, a context $\Gamma = [x : P][y : B]*$ lists such bindings, where $*$ denotes the end of the context (though it is often omitted).

A term t is said to be well-typed under a context Δ if it can be derived using the inference rules of CoC, denoted as $\Delta \vdash t$. If the type of t is T, we write $\Delta \vdash t : T$.

2.2 Related Work

Previous work in the field has explored various approaches to improving performance, but to my knowledge, there is no proof-engine that achieves the goal of performance scaling nearly linearly with the size of programs being verified. Several studies have highlighted the challenges of scaling proof assistants such as Coq and Lean, especially in the context of large, real-world verification tasks. In particular, research has focused on optimizing the handling of contexts, variable substitutions, and the underlying data structures used to represent proof terms.

2.2.1 Bottom-Up β-Reduction

As mentioned in the paper by Gross et al. [8], one of the reasons that a proof-engine could suffer from performance issues is due to a lack of subterm sharing. Consider the following untyped term:

$$(\lambda x, (x\ (f\ z)))\ (\lambda y, (y\ (f\ z))).$$

A basic, naive representation of this term allocates memory for each variable and constructs a tree-like structure to represent how the variables are related. This approach is both space-inefficient and time-consuming. For instance, the subterm $(f\ z)$ appears twice, under two different binders, but this representation doesn't provide a simple way to identify such repetitions without searching the entire tree.

A solution, proposed by Shivers and Wand in 2010 [4], is to represent terms from the lambda calculus using a directed acyclic graph (DAG). The core idea of their approach is as follows: no variable has more than one node in the DAG, each λ binds a unique variable and maintains a reference to it, and terms generally maintain *uplinks* to other terms that reference them. Our approach builds directly on this structure. `MEngine` enables efficient representation of large subterms by allowing shared subexpressions to be stored once and referenced multiple times. In many cases, we can avoid duplicate computation by preserving this sharing.

2.2.2 Performance Engineering of Proof-Based Software Systems at Scale

As part of his PhD thesis [9], Gross presents a survey of performance issues in Coq, along with several micro- and macro-benchmarks. His work includes several technical contributions, but we focus on a select few benchmarks from his work as case studies in Chapter 4.

2.2.3 Towards a Scalable Proof Engine

In 2023, Gross, Erbsen, Philipoom, Agrawal, and Chlipala addressed the challenges of building performant proof-engines [8]. The paper focuses on equational rewriting within Coq and identifies the related performance bottlenecks. The authors found that even when no full matches occur for a rewrite rule, certain rewrite tactics can still exhibit cubic complexity due to the overhead of partial matches and existential-variable (evar) creation. Additionally, they observed that proof-term size grows quadratically with the number of binders, and that the quadratic overhead in type-checking arises from Coq's handling of let-bindings and substitutions. These inefficiencies further highlight the difficulty of scaling proof-engines for large applications.

2.2.4 Slowness in Coq

Despite its expressiveness, Coq has long been known to suffer from various performance issues that can hinder large-scale proof development [8–10]. Many of these bottlenecks stem from core components of the system, including its kernel, tactic engine, and unification algorithm. The Coq issue tracker on GitHub documents numerous open reports that illustrate common forms of slowdown. For example, performance degrades significantly when repeatedly importing files with extensive notation overloads or when applying rewrite tactics like `setoid_rewrite` and `rewrite_strat`.

2.2.5 Bedrock2

Bedrock2 is a low-level programming language and verification framework built on Coq, designed for verifying memory-manipulating programs [5]. It supports basic control structures, arithmetic, and memory operations, and it uses separation logic to model memory. While Bedrock2 is a very effective tool for creating verified low-level programs, it suffers from performance challenges in large-scale proofs.

Chapter 3

Design Overview

In this section we will describe the overall design choices implemented in **MEngine** in detai and, in certain areas, the alternative choices that have been or could've been considered.

3.1 Kernel

3.1.1 Expressions

At the center of the kernel is the `Expression` type, a tagged union struct in C with various constructors.

```
typedef enum
{
  VAR_EXPRESSION,
  LAMBDA_EXPRESSION,
  APP_EXPRESSION,
  FORALL_EXPRESSION,
  TYPE_EXPRESSION,
  PROP_EXPRESSION,
  FIX_EXPRESSION,
  HOLE_EXPRESSION,
  MATCH_EXPR_EXPRESSION,
} ExpressionType;

struct Expression
{
  ExpressionType type;
  union
  {
    VarExpression var;
    LambdaExpression lambda;
    AppExpression app;
    ForallExpression forall;
    TypeExpression type;
    PropExpression prop;
    HoleExpression hole;
    FixExpression fix;
    MatchExprExpression matchExpr;
  } value;
};
```

Notably, we do not currently support inductive types. We include `FixExpression` and `MatchExprExpression` as special cases, discussed in Chapter 4.

Each expression variant includes several pieces of information, including a context in which the expression is valid, its type, and references to where it is referred to by other expressions (referred to as uplinks). Expressions form directed acyclic graphs by referencing other `Expression` objects: for example, an application (`AppExpression`) will refer to a function expression and an argument expression. Uplinks are not considered part of this graph and exist solely for efficient traversal. Alternatively, an uplink can refer to a context.

```
typedef enum
{
  LAMBDA_BODY,
  APP_FUNC,
  APP_ARG,
  FORALL_BODY,
  CTX_VAR,
  HOLE_TYPE
} Relation;

typedef struct
{
  Expression *expression;
  Context *context;
  Relation relation;
} Uplink;
```

We now describe several important expression types in detail.

- A `VarExpression` represents a variable binding, either for a term or a type. It includes a human-readable name (used only for display), its type as an `Expression`, and the minimal context Γ in which the type is well-formed (i.e., such that $\Gamma \vdash A$ where A is the variable's type). Variables are treated as unique by pointer; that is, two variables with the same name and type are considered distinct unless they are represented by the same pointer.

- An `AppExpression` represents function application, consisting of a function expression and an argument expression. It also stores the resulting type of the application and the minimal context in which both the function and the argument are well-typed. Unlike a `VarExpression`, which is unique by pointer, there may be many distinct `AppExpression` instances that are syntactically identical.

- A `LambdaExpression` encodes a lambda abstraction. It includes a bound variable, a body expression, and the full type of the lambda (a `ForallExpression`). The context reflects the environment in which the lambda body is valid, excluding the bound variable.

- A `ForallExpression` has a similar structure to `LambdaExpression`, and is used to represent dependent function types. It includes a bound variable, the body (which depends on that variable), and the context under which the expression is well-typed.

- The `TypeExpression` and `PropExpression` constructors represent the `Type` and `Prop` universes respectively. These are singletons in the system and are shared wherever those universes are referenced. Neither contain additional data.

- The `HoleExpression` represents a typed hole, which may later be instantiated. It includes a name for display, a required return type, the defining context, and uplinks.

```
typedef struct
{
  char *name;
  Expression *type;
  Context *context;
  DoublyLinkedList *uplinks;
  RewriteProof *rresult;
} VarExpression;

typedef struct
{
  Expression *func;
  Expression *arg;
  Expression *cache;
  Expression *type;
  Context *context;
  DoublyLinkedList *uplinks;
  RewriteProof *rresult;
} AppExpression;

typedef struct
{
  Context *context;
  Expression *bound_variable;
  Expression *type;
  Expression *body;
  DoublyLinkedList *uplinks;
  RewriteProof *rresult;
} LambdaExpression;

typedef struct
{
  Context *context;
  Expression *bound_variable;
  Expression *type;
  Expression *body;
  DoublyLinkedList *uplinks;
} ForallExpression;

typedef struct
{
  DoublyLinkedList *uplinks;
} TypeExpression;

typedef struct
{
  DoublyLinkedList *uplinks;
} PropExpression;

typedef struct
{
  char *name;
  Expression *return_type;
  Context *defining_context;
  DoublyLinkedList *uplinks;
} HoleExpression;
```

To initialize an Expression, we provide two sets of APIs: one with explicit context management required and one which is self-managing. We start by describing the one with explicit context management required:

```
// Initialize a new variable expression with a given name, type,
// and defining context. The variable's type must be valid in the defining context.
Expression *init_var_expression_wc(
           const char *name, Expression *type, Context *defining_context);
```

```
// Initialize a new lambda expression with a bound variable, body,
// and context. The body must be valid in the given context.
Expression *init_lambda_expression_wc(
           Expression *bound_variable, Expression *body, Context *context);

// Initialize a new application expression with a function, argument,
// and context. The function and argument must be valid in the given context.
Expression *init_app_expression_wc(
           Expression *func, Expression *arg, Context *context);

// Initialize a new forall expression with a bound variable, body,
// and context. The body must be valid in the given context.
Expression *init_forall_expression_wc(
           Expression *bound_variable, Expression *body, Context *context);
```

In each of these functions, type-checking is performed according to the rules of CoIC (Chapter 2) before returning an `Expression` pointer. For example, when initializing a variable `"x"` with type `nat` in a given defining context Γ, we first check whether $\Gamma \vdash$ `nat`. If this judgment does not hold, a null pointer is returned to indicate failure.

For `init_lambda_expression_wc` and `init_forall_expression_wc`, we require that the body is well-typed under the given context. However, the context stored in the resulting `Expression` is not the one passed in but rather the result of `context_minus(context, bound_variable)`. This yields the minimal context in which the lambda or forall expression is valid—that is, a context that no longer depends on the bound variable.

Additionally, the type of each expression is computed and stored in the `Expression` struct at initialization time. For instance, the application constructor `init_app_expression_wc` first verifies that the function has a dependent function type (i.e., a `FORALL_EXPRESSION`) and that the argument's type matches the function's domain. Once verified, it performs the appropriate substitution according to the CoIC typing rule.

```
Expression *constr_app_type(Expression *func, Expression *arg) {
  Expression *func_type = get_expression_type(func);
  Expression *variable = func_type→ value.forall.bound_variable;
  Expression *expected_arg_type = get_expression_type(variable);
  Expression *actual_arg_type = get_expression_type(arg);
  Expression *return_type = func_type→ value.forall.body;

  if (congruence(actual_arg_type, expected_arg_type)) {
    return subst(return_type, variable, arg);
  }

  return NULL;
}
```

Each of these initialization routines enforces the kernel invariant that every constructed expression is well-typed with respect to the context stored in the resulting `Expression`. This design ensures that, once created, an `Expression` can be assumed valid by downstream clients of the kernel without the need for rechecking. As a result, this API not only guarantees the correctness of individual expressions but also enables efficient and modular construction of larger terms, since each component can safely assume the well-typedness of its subterms.

As previously mentioned, the difference between the two sets of APIs lies in how contexts are handled.

```
Expression *init_var_expression(const char *name, Expression *type);
Expression *init_lambda_expression(Expression *bound_variable, Expression *body);
Expression *init_app_expression(Expression *func, Expression *arg);
Expression *init_forall_expression(Expression *bound_variable, Expression *body);
```

Instead of requiring a context to be provided as input, these functions compute the minimal context required for the resulting term to be valid. For example, in the case of `init_var_expression`, this context is the context of the type, extended with the variable binding itself. In the case of `init_app_expression`, the context is the result of directly joining the function and argument contexts:

```
context_add(get_expression_context(func), get_expression_context(arg)).
```

Otherwise, the functionality of these functions remains the same.

```
Expression *init_type_expression();
Expression *init_prop_expression();
```

As mentioned earlier, the expressions representing `Type` and `Prop` are singletons. Accordingly, `init_type_expression` and `init_prop_expression` serve as getters for these singleton expressions. The need for the first set of APIs becomes apparent when we later discuss holes: to fill a hole, the candidate expression must be valid under the hole's defining context, so we must ensure that the context is preserved correctly.

Holes are placeholders for expressions that are not yet known or constructed. They allow us to defer parts of term construction until more information is available, supporting an incremental or interactive style of proof development. A hole is created via the following function:

```
Expression *init_hole_expression(char *name, Expression *type, Context *context);
```

This function initializes a hole with a user-friendly name, an expected return type, and a defining context. The defining context captures the assumptions under which the hole is expected to be filled. This context is crucial for ensuring that any term used to fill the hole is well-scoped and well-typed under the assumptions available at the point of hole creation. It is assumed that anywhere where the hole is referenced, it is a legal operation to fill the hole with a suitable term.

Before a term can be substituted into a hole, we must ensure it is type-compatible and context-valid. This check is performed by `can_fill`, which returns `true` if and only if:

- The term's type matches the hole's expected return type.
- The term's context is valid under the hole's defining context.

```
bool can_fill(Expression *hole, Expression *term);
```

The actual filling of a hole is performed by `fill_hole`. This function first recursively fills any sub-holes required by the term. Then, it updates all `uplinks` referencing the hole, by directly modifying the parent expressions in place to point to the new term. This approach allows us to propagate updates to the entire term tree efficiently.

```
void fill\_hole(Expression *hole, Expression *term);
```

Together, `can_fill` and `fill_hole` implement a form of hole unification, where expressions can be incrementally completed, while preserving type soundness and scoping discipline.

3.1.2 Contexts

Contexts in our system play a foundational role in tracking the assumptions under which terms are defined. They serve a purpose similar to Coq's `Section` mechanism, but unlike Coq, contexts are explicitly constructed and manipulated.

At the base of the system is the empty context, which is a singleton similar to `Type` and `Prop`. It is created and later returned by calls to the following function:

```
Context *context_create_empty();
```

Larger contexts are built incrementally by extending an existing context with a new variable binding. Each context node holds the type of a variable, a pointer to its parent context, and the length of the context. This means a context $\Gamma = x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n$ is represented as a chain of `Context` structs, each pointing back to its predecessor.

```
typedef struct Context {
  Expression *var_type;
  struct Context *parent;
  int length;
} Context;
```

The type of a variable being added to a context must itself be well-formed under the parent context, which maintains the invariant that every type annotation in the context is meaningful with respect to the bindings that precede it. To enforce this invariant, we implement two helper procedures: `valid_in_context` and `valid_to_add_to_context`.

```
bool valid_in_context(Expression *expr, Context *context) {
  Context *curr_expr_ctx = get_expression_context(expr);
  if (context_is_ancestor(curr_expr_ctx, context)) {
    return true;
  }

  while (!context_is_empty(curr_expr_ctx)) {
    Expression *curr_var = curr_expr_ctx->var_type;
    if (context_find(context, curr_var) == NULL) {
      return false;
    }
    curr_expr_ctx = curr_expr_ctx->parent;
  }
  return true;
}

bool valid_to_add_to_context(Expression *expr, Context *context) {
  if (expr->type != VAR_EXPRESSION) {
    return false;
  }

  Expression *expr_type = get_expression_type(expr);
  return valid_in_context(expr_type, context);
}
```

Variables themselves are pointer-unique, so a single variable expression can appear in multiple contexts, provided that its type remains valid.

The basic operation on `Context` is insertion. Assuming that the expression to be added is a variable that is valid in the given context, we either return the given context if the variable is already in the context, or we return an extended context.

Additionally, we support two operations for largely composing and simplifying contexts: `context_add` and `context_minus`, respectively.

`context_add` takes two contexts and produces a new context that includes all variable bindings from both. This operation is not simply concatenation, since the two contexts may have overlapping or redundant bindings. Instead, we traverse the ancestors of one context and incrementally insert any missing bindings into the other. Insertion is guarded by validity checks: a variable is only added if its type is well-formed in the current context and it hasn't already been inserted. This operation is symmetric and produces the same context up to variable binding ordering.

`context_minus`, on the other hand, computs the minimal context in which a binder (e.g., λ or $\forall$) is valid. Given a context Γ and a variable x, we remove x and all bindings after x from Γ. After removing the subtrahend, we attempt to reinsert as many remaining bindings as possible into the resulting context, but only those whose types remain valid in the reduced context.

```
Context *context_add(Context *context_A, Context *context_B) {
  if (context_A == NULL || context_B == NULL) {
    return NULL;
  }

  if (context_A == context_B) {
    return context_A;
  }

  Context *start_context = context_A;
  Context *add_from_context = context_B;

  DoublyLinkedList *add_from_ancestors = context_ancestors(add_from_context);
  int n = dll_len(add_from_ancestors);

  Context *result = start_context;
  for (int i = 0; i < n; i++) {
    Expression *curr_add_from_expr = dll_at(add_from_ancestors, i)->data;
    if (context_find(start_context, curr_add_from_expr) == NULL) {
      result = context_insert(result, curr_add_from_expr);
    }
  }

  return result;
}

Context *context_minus(Context *context, Expression *subtrahend) {
  Context *until_subtra = context_find(context, subtrahend);
  if (until_subtra == NULL) {
    return context;
  }

  DoublyLinkedList *given_ancestors = context_ancestors_until(context, subtrahend);
  int n = dll_len(given_ancestors);
  Context *result = until_subtra->parent;

  for (int i = 0; i < n; i++) {
    Expression *curr_vartype = dll_at(given_ancestors, i)->data;
    if (valid_to_add_to_context(curr_vartype, result)) {
      result = context_insert(result, curr_vartype);
    }
  }
  return result;
}
```

3.1.3 Substitutions and Reductions

Substitution and reduction are the necessary mechanisms in our system, so we describe them here. The substitution procedure `subst` recursively replaces occurrences of an expression `old_e` with another expression `new_e` [`old_e` $\to$ `new_a`. Additionally, by using Lambda-DAGs (described in Chapter 2), we need to respect the convention that each lambda expression (or forall expression) binds a unique variable. Thus, this necessitates the implementation of what we call *parallel substitutions*. If we are substituting into a lambda or forall expression with a bound variable $a : T$, we may initialize a new variable of a similar type (potentially modified from the original as result of the substition), $a' : T$[`old_e` $\to$ `new_a`], then continue the substitution in the body with a list of substitutions: [`old_e` $\to$ `new_a`][$a \to a'$].

```
Expression *p_subst(Expression *expression,
          DoublyLinkedList *old_exprs, DoublyLinkedList *new_exprs) {
  int n = dll_len(old_exprs);
  if (n != dll_len(new_exprs)) {
    return NULL;
  } else if (n == 0) {
    return expression;
  }

  Context *e_ctx = get_expression_context(expression);
  bool needs_substitution = false;
  for (int i = 0; i < n; i++) {
    Expression *old_e = dll_at(old_exprs, i)→data;
    if (context_find(e_ctx, old_e)) {
      needs_substitution = true;
      break;
    }
  }

  if (!needs_substitution) {
    return expression;
  }

  switch (expression→type) {
    case (VAR_EXPRESSION):
    case (HOLE_EXPRESSION): {
      for (int i = 0; i < n; i++) {
```

```
    Expression *old_e = dll_at(old_exprs, i)→ data;
    Expression *new_e = dll_at(new_exprs, i)→ data;
    if (expression == old_e) {
      return new_e;
    }
  }
  return expression;
}
case (APP_EXPRESSION): {
  Expression *app_func = expression→ value.app.func;
  Expression *app_arg = expression→ value.app.arg;
  Expression *new_app_func = p_subst(app_func, old_exprs, new_exprs);
  Expression *new_app_arg = p_subst(app_arg, old_exprs, new_exprs);

  if ((app_func == new_app_func) &&(app_arg == new_app_arg)) {
    return expression;
  }

  if (forms_redex(new_app_func, new_app_arg)) {
    Expression *reduced = reduce(new_app_func, new_app_arg);
    return reduced;
  } else {
    return init_app_expression(new_app_func, new_app_arg);
  }
}
case (FORALL_EXPRESSION): {
  Expression *forall_var = expression→ value.forall.bound_variable;
  int search_result = dll_search_for_idx(old_exprs, forall_var);
  if (search_result != -1) {
    Expression *new_forall_var = dll_at(new_exprs, search_result)→ data;
    Expression *forall_body = expression→ value.forall.body;

    return init_forall_expression(new_forall_var,
        p_subst(forall_body, old_exprs, new_exprs));
  }
  Expression *forall_var_ty = get_expression_type(forall_var);
  Expression *new_forall_var_type = p_subst(
    forall_var_ty, old_exprs, new_exprs);
  Expression *new_forall_var = init_var_expression(
    forall_var→ value.var.name, new_forall_var_type);

  dll_insert_at_tail(old_exprs, dll_new_node(forall_var));
  dll_insert_at_tail(new_exprs, dll_new_node(new_forall_var));
```

```
      Expression *forall_body = expression→ value.forall.body;
      Expression *new_body = p_subst(forall_body, old_exprs, new_exprs);

      dll_remove_tail(old_exprs);
      dll_remove_tail(new_exprs);

      return init_forall_expression(new_forall_var, new_body);
    }
    ...
  }
}
```

Once substitution is implemented, beta-reduction is simple: given an application function and argument, where the application function is a lambda expression, take the bound variable and substitute for the application argument in the body.

3.2 Engine

This section describes the tactics implemented in the proof-engine. Unlike Coq, our prototype does not include an intermediate tactic language or an Ltac-style layer. Instead, all tactics are implemented directly in C. Since equational rewriting has received the most attention in Coq performance engineering, we mainly focus on rewriting.

3.2.1 Rewriting

Our rewriter is a bottom-up recursive procedure inspired by the rewriting strategy proposed in [8]. The rewrite procedure takes in as input an expression to rewrite in and a lemma to rewrite with. It starts by recursively traversing to the leaves of the expression DAG and applies a rewrite-head function, which matches the input expression against the lefthand side of the lemma.

```
RewriteProof *rewrite_head(Context *goal_context, Expression *expr,
                           Expression *lemma) {
  Expression *lemma_ty = get_expression_type(lemma);

  if (lemma_ty→ type == FORALL_EXPRESSION) {
    UnificationResult *unification_result =
        unify_and_instantiate(goal_context, lemma, lemma_ty, expr);
    if (unification_result == NULL)
        return init_rewrite_proof(expr, expr, build_eq_refl(expr), dll_create());

    Expression *instantiated_lemma = unification_result→ lemma_instantiation;
    if (instantiated_lemma != NULL) {
      Expression *lhs = get_lhs_eq(get_expression_type(instantiated_lemma));
      Expression *rhs = get_rhs_eq(get_expression_type(instantiated_lemma));

      if (expr_match(lhs, expr)) {
        return init_rewrite_proof(expr, rhs, instantiated_lemma,
                                  unification_result→ new_goals);
      }
    }

    return init_rewrite_proof(expr, expr, build_eq_refl(expr), dll_create());
  }

  Expression *lhs = get_lhs_eq(lemma_ty);
  Expression *rhs = get_rhs_eq(lemma_ty);
  if (expr_match(lhs, expr)) {
    return init_rewrite_proof(expr, rhs, lemma, dll_create());
  } else {
    return init_rewrite_proof(expr, expr, build_eq_refl(expr), dll_create());
  }
}
```

If the lemma contains no binders, and is simply of the form `lemma` : `eq T lhs rhs`, we can match the input expression e against lhs to see if they are equivalent. In case of success, we return a `RewriteProof` object representing an input expression, output expression, and proof of their equality:

```
typedef struct {
  Expression *expr;
  Expression *rewritten_expr;
  Expression *equality_proof;
  DoublyLinkedList *remaining_goals;
} RewriteProof;
```

In case of failure, we simply instead return a `RewriteProof` object where the input and output expressions are identical and the equality proof is given by `eq_refl`, signifying that no rewriting has occurred.

If the lemma contains binders, we need to perform instantiation to eliminate the leading `forall` quantifiers. We recursively attempt to construct an application of the lemma by either unifying its quantified variables with subterms of the input expression or filling them with holes if unification fails. This process accumulates any remaining open holes in a list. Once the lemma has been fully applied and no more binders remain, we return a `UnificationResult` that packages the instantiated lemma along with the list of remaining open goals.

The holes are useful because they allow partially applying a lemma even when some arguments are not yet known, enabling the proof-engine to defer their resolution. For example, consider the lemma:

$$\texttt{mult\_0\_l} : \forall x, 0 = 0 \ * \ x.$$

Suppose we want to rewrite the lefthand side of a goal $0 = 0 * y$ using this lemma. From the expression 0, we cannot immediately infer that x should be instantiated with y. However, the shape of the goal suggests that the lemma should apply, so we shouldn't fail outright either. Instead, the engine introduces a hole $?x$ and applies the lemma as `mult_0_l` $?x$, producing the statement $0 = 0* \ ?x$. This partially instantiated equality is still usable in rewriting, and the remaining goal will eventually unify $?x$ with y. In this way, holes allow the engine to proceed with rewriting even in the presence of incomplete information, deferring certain decisions to later unification or tactic steps.

Additionally, we implement a major optimization: we begin by checking whether the input expression could potentially be unified with the instantiated form of the lemma using `could_apply`. This check is performed before attempting full unification to avoid unnecessary computations. Specifically, `could_apply` checks whether the input expression and the lemma's body are compatible based on their structure, accounting for the presence of bound variables (as in `forall` quantifiers) and ignoring them.

On successful unification, the lemma is instantiated with the expression, and if the lefthand side of the instantiated lemma matches the input expression, a `RewriteProof` object is returned with the rewritten expression (righthand side) and the associated proof, along with any new subgoals/holes generated during unification.

We can now describe the rewrite procedure in full. We start with a call to an exposed function `rewrite`:

```
RewriteProof *rewrite(Context *goal_context, Expression *expr, Expression *lemma)
```

This function acts as a dispatcher to other functions specific to the `ExpressionType`: `AppExpression`, `LambdaExpression`, etc. `rewrite_app` begins by recursively calling `_rewrite`—which has the same function signature as `rewrite`—on its application function and argument. The only difference between `_rewrite` and `rewrite` is that the latter is responsible for clearing `rrewrite` proofs. Once these recursive calls return, assuming the returned expressions differ from the originals, we build a new expression defined by the application of the new function and argument, with the proof provided by an application of the axiom `app_cong`.

$$\begin{aligned}\texttt{app\_cong} :&\forall (A : \texttt{Type})(B : \texttt{Type})\\ &(f : A \to B)(g : A \to B),\\ &(x : A)(y : A),\\ &(\texttt{eq}\ (A \to B)\ f\ g) \to (\texttt{eq}\ A\ x\ y) \to\\ &\text{eq}\ B\ (f\ x)\ (g\ y)\end{aligned}$$

At this point, we attempt to apply `rewrite_head` on the newly formed expression. If rewrite_head fails, we return the RewriteProof explaining how the original expression (expr) converts to the new application expression, with the proof being provided by the app_cong axiom. If rewrite_head is successful, we use eq_trans, another axiom, to prove how the three terms are related. Finally, we cache the result with the expression.

```
RewriteProof *rewrite_app(Context *goal_context, Expression *expr,
                          Expression *lemma) {
  Expression *func = expr->value.app.func;
  Expression *arg = expr->value.app.arg;
  RewriteProof *rw_func_proof = _rewrite(goal_context, func, lemma);
  RewriteProof *rw_arg_proof = _rewrite(goal_context, arg, lemma);

  RewriteProof *mid_rewrite_proof;

  if (nothing_rewritten(rw_func_proof) &&nothing_rewritten(rw_arg_proof)) {
    mid_rewrite_proof =
        init_rewrite_proof(expr, expr, build_eq_refl(expr), dll_create());
  } else {
    DoublyLinkedList *merged = dll_merge(rw_func_proof->remaining_goals,
                                         rw_arg_proof->remaining_goals);
    mid_rewrite_proof =
        init_rewrite_proof(expr,
                           init_app_expression(rw_func_proof->rewritten_expr,
                                               rw_arg_proof->rewritten_expr),
                           build_app_cong(rw_func_proof, rw_arg_proof), merged);
  }

  Expression *mid = mid_rewrite_proof->rewritten_expr;
  Expression *fx_mid = mid_rewrite_proof->equality_proof;
  DoublyLinkedList *mid_goals = mid_rewrite_proof->remaining_goals;

  RewriteProof *rewritten_mid = rewrite_head(goal_context, mid, lemma);
  RewriteProof *result;
  if (nothing_rewritten(rewritten_mid)) {
    result = init_rewrite_proof(expr, mid, fx_mid, mid_goals);
  } else {
    DoublyLinkedList *merged =
        dll_merge(mid_goals, rewritten_mid->remaining_goals);
    result = init_rewrite_proof(
        expr, rewritten_mid->rewritten_expr,
        build_eq_trans(mid_rewrite_proof, rewritten_mid), merged);
  }

  free_rewrite_proof(mid_rewrite_proof);
  free_rewrite_proof(rewritten_mid);

  set_rresult(expr, result);
  return result;
}
```

In the case of rewriting lambda abstractions, we take care to reconstruct the body under the binder and apply functional extensionality over the resulting proof, rather than building the equality directly from the new expression. See [8] for more details on the approach taken.

To avoid duplicate computation, our rewriting procedure incorporates caching at every expression node. After rewriting an expression, we store the resulting `RewriteProof` directly in the `Expression` as the `rresult` attribute. Caching in this way avoids redundant work when the same subexpression appears multiple times in the DAG, allowing subsequent rewrites to retrieve the result in constant time. Efficient reuse becomes especially important when a very large subexpression is referenced repeatedly, enabling us to maintain linear time performance with respect to the size of the DAG.

At the end of the call to the top-level `rewrite` function, we recursively clear the cached rewrite results. Clearing the cached rewrite results ensures that subsequent rewrites, especially those using a different lemma, do not incorrectly reuse stale data. Because rewrite results are stored directly on expression nodes without tracking which lemma was used, retaining these caches could lead to incorrect behavior. Alternative, we could've considered using an external caching mechanism, such as a hash table keyed by both expression and lemma, to allow reuse of rewrite results across calls.

Additionally, we implement a variant of rewrite capable of handling a user-specified list of lemmas rather than a single one. The main modification lies in the `rewrite_head` function, which now iteratively attempts to apply each lemma in the list until no further rewriting is possible. The proofs are chained together using `eq_trans`, which allows for chaining multiple rewrites driven by different lemmas without requiring separate calls to `rewrite`.

```
RewriteProof *rewrites_head(
    Context *goal_context, Expression *expr, int n, va_list lemmas) {
  RewriteProof *current_proof = init_rewrite_proof(
    expr, expr, build_eq_refl(expr), dll_create());
  Expression *current_expr = expr;

  while (true) {
    bool rewrote = false;
    va_list copy;
    va_copy(copy, lemmas);
    for (int i = 0; i < n; i++) {
      Expression *ith_lemma = va_arg(copy, Expression *);
      Expression *ith_lemma_ty = get_expression_type(ith_lemma);
      UnificationResult *unification_result = unify_and_instantiate(
        goal_context, ith_lemma, ith_lemma_ty, current_expr);
      if (unification_result == NULL) continue;
      Expression *instantiated_lemma = unification_result->lemma_instantiation;

      if (instantiated_lemma != NULL) {
        Expression *lhs = get_lhs_eq(get_expression_type(instantiated_lemma));
        Expression *rhs = get_rhs_eq(get_expression_type(instantiated_lemma));
        if (expr_match(lhs, current_expr)) {
          RewriteProof *curr_to_next = init_rewrite_proof(
            current_expr, rhs, instantiated_lemma, unification_result->new_goals);
          Expression *proof_of_curr_to_next = build_eq_trans(
            current_proof, curr_to_next);
          current_proof = init_rewrite_proof(
            expr, rhs, proof_of_curr_to_next,
            dll_merge(current_proof->remaining_goals,
                curr_to_next->remaining_goals));
          current_expr = rhs;
          free_rewrite_proof(curr_to_next);
          rewrote = true;
        }
      }
    }
    va_end(copy);
    if (!rewrote) return current_proof;
  }
}
```

The rewrite tactic described above operates directly on expressions, rewriting the lefthand sides of equalities in a forward manner. However, this is not generally how proof-engines manipulate goals. Instead, users typically specify a goal they wish to prove, and the engine generates a goal state or hole, with the expected type. In the case of equational rewriting, this goal has the form `eq ...`. Each rewriting step then provides a proof that the current goal is equivalent to a new, simpler goal, effectively transforming the problem into a sequence of equivalence-preserving subgoals.

We provide `rewrite_transform`, which operates not on arbitrary expressions but directly on goals. Given a goal expression (a hole with an expected return type) and a rewrite lemma, this function attempts to transform the goal's expected type by applying the lemma. If successful, it produces two things: a new hole with the rewritten goal type, and a term that satisfies the original goal by invoking `eq_subst` with the rewrite proof and the new goal as an argument. The new goal may then be resolved independently.

$$\texttt{eq\_subst} : \forall (P : \mathit{Prop})(Q : \mathit{Prop})$$
$$\texttt{eq Prop}\ P\ Q \to Q \to P.$$

In practice, a single lemma may not be sufficient to simplify the goal fully. For this reason, we also support `rewrites_transform`, which accepts an arbitrary list of lemmas and applies them exhaustively. Each application rewrites the current goal type and updates the proof term accordingly, chaining substitutions via nested uses of `eq_subst`. This process continues until no lemma results in further simplification.

Together, `rewrite_transform` and `rewrites_transform` provide a bridge between our rewriting engine and goal-directed proof construction.

3.2.2 Other Tactics

We also provide basic implementations of other tactics for evaluation purposes. These include `intro` and `eapply`. All of these are hole-oriented tactics designed to be used together to solve a goal.

The `intro` tactic handles goals whose expected type is a ∀ expression. When the goal has type `forall x : T, U`, the tactic responds by extending the local context with the bound variable x, then generating a new goal of type U in that extended context. The original goal is filled with a lambda abstraction binding x and returning the new goal.

```
IntroReturn *intro(Expression *goal) {
  if (goal→ type != HOLE_EXPRESSION) return NULL;

  Expression *goal_ty = get_expression_type(goal);
  if (goal_ty→ type != FORALL_EXPRESSION) return NULL;

  Expression *goal_ty_bv = goal_ty→ value.forall.bound_variable;
  Expression *goal_ty_body = goal_ty→ value.forall.body;

  Context *new_context = context_insert(get_expression_context(goal), goal_ty_bv);

  Expression *new_goal = init_hole_expression("Goal", goal_ty_body, new_context);
  Expression *proof_of_original = init_lambda_expression(goal_ty_bv, new_goal);

  if (can_fill(goal, proof_of_original)) {
    fill\_hole(goal, proof_of_original);
    return init_intro_return(goal, new_goal, proof_of_original);
  }
  return NULL;
}
```

The `eapply` tactic uses unification to match a goal against the type of a lemma, potentially generating new subgoals. The unification procedure `eunify` is responsible for the unification algorithm. Unlike the usual approaches that first instantiate a lemma and then unifies it with the goal, our procedure performs both steps at once. As it traverses the lemma's type, which we expect to be several leading ∀s, it attempts to find instantiations for each bound variable on-the-fly.

When a matching instantiation cannot be found for a bound variable, we create a hole with the expected type and apply the lemma to this hole. These holes are appended to a list of remaining subgoals.

Once the lemma is fully instantiated and applied, the result is returned along with the generated goals. If the resulting term can satisfy the initial goal, the hole is filled and the new subgoals are returned. Otherwise, the tactic fails.

```
UnificationResult *eunify(Expression *lemma, Expression *goal) {
  Context *goal_context = get_expression_context(goal);
  Expression *expr = get_expression_type(goal);

  Expression *current_lemma_app = lemma;
  Expression *current_lemma_app_ty = get_expression_type(current_lemma_app);
  DoublyLinkedList *remaining_open = dll_create();

  while (current_lemma_app_ty->type == FORALL_EXPRESSION) {
    Expression *bound_variable = current_lemma_app_ty->value.forall.bound_variable;
    Expression *hole_subst = _unify(
        get_innermost_body(current_lemma_app_ty), expr, bound_variable);

    if (hole_subst == NULL) {
      Expression *hole_to_fill = init_hole_expression(
        bound_variable->value.var.name,
        get_expression_type(bound_variable),
        goal_context);
      current_lemma_app = init_app_expression(current_lemma_app, hole_to_fill);
      dll_insert_at_tail(remaining_open, dll_new_node(hole_to_fill));
    } else {
      current_lemma_app = init_app_expression(current_lemma_app, hole_subst);
    }

    current_lemma_app_ty = get_expression_type(current_lemma_app);
  }

  return init_unification_result(current_lemma_app, remaining_open);
}
```

```
DoublyLinkedList *eapply(Expression *goal, Expression *lemma) {
  UnificationResult *unification_result = eunify(lemma, goal);
  Expression *instantiated_lemma = unification_result→ lemma_instantiation;
  DoublyLinkedList *new_goals = unification_result→ new_goals;

  if (can_fill(goal, instantiated_lemma)) {
    fill\_hole(goal, instantiated_lemma);
    return new_goals;
  }

  return NULL;
}
```

Chapter 4

Case Studies

In this section, we will consider the following four questions over three different examples:

1. What performance do we get from Coq?
2. With conventional methods, how good can we expect the performance to be?
3. How fast does my design perform?
4. What is a reasonable lower bound?

4.1 A Basic Rewriting Example

We start by looking at a very basic example where we simplify an expression, under no binders, using rewrite rules. Consider the following expression:

$$g\,(f\,(f\,(\cdots f\,(a)\cdots))) \tag{4.1}$$

where a has some type and f, g are unary functions over that type. For simplicity, we'll call the type `nat`, so $a : \texttt{nat}$ and $f, g : \texttt{nat} \to \texttt{nat}$. If a is a fixed point for the function f and we had a theorem expressing the equality, we could expect to rewrite away all the occurrences of f to be left with $g(a)$.

Note that in this example, there are $\Theta(n)$ rewriting locations, so we might expect both the proof-term size and time complexity to grow linearly. As shown in Figure 4.2, the time complexity is **superlinear** in Coq!

After declaring `eq`, `nat`, etc as variable assumptions in Coq, and telling Coq that `eq` is an equivalence relation, we used the code in Figure 4.1 to generate the desired expression and rewrite back to $g(a)$.

Not only does our approach outperform Coq (which is expected, since our approach was written in C), it outperforms *asymptotically*, achieving $\Theta(n)$ time performance.

```
Fixpoint repeat_f (n : Datatypes.nat) : nat :=
  match n with
  | 0 ⇒ a
  | S n' ⇒ f (repeat_f n')
  end.

Definition gfa (n : Datatypes.nat) : nat := g (repeat_f n).

Goal eq nat (gfa __N__) (g a).
Proof.
  unfold gfa. cbn [repeat_f].
  Time (repeat rewrite eq_fa_a).
  (* or Time (rewrite_strat topdown eq_fa_a). *)
  (* or Time (rewrite_strat bottomup eq_fa_a). *)
  apply eq_refl.
Qed.
```

Figure 4.1: Template Coq code for generating and proving `eq nat g(f(f(...f(a)..))) (g(a))`.

4.2 Rewriting under Binders

Consider the following example using let-bindings from Gross's PhD thesis [9]:

$$\begin{aligned}
&\texttt{let } v_1 := \; v_0 + v_0 + 0 \texttt{ in} \\
&\vdots \\
&\texttt{let } v_n := \; v_{n-1} + v_{n-1} + 0 \texttt{ in} \\
&v_n + v_n + 0,
\end{aligned} \tag{4.2}$$

where the goal is to rewrite away all occurrences of $+0$, of which there are $\Theta(n)$ of. Note that in this example, maintaining the let-bindings is crucial. Performing zeta-reduction on this term to get rid of the let-bindings actually causes the number of rewrite locations to spike up to $\Theta(2^n)$.

Coq's default behavior is to **silently inline let-bindings**! Because of this behavior, it is common in large Coq developments to define constants such as `Let_In` to bypass this behavior. The definition of `Let_In` to escape Coq's default behavior of inlining let-in expressions is adapted from [9]:

```
Definition Let_In {A P} (x : A) (f : forall a : A, P a) : P x
:= let y := x in f y.
Lemma Let_In_def : @Let_In = fun A P x f ⇒ let y := x in f y.
Proof. reflexivity. Qed.
```

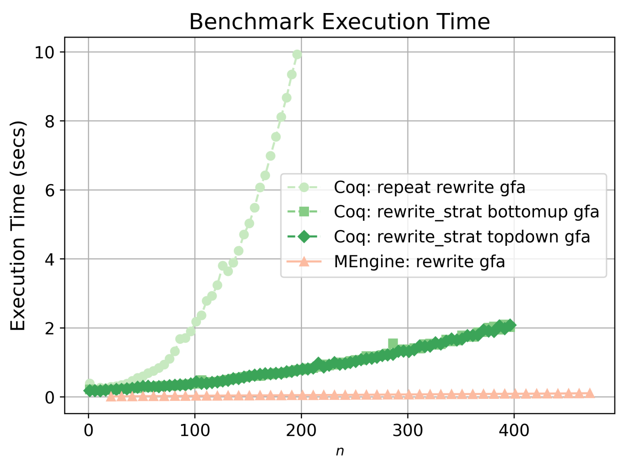

Figure 4.2: Execution time to prove `eq nat (g(f(f(...f(a)..))))  (g(a))` using three different rewriting strategies implemented in Coq, versus our approach. A rolling average with a window size of 5 has been applied to clearly demonstrate the trends.

```
Global Strategy 100 [Let_In].
Hint Opaque Let_In : rewrite.
Global Instance Let_In_nd_Proper {A P}
: Proper ((@eq A) ==>
    pointwise_relation _ (@eq P) ==>
    (@eq P)) (@Let_In A (fun _ ⇒ P)).
Proof. cbv; intros; subst. apply app_cong.
apply lambda_extensionality. exact H0. exact H.
Qed.
```

This approach can be helpful in certain scenarios, but does not completely alleviate the performance issues[1]. Furthermore, it is cumbersome and takes an unreasonable amount of effort to understand and apply. In both approaches, the user must use the `setoid_rewrite` tactic to remove +0's, since the `rewrite` tactic is unable to rewrite under binders.

In our proof-engine, we take three different approaches. In **Approach A**, we naively build an expression with (nearly) no subterm sharing. This approach is the equivalent of evaluating out let-bindings. In **Approach B**, we can take advantage of native sharing to avoid wasting time redoing work:

```
    Expression *vnm1 = init_var_expression("v0", nat);

    for (int i = 1; i <= n_depth; i++) {
        Expression *doubled = init_app_expression(init_app_expression(
            add, vnm1), vnm1);
        vnm1 = init_app_expression(init_app_expression(add, doubled), 0);
    }
```

[1]See: https://github.com/rocq-prover/rocq/issues/12510

```
Expression *expr = init_app_expression(
init_app_expression(add,
  init_app_expression(init_app_expression(add, vnm1), vnm1)),
0);
```

In **Approach C**, we can define an equivalent to `Let_In` to combine the modern approach in Coq with our proof-engine. A goal in this project was to make the `Let_In` obsolete, by achieving similar performance results between **Approach B** and **Approach C**.

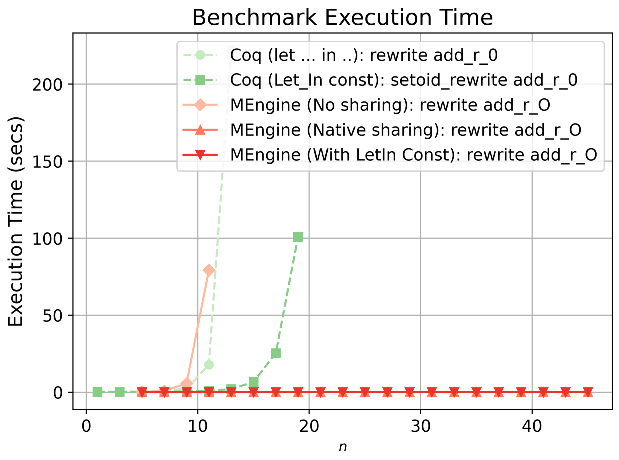

Figure 4.3: Execution time of removing +0's from equation 4.2.

4.3 Symbolic Execution

The next example is inspired by Bedrock2 to show how we compare in real-world usecases. Consider the following language, which is a simplified subset of the language implemented in the Bedrock2 project [11].

```
Variable bopname : Type.
Variable bopname_add : bopname.
Variable bopname_sub : bopname.

Inductive expr : Type :=
  | expr_literal : Z → expr
  | expr_var : string → expr
  | expr_op : bopname → expr → expr → expr.

Variable cmd : Type.
Variable cmd_skip : cmd.
Variable cmd_set : forall (lhs: string), forall (rhs: expr), cmd.
```

```
    Variable cmd_seq : forall (s1: cmd), forall (s2: cmd), cmd.
    Variable cmd_input : forall (lhs: string), cmd.
    Variable cmd_output : forall (arg: string), cmd.
```

In this language, programs are constructed by chaining together commands via the `cmd_seq` operator. For example, the following program is intended to receive user input as a and then double it:

```
cmd_seq
  (cmd_input a)
  (cmd_set a (bopname_add (expr_var a) (expr_var a))).
```

To define the behavior of such programs, we also need to specify the semantics of the language. Following the approach used in the Bedrock2 project, we adopt an *omnisemantics* model [12], which relates a command and an initial state (memory, local variables, and trace) to a set of possible outcomes (a postcondition), rather than to a single deterministic result:

```
Variable exec : cmd →
list IOEvent →
partial_map word byte →
partial_map string word →
(list IOEvent → partial_map word byte → partial_map string word → Prop) → Prop.
```

An `IOEvent` represents an input or output event, such as the result of an external function call or user input. The `partial_map word byte` type represents memory, while `partial_map string word` represents local variables. Together, these three components capture the full state of the program. The final argument to the command is a predicate on the final state. Altogether, the execution semantics assert that a command, when run from a given state, must satisfy the given postcondition. The following are the rules defining the `exec` relation:

```
Variable exec_skip : forall
(t : list IOEvent) (m : partial_map word byte) (l : partial_map string word)
(post : list IOEvent → partial_map word byte → partial_map string word → Prop),
post t m l → exec cmd_skip t m l post.
Variable exec_set : forall
(t : list IOEvent) (x : string) (e : expr) (m : partial_map word byte)
(l : partial_map string word)
(post : list IOEvent → partial_map word byte → partial_map string word → Prop)
(v : word),
·eq (option word) (eval_expr m l e) (option_some word v) →
post t m (partial_map_put string word l x v) →
exec (cmd_set x e) t m l post.
Variable exec_seq : forall
```

```
(t : list IOEvent) (c1 c2 : cmd) (m : partial_map word byte)
(l : partial_map string word)
(post : list IOEvent → partial_map word byte → partial_map string word → Prop),
exec c1 t m l (fun
    (t' : list IOEvent) (m' : partial_map word byte) (l' : partial_map string word)
        ⇒ exec c2 t' m' l' post) → exec (cmd_seq c1 c2) t m l post.
Variable exec_input : forall
(t : list IOEvent) (lhs : string) (m : partial_map word byte)
(l : partial_map string word)
(post : list IOEvent → partial_map word byte → partial_map string word → Prop),
(forall v : word,
   post (list_cons IOEvent (IOEvent_IN v) t) m
       (partial_map_put string word l lhs v)) →
exec (cmd_input lhs) t m l post.
```

To automate proofs of programs built with these semantics, we wrote a tactic and a test program. The generated program consists of a sequence of operations structured as follows:

$$\begin{aligned}
&\texttt{cmd\_input}\ x\\
&\texttt{cmd\_set}\ t\ (\texttt{expr\_var}\ x)\\
&\texttt{cmd\_set}\ t\ (\texttt{expr\_op bopname\_add}\ (\texttt{expr\_var}\ t)\ (\texttt{expr\_var}\ t))\\
&\texttt{cmd\_set}\ t\ (\texttt{expr\_op bopname\_sub}\ (\texttt{expr\_var}\ t)\ (\texttt{expr\_var}\ x))\\
&\vdots\ \text{(repeated n times)}\\
&\texttt{cmd\_set}\ t\ (\texttt{expr\_op bopname\_add}\ (\texttt{expr\_var}\ t)\ (\texttt{expr\_var}\ t))\\
&\texttt{cmd\_set}\ t\ (\texttt{expr\_op bopname\_sub}\ (\texttt{expr\_var}\ t)\ (\texttt{expr\_var}\ x))
\end{aligned}$$

Each command is sequenced appropriately using `cmd_seq`.

To prove the correctness of such generated programs, we use the `repeat_exec` tactic:

```
Ltac solve_eq_by_rewriting :=
let step :=
    first [
        rewrite partial_map_get_put_same
    |   rewrite partial_map_get_put_diff; apply not_eq_string_b_a
    |   rewrite binop_add_to_word_sub
    |   rewrite binop_add_to_word_add
    ]
in
simpl;
repeat step;
apply eq_refl.
```

```
Ltac repeat_exec :=
repeat match goal with
| [ |- exec (cmd_seq _ _) _ _ _ _ ] ⇒
    apply exec_seq
| [ |- exec (cmd_set _ _) _ _ _ _ ] ⇒
    try rewrite partial_map_put_put_same;
        eapply exec_set; try solve_eq_by_rewriting
| [ |- exec cmd_skip _ _ _ _ ] ⇒
    apply exec_skip
end.
```

Finally, we attempted to solve the following goal:

```
Goal let n := __N__ in let t := (list_nil IOEvent) in
    forall (m : partial_map word byte) (l : partial_map string word),
        exec (generated_cmd n a b) t m l
        (fun (t' : list IOEvent) (m' : partial_map word byte)
            (l' : partial_map string word) ⇒
            and (·eq (partial_map word byte) m' m)
                (ex word (fun v : word ⇒
                    and (·eq (list IOEvent) t' (list_cons IOEvent (IOEvent_IN v)
                            (list_nil IOEvent)))
                        (·eq (option word) (partial_map_get string word l' a)
                                        (option_some word v))))).
Proof.
    unfold generated_cmd;
    cbn [repeated_cmds];
    intros;
    apply exec_seq;
    apply exec_input;
    intros.
    Time repeat_exec.
Admitted.
```

Here, `__N__` is a placeholder for the desired number of repetitions. Note that Bedrock2 similarly implements a tactic for reasoning about straightline code, and this design is loosely inspired by it. For reasons that are apparent in Figure 4.4, we don't bother solving the postcondition and instead focus on the amount of time it takes to get to the postcondition.

In `MEngine`, we implement a similar straightline tactic for solving `exec` goals, though with important differences due to our core language design. Since `MEngine` does not directly implement inductive types, we added a small set of primitives `FixExpression` and `MatchExprExpression`, along with normalization procedures that govern evaluation of expressions. Unlike the Coq version, our symbolic execution in `MEngine` always produces a proof term. Moreover, we take care to ensure that each hole is initialized with the correct

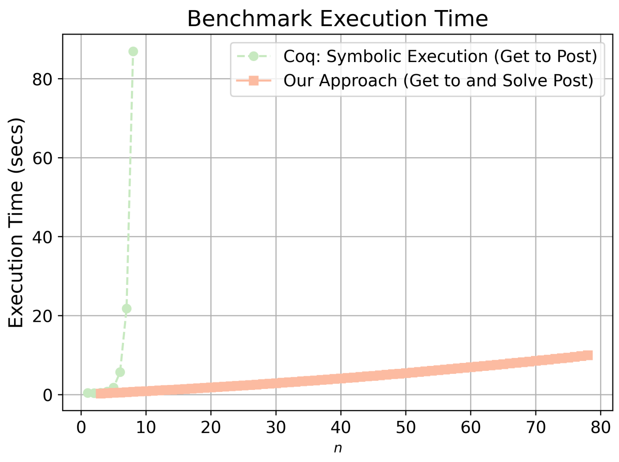

Figure 4.4: Results of solving the Bedrock2-like symbolic-execution goals in Coq and `MEngine` for values of n.

defining context.

The results of running this benchmark in Coq and `MEngine` for different values of n are shown in Figure 4.4. As shown, after $n := 4$, Coq was spending an unreasonable amount of time to solve the goal, while our approach required only a few seconds.

Chapter 5

Conclusion and Future Work

Overall, this project has reached the initial goal, which was to build a standalone proof-engine that achieves asymptotically better performance than Coq over several benchmarks. Chapter 3 outlines the overall design and choices made, while Chapter 4 gives several examples of benchmarks comparing standard and modern approaches in Coq vs. our approach. However, there are several engineering challenges ahead for this design to be useful.

5.1 Memory Management

Currently, our approach is sloppy in terms of memory management. At the scale needed to compete with Coq, poor memory management is not a problem, but looking ahead, our goal is to support a fully fleshed-out memory-management system.

While we could implement a reference-counting algorithm for garbage collection, the Shivers and Wand paper on Lambda-DAGs ([4]) contains a cleaner approach that leverages the uplinks we are already using. The caveat is that their technique depends on the so-called "single-DAG requirement," which stipulates that each subterm belongs to exactly one rooted DAG. This assumption does not hold in our setting, where a term can appear in multiple rooted DAGs. For example, a variable `eq` might be used as part of multiple DAGs representing the types of `eq_refl` or `app_cong`. At present, we believe the technique from the Shivers and Wand paper can be extended by additionally maintaining uplinks to context nodes and a top-level runtime system that tracks freshly allocated expressions and contexts that are still referenceable but not currently referenced by other objects.

5.2 Optimizing Conversions and Congruence

Currently, we do not implement any way of tracking conversions between one expression and another. Two expressions could be equivalent via a series of rewrites or reductions, but we do not maintain this information. One could consider adding some form of convertibility structure to capture these relationships and manipulate them for efficiency. A particularly attractive option is e-graphs [13], an efficient data structure for representing congruence relations. While we have not implemented them at present, they remain an interesting feature to consider for future work.

5.3 Benchmarking

Our next goal is to improve the benchmark suite. The current examples, while sufficient to illustrate our asymptotic goals, are limited in scope and drawn mostly from Coq. We'd like to include benchmarks from Lean as well, both to broaden the comparison and to explore how the system handles different proof engineering idioms. More generally, we hope to extend the benchmark set to cover more realistic automation developments.

5.4 Conclusion

In this thesis, we have described and demonstrated the design and implementation of a scalable proof-engine, including built-in subterm sharing, efficient equational rewriting, and incremental type-checking. While the core functionality is in place and demonstrably behaves correctly in many representative examples, there is still significant work to be done to make the design more usable. The author believes that the components presented are sound and that the remaining gaps are largely issues of engineering. Looking forward, we hope that this framework can serve as a foundation for further work on practical realization of scalable formal systems.

References

[1] *Rocq: A Proof Assistant for High-Performance Proof Development.* https://github.com/rocq-prover/rocq. Accessed: 2025-04-26. 2025.

[2] *Lean 4: The Next Generation of Lean.* https://github.com/leanprover/lean4. Accessed: 2025-04-26. 2025.

[3] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1202–1219. DOI: 10.1109/SP.2019.00005.

[4] O. Shivers and M. Wand. "Bottom-up Beta-reduction: Uplinks and Lambda-DAGs". In: *Fundam. Inf.* 103.1–4 (Jan. 2010), pp. 247–287. ISSN: 0169-2968.

[5] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. "Integration verification across software and hardware for a simple embedded system". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 604–619. ISBN: 9781450383912. DOI: 10.1145/3453483.3454065. URL: https://doi.org/10.1145/3453483.3454065.

[6] C. Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions". In: *All about Proofs, Proofs for All*. Ed. by B. W. Paleo and D. Delahaye. Vol. 55. Studies in Logic (Mathematical logic and foundations). College Publications, Jan. 2015. URL: https://inria.hal.science/hal-01094195.

[7] F. Pfenning and C. Paulin-Mohring. "Inductively defined types in the Calculus of Constructions". In: *Mathematical Foundations of Programming Semantics*. Ed. by M. Main, A. Melton, M. Mislove, and D. Schmidt. New York, NY: Springer-Verlag, 1990, pp. 209–228. ISBN: 978-0-387-34808-7.

[8] J. Gross, A. Erbsen, J. Philipoom, R. Agrawal, and A. Chlipala. "Towards a Scalable Proof Engine: A Performant Prototype Rewriting Primitive for Coq". In: *Journal of Automated Reasoning* 68.3 (Aug. 2024). ISSN: 1573-0670. DOI: 10.1007/s10817-024-09705-6. URL: http://dx.doi.org/10.1007/s10817-024-09705-6.

[9] J. S. Gross. “Performance Engineering of Proof-Based Software Systems at Scale”. Ph.D. Thesis. Cambridge, MA: Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Feb. 2021, p. 258. URL: https://hdl.handle.net/1721.1/130763.

[10] J. Gross and A. Erbsen. *10 Years of Superlinear Slowness in Coq*. Machine Intelligence Research Institute, Berkeley, CA, USA & MIT CSAIL, Cambridge, MA, USA. 2023.

[11] A. Erbsen. “Foundational Integration Verification of Diverse Software and Hardware Components”. Ph.D. thesis. Massachusetts Institute of Technology, Feb. 2023. URL: https://hdl.handle.net/1721.1/150216.

[12] A. Charguéraud, A. Chlipala, A. Erbsen, and S. Gruetter. “Omnisemantics: Smooth Handling of Nondeterminism”. In: *ACM Trans. Program. Lang. Syst.* 45.1 (Mar. 2023). ISSN: 0164-0925. DOI: 10.1145/3579834. URL: https://doi.org/10.1145/3579834.

[13] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. “egg: Fast and extensible equality saturation”. In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.