Project Report

Terminal Functions Library

Roshan Y Singh

Abstract

We present a symbolic evaluation engine for unary mathematical expressions using rewrite rules, algebraic identities, and commutation logic. It features a fast evaluator for both symbolic simplification and direct computation modes. This report describes the design, implementation, rule system, simplification logic, and discusses the tradeoffs in performance between naive evaluation and symbolic simplification.

Contents

1 Introduction **2**
1.1 Motivation 2

2 System Design and Architecture **2**
2.1 AST Representation 2
2.2 Expression Parser 3

3 Rule System and Simplification Logic **3**
3.1 Rule Format 3
3.2 Types of Rules 3
3.3 Rule Loader 3
3.4 Commutation Closure 3

4 Simplification Engine **4**
4.1 Core Algorithm 4
4.2 Sample Transformations 4

5 Evaluation System **4**
5.1 Recursive Evaluation 4
5.2 Simplified vs Direct Evaluation 4

6 Results and Output **5**
6.1 Rewrite Rules Used 5
6.2 Sample Output 5
6.3 Analysis 5

7 Implementation Details **5**
7.1 Technologies Used 5
7.2 Modules 6
7.3 Performance Tools 6

8 Conclusion **6**

9 Future Work **6**

1 Introduction

Symbolic simplification is a critical aspect of many mathematical engines, including compilers, computer algebra systems, and proof assistants. These systems frequently manipulate expressions by applying known algebraic identities and rules, such as:

$$\arcsin(\sin(x)) = x \quad \text{and} \quad ||x|| = |x|$$

This work focuses on building a lightweight symbolic simplification engine capable of operating entirely within a terminal. Expressions are evaluated either directly or using a symbolic simplifier before numerical evaluation.

1.1 Motivation

In scientific computing and symbolic mathematics, applying simplification before computation can drastically reduce the number of operations required, especially in nested expressions. Our system provides:

- Rule-based symbolic simplification.
- Commutation-aware transformations.
- Performance benchmarking for raw vs simplified evaluations.

2 System Design and Architecture

Our engine uses an Abstract Syntax Tree (AST) to represent expressions. The core components include the expression representation, rule loading and parsing, simplification logic, and the evaluator.

2.1 AST Representation

Expressions are built using the following structure:

Listing 1: Structure of an Expression Node

```
struct Expr {
    std::string func;         // e.g., "sin", "abs"
    Expr* child = nullptr;    // unary expression
};
```

Each node in the AST represents a unary function applied to its child. This design supports deeply nested expressions like:

$$\tan(\arcsin(\sin(x^2)))$$

2.2 Expression Parser

The function ‘parse_expr(std::string s)‘ parses input strings like:

```
abs(abs(-5))
```

into ASTs. The parser is recursive and handles both functions and constants. Parentheses are used to distinguish function application, and constants are parsed as leaf nodes.

3 Rule System and Simplification Logic

3.1 Rule Format

Rules are defined in a ‘rules.txt‘ file using the format:

```
lhs = rhs
```

For example:

```
arcsin(sin(x)) = x
abs(abs(x)) = abs(x)
```

3.2 Types of Rules

Rules are categorized as:

1. **Identity:** $f(x) = x$
2. **Idempotent:** $f(f(x)) = f(x)$
3. **Commutative:** $f(g(x)) = g(f(x))$
4. **Implication:** $h(f(x)) = x^2$

3.3 Rule Loader

The function ‘load_rules()‘ reads the file and stores parsed rules in categorized lists. It uses a simple parser to extract left-hand and right-hand expressions and maps functions for quick lookup.

3.4 Commutation Closure

To apply nested commutation rules, we compute a transitive closure using a Floyd–Warshall-like algorithm:

- Construct a graph with function names as nodes.
- An edge exists from $f \to g$ if $f(g(x)) = g(f(x))$ is a valid rule.
- Closure enables matching nested commutations like $f(g(h(x)))$.

This ensures correct simplification order across commutable expressions.

4 Simplification Engine

4.1 Core Algorithm

The function ‘simplify_node(Expr* root)‘ performs simplification recursively:

1. Traverse the expression tree.
2. At each node, check if any rewrite rule applies.
3. If yes, apply it and restart from the modified subtree.
4. For commutative functions, reorder sub-functions in a normalized manner.
5. For idempotents, collapse repeated calls.

4.2 Sample Transformations

Examples of successful simplifications include:

- $\arcsin(\sin(x)) \Rightarrow x$
- $\mathrm{abs}(\mathrm{abs}(\mathrm{abs}(-5))) \Rightarrow \mathrm{abs}(-5) \Rightarrow 5$
- $g(h(f(x))) \Rightarrow x^2 \Rightarrow$ evaluated

5 Evaluation System

5.1 Recursive Evaluation

The function ‘eval(Expr* root)‘ evaluates an expression by applying the corresponding mathematical function:

Listing 2: Recursive Evaluation of Expression

```
double eval(Expr* root) {
    if (is_constant(root)) return root->value;
    double val = eval(root->child);
    return apply_function(root->func, val);
}
```

Functions like ‘sin‘, ‘abs‘, ‘tan‘ are defined using standard math library calls.

5.2 Simplified vs Direct Evaluation

The function ‘compute(Expr* root, bool simplify)‘ evaluates an expression in either mode:

- If ‘simplify‘ is true, ‘simplify_node()‘ is called before ‘eval()‘.
- Otherwise, direct recursive evaluation is performed.

6 Results and Output

6.1 Rewrite Rules Used

```
# Basic rewrite rules
arcsin(sin(x)) = x
abs(abs(x)) = abs(x)

# Commutation rules
f(g(x)) = g(f(x))
h(g(x)) = g(h(x))

# Implication rule
h(f(x)) = x*x

# Identity
g(x) = x
```

6.2 Sample Output

```
tan(arcsin(sin(1.234^2))): ON=2.85603 (0.013284 ms), OFF=2.85603 (0.00795 ms)
abs(abs(abs(-5))):         ON=5 (0.001662 ms), OFF=5 (0.000628 ms)
g(h(f(3))):                ON=10 (0.002342 ms), OFF=16 (0.000696 ms)
...
Average ON-time: 0.046 ms
Average OFF-time: 0.008 ms
```

6.3 Analysis

The system shows significant gains for nested expressions:

- Simplification avoids redundant computations.
- In some cases, computation cost drops from multiple functions to one.
- For shallow expressions, simplification adds minor overhead.

7 Implementation Details

7.1 Technologies Used

- C++ for implementation (STL and math libraries).
- ‘rules.txt‘ file for external rule configuration.
- GNU ‘g++‘ compiler for builds.

7.2 Modules

1. `main.cpp` – driver and benchmarking
2. `expr.hpp` – AST definitions
3. `parser.cpp` – input parsing logic
4. `rules.cpp` – rule loader and matcher
5. `simplifier.cpp` – tree simplification functions
6. `evaluator.cpp` – numeric computation

7.3 Performance Tools

Execution time measured using ‘chrono‘ for nanosecond accuracy. Each expression is tested in both modes.

8 Conclusion

This project demonstrates a symbolic simplifier using rewrite rules to enhance mathematical evaluation. While direct computation is often fast, simplification becomes crucial for nested expressions. The engine is:

- Modular and extensible with user-defined rules.
- Efficient for complex symbolic tasks.
- A strong base for future symbolic computation or integration into compilers.

9 Future Work

- Extend to binary operators like ‘+‘, ‘*‘, etc.
- Add pattern-matching for general trees.
- Support user-defined functions.