A Domain-Specific Language for LLM-based Program Optimization (Work-in-Progress)

Yingfei Xiong

Peking University

Bidirectional Transformation Workshop @ Tokyo · July 2026

Or, Guiding an LLM to Learn a Programming Language through Bidirectional Transformation (Work-in-Progress)

Yingfei Xiong

Peking University

Bidirectional Transformation Workshop @ Tokyo · July 2026

Why Program Optimization Matters

Performance

Lower latency and higher throughput for compute-intensive workloads.

Efficiency

Less CPU time, memory traffic, and energy for the same computation.

Scale

Small improvements can matter when software runs repeatedly or at large scale.

Optimization is a continuing software-engineering task.

Compilers Automate Optimization, but Only to a Limited Extent

Compiler optimizations

- Apply established optimizations automatically.
- Carefully designed analyses ensure soundness.

→

Remaining bottleneck

- Optimization rules must be written by humans.
- Only a limited set of optimizations is implemented.

Can an LLM help write optimization rules?

From Optimization Patches to Reusable Rules

Evidence

A real-world performance patch.

→

LLM

Generalizes the edit and proposes a rule.

→

Deployment

The rule optimizes new code.

Potential benefit

Rule generation can expand optimization knowledge beyond what engineers can encode manually.

The Central Challenge: Soundness

A plausible rewrite

```
x / 8 ⟹ x >> 3
```

The shift may be cheaper than signed integer division.

A counterexample

```
x = -7
x / 8 = 0
x >> 3 = -1
```

The rewrite is not valid for all signed values.

A generated rule is useful only if its soundness is established.

Our Direction: A Verifiable Transformation Language

Language

An LLM expresses a program transformation rule in our language

→

Verification

Our system automatically verifies the soundness of the rule.

Design principle

Only verified rules are used for optimization

Four Challenges

1 · Express soundness conditions

Source patterns alone cannot state many value, effect, or memory requirements.

2 · Verify program transformations

Existing bounded verifiers check programs, not transformation rules.

3 · Return readable feedback

Verifier failures refer to generated low-level artifacts, while the LLM edits rules.

4 · Preserve rule quality

A rule that satisfies soundness may still be narrow or unrelated to performance.

Challenge 1: Patterns Do Not Capture Soundness Conditions

An Optimization

```
x / 8
  ↓
x >> 3
```

Analysis

- Bit shifting is usually faster than signed integer division.
- Soundness holds only when x is non-negative.

Existing program transformation languages are good at expressing pattern matching and changes, but not soundness conditions.

Solution 1: Analysis-Backed Conditions

```
rule:
    divide_by_8_to_shift
match:
    $X / 8
rewrite:
    $X >> 3
requires:
    eval($X) >= 0
```

Integrated Program Analysis

- Type and constant evaluation.
- Effect and control-flow checks.
- Pointer analysis.
- Interval analysis.

Only perform a transformation when the soundness condition can be proved by the analysis.

Challenge 2: Verifiers Check Programs, Not Rules

A transformation rule denotes many concrete program pairs:

$$\textbf{For every legal substitution } \sigma\textbf{: } C(\sigma) \Longrightarrow P(\sigma) \equiv Q(\sigma)$$

Available foundation

Bounded symbolic checkers can verify assertions over a concrete program model.

Missing support

We need to verify the equivalence of the original and rewritten fragments for all possible programs.

Solution 2: Reduce to the Correctness of a Single Program

```
#define L 20

void x_side_effect(int *memory)
  __CPROVER_accesses(
    _object_upto(memory, L * sizeof(int)));
  __CPROVER_deterministic();

int x(int *memory)
  __CPROVER_reads(
    _object_upto(memory, L * sizeof(int)));
  __CPROVER_assigns()
  __CPROVER_deterministic();

int main(void)
{
    int memory1[L];
    int memory2[L];

    __CPROVER_assume(
        __CPROVER_array_equal(memory1, memory2)
    );
    x_side_effect(memory1);
    int x1_return = x(memory1);

    x_side_effect(memory2);
    int x2_return = x(memory2);

    __CPROVER_assume(x1_return >= 0);
    __CPROVER_assume(x2_return >= 0);

    int out1 = x1_return / 8;
    int out2 = x2_return >> 3;

    __CPROVER_assert(
        out1 == out2, "expr outputs equal"
    );

    __CPROVER_assert(
        __CPROVER_array_equal(memory1,
          memory2), "memories equal"
    );
}
```

Challenge 3: Low-Level Feedback Is Hard to Use

1 · Rule without requires

```
match: $X / 8
rewrite: $X >> 3
```

3 · LLM struggles

The output contains low-level details that are not present in the rule.

2 · Raw CBMC feedback

```
** Results:
product-harness.c function main
[main.assertion.1] line 29
expression output equivalence: FAILURE

Trace for main.assertion.1:
x1_return=-2147483643
x2_return=-2147483643
out1=-268435455
out2=-268435456

Violated property: out1 == out2
VERIFICATION FAILED
```

Solution 3: Lift Feedback to Rule Semantics

Semantic lifting as a bidirectional transformation

Rule level

Candidate rule · lifted diagnosis

Forward lowering →

← Backward lifting

Verifier level

Converted C code · CBMC failure trace

1 · Apply semantic lifting

- Maintain provenance across both transformation directions.
- Follow assertion IDs and trace values backward through the map.
- Reconstruct a rule-level diagnosis.

2 · Lifted feedback

```
verification failed
property: output equivalence
metavariable binding: $X = -2147483643
match output: -268435455
rewrite output: -268435456
```

Inspired by Guan et al., Semantics Lifting for Syntactic Sugar *(OOPSLA 2024) and* Localizing Type Errors for Syntactic Sugar by Lifting *(OOPSLA 2026).*

Challenge 4: Soundness Does Not Mean Usefulness

Too Specific

- Commit-specific names or literals prevent reuse.
- Over-constrained soundness conditions describe an empty or nearly empty domain.

Not an Optimization

- An identity-like replacement makes no meaningful change.
- The rule preserves behavior but lacks a plausible performance benefit.

Formal verification checks semantic preservation under premises; it does not establish generality or performance value.

Solution 4: An LLM Quality Gate

Candidate rule

Already parsed and verified.

→

Independent review

A second LLM assesses generality, non-vacuity, and optimization intent.

→

Decision

Accept, request repair, or discard.

End-to-End Workflow

Optimization commit

Concrete edit

→

LLM proposal

Typed rule + premises

→

Verification loop

Parse · prove · lift feedback · repair

→

Quality gate

General and optimization-relevant?

Output: a reusable, verified transformation rule

Initial Evaluation: Language Expressiveness

5,290

optimization commits

1,545

commits where our system generated a verified transformation rule

In our previous work, an LLM generated Semgrep rules to locate potential optimizations and succeeded on 2,220 commits.

Other Planned Evaluations

- Evaluate whether our system can reproduce existing human-written optimizations.
- Evaluate how much our system can improve the performance of real-world programs.
- Evaluate the soundness and usefulness of the generated rules.

Summary

- LLMs can help expand optimization knowledge, but generated rules require stronger soundness guarantees than syntax matching.
- A verifiable transformation language makes soundness premises explicit and connects them to program analysis.
- Learning a new language is difficult for an LLM, especially when the feedback is low-level.
- Semantic lifting, formulated as a bidirectional transformation, helps the LLM understand low-level feedback.