Don't Transform the Code, Code the Transforms: Towards Precise Code Rewriting using LLMs

Chris Cummins **Volker Seeker** **Jordi Armengol-Estapé** **Aram H. Markosyan**
Gabriel Synnaeve **Hugh Leather**

Meta
cummins@meta.com

Abstract

Tools for rewriting, refactoring and optimizing code should be fast and correct. Large language models (LLMs), by their nature, possess neither of these qualities. Yet, there remains tremendous opportunity in using LLMs to improve code.

We explore the use of LLMs not to *transform code*, but to *code transforms*. We propose a chain-of-thought approach to synthesizing code transformations from a small number of input/output code examples that incorporates execution and feedback. Unlike the direct rewrite approach, LLM-generated transformations are easy to inspect, debug, and validate. The logic of the rewrite is explicitly coded and easy to adapt. The compute required to run code transformations is minute compared to that of LLM rewriting.

We test our approach on 16 Python code transformations and find that LLM-generated transforms are perfectly precise for 7 of them and less imprecise than direct LLM rewriting on the others. We hope to encourage further research to improving the precision of LLM code rewriting.

1 Introduction

A code transformation $f(c) \rightarrow c'$ is a function that rewrites input code c to produce c'. A multitude of software tasks can be expressed as code transformations from compiler optimizations to legacy code refactoring. Traditional rule-based code transformations are challenging to implement, and there is increasing interest in using LLMs to estimate them [1–7]. However, in contrast to rule-based transformations, the logic of LLMs is opaque, provides no correctness guarantees, and is hard to debug when incorrect. Instead, we propose using the LLM as a *generator* for code transformations, $g(C, C') \rightarrow f$, where the output is an implementation of a code transformation, $f()$, inferred from input/output code examples (C, C') before and after a transformation has been applied.

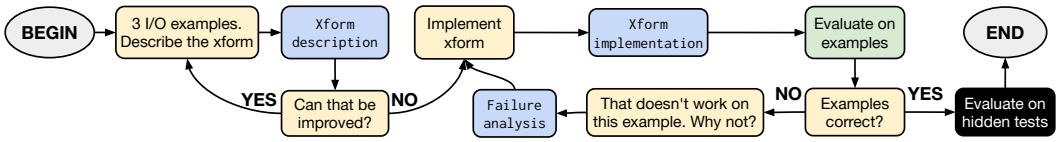

Figure 1: Our chain-of-thought approach to synthesizing code transformations (xforms) using LLMs. Yellow boxes are prompts, blue are LLM outputs.

Machine Learning for Systems Workshop at (NeurIPS 2024).

2 Synthesizing Code Transformations from Input/Output Examples

We present a chain-of-thought [8] approach for the efficient synthesis of code transformations from input/output examples, shown in Figure 1. The approach requires only a small number of input/output examples to successfully generate a code transformation implementation. Key to our approach is the plentiful use of loopback iterations to encourage the model to introspect on its own output, and to speculate as to the cause of failures before attempting to fix them. The methodology is as follows:

1. We first present three input/output code examples and prompt the model to describe the underlying transform in precise natural language.
2. We allow the model to iterate on this description up to 10 times. We found that the initial description often under-explains the transformation, particularly the handling of edge cases.
3. Once the model assesses that the description is adequate, we provide this description, along with the original input/output code examples, and prompt the model to generate an implementation of the transform.
4. We take the model-generated transform implementation and execute it in a sandboxed environment against 10 input/output examples, including the ones it has seen. If the transform fails to produce the correct output, or if it crashes, we provide the counterexample or error message.
5. In case of failure we then perform an additional introspection step in which we ask the model to explain why the problem occurred.
6. We prompt the model with the previous incorrect code, the failing counterexample (both expected output and actual output), and the failure analysis generated in the previous step.
7. This process repeats until the transform works on all examples, up to a maximum of 50 iterations. We then evaluate the quality of the generated transform on unseen input/output examples.

In this work we target Python and formulate code transformations as Abstract Syntax Tree (AST) rewrites using the format: `def xform(code: ast.AST)-> ast.AST`.

Table 1: Python code transformations, and the performance of two approaches: *Transform the code*, and *Code the transform*. We show F1 scores, with precision (how accurately the transformation is applied) and recall (how often transformation opportunities are identified) in parentheses.

Class	Transform	Description	*Transform the code*	*Code the transform*
Arithmetic	Add / subtract zero	Simplify $x + 0 \rightarrow x$ and $x - 0 \rightarrow x$.	0.92 (0.85, 1.00)	**1.00** (1.00, 1.00)
	Constant folding	Evaluate integer literal expressions in-place, e.g. $x = 10 + 15 \rightarrow x = 25$.	0.95 (0.91, 1.00)	**1.00** (1.00, 1.00)
	Divide / multiply by one	Simplify $x \div 1 \rightarrow x$ and $x \times 1 \rightarrow x$.	0.93 (0.88, 1.00)	**1.00** (1.00, 1.00)
Boolean	Collapse nested ifs	Recursively flatten nested `if` conditionals to a compound conditional.	0.81 (0.68, 1.00)	**1.00** (1.00, 1.00)
	De Morgan's law	Rewrite `!(a & b)` $\rightarrow$ `!a | !b`.	0.82 (0.69, 1.00)	**1.00** (1.00, 1.00)
	Reorder conditional	Flip the branches in `if not/else` conditionals to `if/else`.	0.52 (0.35, 1.00)	**0.93** (0.86, 1.00)
Liveness	Dead code elimination	Remove `if` conditionals if the branch condition statically evaluates to `False`.	0.93 (0.88, 1.00)	**0.99** (0.99, 0.99)
	Redundant fn. elimination	Remove function definitions, and their calls, if the function contains no instructions.	0.93 (0.87, 1.00)	**1.00** (1.00, 1.00)
	Unused var. elimination	Remove declared but unused variables.	0.87 (0.77, 1.00)	**0.98** (0.96, 1.00)
Loops	List comprehension	Rewrite `for` loop as list comprehension.	0.60 (0.43, 1.00)	**0.90** (0.86, 0.95)
	List comp. w. condition	As above but the loop body has a conditional.	0.62 (0.45, 1.00)	**0.82** (0.73, 0.93)
	Loop dupe	Duplicate loops (not semantics preserving).	0.50 (0.34, 1.00)	**0.99** (1.00, 0.99)
	Loop unroll	Fully unroll loops with statically known `range()` iteration bounds.	0.82 (0.70, 1.00)	**0.98** (0.99, 0.98)
Optimization	Dot product to torch	Replace `for` loop that computes vector dot product with torch API.	0.66 (0.49, 1.00)	**0.94** (0.95, 0.93)
	Pointwise add to torch	Replace `for` loop that computes pointwise add with torch API.	0.57 (0.40, 1.00)	**0.97** (0.94, 1.00)
	Torch zero grad	Replace `m.zero_grad()` with a loop over model parameters, assigning to `None`.	0.87 (0.77, 1.00)	**1.00** (1.00, 1.00)
Overall			0.75 (0.60, 1.00)	**0.97** (0.95, 0.99)

1 | PROMPT

Here are 3 examples of python code before and after a transformation has been applied to them:

Example 1

Before:

```
def encode_shift(s: str):
    for i in range(2):
        print('hello, world!')
    # snip...
```

After:

```
def encode_shift(s: str):
    print('hello, world!')
    print('hello, world!')
    # snip...
```

< snip 2 more examples ... >

MODEL

The transformation being performed is the replacement of for loops that iterate a fixed number of times with repeated statements. Specifically, any for loop that has a range-based iterator with a constant upper bound is replaced by repeating the loop body that many times. This transformation effectively "unrolls" the loop, replacing it with a sequence of identical statements, one for each iteration that would have been performed by the loop.

Complete the python code that performs the transformation:

```
import ast
def xform(tree: ast.AST) -> ast.AST:
    pass  # implement this function
```

< snip additional instructions on expected output format ... >

2 | MODEL RESPONSE

```
import ast

class LoopUnroller(ast.NodeTransformer):
    def visit_For(self, node):
        if (
                isinstance(node.iter, ast.Call) and
                isinstance(node.iter.func, ast.Name) and
                node.iter.func.id == 'range' and
                len(node.iter.args) == 1 and
                isinstance(node.iter.args[0], ast.Constant)
            ):
            num_iterations = node.iter.args[0].n
            new_body = []
            for _ in range(num_iterations):
                new_body.extend(
                    [ast.copy_location(n, node) for n in node.body]
                )
            return ast.With(
                items=[], body=new_body, lineno=node.lineno
            )
        return self.generic_visit(node)

def xform(tree: ast.AST) -> ast.AST:
    unroller = LoopUnroller()
    return unroller.visit(tree)
```

3 | PROMPT

That doesn't work on this input:

```
def fib4(n: int):
    # snip ...
    for i in range(2):
        x += 1
    # snip ...
```

The correct output should be:

```
    # snip ...
```

but instead was:

```
def fib4(n: int):
    # snip ...
    with :
        x += 1
        x += 1
    # snip ...
```

MODEL

The failure occurred because the `ast.With` node is not suitable for replacing the loop body in this case. The `with` statement in Python is used for context management, such as opening files or acquiring locks, and it's not a valid replacement for a simple sequence of statements. In this case, we should have simply returned a list of statements instead of wrapping them in an `ast.With` node.

Try again.

4 | MODEL RESPONSE

```
import ast

class LoopUnroller(ast.NodeTransformer):
    def visit_For(self, node):
        if (
                isinstance(node.iter, ast.Call) and
                isinstance(node.iter.func, ast.Name) and
                node.iter.func.id == 'range' and
                len(node.iter.args) == 1 and
                isinstance(node.iter.args[0], ast.Constant)
            ):
            num_iterations = node.iter.args[0].n
            new_body = []
            for _ in range(num_iterations):
                new_body.extend(
                    [ast.copy_location(n, node) for n in node.body]
                )
            return new_body
        return self.generic_visit(node)

def xform(tree: ast.AST) -> ast.AST:
    unroller = LoopUnroller()
    if isinstance(tree, ast.Module):
        tree.body = [unroller.visit(n) for n in tree.body]
    elif isinstance(tree, ast.FunctionDef):
        tree.body = [unroller.visit(n) for n in tree.body]
    else:
        raise ValueError("Unsupported AST type")
    return tree
```

Figure 2: An example LLM dialog, showing how the results of previous queries are used to generate natural language descriptions and failure analyses that are integrated into the chain-of-thought.

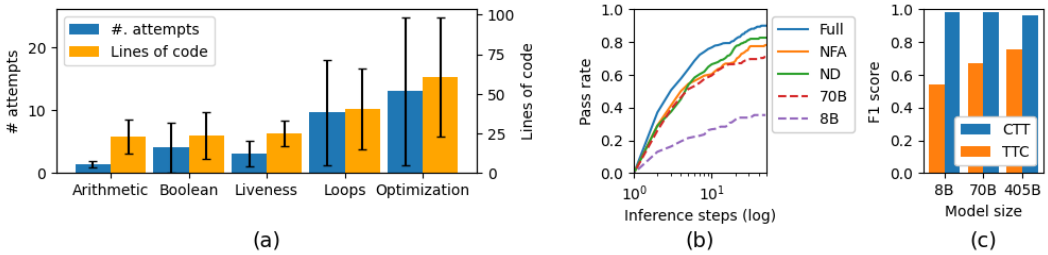

Figure 3: (a) the average number of attempts needed to synthesize transforms of each class correlates with the size of the transforms; (b) more attempts are required if chain-of-thought steps are removed or smaller models used; (c) synthesized transforms (CTT) of all model sizes work similarly well, but code rewriting ability (TTC) scales with model size.

3 Experiments

We evaluate the performance of our *Code the transforms* approach (CTT) and compare it against a *Transform the code* (TTC) approach in which the LLM is used to directly rewrite the code. For TTC we provide the model 10 examples and prompt it to apply the same transformation to unseen codes in turn. For CTT we use the examples to synthesize a transform in the manner described previously and then test the synthesized transform against unseen codes. We use the 405B parameter Llama 3.1 model [9] and sample it with temperature 0.25. We repeat the experiment 10 times.

Benchmarks. We assemble a dataset of 480 input/output Python programs covering 16 code transformations, summarized in Table 1. We aim to cover a range of uses and complexities from simple semantics-preserving rewrites (e.g. *constant folding*) to more substantial code optimizations (e.g. *dot product to torch*). We generated the example programs by adapting HumanEval solutions [10].

Our benchmark comprises 30 input/output Python code pairs for each of the transformations: 10 public examples available to the LLM, 10 hidden examples, and a further 10 examples of code where the transformation does not apply. The average length of each program is 11 lines of code.

Metrics. Code transformation requires that two tasks be completed successfully: *identifying* regions of code eligible for transform, and *executing* the transformation on identified code regions. To assess these properties we use the binary metrics of Precision and Recall. For a particular `(input,expected_out,actual_out)` tuple, the transform is *precise* if `actual_out == expected_out`, and successfully *recalled* if `actual_output != input && expected_out != input`. F1 is the harmonic mean of precision and recall. Scores are calculated over 10 runs.

Results. Table 1 compares the performance of both approaches. CTT has higher precision than TTC across every problem (overall 0.95 vs 0.60). Although overall scores are high, CTT still occasionally fails. Figure 2 shows an example failure and the model self-correcting. Qualitatively, we found that while CTT does a good job at comprehending the problem, it often struggles to turn that into working code. For example, the simple *arithmetic* transforms required only 1.5 attempts on average to synthesize a transform, whereas *optimization* transforms, which require more substantial code changes, require 11.8 (Figure 3a). Typically we found errors made by CTT to be easy to debug. Synthesized transforms average 34.4 lines of code (Figure 3a), compared to TTC which requires reviewing every output to check for errors. CTT performs slightly worse than TTC on recall (0.99 vs 1.00), as it would occasionally miss opportunities to apply a transformation that can be applied.

Ablations. Figure 3b compares the rate at which code transformations are synthesized when using our full chain-of-thought approach (Full), when the failure analysis step is removed (NFA), and when the natural language description step is removed (ND). Removing these steps from the chain-of-thought reduces the rate of transform synthesis efficiency.

We also repeated the full chain-of-though experiments using the smaller 70B and 8B parameter Llama 3.1 variants. Interestingly, we discovered that while the smaller models require many more inferences to synthesize transforms (Figure 3b), the transforms synthesized by all models perform equally well (Figure 3c), suggesting a compute/inference budget tradeoff. For TTC we see the expected result that smaller models are worse at directly rewriting code.

4 Discussion

We cannot afford the burden of reviewing and testing every piece of code an LLM touches. We propose an alternative formulation that reduces reviewing and testing costs by having the LLM instead generate code transformations. We show that this is more precise than using the LLM directly, but there is still a way to go. For example, we see in Figure 2 that although the model-generated code provides the expected transformation, there are obvious improvements that a human developer would make. We suspect that reinforcement learning and bootstrapped fine-tuning [11] could improve performance.

References

[1] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv:2308.10620*, 2023.

[2] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large Language Model-Based Agents for Software Engineering: A Survey. *arXiv:2409.02977*, 2024.

[3] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A Survey on Large Language Models for Code Generation. *arXiv:2406.00515*, 2024.

[4] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning Performance-Improving Code Edits. *arXiv:2302.07867*, 2023.

[5] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta Large Language Model Compiler: Foundation Models of Compiler Optimization. *arXiv:2407.02524*, 2024.

[6] Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to generate source code optimizations in C/C++. *TSE*, 2024.

[7] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler. In *CGO*, 2024.

[8] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*, 2022.

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 Herd of Models. *arXiv:2407.21783*, 2024.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374*, 2021.

[11] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping Reasoning with Reasoning. *NeurIPS*, 2022.