# Inductive Program Synthesis: Simplifying Math Expressions

Shifa Somji, Eylon Caplan, Arvind Ramaswami

December 11, 2024

## Introduction

Our high level problem is to use inductive program synthesis (IPS) to simplify math expressions automatically. Simplifying expressions often involves subjective interpretation and depends on the context in which the expression is used. For instance, an expression simplified for human readability might prioritize compactness, while one simplified for computational efficiency might prioritize fewer costly operations. Our goal is to create a system that defines a clear, adaptable standard for what it means for an expression to be simplified, allowing it to address various contexts and user needs effectively.

IPS involves automating programming tasks by constructing programs from high-level user-provided specifications [4]. These specifications typically include a set of example input-output behaviors, which act as benchmarks for the generated program's correctness. The IPS system then generalizes from these examples to create a program that meets the provided requirements. This approach allows for problems to solved where defining an explicit algorithm is difficult, but providing sufficient examples can guide the process. By leveraging IPS for math expression simplification, we aim to automate the reasoning process.

## Overview

Relevant existing works look at using LLMs for mathematical reasoning. LOGIC-LM integrates LLMs with symbolic solvers to improve logical problem-solving, utilizing LLMs to translate a natural language problem into a symbolic formulation and use a deterministic symbolic solver to perform inference on the formulated problem [2]. Another work proposes an approach to utilize a policy gradient to learn to select in-context examples from a small amount of training data and construct a corresponding prompt [3].

Our high-level approach is to combine IPS with an LLM to generate data and guide the simplification process. Our solution involves the following steps:

1. Ask LLM for a set of heuristic indicators (i.e., fewer number of terms, more factored terms)

2. Generate simplification candidates using an LLM
3. Check equivalence of candidate with original expression using SMT solver
4. Calculate heuristic score(s) for each candidate
5. Recurse with BFS if score is still increasing

Similar to existing work, our solution combines LLMs and symbolic solvers to verify our simplification. However, we guarantee equivalence of our original and simplified expressions by using a SMT solver, which is different from most relevant existing work.

# Implementation

## Formal Encoding

We formally define the input and output of our system. We provide a set of simplification examples, $\Pi$, to an LLM, to obtain $H$, a set of heuristic functions, $h_i$. Heuristic functions could include the number of terms, factoring terms, and standard forms for polynomials. We then generate potential simplification candidates using an LLM, by providing $H$ and $x$, our initial expression. For each potential candidate, $x', x'', \ldots$, we check equivalence with $x$ using an SMT solver. For each candidate, we also calculate a heuristic score – how well the candidate performs on the heuristic functions. We recurse with breadth first search, as long as the heuristic score is still increasing.

## Dataset Generation

We few-shot prompted an LLM to generate forms of mathematical expressions, as shown in Table 1. We had the model combine pairs of these forms and provide examples of each. Through human curation and validating that the before/after expressions are equivalent, we obtain a list of 16 sets of 5 pairs, where each set of 5 follows approximately the same form. We also include the reversed pairs, doubling the dataset to 32 sets of 5 pairs for a total of 160 examples. This allows us to use up to 4 pairs of examples for the inductive program to learn the 'style' of simplification, and to attempt to do so on a fifth expression.

| **Forms** |
|---|
| Write reciprocal as $x^{-1}$ |
| Simplify powers of powers |
| Factor out terms |
| Reduce the number of fractions |

Table 1: Select forms for generating the data

## Equivalence Checking

In order to check the equivalence of the two expressions, we first parse them into equations using regex parsing and SymPy. We then use two methods to check for equality: an exact method and a probabilistic method. In the exact method, we use SymPy to convert the expressions to Z3 and check for equivalence. In the probabilistic method, we assign random floats to variables and check if the results of the expressions are close for ten trials.

Using two methods is beneficial for stress-testing and for more complex equations. For future work, it can help to have a fast equivalence checking method in addition to a slower and more accurate method.

## Overall Pipeline

Our overall pipeline first generates guiding principles, $H$, given $\Pi$, some example before and after pairs of mathematical expressions, using the prompt in Figure 1. We generate potential candidates, $x'$, until a specific branching factor is reached and discard candidates that are the same as the original expression, $x$, or have already been generated. We score the candidates based on the average of two scores (on a scale of 1 to 5): how well they follow the guiding principles, $H$, and how well they follow they conform to the form shown in the before and after examples, $\Pi$. Note that we only keep a potential candidate if its score is larger than its parents, as a way to guarantee that at every step, we are looking at better simplification candidates. We do not include our other prompts in this report for brevity, but they can be found in our github repo [1].

### INSTRUCTION ###
I want to figure out what form to put a math expression in given some examples. You will be given some example pairs of mathematical expressions before and after being put in this specific form. Then, I want you to write some guiding principles that describe how to reach the form. For example, a principle could be "distribute the minus sign", "rationalize denominator", "simplify to the least number of terms possible", etc. Here are the examples of before/after being put in the form:

### EXAMPLE PAIRS ###
before: "$55 + x^{-3} + y^{-2} + 7$", after: "$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 1/x^3 + 1/y^2 + 7$"
before: "$21 + z^{-2} + 4 + x^{-3}$", after: "$7 + 6 + 5 + 3 + 4 + 1/z^2 + 1/x^3$"

### INSTRUCTION ###
Now, write 1-3 guiding principles that describe how to reach the form. Output your response as line-separated principles, like this:

### PRINCIPLES ###
principle #1
principle #2
principle #3

Figure 1: Prompt for generating guiding principles, given some example before and after expression pairs.

# Results

Figure 2 shows a comparison between our method (inductive BFS) and a few-shot LLM, that is prompted to simplify an expression in one prompt. Our method has a higher average final score, heuristic score, and few-shot score. Our method is also guaranteed to return an equivalent simplified expression because of our Z3 equivalence checking, which is not the case for the few-shot LLM. Lastly, our method requires many more LLM calls, in contrast to the baseline.

| Method | Avg. Final Score | Avg. Heuristic Score | Avg. Few-Shot Score | Equivalent | Avg. # of LLM Calls |
|---|---|---|---|---|---|
| Few-shot LLM | 2.275 | 1.025 | 3.525 | 99.38% | 1 |
| Inductive BFS (ours) | 3.503 | 3.694 | 3.688 | 100% | 22.419 |

Figure 2: Performance for the baseline vs. our method.

Figure 3 shows how the scores change as we iterate further in the tree, with each of the three scores increasing over time.

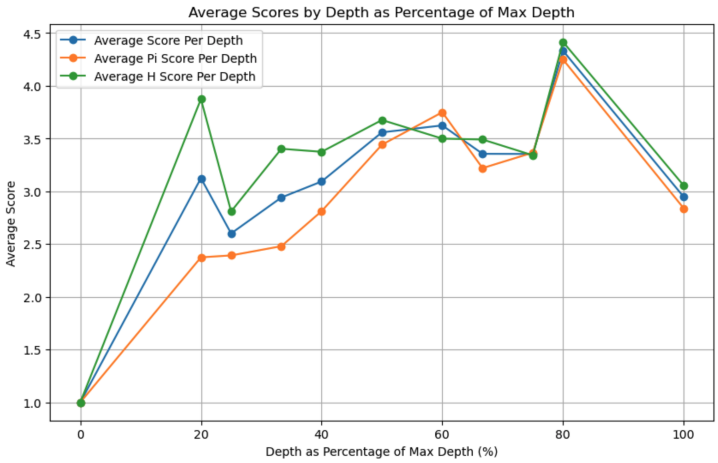


Figure 3:

Lastly, Figure 4 shows a box plot of the maximum score we achieved, with the average $H$, $\Pi$, and overall score around 3.5.

# Reflection

Throughout the course of this project, we learned how to apply inductive program synthesis to the task of simplifying math expressions. We also learned about the inherent trade-offs in balancing generalization and specificity of math

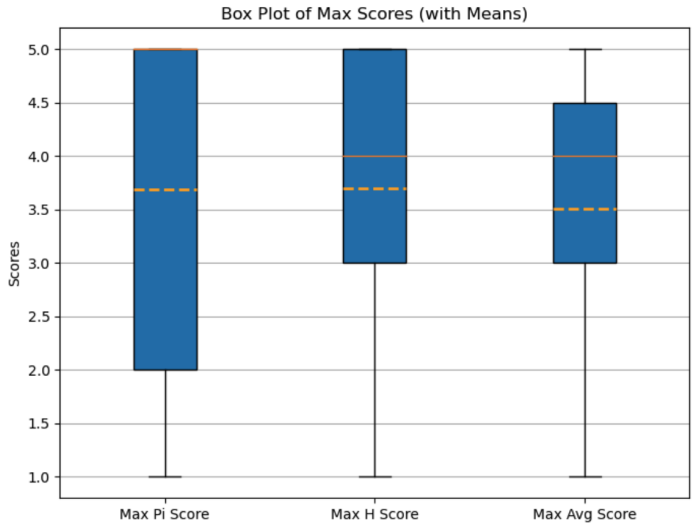


Figure 4: Maximum scores achieved, and their average.

expressions, highlighting the importance of integrating user feedback in the future to refine the definition of "simplification" in different contexts.

We experienced technical challenges with combining forms when creating the dataset and generating different simplification candidates. When making our dataset, we combined two forms and asked the LLM to provide examples of each. This was very difficult to achieve with prompting and required some iterations of our prompt. Additionally, the LLM would often generate the same simplification candidates. As a way to avoid this, we incrementally turned up the temperature, making the output more diverse, especially when we did not have enough potential candidates.

In the future, we can increase the complexity of our mathematical expressions and allow users to guide the simplification process, specifying preferred forms and constraints at each step. We can also optimize the system to handle large expressions more efficiently.

# Teamwork

1. Eylon constructed the dataset via repeated prompting of an LLM. He wrote all prompts for generating heuristics, generating a next simplification step, and the two methods of scoring. He implemented the search algorithm using these prompts and the code for metrics and analyzing the results.

2. Arvind designed and wrote the equivalence checking code. This involved writing a special parser for the format of the dataset. He then wrote two

equivalence checkers: both the random pointwise checker and the SMT checker, cleanly leaving room for future work.

3. Shifa assisted with high-level decisions regarding the dataset and project specification. She made the final presentation, particularly the overview and motivation sections. Finally, she synthesized the group's work into the final report.

## Course Topics

Our project built directly on several core concepts from the course. We extensively used SMT solving through Z3 for expression equivalence verification, drawing on our coursework in first-order theories and SMT solvers. The theoretical foundations in representing and reasoning about arithmetic expressions proved essential for our equivalence checking system.

Our work also highlighted areas where additional background would have been valuable. While we implemented a form of program synthesis, deeper coverage of dedicated synthesis techniques and efficient search strategies could have informed alternatives to our BFS approach. Given the subjective nature of expression simplification, methods for incorporating user preferences into the synthesis process would have helped us better handle what makes an expression "simpler." More background on domain-specific languages would have also benefited our expression representation system.

# References


[1] `https://github.com/eyloncaplan/exp_simp`.

[2] Ahn, J., Verma, R., Lou, R., Liu, D., Zhang, R., and Yin, W. Large language models for mathematical reasoning: Progresses and challenges. *arXiv preprint arXiv:2402.00157* (2024).

[3] Lu, P., Qiu, L., Chang, K.-W., Wu, Y. N., Zhu, S.-C., Rajpurohit, T., Clark, P., and Kalyan, A. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. *arXiv preprint arXiv:2209.14610* (2022).

[4] Polozov, O., and Gulwani, S. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2015), pp. 107–126.