Program Augmented LLMs: CS 224U Lit Review

Riya Sankhe
riyas@stanford.edu

Jenny Yang
jjyang1@stanford.edu

Jasmine Bilir
jbilir@stanford.edu

1 Introduction and Problem Overview

Recent studies have demonstrated that Large Language Models (LLMs) have an impressive ability to decompose a problem into steps and answer a question when prompted with a few input-output exemplars ("few-shot prompting"). In particular, the widely used chain-of-thought (COT) method presents the model with the explicit intermediate steps that are required to reach the final answer for a particular task. (Wei et al., 2023) Then, the model is expected to apply a similar decomposition to the actual test example, and consecutively reach an accurate final answer (Ling et al., 2017; Amini et al., 2019). However, when CoT prompting is applied to numerical reasoning problems, LLMs often make logical and arithmetic mistakes in their final solution outputs, even when the problem is decomposed correctly (Hendrycks et al., 2021; Madaan and Yazdanbakhsh, 2022).

To tackle this problem, we turn to the novel 'Program of Thoughts' (PoT) prompting method, where LLMs generate executable programs as reasoning steps and then offload the solution execution to a runtime such as a Python Interpreter. This falls under the broader task of generating code from a natural language description (NL2Code) that has attracted widespread interest in both academia and industry (Daoguang Zan, 2022).

In this literature review, we first look at a comprehensive survey of LLMs applied to the NL2Code Task (Daoguang Zan, 2022) and then study two papers that apply the Program of Thoughts prompting method to numerical reasoning problems, our area of interest (Wenhu Chen, 2022; Luyu Gao, 2021). We then look at studies that tackle future challenges and extensions to code generation for problem-solving. Specifically, we first look at studies that modify prompting methods to reduce the amount of human effort required to manually annotate examples, such as in (Shizhe Diao, 2023), or prompting for self-debugging purposes as in (Xinyun Chen, 2023). We then look at applications of Program of Thoughts prompting in other areas, including visual question answering (Dídac Surís, 2022) and science question-answering and tabular reasoning tasks (Pan Lu, 2023). We also compare these papers to the traditional CoT approaches. For example, (Yinya Huang, 2023) proposes a dataset to use formal logic and analytical problem-decomposition to solve problems using CoT. These papers suggest that CoT can be modified to effectively answer numerical reasoning problems without mathematical errors, but that different decomposition rules and/or tailored training data that better capture how humans develop answers to complex problems are required.

Overall, with a better understanding of CoT, PoT, upcoming applications, and logic-based counterparts, we will consider how mixing and matching facets from these frameworks might merit an interesting path of research.

2 Article Summaries

2.1 Introduction to CoT

In "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Wei et al. (2023) explore the idea of encouraging models to use intermediate reasoning steps in a chain of thought format, suggesting this significantly improves the complex reasoning abilities of large language models. More formally, they demonstrate that this behavior emerges without fine-tuning in LLMs via a method called "chain-of-thought prompting," where they provide a few examples of chain of thought behavior in prompting. There are four main benefits of using chain-of-thought prompting to facilitate reasoning: 1) it allows models to decompose multi-step problems, 2) it provides an interpretable window into the behavior of the model to reveal how it arrived at an answer, 3) it can be used for tasks like math word problems, commonsense reasoning, and symbolic manipulation, with the potential to be applicable to any tasks that are solved via language, 4) it can be applied to any sufficiently large language models simply with examples of chain of thought sequences. The authors demonstrate that using chain-of-thought prompting improves performance on a range of arithmetic, commonsense, and symbolic reasoning tasks via experimentation on 12 datasets.

2.2 Introduction to NL2Code Task

In "Large Language Models Meet NL2Code: A Survey," Daoguang Zan (2022) present a comprehensive survey of how LLMs are being used to generate code from natural language, and existing benchmarks, challenges and opportunities in the NL2Code task landscape. They start by analyzing 27 existing LLMs that have been applied to the NL2Code task in recent papers. For comparison, they evaluate their performance on the HumanEval Benchmark in a zero-shot manner, using the pass@k test metric [1]. This is one of the most popular benchmarks for this task and consists of 164 hand-written Python programming problems with corresponding test cases. In their evaluations, the authors find that (1)large model size, (2)training on large corpuses of high-quality pre-processed data, and (3) keen fine-tuning of hyperparameters were key factors to obtain a successful LLM for this task. Larger models tended to have more parameters and were thus more expressive, and also tended to have lower syntax error rates. In terms of data, the removal of likely auto-generated or unfinished code files, as well as the filtering out of uncommon code pieces helped ensure a high-quality corpus and improved model performance. Thirdly, fine-tuned hyperparameters including batch size, window size, warmup steps, sampling temperature also helped improve model performance. While the performance of existing LLMs varies widely on the benchmark used, the Codex (Chen et al., 2021) CodeGen-Mono (Nijkamp et al., 2023), and PanGu-Coder (Christopoulou et al., 2022) models exhibit impressive results at varying model sizes.
In summarizing existing benchmarks, the authors note that they vary widely in terms of size, language and complexity, however, are often limited by number of instances. This is because the instances have to be hand-written to ensure that LLMs have not seen them during training.
Lastly, the authors present existing opportunities and challenges in the space. They note two particularly interesting challenges in the space. First, they note that complex problems with multiple conditions or steps can be particularly difficult for LLMs, which suggests the potential merit of incorporating CoT/ PoT approaches to break down complex problems and improve performance. Second, they note that LLMs often have poor judgment ability and cannot determine when a programming problem is solvable or not, which may be especially relevant in the numerical reasoning context.

2.3 PoT: Combining CoT with NL2Code

Building on this idea of incorporating a CoT approach into LLM code generation, in "Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks," Wenhu Chen (2022) propose the Program of Thoughts (PoT) framework for solving numerical reasoning problems. In PoT prompting, the model expresses reasoning via executable Python programs and then delegates the actual computational task to an external Python interpreter. Thus, the LLMs are only responsible for expressing the problem reasoning process in the programming language, not the final answer. Moreover, PoT is also significantly different from direct code generation. First, it stimulates models to break down equations into a multi-step thought process, and second, it binds semantic meanings to variables to ground the model in language. They evaluated PoT across five Math word problem datasets (MWP) and three financial datasets. These datasets cover various input formats including text, tables, and conversation. Under both few-shot and zero-shot settings, PoT significantly outperformed CoT,and direct code generation across all the evaluated datasets. Under the few-shot setting, the average gain over CoT was around 8% for the MWP datasets and 15% for the financial datasets. Under the zero-shot setting, the average gain over CoT was around 12% for the MWP datasets. They also combine PoT with self-consistency (SC), which achieves the best-known results on all the evaluated MWP datasets and near SoTA results on the financial datasets. In their ablation studies, the authors find that increasing the number of shots (labeled samples) helps more for datasets with more diverse questions, semantic binding of meaning to variables helps improve

[1] see Appendix A.1 for more details on pass@k

model performance, and PoT especially helps in comparison to CoT on questions that require complex arithmetic like polynomial and combinatoric questions.

Another such paper is "Program-Aided Language Model (PAL)", which uses an LLM to read natural language problems, generates programs as the intermediate reasoning steps, and executes the Python code to obtain the solution. (Luyu Gao, 2021) As a result, the only learning task for the LLM is to decompose the natural language problem into runnable steps; the model does not see the final answers. Specifically, the LLM generates both its chain-of-thoughts in natural language as well as augmenting each natural language step with its corresponding programmatic statement. This teaches the model to generate a program that will provide the answer for a given question rather than relying on the LLM to perform the calculation correctly. The natural language intermediate steps are generated with comment syntax (#) so that the entire output can be fed directly into the interpreter. They validate the performance of PAL on mathematical, symbolic, and algorithmic reasoning tasks, comparing the performance of PAL with direct prompting and chain of thought prompting. The authors found that across all tasks, PAL with GPT-3 Codex, a code generation language model, achieves few-shot state of the art performance. The jump in performance with PAL over CoT remains consistent even with weaker LMs.

2.4 Prompt Experimentation

In "Teaching Large Language Models to Self Debug," Xinyun Chen (2023) take a step further from using LLM for code generation. Specifically, they propose "Self Debugging" as a methodology to teach language models to debug their predicted program code via few-shot demonstrations. By leveraging feedback messages and reusing failed predictions, this methodology helps improve sample efficiency and also achieves state-of-the-art performance on the Spider dataset for text-to-SQL generation, TransCoder for C++-to-Python translation, and MBPP for text-to-Python generation. The authors use three types of feedback messages that can be automatically acquired and generated with code execution and few shot prompting. First, they explore self-debugging with simple feedback where the feedback is a sentence that indicates the code correctness without more detailed information. Second, they explore self debugging with unit tests, where the model is also presented with the execution results of unit tests in the feedback message. Lastly, they explore self-debugging via code explanations, where the model has to self-debug by explaining the generated code. On text-to-SQL generation where there is no unit test specified for the task, leveraging code explanation for self-debugging. consistently improves the baseline by 2 - 3%, and provides a performance gain of 9% on the hardest problems. For code translation and text-to-Python generation tasks where unit tests are available, self-debugging significantly increases the baseline accuracy by up to 12%.

Another creative manipulation of prompting is explored in "Active Prompting with Chain-of-Thought for Large Language Models.", the CoT approach can be challenging as it requires additional manual annotation from human beings, and the number of adequate samples that can be produced in this manner is limited in size, number, and question scope. Likewise, questions requiring greater technical complexity (like programming) require industry professionals to dedicate valuable time to generating quality samples. To reduce this complexity, Shizhe Diao (2023) propose "active CoT prompting" wherein the model generates its own reasoning annotations with respect to a query, and then scores itself based on uncertainty. For cases where uncertainty is high, it seeks for additional human annotation, greatly reducing the human interaction required. Annotation occurs in a standard few-shot prompting, and uncertainty is then calculated over the k examples used. After many questions are processed in this manner, the top-n most uncertain are provided and retrained using human annotation. Inferences and re-evaluation on the remaining samples is then conducted, and can be evaluated for whether greater human interaction is needed or whether the model has achieved a self-confidence threshold. Using datasets typical to chain-of-thought evaluation, including commonsence resoning, arithmetic reasoning, symbolic reasoning, etc., this model outperformed standard chain-of-thought and several other chain-of-thought variants, including random and auto CoT. More specifically in airthmetic reasoning, "compared with the competitive baseline, self-consistency, Active-Prompt (D) outperforms it by an average of 2.1% with code-davinci-002. Larger improvement is observed with textdavinci-002, where ActivePrompt (D) improves over self-consistency by 7.2%" measured in answer accuracy (Shizhe Diao, 2023). Overall, it appears that when used in tandem with large industry-accepted LLMs, active-prompting can show improvements in numerical questions. In their discussion, (Shizhe Diao, 2023) report that performance on a wide-range of metrics depended heavily on the number of few-shot exemplars and tuning of uncertainty thresholds and tuning parameters.

2.5 Applications of PoT

One application of separating chain-of-thought from programming and execution is seen in “ViperGPT: Visual Inference via Python Execution for Reasoning” (Dídac Surís, 2022). The authors of ViperGPT propose a framework to answer complex visual questions by integrating a code-generation model to interpret the image or video. To do this, they use an API and Python interpreter to run the generated code. Specifically, they use GPT-3 Codex, a LLM for code generation, to perform this task. The input prompt to the LLM includes an API, the specification for the API with function signatures and docstrings, and the query for the sample under consideration. The output is a Python function definition that leverages the relevant functions provided, which is then compiled and executed. They evaluated their results in multiple steps: visual grounding, which is the task of identifying the bounding box that best corresponds to a natural language query, compositional image question answering, which requires decomposing complex questions into simpler tasks, external knowledge-dependent image question answering, and video causal and temporal reasoning. To do these tasks, they provided ViperGPT with the API for several modules, leveraging pretrained models and allowing ViperGPT to query external knowledge bases in natural language. This approach required no further training and achieved state of the art results on various complex visual tasks.

Another application introduces a “planner” model that helps orchestrate distribution of tasks. Building off of recent momentum in LLMs, the Pan Lu (2023) of “Chameleon: Plug-and-Play Compositional Reasoning with Large Language Models” are seeking to improve the current coverage and reliability of models on questions which may have multi-modal elements, layers of background complexity, or need for precise mathematical reasoning. In these cases, the authors explain how noteworthy LLMs, including GPT-4, are weak in these areas due to an “inability to access up-to-date information, utilize external tools, or perform precise mathematical reasoning” in particular. In elaborating, they likewise offer similar analysis to these gaps as in Wenhu Chen (2022). Their novel response to this problem was to develop Chameleon, a natural language planner that operates on top of and orchestrates the collaborative use of LLMs, off-the-shelf computer vision models, python programs, and search engine information retrieval. The job of the Chameleon model is to parse tasks and schedule them in a logical form such that intuitive groupings can be tackled by certain tools before being passed in a different form to the next set of tools. In other words, the programmer makes modular step sizes and runs them through a sequence of tools like an assembly line. The planner is an additional LLM prompted and trained in a few-shot setup to first develop a sequence of stages in which to approach the query problem, then to ensure that the tools are orchestrated in this manner. Each intermediary answer is passed to the next scheduled tool and so forth until the ultimate answer is reached. The primary LLMs used to implement Chameleon experiments were ChatGPT and GPT-4 as specified by the authors (Pan Lu, 2023). In experiments averaged accross all subjects, Chameleon implemented with ChatGPT scored higher (79.93% accuracy) than CoT implemented with ChatGPT (78.31% accuracy). However, Chameleon did not outperform larger multi-modal CoT models including MM-COT Large (Pan Lu, 2023). In qualitative analysis, Chameleon developed "fewer distinct programs" meaning that in comparison to other CoT models chameleon was more consistent and high-level in reasoning capabilities. Strongly inspired by program-of-thought and chain-of-thought models, chameleon offers yet another approach to breaking down tasks of complex compositional reasoning into staged subsets, and its strength is its ability to exploit the existing tools available today.

2.6 Logic Based Alternatives

Finally, we sought to explore the role of logic in model reasoning. Prior to “MetaLogic: Logical Reasoning Explanations with Fine-Grained Structure,” available datasets and measures of success for chain-of-thought, program-of-thought, and other types of multi-step reasoning models are trained and evaluated on datasets that do not include degrees of certainty or sufficiently tackle all types of inference used in problem-solving. Seeking to address this issue, the authors of this paper develop a dataset based on the idea of a problem metagraph (Yinya Huang, 2023). The metagraph is their developed dataset graph where nodes must represent a logical statement like premise or conclusion while edges represent whether a node is supporting or rebutting another. Each logical statement within a node contains standard logical form, but additionally include "modal" characteristics, namely degrees of certainty associated with the statement. When encompassed all-together in either graph or matrix form, the complexity of the dataform establishes how the model should be thinking about the problem at each stage, with what degree of conviction, and with what supporting information. Models trained on this type of framework can then be evaluated and fintuned on multiple levels of inference and resoning, including "Incorrect inference type: the model predicts the correct structure, but over one of the inference steps has the inverse type ..., (G2) Incorrect rebuttal: missing or incorrectly predict a rebuttal step; (G3) Incorrect conclusion: mismatched conclusion node at the end of the reasoning chain" etc.(Yinya Huang, 2023). The authors propose that this more accurately matches the cognitive processes humans go through when reasoning. Likewise, they believe that structuring datasets in this manner is more likely to improve numerical reasoning in the CoT context. While the authors conduct preliminary experiments into gauging the success of this dataset, the paper's results adhere to the idea that this is meant to encourage better and more thorough approaches to evaluating multi-step reasoning problems but is not a comprehensive and complete dataset. Accordingly, when testing current CoT models on this dataset, results showed a substantive gap in performance when compared to standard CoT datasets. Specific information on CoT model performance when trained on MetaLogic was not provided.

3 Compare and Contrast

3.1 Decomposing Problems and Rationale Generation for Solving Complex Problems

The improvement in task performance brought about by prompting a model to produce step-by-step reasoning for a solution to a problem is a key theme across all our papers. This strategy is the basis of CoT prompting, as described in Wei et al. (2023).

In studying the NL2Code Task, Luyu Gao (2021) also specifically cites rationale generation and the decomposition of a problem into smaller parts as a potential pathway to improve the performance of LLMs when generating code for problem-solving. This is precisely the combination of a CoT approach with code generation, explored both in Luyu Gao (2021) and (Wenhu Chen, 2022). As suggested by Daoguang Zan (2022), incorporating problem decomposition did lead to performance improvements for LLMs in the applications studied. For example, the PAL paper shows that incorporating PoT and CoT prompting led to problem-solving rate improvements across 8 mathematical reasoning datasets as compared to directly using an LLM like Codex (Luyu Gao, 2021). Similarly, in Wenhu Chen (2022), PoT prompting leads to a 26.4% improvement on average as compared to using Codex directly across 5 math word problem datasets.

Thus, it is clear that incorporating step-by-step reasoning when generating answers from LLMs through new prompting techniques is a key trend in the field, and has led to performance improvements across a variety of different types of problems.

3.2 Importance of Semantic Binding

Semantic Binding is the association of semantic meaning to variable names when LLMs generate code. By using semantically meaningful variable names, a program can also be a natural representation to convey human thoughts. For example, in a question on how much time a train A and B take to travel given a specific distance and time, the LLM might use names like `train_A_distance` and `train_b_distance` instead of x and y. By analyzing variants that substitute meaningful names with random characters, both Wenhu Chen (2022) and Luyu Gao (2021) found that removing the binding will, in general, hurt the model's performance. The decrease in performance is especially significant for more complex problems.

3.3 Reducing Human Prompt-Engineering Workload

Our papers also exhibit a trend toward developing techniques to reduce human engineering workload when it comes to prompting. CoT and PoT prompting depends on human engineering: they require humans to select a few exemplars to annotate with rationales and then answers. However, this can cause issues. First, there is significant variation in the nature of different questions / numerical reasoning problems in terms of difficulty, scope and domain. Second, annotating examples can be time-consuming and expensive, which results in increased costs and reduced availability of annotated examples.

We see several papers in this study make strides to tackle this problem. Xinyun Chen (2023) explore a framework for iterative debugging of generated LLM code by automatically modifying prompts to include feedback messages for subsequent stages. This ensures that one does not need to re-engineer prompts for debugging purposes. Instead, the LLM is used to first predict candidate programs, infer program correctness and use automatically produced feedback messages for subsequent debugging steps. The authors found that their self-debugging framework improved baseline accuracy across a range of benchmarks with no required human intervention. Similarly, the active prompting framework proposed in Shizhe Diao (2023) aims to judiciously select the most helpful and informative examples for annotation, reducing the human engineering workload by using an uncertainty-based annotation strategy.

3.4 Integrating External Tools and Resources

Our papers also display an interest in augmenting LLMs with access to external tools and resources, as well as exploring the integration of external tools and plug-and-play modular approaches. For example, Wenhu Chen (2022) and Luyu Gao (2021) use external interpreters to execute numerical reasoning tasks more effectively. In another vein, ViperGPT utilizes a provided API to incorporate computer vision models that equip LLMs with the ability to perform visual reasoning tasks. The authors of Chameleon take inspiration from ViperGPT and employ diverse types of tools, including off-the-shelf computer vision models, web search engines, Python functions, and rule-based modules. This allows Chameleon to address a wide range of reasoning tasks, including lifting state-of-the-art on two diverse benchmarks, ScienceQA and TabMWP.

3.5 Key Differences

Differing from our other sources, Yinya Huang (2023) have posited that the path to strengthening CoT and POt requires more rigorous attention to problem breakdown, both in the ways of formal and informal logic. Drawing on ideas from cognitive decision-making, they propose altering the way in which LLMs consider and approach complex reasoning tasks. Where papers like Chameleon (Pan Lu, 2023) and ViperGPT (Dídac Surís, 2022) attempt to add layers of tools or where active-prompting and self-debugging turn to re-prompting and self-editing, this paper suggests that LLMs need to have different structures (and differently structured datasets) to sufficiently model how a human might consider these kinds of tasks.

4 Future work

This literature review stimulated a few open opportunities and challenges for future work:

- Wenhu Chen (2022) applies PoT prompting to generate code to solve numerical reasoning problems and achieve SoTa performance on multiple math word problem datasets. However, the accuracy on complex types of problems, including geometric problems, probability problems, or polynomial problems is still low, at 10%, 38% and 50% respectively. The authors' error analysis indicates that they have two main types of errors: value grounding errors wherein the model fails to assign correct values to the variables related to the question, and logic generation errors where the model fails to generate the correct computation process (Wenhu Chen, 2022). We would like to see if the prompting methods explored above could improve performance on errors for more complex problems, specifically self-debugging prompting and active prompting. In the context of self-debugging, we could provide feedback on the code correctness to see if the model can identify implementation errors. In the context of active prompting, we could use the framework from Shizhe Diao (2023) to determine the most important and helpful questions to annotate for the datasets and see if the model performance improves.

- In Pan Lu (2023), an LLM was used as a scheduler and planner for sub-tasks and the appropriate tool to solve them. The LLMS used were GPT-4 and GPT-3 derivatives. However, given the size and the expense of these models, we wonder whether a smaller model that is more publicly available (like BERT or LLAMA) might be able to achieve comparable results.

- We could also leverage the Chameleon architecture to complete adjacent numerical reasoning tasks. In their paper, the authors focus on science question answering and mathematical tabular reasoning (Pan Lu, 2023). We could extend this to mathematical word problem or financial datasets as seen in Wenhu Chen (2022). We could also focus on the task of a standardized test like the ACT Math section, crafting a series of modules and using the Chameleon architecture to delegate the question to the specific modules. Combining what we have learned from other papers like ViperGPT and PoT, we could use creative prompting methods and code generation to execute the relevant queries proposed by our system

References

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2357–2367, Minneapolis, Minnesota. Association for Computational Linguistics.

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. 2021. Evaluating large language models trained on code.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. Pangu-coder: Program synthesis with function-level language modeling.

Fengji Zhang Dianjie Lu Bingchao Wu Bei Guan Yongji Wang Jian-Guang Lou Daoguang Zan, Bei Chen. 2022. Large language models meet nl2code: A survey. *arXiv preprint arXiv:2212.09420*.

Carl Vondrick Dídac Surís, Sachit Menon. 2022. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2211.10435*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 158–167, Vancouver, Canada. Association for Computational Linguistics.

Shuyan Zhou Uri Alon Lengfei Liu Yiming Yang Jamie Callan Graham Neubig Luyu Gao, Aman Madaan. 2021. Pal: Program-aided language models. *arXiv preprint arXiv:2108.00648*.

Aman Madaan and Amir Yazdanbakhsh. 2022. Text and patterns: For effective chain of thought, it takes two to tango.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis.

Hao Cheng Michel Galley Kai-Wei Chang Ying Nian Wu Song-Chun Zhu Jianfeng Gao Pan Lu, Baolin Peng. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*.

Yong Lin Tong Zhang Shizhe Diao, Pengcheng Wang. 2023. Active prompting with chain-of-thought for large language models. *arXiv preprint arXiv:2303.08128*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Xinyi Wang William W. Cohen Wenhu Chen, Xueguang Ma. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Nathanael Schärli Denny Zhou Xinyun Chen, Maxwell Lin. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Ruixin Hong3 Xiaodan Liang Changshui Zhang Dong Yu Yinya Huang, Hongming Zhang. 2023. Metalogic: Logical reasoning explanations with fine-grained structure. *arXiv preprint arXiv:2302.12246*.

A Appendix

A.1 Definition of pass@k

For each programming problem, n candidate code solutions are sampled and then k are randomly picked. If any of these k code solutions pass the given test cases, the problem can be regarded as solved. So pass@k can be thought of as the proportion of solved problems