Markup Language for Mathematical Reasoning with LLMs

Ryutaro Yamauchi[1], Sho Sonoda[2,4], Akiyoshi Sannai[3,4], and Wataru Kumagai[4,2]

[1] ALBERT Inc., Shinjuku, Tokyo, Japan
ryutaro_yamauchi@albert2005.co.jp
[2] Center for Advanced Intelligence Project, RIKEN, Chuo, Tokyo, Japan
sho.sonoda@riken.jp
[3] Kyoto University, Department of Physics, Graduate School of Science, Kyoto, Japan
sannai.akiyoshi.7z@kyoto-u.ac.jp
[4] The University of Tokyo, Hongo, Tokyo, Japan
kumagai@weblab.t.u-tokyo.ac.jp

Introduction – Large language models (LLMs) can solve diverse tasks without additional training, despite being trained on a simple task of predicting the next token, when conditioned on an appropriate *prompt* [1]. LLMs have shown remarkable performance in various natural language processing (NLP) tasks, including arithmetic inference. And it is reported that effective prompts can improve LLMs' performance. For example, we can improve LLMs' arithmetic reasoning ability by having them output intermediate steps to solve a problem rather than having them output the final step directly. This technique is called Chain-of-Thought (CoT) [8, 4].

One challenge in mathematical reasoning with LLMs is how to handle errors in calculations or reasoning that may occur in LLMs' output [3]. Many previous studies have taken the approach of reducing the occurrence of errors themselves by increasing the model size [7], tuning the dataset [5], or integrating with external tools [2, 9]. However, since LLMs are probabilistic language generative models, we cannot completely eliminate LLMs' errors. And since LLMs can make mistakes, the integration of LLMs and external tools does not always work. Therefore, to further enhance LLMs' mathematical reasoning abilities, a different approach is needed, which is to make LLMs *recognize and correct errors* that occur in their reasoning.

In this study, we attempt to correct errors that occurred within the CoT when using LLMs to solve mathematical problems by integrating with an external tool (Python REPL). Specifically, we have LLMs output both CoT and Python codes and feed back the results of Python code execution to LLMs. Our study differs from previous studies [2] in that we do not have LLMs directly generate Python codes to solve the problem but rather use Python codes as part of CoT. However, we found that simply feeding back Python execution results did not lead LLMs to behave as we expected: when we let them write codes in CoT, LLMs did not wait for the execution results but also output the fake execution results. Also, when LLMs make mistakes in CoT, they tend to think that the code is incorrect, even if they write the correct Python code and get the results of that execution. To avoid these problems, we defined an XML-like markup language, gave its grammar to LLMs, and used it to interact with LLMs. The markup language includes `THINK` tag for CoT, `PYTHON` tag for Python code description, and `OUTPUT`

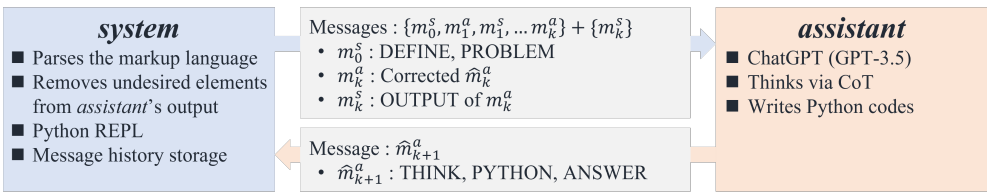

Figure 1: An overview of the mathematical reasoning process. *system* and *assistant* continue to interact until an answer is obtained. All messages are structured in the markup language.

tag for Python code execution results, etc. By using these tags, the output text of LLMs is structured. As a result, we can remove fake Python execution result that LLMs output, and we can make LLMs correct the errors in CoT based on the Python code execution results. Our method achieved a success rate of 65.6% in the MATH dataset [3].

Interact with ChatGPT using markup language - In this study, we used OpenAI ChatGPT (GPT-3.5-Turbo) [6] via API, which is an LLM that takes a sequence of messages and predicts the next message. The API specification allows us to set three roles as the speaker of a message: *assistant*, *user*, and *system*, where *assistant* is an AI, *user* is a human and *system* is a conversation context manager. As shown below, we designed the mathematical reasoning process as an interaction between *system* and *assistant*. First, as *system*, we present the problem to be solved and define the grammar of the markup language and rules of reasoning (m_0^s). Then, when *assistant* returns an output ($\hat{m}_k^a$), *system* analyzes it, and if it contains Python code, *system* returns the execution result (m_k^s). If *assistant* outputs any undesired elements, such as fake Python execution results, *system* removes them from the *assistant*'s output (m_k^a). We made *system* and *assistant* repeat this process until *assistant* outputs the answer (Fig. 1).

The markup language we have defined is based on XML syntax and is a set of elements consisting of content enclosed in a start tag `<TAG>` and end tag `</TAG>`. Since the dataset used to train LLMs is assumed to include text written in XML and HTML, LLMs can write the markup language that is similar to them. The markup language includes the tags `DEFINE`, `PROBLEM`, `ANSWER`, `THINK`, `PYTHON`, and `OUTPUT`. `DEFINE` tag is for defining tags and rules. The grammar of the language and the tags are defined by `DEFINE` tag. `PROBLEM` and `ANSWER` tags are for describing the problem and answer, and the solving process ends when *assistant* outputs `ANSWER` tag. `THINK` tag is for describing thoughts. By defining `THINK` tag as "a tag for describing thinking step-by-step", we induce Zero-shot-CoT [4]. `PYTHON` tag is for describing Python code. When *assistant* uses `PYTHON` tag, *system* returns the execution result using `OUTPUT` tag. By using these tags, all messages, including the initial system prompt, are written in the markup language, thus strongly conditioning *assistant* to output in the markup language.

```
<DEFINE type="rule" role="system">
The system interprets the text received from the assistant
The system returns the execution results to the assistant u
</DEFINE>

<DEFINE type="tag" name="DEFINE">This tag defines a rule or
<DEFINE type="tag" name="THINK">Thinking done step by step.
<DEFINE type="tag" name="PYTHON">Executable python code. Th
<DEFINE type="tag" name="PROBLEM">Problem to be solved.</DE
<DEFINE type="tag" name="ANSWER">The answer to the problem.
```

Figure 2: The proposed markup language.

We instructed LLMs to trust the contents of `OUTPUT` tags rather than the contents of `THINK` tags. This enables LLMs to ignore errors in CoT by referring to the results of Python code execution. Without this instruction, LLMs would either assume that the Python code was incorrect and fall into a loop of repeated debugging or ignore the contents of the `OUTPUT` tag.

Experiments and Results - We evaluated our method on the MATH dataset [3], which contains 12,500 challenging competitive math problems. We sampled 90 problems from the dataset and solved them using our method. We generated five answers for each question and evaluated them using two criteria: 1. whether our method answered the problem correctly at least once, and 2. whether our method answered the problem correctly by a majority vote. As a result, 75 problems (83.3%) were answered correctly by the proposed method at least once, and 59 problems (65.6%) were answered correctly by the majority vote. These scores are higher than the score of Minerva 540B (50.3 %) [5], which was fine-tuned with technical content.

At the conference, we will report on the effects of `OUTPUT` tag priority instruction in addition to the details of the above experiment.

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[2] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

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

[4] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213, 2022.

[5] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 3843–3857. Curran Associates, Inc., 2022.

[6] OpenAI. OpenAI: Introducing ChatGPT. https://openai.com/blog/chatgpt, 2022.

[7] OpenAI. Gpt-4 technical report, 2023.

[8] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.

[9] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.