Making LLMs Program Interpreters via Execution Trace Chain of Thought

Koshi Eguchi [1] **Takuya Akiba** [1]

Abstract

Programmatic representations constitute policies, reward functions, environment models, and skill libraries for autonomous agents. However, their practical value hinges on large language models (LLMs) that can understand and reason about code, not merely generate it. A crucial aspect of this reasoning is the ability of LLMs to predict the outcome of the code (or "execute" it), a critical yet less developed area. Improving this capability is essential for verifiable policies, self-auditing reward functions, and debuggable environment models within program-centric agents.

To address this, we propose *ET-CoT (Execution Trace Chain of Thought)*, an approach where LLMs learn to generate a detailed and systematic program execution trace as a chain of thought to predict program outcomes. Taking Python as an example, we designed a program-execution trace format inspired by recent theoretical advances. Next, we developed a new Python interpreter called *PyTracify*, which outputs these traces during execution. We then generated a large number of traces and fine-tuned an LLM using them. This ET-CoT approach allows the LLMs to execute Python programs consistently by generating the trace as a CoT. Specifically, our fine-tuned model outperforms other models of comparable size on code execution benchmarks such as CRUXEval-O and LiveCodeBench.

1. Introduction

In recent advancements within large language model (LLM) development, performance on code generation tasks has improved remarkably, reaching notably high level of proficiency. However, a significant challenge persists concerning predicting the actual outcomes of code. For humans, tracing program instructions sequentially and simulating their behavior to predict execution results is often simpler than writing the code itself and can even be considered a foundational prerequisite. This highlights a pronounced difference between the current capabilities of AI and human cognition. Such a discrepancy suggests that LLMs may not yet possess a deep, genuine understanding of program execution principles or may lack the robust step-by-step reasoning abilities that enable humans to logically deduce outcomes by meticulously following each operational stage.

Recent theoretical work indicates that the design of a chain of thought (CoT) is essential for LLMs to execute complex processes, such as programs accurately. Recent theoretical research suggests that by employing CoT, autoregressive transformers can, in principle, attain computational capabilities. Conversely, it has been shown that without CoT, the computational power of transformers is limited to highly restricted classes, such as uniform TC^0. Therefore, the capacity to generate an appropriate CoT that corresponds to each computational step of a program, and scales in length with the computational complexity, is considered indispensable for LLMs to achieve reliable program execution.

To address this, we propose *ET-CoT (Execution Trace Chain of Thought)*, an approach where LLMs learn to generate a detailed and systematic program execution trace as a chain of thought to predict program outcomes. This method involves the LLM generating a detailed "execution trace," which describes the program's execution process, to serve as its CoT. First, based on theoretical considerations, we designed an execution trace format that represents each program step in a detailed and unambiguous manner. Next, we developed PyTracify, a new Python interpreter capable of outputting traces conforming to this designed format while executing Python programs. Subsequently, leveraging PyTracify, we constructed an extensive dataset composed of Python codes, their corresponding execution traces, and the final outputs, derived from a diverse range of collected Python programs. This dataset was then utilized to fine-tune an LLM. Through this training, the LLM learns to generate an execution trace as a CoT and then predict the final output based on this generated trace.

The performance of the LLM fine-tuned with our ET-CoT was evaluated on standard code execution benchmarks,

[*]Equal contribution [1]Sakana AI, Tokyo, Japan. Correspondence to: Takuya Akiba <takiba@sakana.ai>, Koshi Eguchi <koshi@sakana.ai>.

ICML 2025 Workshop on Programmatic Representations for Agent Learning, Vancouver, Canada.

Figure 1: Example of Python code and its trace.

Code:

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

fibonacci(1)
```

PyTracify Trace:

```
0 Statement
1 FunctionDef def fibonacci(n):
0 Statement
1 Expr fibonacci(1)
2 Call fibonacci(1)
2 CallArg0 1
3 Constant 1
2 Statement
3 If if n <= 1:
4 Compare n <= 1
4 CompareLeft n
5 Name n = 1
4 CompareRight 1
5 Constant 1
4 CompareResult 1 <= 1 = True
3 IfCond True
3 Statement
4 Return return n
5 Name n = 1
2 Call fibonacci(1) = 1
```

namely CRUXEval-O and LiveCodeBench (LCB-Exec). The results demonstrated that our ET-CoT model surpassed the performance of existing models with comparable parameter sizes. These findings underscore the effectiveness of our ET-CoT approach.

Contributions. ① We proposed a novel approach, termed Execution Trace Chain of Thought (ET-CoT), which involves generating detailed execution traces as CoT to enhance the ability of LLMs to comprehend programs and predict their execution outcomes. ② To implement this approach, we designed a systematic execution trace format. ③ We developed PyTracify, a Python interpreter that outputs execution logs conforming to this trace format. ④ Leveraging PyTracify, we constructed a large-scale training dataset comprising over 160,000 samples. ⑤ Through these contributions, we experimentally demonstrated a substantial improvement in the program execution capabilities of LLMs.

2. Method

2.1. Implications from Theory of Computation

Recent theoretical work has shown, under a set of assumptions, that an autoregressive transformer is computationally universal (i.e., Turing-complete) (Schuurmans et al., 2024), while another study, working under different assumptions, demonstrates that an autoregressive transformer can perform the computations in the complexity class P (Merrill & Sabharwal, 2024). This is a positive result, suggesting that, in a CoT setting, a real LLM could in principle "become a computer," i.e., understand and execute programs. However, the result is purely theoretical: it relies on several idealised (and arguably unrealistic) assumptions. Real-world LLMs are trained by gradient descent under practical constraints, so whether such universality can ever be achieved in practice remains an open question.

Conversely, there are negative, upper-bound results showing that, without CoT, an autoregressive transformer is far from computationally universal (Merrill & Sabharwal, 2024; 2023). Specifically, without CoT, its computational power is bounded above by uniform TC^0. This insight highlights the crucial role of CoT design if we want an LLM to "think like a computer" (for example, to run programs). Trying to force the model to perform computations that exceed TC^0 in zero or a constant-length CoT will inevitably fail. Therefore, an LLM that can execute programs consistently and reliably is likely to need to be able to emit CoT traces whose length grows at least proportionally to the required computational effort (i.e., the number of computation steps).

2.2. Execution Trace Format

Our goal is to create an LLM that, given a Python program, can "execute" the program while generating a CoT that satisfies the conditions above. We call this CoT trajectory an *(execution) trace*.

The most crucial aspect in designing an execution trace that meets the requirements discussed in Section 2.1 is that it contains no leaps and that each step is computationally "easy" enough for LLMs. We aim to make the trace systematic, similar to how a human understands a program and simulates its execution. That is, we essentially go line by line (more precisely, statement by statement), recursively parsing and understanding the program, while simultaneously evaluating it bottom-up.

Each trace entry consists of a triplet: nest depth, mnemonic, and operation. The nest depth is included to help LLMs recognize and reason the state of recursive processes. The mnemonic primarily represents AST node names. The operation describes what actually happens there (such as the raw code about to be evaluated, or the action or value that results from the evaluation). See Figure 1 for an example.

One interesting design point is the handling of memory. To execute a program, it is necessary to write and read the values of variables and arrays. From our preliminary experiments, we concluded that it is not required to handle this explicitly. In other words, for operations like variable assignments, it is sufficient to simply record that they occurred, just like other entries in the trace. For example, if

the trace contains a record of writing to a variable `a` (such as `a=3`) the LLM can refer to it and correctly read the latest value when variable `a` is next read.

2.3. Implementation of PyTracify

To create a training dataset, we implemented PyTracify, a Python interpreter that, when given a Python program as input, executes the program while outputting a trace in the format of Section 2.2. It is implemented in Python. Parsing is done using the `ast` module, and it recursively executes the program while managing stack frames.

2.4. Dataset Construction

We constructed a comprehensive dataset totaling 160,017 samples by executing Python code snippets with PyTracify and capturing their execution traces. The primary goal of this dataset is to train models to predict program outputs by understanding their step-by-step execution. The format of each sample in our dataset is inspired by DeepSeek's R1 methodology, as illustrated in Table 1. Specifically, the execution trace generated by PyTracify is enclosed within `<think>` and `</think>` tags, while the final program output is provided within `<answer>` and `</answer>` tags.

2.4.1. Dataset Composition

Our dataset is composed of five main sources:

- **Nan-Do AtCoder and LeetCode (61,350 samples):** This collection combines 35,685 samples from AtCoder programming contest problems[1] and 25,665 samples from LeetCode problems[2] provided by Nan-Do on Hugging Face. The LeetCode subset underwent a decontamination process, which is detailed in Section 2.4.2.
- **APPS (Hendrycks et al., 2021) (44,614 samples):** We incorporated 44,614 samples from the APPS dataset. This subset was also decontaminated, as described in Section 2.4.2.
- **MBPP (Austin et al., 2021) (1,365 samples):** From the Mostly Basic Python Problems (MBPP) dataset, we generated 1,365 samples. For problems containing multiple `assert` statements, each assertion was treated as a unique test case, leading to the creation of distinct data points with corresponding inputs and outputs.
- **PyX (Ding et al., 2024) (13,809 samples):** We selected 13,809 samples from the `semcoder/PyX` dataset hosted on Hugging Face[3].
- **Custom Datasets (38,879 samples):** To address specific challenges observed in language models' code understanding capabilities, we developed custom datasets. First, we created a *String Function Behavior dataset (12,000 samples)*. CruxEval (Gu et al., 2024) highlights the importance of understanding standard library functions, having constructed its dataset using 69 such functions, 47 of which are string-related. We observed that LLMs, due to their token-based processing, often struggle with character-level string manipulations. To target this, we created a dataset focusing on eight commonly challenging string functions: `len`, `slice`, `replace`, `rpartition`, `find`, `join`, `removeprefix`, and `rstrip`. For each function, we generated 1,500 samples by applying it to randomly generated strings with lengths varying from 3 to 20 characters, resulting in $1,500 \times 8 = 12,000$ samples. Second, we developed a *Tokenizer Vocabulary Length dataset (26,879 samples)*. This dataset primarily focuses on vocabulary items for which the model initially struggled to correctly predict the length of a vocabulary token. In our experiments, we utilized Llama3.1-8B-Instruct (Grattafiori et al., 2024) as the base model, which we fine-tuned using our PyTracify datasets. Therefore, to better support this model, we created this specific dataset to enhance its understanding of token lengths within the Llama3 tokenizer's vocabulary. Samples corresponding to particularly challenging vocabulary items were upsampled.

All of the samples were chosen based on the successful execution and trace generation by PyTracify, with a timeout threshold of 5 seconds per execution.

2.4.2. Decontamination

For our evaluation, we utilized the LiveCodeBench (LCB-Exec) benchmark (Jain et al., 2024), whose code execution tasks are constructed from LeetCode problems. Given that our dataset incorporates samples from APPS (Hendrycks et al., 2021) and the LeetCode portion of the Nan-Do dataset, there was a potential for contamination with the LCB-Exec evaluation set. To mitigate this and ensure a fair assessment, we performed a decontamination process. Specifically, we adapted a script from the Open-R1 repository[4], employing an n-gram size of 8, to remove overlapping problems. This decontamination was applied to both the APPS dataset and

[1] https://huggingface.co/datasets/Nan-Do/atcoder_contests

[2] https://huggingface.co/datasets/Nan-Do/leetcode_contests

[3] https://huggingface.co/datasets/semcoder/PyX

[4] https://github.com/huggingface/open-r1/blob/main/scripts/decontaminate.py

Table 1. **The training prompt format for the ET-CoT training dataset.** The user supplies the Python code for execution and any required standard input values. The assistant then provides the execution trace (generated using PyTracify), encapsulated within <think> tags, followed by the code's final output, encapsulated within <answer> tags.

> You are a highly capable assistant. Your task is to estimate the output of the given Python code. The reasoning process and output are enclosed within <think> </think> and <answer> </answer> tags, respectively, i.e., <think> reasoning process here</think><answer> output here </answer> User: <code>code</code> <input>input</input>. Assistant:<think>trace</think><answer>output</answer>

the LeetCode samples from the Nan-Do dataset, particularly targeting problems corresponding to those in LCB-Exec (as detailed in Sections 3.3 and A.3 of the LCB paper (Jain et al., 2024)). This step was crucial for an unbiased evaluation of our model's performance on LCB-Exec.

3. Experiments

3.1. Setup

Benchmarks. For evaluating program execution reasoning, we utilize the CruxEval-O benchmark (Gu et al., 2024) and the code execution task from LiveCodeBench (LCB-Exec for short) (Jain et al., 2024). Referencing the evaluation methodology in the SemCoder paper (Ding et al., 2024), we compared our model against a suite of baseline models: Code Llama (Rozière et al., 2024), StarCoder2 (Lozhkov et al., 2024), DeepSeekCoder (Guo et al., 2024), Magicoder (Wei et al., 2024), as well as SemCoder (Ding et al., 2024) itself, all prompted in the benchmark-specified reasoning (chain-of-thought) format. Consistent with the SemCoder paper, the inferences for these baseline models—Code Llama, StarCoder2, DeepSeekCoder, Magicoder, and SemCoder—follow the original inference settings of each benchmark. Results are reported with pass@1. Our model is designed to produce deterministic outputs. Therefore, in all inference tasks involving our model, we use a temperature of 0 and top-k sampling of 1, as there is no inherent need to introduce diversity into the outputs. The inference prompt format consists of the user-provided code and input, mirroring the "User" portion of the training format detailed in Table 1. The model is then expected to generate the subsequent reasoning and answer. We evaluate the correctness by comparing the content within the generated `<answer>` tags against the ground truth expected output.

Training. We fine-tuned Llama3.1-8B-Instruct with our ET-CoT dataset detailed in Section 2.4. It was fine-tuned for 3 epochs on a single server equipped with 8 NVIDIA H100 GPUs. We used the AdamW optimizer (Loshchilov & Hutter, 2019) with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and an $\epsilon =$ 1e-8. We employed a cosine learning rate decay schedule, starting from an initial learning rate of 2e-5 and decaying to 4e-6. The training was conducted with a batch size of 64 and a context length of 8192 tokens.

Table 2. **Performance comparison on code execution benchmarks.** All results are reported with pass@1.

Model	Size	Benchmark	
		CruxEval-O	LCB-Exec
CodeLlama-Python	13B	36.0	23.2
CodeLlama-Inst	13B	41.2	25.7
StarCoder2	15B	46.2	33.6
StarCoder2-Inst	15B	50.9	29.6
CodeLlama-Python	7B	34.0	23.0
CodeLlama-Inst	7B	36.8	30.7
StarCoder2	7B	34.5	26.3
Magicoder-CL	7B	35.5	28.6
Magicoder-S-CL	7B	35.8	30.0
DeepSeekCoder	6.7B	41.2	36.1
DeepSeekCoder-Inst	6.7B	43.2	34.0
Magicoder-DS	6.7B	41.9	38.8
Magicoder-S-DS	6.7B	43.5	38.4
SemCoder	6.7B	65.1	59.7
SemCoder-S	6.7B	63.9	61.2
Llama3.1-8B	8B	11.1	35.4
Llama3.1-8B + ET-CoT	8B	**67.6**	**88.9**

3.2. Result

As shown in Table 2, Llama3.1-8B + ET-CoT achieved a remarkably high accuracy of 88.9% on LCB-Exec and 67.6% on CruxEval-O. This represents a significant improvement compared to the performance of the baseline Llama3.1-8B model before fine-tuning.

Furthermore, the proposed model demonstrated superior results to existing leading code LLMs of comparable size. Notably, our model surpassed SemCoder, which is also trained explicitly for code execution prediction. This suggests that the ET-CoT approach significantly enhances the accuracy and reliability of program execution by LLMs.

4. Conclusions

We introduced ET-CoT, a novel approach that significantly enhances LLM program execution capabilities by training models to generate detailed execution traces as a chain of thought. Our fine-tuned Llama3.1-8B model using ET-CoT outperformed existing leading code LLMs of comparable size, demonstrating that this method substantially improves the accuracy and reliability of program execution by LLMs.

As future developments for this research, we plan to investigate whether ET-CoT can lead to the acquisition of other capabilities beyond simple execution.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Ding, Y., Peng, J., Min, M. J., Kaiser, G., Yang, J., and Ray, B. Semcoder: Training code language models with comprehensive semantics. *arXiv preprint arXiv:2406.01006*, 2024.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Wyatt, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Guzmán, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Thattai, G., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Zhang, J., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Prasad, K., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Lakhotia, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Tsimpoukelli, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Zhang, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Maheswari, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Albiero, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Wang, X., Tan, X. E., Xia, X., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Srivastava, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Teo, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Dong, A., Franco, A., Goyal, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Liu, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Gao, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Le, E.-T., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Kokkinos, F., Ozgenel, F., Caggioni, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Inan, H., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Zhan, H., Damlaj, I., Molybog, I., Tufanov, I., Leontiadis, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Lam, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Jagadeesh, K., Huang, K., Chawla, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Liu, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Mehta, N., Laptev, N. P., Dong, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P.,

Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Parthasarathy, R., Li, R., Hogan, R., Battey, R., Wang, R., Howes, R., Rinott, R., Mehta, S., Siby, S., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Mahajan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Patil, S., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Deng, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Koehler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wu, X., Wang, X., Wu, X., Gao, X., Kleinman, Y., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Zhao, Y., Hao, Y., Qian, Y., Li, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., Zhao, Z., and Ma, Z. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Gu, A., Roziere, B., Leather, H. J., Solar-Lezama, A., Synnaeve, G., and Wang, S. CRUXEval: A benchmark for code reasoning, understanding and execution. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/gu24c.html.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., and Liang, W. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder 2 and the stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

Merrill, W. and Sabharwal, A. A logic for expressing log-precision transformers. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

Merrill, W. and Sabharwal, A. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=NjNGlPh8Wh.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Schuurmans, D., Dai, H., and Zanini, F. Autoregressive large language models are computationally universal. *CoRR*, abs/2410.03170, 2024. doi: 10.48550/ARXIV.2410.03170. URL https://doi.org/10.48550/arXiv.2410.03170.

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.