Towards Bridging Classical and Neural Computation through a Read-Eval-Print Loop

David W. Zhang [*1] **Michaël Defferrard** [*1] **Corrado Rainone** [*1] **Roland Memisevic** [1]

Abstract

Humans rely on step-by-step reasoning to solve new problems, each step guided by the feedback of its effect on a potential solution. For complicated problems, such a sequence of step-by-step interactions might take place between the human and some sort of software system, like a Python interpreter, and the sequence of operations so obtained would then constitute an algorithm to solve a particular class of problems. Based on these ideas, this work proposes a general and scalable method to generate synthetic training data, which we in turn use to teach a Large Language Model to carry out new and previously unseen tasks. By tracing the execution of an algorithm, through careful transformations of the control flow elements, we can produce "code traces" containing step-by-step solutions for a range of problems. We empirically verify the usefulness of training on such data, and its superiority to tracing the state changes directly.

1. Introduction

Neural computations, as performed, for example, by large language models (LLMs) (Radford et al., 2018; 2019), are "informal": they match patterns between distributed representations. This approach allows for reasoning shortcuts and analogies but can lead to hallucinations. In contrast, classical computations performed by a Turing machine or equivalent virtual machine (VM) are formal, providing guarantees but with reduced flexibility. We herein propose to bridge those two computing paradigms by making neural and classical computation interact through a Read-Eval-Print Loop (REPL), making informal reasoning drive formal computations.

[*]Equal contribution [1]Qualcomm AI Research. Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.. Correspondence to: David Zhang <davizhan@qti.qualcomm.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

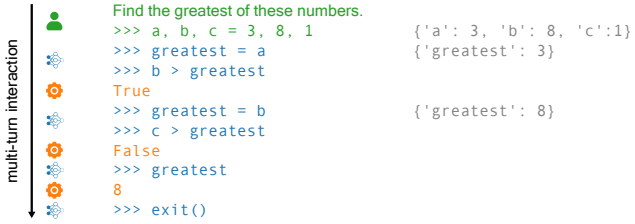


Figure 1: Interaction between the user writing a prompt, the LLM writing code, and the Python interpreter inserting responses. The resulting trace resembles an interactive session within the Read-Eval-Print Loop (REPL), and is used to train an LLM to interactively solve problems step-by-step. Statements correspond to state transitions (grey), and are used as the "state trace" baseline, as detailed in section 3.

Such interactions have been proposed by making an LLM write programs for an interpreter to execute (Chen et al., 2023; Gao et al., 2023). We propose instead to work at an intermediate level of abstraction, where the interpreter manipulates the data but the LLM controls the execution flow. Figure 1 illustrates such an LLM–interpreter interaction. The VM's evaluation of expressions are used by the LLM to make decisions on what to execute next, and the execution of statements manipulates the data, inducing state transitions in the VM. This multi-turn interaction with the VM requires the LLM to plan ahead, but also allows it to inspect the data and backtrack where applicable. We experimentally demonstrate that this level of abstraction yields generalization benefits.

We generate training data by tracing the execution of algorithms, resulting in code traces that consist of a sequence of interactions with the Python interpreter through its read–eval–print loop (REPL). This sequence of interactions effectively teaches the LLM how to execute algorithms by demonstrating how the Python REPL can be leveraged for solving a given problem instance.

Finally, we explore whether we may omit the VM and have the LLM simulate it. Code traces then constitute a training signal in the form of sequences of reasoning steps, akin to those elicited by Chain of Thought (CoT) prompting (Wei et al., 2022; Kojima et al., 2023; Zhou et al., 2023). This

```python
def exchange_sort(A: list) -> list:
    for i in range(len(A)):
        for j in range(i+1, len(A)):
            if A[j] < A[i]:
                A[i], A[j] = A[j], A[i]
    return A
```

(a) Algorithm.

```
>>> A = [28, 25, 62, 50, 97]
>>> forloop0 = iter(range(len(A)))
>>> i = next(forloop0)
>>> forloop1 = iter(range(i+1, len(A)))
>>> j = next(forloop1)
>>> A[j] < A[i]
True
>>> A[i], A[j] = A[j], A[i]
...
>>> i = next(forloop0)
StopIteration
>>> A
[25, 28, 50, 62, 97]
```

(b) Code trace.

```
{'A': [28, 25, 62, 50, 97]}
{'i': 0}
{'j': 1}
{'A': [25, 28, 62, 50, 97]}
{'j': 2}
...
{'i': 2}
{'j': 3}
{'A': [25, 28, 50, 62, 97]}
{'j': 4}
{'i': 3}
{'i': 4}
[25, 28, 50, 62, 97]
```

(c) State trace.

Figure 2: Two ways of tracing the execution of an algorithm (a): executed code statements (b) and state changes (c). Without control flow, code statements correspond to state transitions.

comes with the advantage of allowing for the generation of faithful and correct chains-of-thought at scale.

Our main contribution is that we enable LLMs to interact with the Python interpreter through its Read-Eval-Print Loop (REPL). This allows grounded step-by-step reasoning where the reasoning steps happen at a meta-level. We enable these interactions with a scalable data generation method wherein we trace the algorithm execution on diverse inputs, which we then use to finetune the LLM.

2. Algorithmic traces in interactive sessions

In this section, we explain the format in which the large language model (LLM) interacts with the Python virtual machine (VM), and how we generate algorithmic traces in that format. While we emphasize Python for concreteness, the general framework applies to other programming languages as well. Python, being a high-level general-purpose language, enables the implementation of commonly taught algorithms without the need to manage low-level memory operations such as memory allocation. It is a programming language designed to be easily readable and relatively close to pseudo-code; it therefore effectively conveys the essence of an algorithm, which is precisely what we aim to teach to the LLM.

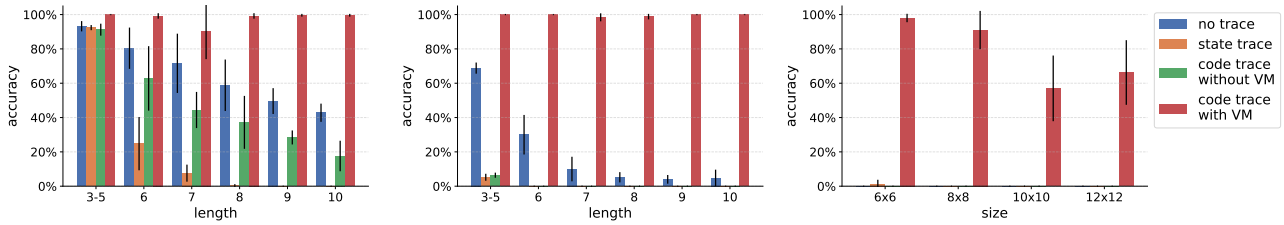
Ideally, the format of the algorithmic traces should mimic what a pre-trained LLM observed in its training dataset. For Python there exists a natural format to achieve this: the Python interactive session, also referred to as the read–eval–print loop (REPL). Herein, each code line starts with `>>>` and ends with a line break. The interpreter executes the code and updates its state, consisting of the objects in the global and local namespaces. When the code line generates a result and the result is not `None`, then the interpreter prints the result into the subsequent line. Text data in this format can be found on the web: for example in docstrings of Python code, or in tutorials on Python programming. It is however relatively scarce and each example usually only consists of a few back-and-forths between the programmer and the machine. Because of this, we implemented a method to synthetically generate such data.

2.1. Generating code traces

We generate synthetic data by tracing Python functions that implement common algorithms used for teaching data structures and algorithms, such as Bubble Sort, Exchange Sort, or A* search. These algorithms typically run a sequence of statements that are chained together in a specific order to produce the correct result. We trace the Python code at the level of a function, which allows control over which Python code we track, and which code is executed in the background without explicitly appearing in the code trace. The execution of function or method calls such as `len(A)` is, for example, not traced.

Consider as an example the function `exchange_sort` shown in Figure 2a, and its argument `[28, 25, 62, 50, 97]`. The corresponding code trace shown in Figure 2b starts with an assignment of the argument to the parameter `A`. Then, we add to the code trace by simulating the function's execution line-by-line. Lines that do not control execution are directly copied into the code trace, while lines that contain control flow statements are transformed into an interactive session. The purpose of this transformation is to explicitly place the decision of what to run next on the agent that interacts with the interpreter. Table 3 lists examples of how we transform for-loop, if-else, while-loop, and return statements. For more details we refer to Appendix A.

Every time we execute a statement in the interactive session, the interpreter might create or update a variable as a result. We track these changes in form of a *state trace*, as shown in Figure 2c. The state trace mimics the format of execution traces from previous works (Lehnert et al., 2024) and serves as one of the baselines in our experiments.



(a) Sorting random words. (b) Words with shared prefix (hard). (c) Searching with A* on mazes.

Figure 3: Size generalization. Bubble sort (a,b) and A* search (c) trained on lists of length 3-5 and mazes of size 6x6; and tested on larger ones. Code traces enable size generalization.

2.2. Learning the classical and neural computer

We generate multiple samples of code traces for each function by tracing execution with different input values. Each sample starts with a prompt describing the algorithm's behavior and the input values. These samples are then used to train the LLM on a standard next token prediction objective. As a consequence, the LLM learns both the role of neural computer, which decides what code to run next, and the classical computer which runs that code, and reports the result if any. Empirically we observe that code execution is the most difficult part to learn and generalize on. We investigate two settings that shed light on the limitations and opportunities in simulating the interpreter.

Quiz. Executing the result of many operations is difficult, because the LLM needs to keep track of the state of the machine *implicitly*. For example the list `A` in exchange sort only appears at the very end of the trace, and all intermediate states of the list remain hidden to the LLM. For the LLM to correctly predict the result, the representations of the intermediate steps need to capture the changes to the list. We introduce quizes as an optional auxiliary task to improve the representation of the LLM. For every line in the code trace we randomly add a variable to the code trace as an expression. As an effect of this, the LLM needs to be ready at any step for predicting the value of some variable in the local namespace.

Multiple algorithms. An LLM trained on the code traces of multiple different algorithms needs to predict the interpreter response for multiple different variable names, sequences of operations, and lengths of code traces. We investigate whether this type of variability can aid the LLMs ability to generalize its predicition for interpreter responses.

3. Experiments

Our experiments aim to answer the overarching question of how well code traces generalize out-of-distribution. For that purpose we compare four different setups: no trace, state trace, code trace without access to the VM, and code trace with access to the VM.

We fine-tune Llama2-7B (Touvron et al., 2023) in all our experiments with QLoRA (Dettmers et al., 2023). See Appendix D for more details on the experimental setup and Appendix E for detailed descriptions of the datasets.

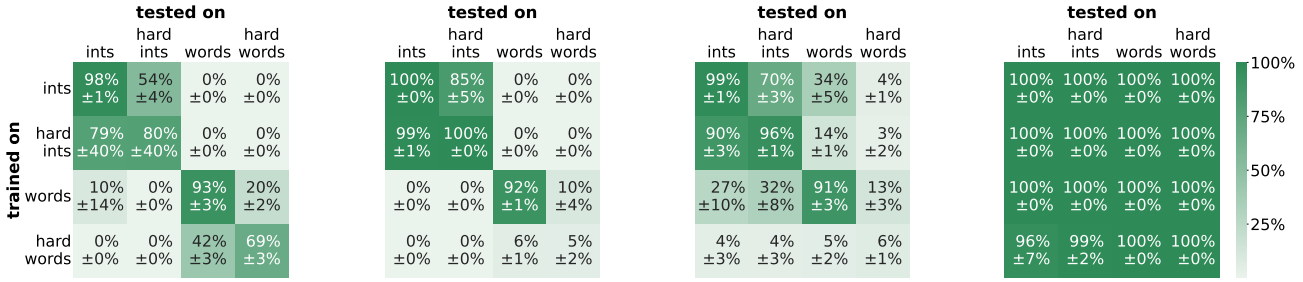
3.1. Bubble sort and A* search

In this experiment, we focus on the model's capability to generalize to longer inputs, or different input types than those seen during training. We consider bubble sort for sorting lists of words or integers, and the A* search algorithm for finding the path through a 2d maze.

For bubble sort we sample input lists from two different distributions: random words, or random words with a shared prefix. We train on lists of length 3 to 5 and test on lengths up to 10. For A* search we sample random mazes of size 6×6 using the backtracking generation algorithm and test on mazes of sizes up to 12×12.

Generalization to longer inputs. We report our results in Figure 3 and Figure 5. We observe that, by teaching the model to use a Python interpreter via training on code traces, we are able to achieve near perfect length generalization with both input distributions for sorting. Conversely, once the VM is taken out of the evaluation loop, direct prediction (i.e., no tracing at all) outperforms all tracing methods across all input lengths, including those encountered during training. Since list sorting is a very common and pedagogical task, we speculate that this might be due to the model having already been exposed to it during pre-training.

We also observe that all tracing methods, as well as no-trace, quickly break down on the hard distribution, unless the VM is used during evaluation; both tracing methods already perform poorly on in-distribution tasks, whilst direct prediction gradually breaks down as the length of evaluation inputs increases. We can also observe that state tracing performs acceptably only in-distribution, and on the easier input distribution.



(a) No trace. (b) State trace. (c) Code trace w/o VM. (d) Code trace with VM.

Figure 4: Generalization between different data distributions.

The results for A* in Figure 3c highlight the difficulty of generalization without the VM in the loop. Even in-distribution generalization fails for all but code traces with VM access. At first sight this appears inconsistent with the results reported in Lehnert et al. (2024), which applies a method similar to state traces. We speculate that this is due to differences in the training regime and dataset size: we fine-tune a pre-trained LLM with QLoRA on 1000 mazes and traces, whilst in Lehnert et al. (2024) an (albeit smaller) transformer model is trained from scratch on 1 million sequences.

Generalization to different input types. In this experiment, we examine generalization between the different input types: random words, random words with shared prefix, random integers, and random integers with shared prefix. We report the results in Figure 4.

The model equipped with access to the VM during evaluation generalizes near perfect, independent of which input type it is trained or evaluated on. With the no trace methods, we observe a block-diagonal pattern, with generalization within a certain input class (words or integers) being much more robust than "off-diagonal" generalization between different classes. In comparison, state traces generalize better between easy and hard integers, but fails on hard words distribution. Amongst those that do not have VM access, code trace without VM generalizes the best between integers and words, but similar to state traces fails on the hard words distribution.

3.2. Simulating the interpreter

In this section we examine the capability of the model to simulate the Python interpreter, and in particular, whether training on multiple algorithms is helpful or not in achieving this, and whether adding quizes to the code as outlined in section 2.2 is helpful.

We evaluate on code traces of bubble sort. For each line that corresponds to the response of the interpreter, we check whether every token in that line is correctly predicted given the ground-truth up until that token. Then, we average over all interpreter responses in all code traces that we evaluate over. The results are reported in Table 1 and Table 2. In Table 1 we observe that, while quizes do not make the model better at simulating the interpeter for in-distribution inputs, they do help it do so when evaluated on the traces for longer inputs. In Table 2, we observe that training on all sorting algorithms but bubble sort can still achieve a high, albeit lower, performance in predicting the interpreter results.

Table 1: Simulating the interpreter. We evaluate how quizes affect the LLM's accuracy in predicting the interpreter responses.

	ints		words	
trained	3-5	7	3-5	7
w/o quizes	$99.46_{\pm 0.1}$	$80.76_{\pm 0.8}$	$99.11_{\pm 0.2}$	$80.58_{\pm 1.3}$
with quizes	$99.25_{\pm 0.2}$	$84.46_{\pm 0.4}$	$98.89_{\pm 0.3}$	$85.04_{\pm 0.7}$

Table 2: Simulating the interpreter. We evaluate how training on multiple sorting algorithms affects the LLM's accuracy in predicting the interpreter responses.

trained on	3-5	7
all	$97.73_{\pm 0.4}$	$77.16_{\pm 0.6}$
all except bubble sort	$90.81_{\pm 1.2}$	$72.95_{\pm 1.0}$

4. Conclusion

We showed how interacting with classic computations allows an LLM to improve generalization to OOD problem instances, problem sizes, and data types. The interactions can be thought of as increasing the level of abstraction the LLM works at, i.e., of making the model manipulate data indirectly through Python statements instead of manipulating it directly in-sequence. A potential benefit of solving problems interactively with an interpreter in the loop is that a model can potentially learn to recover from, and appropriately deal with, exceptions during solution generation. This is an important area for future research.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models, 2021.

Bouzenia, I., Ding, Y., Pei, K., Ray, B., and Pradel, M. Tracefixer: Execution trace-driven program repair, 2023.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=YfZ4ZPt8zd.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms, 2023.

Gandhi, K., Lee, D., Grand, G., Liu, M., Cheng, W., Sharma, A., and Goodman, N. D. Stream of search (sos): Learning to search in language. *arXiv preprint arXiv:2404.03683*, 2024.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines, 2014.

Hao, S., Liu, T., Wang, Z., and Hu, Z. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 45870–45894. Curran Associates, Inc., 2023.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Huang, Q., Vora, J., Liang, P., and Leskovec, J. Mlagentbench: Evaluating language agents on machine learning experimentation, 2024.

Kaiser, L. and Sutskever, I. Neural gpus learn algorithms, 2016.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners, 2023.

Lehnert, L., Sukhbaatar, S., Su, D., Zheng, Q., Mcvay, P., Rabbat, M., and Tian, Y. Beyond a*: Better planning with transformers via search dynamics bootstrapping, 2024.

Li, Y., Gimeno, F., Kohli, P., and Vinyals, O. Strong generalization and efficiency in neural programs, 2020.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization, 2019.

Ni, A., Allamanis, M., Cohan, A., Deng, Y., Shi, K., Sutton, C., and Yin, P. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*, 2024.

Nye, M., Andreassen, A., Gur-Ari, G., Michalewski, H. W., Austin, J., Bieber, D., Dohan, D. M., Lewkowycz, A., Bosma, M. P., Luan, D., Sutton, C., and Odena, A. Show your work: Scratchpads for intermediate computation with language models, 2021. https://arxiv.org/abs/2112.00114.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Recchia, G. Teaching autoregressive language models complex tasks by demonstration, 2021.

Reed, S. and de Freitas, N. Neural programmer-interpreters, 2016.

Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551. Curran Associates, Inc., 2023.

Schuurmans, D. Memory augmented large language models are computationally universal, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R.,

Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Yang, J., Prabhakar, A., Narasimhan, K., and Yao, S. Intercode: Standardizing and benchmarking interactive coding with execution feedback, 2023.

Zaremba, W. and Sutskever, I. Learning to execute, 2015.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., and Chi, E. Least-to-most prompting enables complex reasoning in large language models, 2023.

Table 3: Transforming a program into an interactive session by tracing its execution through control flow.

Static program	Interactive session

`for v in range(1):`
`  w = v + 1`
	`>>> forloop0 = iter(range(1))`
	`>>> v = next(forloop0)`
	`>>> w = v + 1`
	`>>> v = next(forloop0)`
	`StopIteration`

`if c:  # c == False`
`  b = 1`
`else:`
`  b = 0`
	`>>> c`
	`False`
	`>>> b = 0`

`while c:  # c == True`
`  c = False`
	`>>> c`
	`True`
	`>>> c = False`
	`>>> c`
	`False`

`while c:  # c == True`
`  break`
`a = 1`
	`>>> c`
	`True`
	`>>> a = 1`

`return r  # r == (1, 2, 3)`	`>>> r`
	`(1, 2, 3)`
	`>>> exit()`

A. Tracing rules

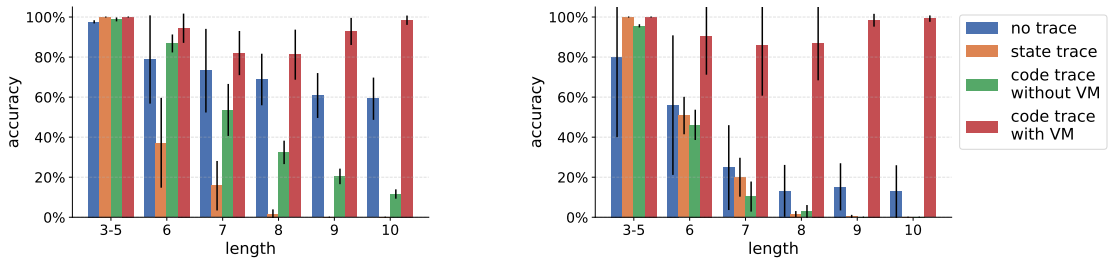
Control flow statements affect the order in which the code is run. More precisely, we identify a line as a control flow statement if and only if its compiled byte code contains an instruction that modifies the byte code pointer, for example like `POP_JUMP_IF_FALSE` or `JUMP_ABSOLUTE`. We turn the code that is part of the control flow statement into interactions with an interpreter through a set of transformation rules.

For-loop. In the for-loop we introduce a new variable `forloop0` that holds the sequence over which the loop iterates. We add a number as suffix to account for nested for-loops and increment the number for the inner ones. At the start of each iteration we explicitly assign the next item in the sequence to the loop variable. We repeat this until no more items remain and the `StopIteration` exception is thrown, which is added to the code trace as a response from the interpreter.

If-else. For conditional statements, we pass the condition as an expression to the interpreter. The interpreter evaluates the expression and prints the result in the next line. Different from how the Python interpreter commonly works, we do not explicitly cast the condition to a boolean. Note that Python can take non-boolean values as conditions which are then evaluated based on the truthiness rules. Consider for example the case where the variable `c` references the list `['a', 'b']`. Here, `c` is a valid condition that evaluates to `True`. Instead of adding the boolean result to the next line, we add the original list. The body of the conditional only appears in the interactive session if it is actually run.

While-loop. Similar to if-else statements, we expose the condition as an expression in the interactive session. At the beginning of each iteration in the loop we evaluate the condition and repeat the body of the loop as long as the condition is true.

Return. We add the return value as an expression to the code trace and conclude the interactive session with a call to the `exit` function.



(a) Random integers.

(b) Random integers with a shared prefix (hard).

Figure 5: Size generalization. Bubble sort trained on lists of integers of length 3-5 and tested on larger ones. Code traces enable size generalization.

B. More length generalization results

C. Related work

A wide range of neural architectures have been proposed that mimic aspects of a classic computer to perform step-by-step program execution. These include the Neural Turing Machine (Graves et al., 2014), Pointer Networks (Vinyals et al., 2015), Neural GPUs (Kaiser & Sutskever, 2016), and the Neural Program Interpreter (Reed & de Freitas, 2016) and related work (e.g., (Li et al., 2020)). In our work, we restrict our attention to pre-trained LLMs, and instead focus on data generation as the means to instill program execution abilities in these models. More importantly, we do not consider a specialized instruction set and instead train models to *interact* with a Python interpreter.

Letting an LLM interact with Python to solve reasoning tasks has recently been proposed by (Gao et al., 2023), as well as (Chen et al., 2023). Instead of generating a solution directly, a model is trained or prompted to generate Python code, whose output serves as the solution. In light of these, and in contrast to that work, our work can also be viewed as generating solutions as code, while however by-passing the code generation stage, and instead directly generating the step-by-step execution of a program, that is never explicitly generated itself.

There has been some preliminary recent work on letting a language model learn task-specific, back-tracking based search algorithms, such as A* or DFS (Lehnert et al., 2024; Gandhi et al., 2024). However, that work is confined to domain-specific encodings of task-specific solution paths. This makes them comparable to the special case of *state traces* discussed above. Nye et al. (2021) discuss a more general setting, which includes general Python programs and their traces, albeit without interactivity, making that work comparable to our *without-VM* setting. A non-interactive setting similar to our *no trace* setting has been discussed in a variety of past work as well (for example, Zaremba & Sutskever (2015) and follow-up work).

The Intercode benchmark (Yang et al., 2023) provides an environment for evaluating interactive code generation. While the Python portion, based on MBPP (Austin et al., 2021), bears some similarity to our work, it evaluates iterative code improvements, not interaction with Python for the sake of generating a solution to a given problem. Another benchmark related to interactive code generation, albeit restricted to the context of machine learning research workflows, is the work by Huang et al. (2024).

The use of external APIs is another emerging research theme towards LLM interactions with external resources (see, for example, (Schick et al., 2023; Hao et al., 2023)), which, in contrast to our work, is significantly more restrictive than an open ended, multi-hop interaction with Python. An even more rudimentary way to interact with an external resource is the use of an external, addressable memory (see, for example, Recchia (2021)), and it can be shown that this is sufficient for common LLMs to be Turing complete (Schuurmans, 2023). An interactive Python session offers a wide range of tools and data structures to support external memory use with more flexibility than a simple random access memory. The benefits of this additional flexibility in the context of external memory use by an LLM is beyond the scope of this paper but a potential avenue for future research.

Excerpts from execution traces have recently been used to aid models in code repair (Ni et al., 2024; Bouzenia et al., 2023). This is in contrast to our work, which proposes using full execution traces themselves for training interactive problem solving skills using supervised learning.

D. Detailed experimental setup

We fine-tune Llama2-7B (Touvron et al., 2023) in all our experiments with QLoRA (Hu et al., 2021; Dettmers et al., 2023). We use a rank parameter of 16, a scaling parameter of 64, and a dropout rate of 0.1. We use the AdamW optimiser (Kingma & Ba, 2017; Loshchilov & Hutter, 2019), with a linear decay schedule with 10 warmup steps. Since our training datasets are synthetically generated, we carry out 250 (unless otherwise specified) parameter updates regardless of dataset size. Once training is concluded, we evaluate the model by feeding it prompts from one or more evaluation datasets, and autoregressively generating from them the associated traces, which, as outlined in the previous section, also include the final result. The metrics we report are always averaged, for each run, over all evaluation prompts, and we report mean and variance over 5 different random seeds. In most experiments, we report the 1-0 accuracy (1 if perfectly correct, 0 otherwise) of the final result as our main metric, though we also consider next-token-prediction accuracy in one experiment. During evaluation, the model can either have access to the Python VM, or not. In the first case, the output of the Python statements generated by the model is environment-forced during generation of subsequent tokens, whilst in the second case, this is not the case.

Each run for Bubble Sort is trained on a V100 on 16000 training examples for less than 5h per run for a single epoch. For A* search, we train on 1000 mazes of 6x6, and associated traces; each run takes about 24 hours on a single V100 GPU.

E. Datasets

All our datasets are synthetically generated, and unless otherwise specified, each example in them consists of a string representation of the algorithm inputs (either a list or a maze), the three different traces associated with it (no-trace i.e. final result only, code-trace, and state-trace), and metadata. During generation, we use a deduplication filter to ensure there be no leakage between training and evaluation splits. We store both inputs and traces in string format.

E.1. Lists

We have four different distributions of input lists: ints, ints hard, words, words hard. The ints distribution samples numbers randomly from the range of 1 to 100.000. The ints hard distribution samples numbers with up to six digits randomly, while keeping the first one to three digits the same. The words distribution samples a list of words randomly from a vocabulary with more than 30.000 natural words. In the words hard distribution we restrict the words distribution to only sample lists of words with a shared prefix. The shared prefix length ranges from one to three.

E.2. Mazes

We use the Mazelib library[1] to generate mazes of a desired height and width. In particular, we use the Backtracking Generator algorithm, the most commonly used for maze generation. We show examples of the resulting mazes in Figure 6. The mazes are represented by Python dictionaries, with each cell being a key and a list of its neighbors being the values; every cell is represented as a tuple reporting its Cartesian coordinates, and the model ingests them via the string representation (`repr()`) of these dictionaries. For training, we use a dataset of 1000 3x3 mazes and associated A* algorithm traces. For evaluation, we generate four datasets of 20 examples each for sizes 6x6, 8x8, 10x10, and 12x12.

E.3. quizes

We add a variable with 0.3 probability to the code trace if it has changed since the last time it was quizzed. We sample this for every line in the original code trace and for every variable in the algorithm.

[1] https://github.com/john-science/mazelib

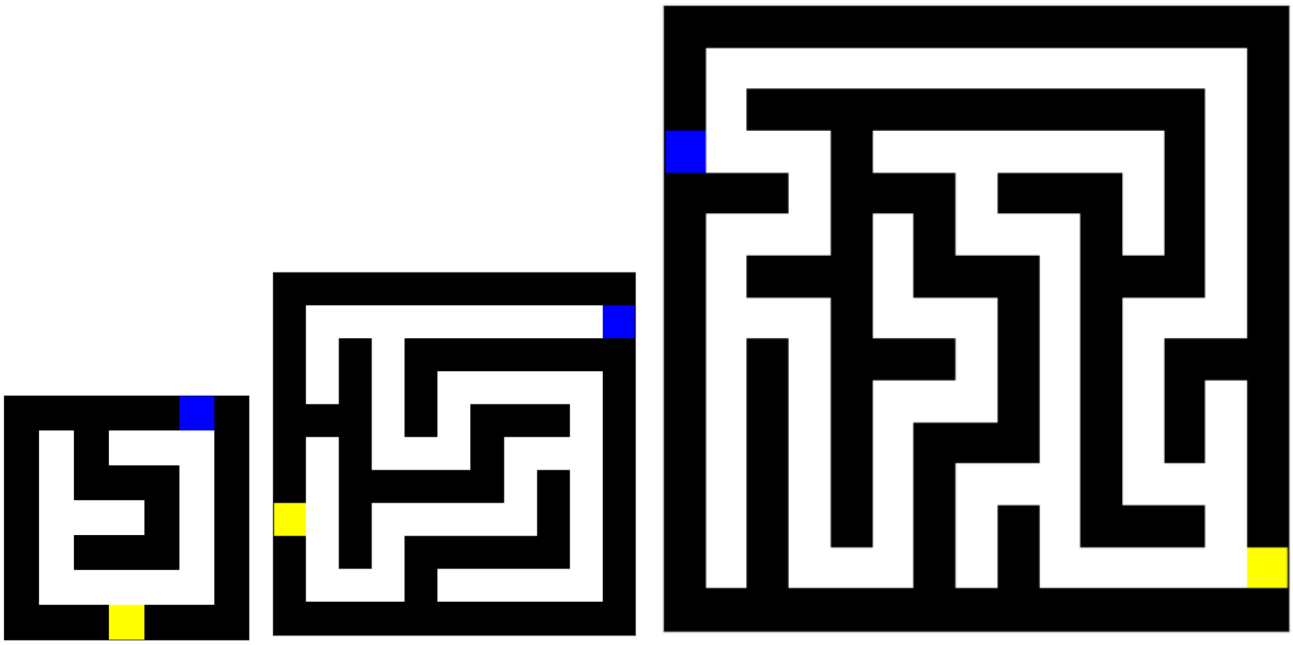


Figure 6: Three example mazes of 6x6, 10x10, and 14x14 size. The yellow and blue cells are the entrances and exits, respectively. Due to how Mazelib is implemented, these size correspond to width and height parameters of (3,3), (5,5), and (7,7).