NEXT: Teaching Large Language Models to Reason about Code Execution

Ansong Ni [1] **Miltiadis Allamanis** [2] **Arman Cohan** [1] **Yinlin Deng** [3] **Kensen Shi** [2]
Charles Sutton [2] **Pengcheng Yin** [2]

Abstract

A fundamental skill among human developers is the ability to understand and reason about program execution. As an example, a programmer can mentally simulate code execution in natural language to debug and repair code (*aka*. rubber duck debugging). However, large language models (LLMs) of code are typically trained on the surface textual form of programs, thus may lack a semantic understanding of how programs execute at run-time. To address this issue, we propose NEXT, a method to teach LLMs to inspect the execution traces of programs (variable states of executed lines) and reason about their run-time behavior through chain-of-thought (CoT) rationales. Specifically, NEXT uses self-training to bootstrap a synthetic training set of execution-aware rationales that lead to correct task solutions (*e.g.,* fixed programs) without laborious manual annotation. Experiments on program repair tasks based on MBPP and HUMANEVAL demonstrate that NEXT improves the fix rate of a PaLM 2 model, by 26.1% and 10.3% absolute, respectively, with significantly improved rationale quality as verified by automated metrics and human raters. Our model can also generalize to scenarios where program traces are absent at test-time.

1. Introduction

Recent years have witnessed the burgeoning of large language models (LLMs) trained on code (Austin et al., 2021; Chen et al., 2021a; Anil et al., 2023; Touvron et al., 2023; Li et al., 2023; Roziere et al., 2023). While those LLMs achieve impressive performance in assisting developers with writing (Chen et al., 2021a), editing (Fakhoury et al., 2023), explaining (Hu et al., 2018), and reviewing (Li et al., 2022) code, they still struggle on more complex software engineering tasks that require reasoning about the runtime execution behavior of programs (Ma et al., 2023). On the other hand, it is not always sufficient for the model to suggest good code solutions, but it is often necessary to provide an explanation to developers to document what the change does and why it is needed. These explanations can help developers better understand the code solutions from models and make more informative decisions. (Cito et al., 2022; Ross et al., 2023; Kang et al., 2023).

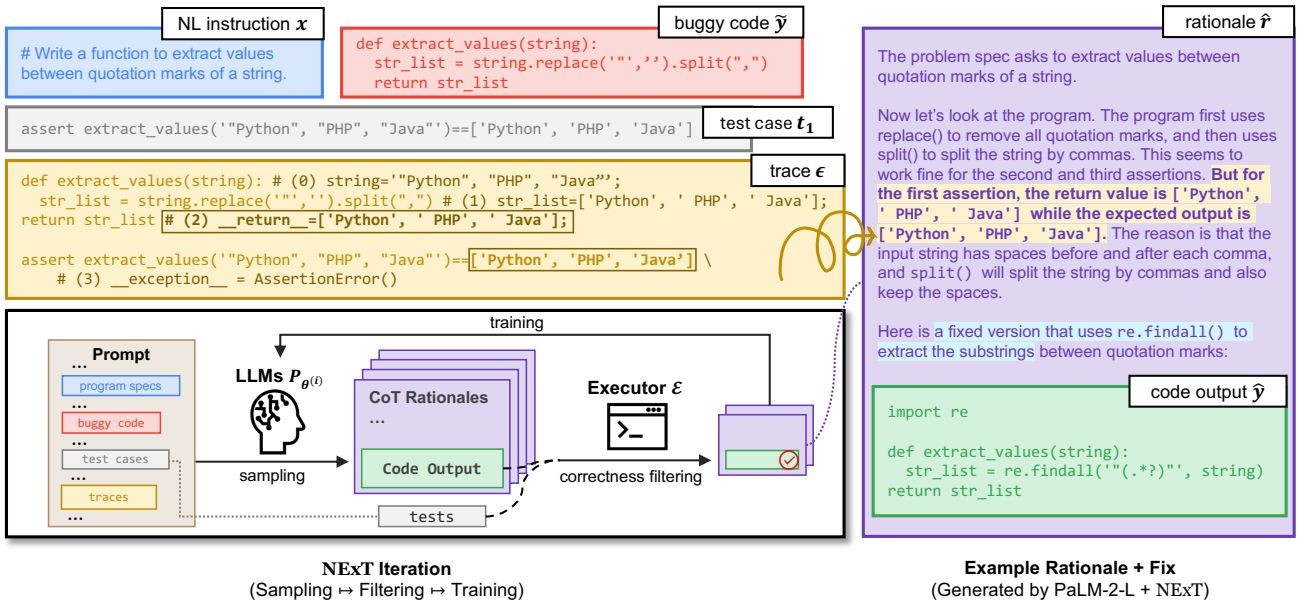
For example, *program repair* (Chen et al., 2018; Li et al., 2020; Le Goues et al., 2019) is the task of fixing bugs in a program. Human developers usually learn to debug and fix code by interacting with code interpreters or debuggers to inspect the variable states of executed lines (Siegmund et al., 2014). Such practice helps them acquire a *mental model* of program execution (Heinonen et al., 2022), so that they could mentally simulate code execution in a more abstract manner using natural language as in rubber duck debugging (Hunt & Thomas, 1999). Therefore, a program repair model would be more helpful to developers if the model could carry out similar reasoning about program execution in order to explain bugs to programmers.

With this inspiration, our goal is to improve the ability of LLMs to reason about program execution when solving coding tasks. In this paper we propose Naturalized Execution Tuning (NEXT), which aims to teach LLMs to reason with code execution by inspecting program execution traces and reasoning about the code's runtime behavior in natural language (NL). At a general level, for a coding task, the main idea is to train a model to generate intermediate NL rationales, as in chain-of-thought reasoning (Wei et al., 2022a), but to provide the model with a trace of the execution of the program in question, so the rationale can be more accurate and grounded on program semantics. Teaching LLMs to reason about program execution in NL would not only offer better interpretability, it could also increase the diversity of solutions predicted by the model (Yin et al., 2023).

Fig. 1 illustrates our proposed approach when applied to program repair. Given an NL task instruction (x in Fig. 1) and a buggy program ($\tilde{y}$), as well as the execution traces of the

*Work done during Ansong and Yinlin's internship at Google DeepMind. [1]Yale University [2]Google DeepMind [3]University of Illinois at Urbana-Champaign. Correspondence to: Ansong Ni <ansong.ni@yale.edu>, Pengcheng Yin <pcyin@google.com>.

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



NEXT Iteration
(Sampling ↦ Filtering ↦ Training)

Example Rationale + Fix
(Generated by PaLM-2-L + NEXT)

Figure 1: NEXT finetunes an LLM to *naturalize* execution traces into the chain-of-thought rationales for solving coding tasks. It performs *iterative self-training* from weak supervision, by learning from samples that lead to correct task solutions.

program (ϵ), an LLM solves the task (*e.g.,* predict the fixed code $\hat{y}$) using chain-of-thought (CoT) reasoning to generate a *natural language rationale* ($\hat{r}$) leveraging the execution information[1]. Intuitively, program traces encode useful debugging information such as line-by-line variable states (*e.g.,* the value of `str_list` in ϵ, Fig. 1) or any exceptions thrown, which could be useful for LLMs to identify and fix bugs by reasoning over the expected and the actual execution results (*e.g.,* *"highlighted text"* in $\hat{r}$). To help LLMs understand execution traces, NEXT represent traces as compact inline code comments (*e.g.,* `# (1) str_list=...` in ϵ, more in §3), without interrupting the original program structure.

While execution traces capture informative runtime behavior, we find it challenging for LLMs to effectively leverage them out-of-box through CoT prompting (§3). Therefore we opt to finetune LLMs on high-quality CoT rationales that reason about program execution (§4). NEXT uses weakly-supervised self-training (Zelikman et al., 2022) to bootstrap a synthetic training set by sampling rationales that lead to correct task solutions (*e.g.,* fixed code $\hat{y}$ in Fig. 1) verified by unit tests (Ye et al., 2022). Using unit tests as weak supervision, NEXT learns to discover task-specific, execution-aware NL rationales without relying on laborious manual annotation of rationales (Chung et al., 2022; Longpre et al., 2023; Lightman et al., 2023) or distilling such data from stronger teacher models (Gunasekar et al., 2023; Mukherjee et al., 2023; Mitra et al., 2023; Fu et al., 2023). NEXT executes this self-training loop for multiple iterations (Anthony et al., 2017; Dasigi et al., 2019), solving more challenging tasks with improved success rate and rationale quality (§5).

We evaluate NEXT with the PaLM 2-L model (Anil et al., 2023) on two Python program repair tasks. Experiments (§5) show that NEXT significantly improves PaLM 2's ability to reason about program execution in natural language, improving the program fix rate on MBPP-R by 26.1% and HUMANEVALFIX-PLUS by 10.3% absolute, respectively. When compared against a strong self-training program repair approach without predicting NL rationales (Ye et al., 2022), our model achieves comparable accuracy with significantly improved sample diversity. Interestingly, while our model learns to reason with pre-existing execution information in input program traces, it also generalizes to the out-of-distribution scenario where execution traces are unavailable at test-time. Finally, to measure the quality of model-generated rationales, we propose a *proxy-based* evaluation approach, which estimates rationale quality using the performance of smaller LLMs when prompted to solve the original task following those rationales from our models. Through both proxy-based evaluation and human annotation, we demonstrate that NEXT produces helpful NL rationales which explain the causes of bugs while suggesting potential fixes. The generated rationales are of significantly higher quality compared to those from the base PaLM 2-L model.

[1]While there are a variety types of execution information that we may provide to an LLM (*e.g.,* variable read/write, runtime environments), in this work we limit the execution information to program states and variable values from the execution trace, which is common information that (human) developers also use.

```
1  def separate_odd_and_even(lst):  # (0) lst=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2      odd_list = []  # (1) odd_list=[];
3      even_list = []  # (2) even_list=[];
4      for n in lst:  # (3) n=1; (5) n=2; (7) n=3; ...; (21) n=10;
5          if n % 2 == 1:
6              even_list.append(n)  # (4) even_list=[1]; (8) even_list=[1, 3]; ...; (20) even_list=[1, 3, 5, 7, 9];
7          else:
8              odd_list.append(n)  # (6) odd_list=[2]; (10) odd_list=[2, 4]; ...; (22) odd_list=[2, 4, 6, 8, 10];
9      return odd_list, even_list  # (23) __return__=([2, 4, 6, 8, 10], [1, 3, 5, 7, 9])
10
11 separate_odd_and_even([1,2,3,4,5,6,7,8,9,10]) == [1,3,5,7,9], [2,4,6,8,10]
```

Figure 2: NEXT represents execution trace as `inline comments`. More details are discussed in §2 and Appendix A.1.

2. Task: Program Repair with Traces

Here we introduce our task of program repair with execution traces using chain-of-thought reasoning.

Program Repair with Execution Traces. As in Fig. 1, given an instruction x and a buggy code solution $\tilde{y}$, automated program repair (Le Goues et al., 2019) aims to generate a fixed program $\hat{y}$ such that $\hat{y}$ passes all test cases $t \in T$ in an executor $\mathcal{E}$, *i.e.,* $\mathcal{E}(\hat{y}, T) = 1$ while $\mathcal{E}(\tilde{y}, T) = 0$. In this paper we focus on the task of program repair using execution traces (Bouzenia et al., 2023). Specifically, a **program trace** ϵ is a sequence of intermediate variable states after executing each statement in $\tilde{y}$ against a test case t. Intuitively, traces record the computation of a program, and can provide useful debugging information (*e.g.,* exceptions) to repair $\tilde{y}$.

To use LLMs to repair programs with traces, we concatenate the task instruction, the buggy code, the test cases, and their execution traces as a prompt (Fig. 1). To help LLMs understand program traces, we design a prompt-friendly trace representation by formatting ϵ as compact inline code comments (*i.e.,* ϵ in Fig. 1), as discussed later.

CoT Reasoning with Execution. We focus on using chain-of-thought reasoning (Wei et al., 2022b) to solve program repair problems by reasoning with execution, where an LLM is prompted to generate an NL rationale $\hat{r}$ together with a fixed program $\hat{y}$ as in Fig. 1. Specifically, we consider rationales that contain reasoning steps to identify and explain bugs in the original code (*e.g.,* the second paragraph in $\hat{r}$, Fig. 1), as well as suggestions to fix the buggy code (*e.g.,* *"a fixed version that uses* `re.findall()` *"* in $\hat{r}$). Since rationales are generated using traces, they often include useful reasoning about program execution that helps localize the bug, such as identifying a counterfactual between the expected and the actual variable values of a statement (*e.g.,* the *"highlighted text"* in $\hat{r}$). Such explanations can be helpful for developers to understand bugs in the original code and the model's fixed solutions (Kang et al., 2023). We therefore aim to improve the quality of NL rationales along with the fix rate by teaching LLMs to reason with execution information.

An LLM-friendly Trace Representation. The raw execution traces collected at runtime contain complete variable states for each executed statement.[2] Encoding all such information in prompts is not feasible given the context limit and computation overhead of LLMs. To address this issue and make execution information more intelligible to LLMs, we propose an *inline trace representation* format, which encodes variable states as inline comments of the traced program. Fig. 2 shows an example. Specifically, each inline comment only encodes changed variables after executing that line. Because statements may be invoked multiple times in non-obvious orders (*e.g.,* in loops like lines 4 to 8 in Fig. 2), we index the variable states based on the execution order (*e.g.,* `(3) n=1;` and `(4) even_list=[1]`), and one may reconstruct the original execution footprint by following those variable states in order. We further compress the trace information for loops by omitting the variable states in intermediate iterations (*e.g.,* "..." in lines 4, 6, and 8). Intuitively, by showing states as pseudo-comments within the original code without interrupting the program structure, our trace representation is significantly more compact than existing approaches that unroll executed lines of code and pair them with line-by-line variable states (*c.f.*, Nye et al., 2021; Bouzenia et al., 2023),[3] while allowing an LLM to leverage its learned code representation to understand the additional execution effect of each statement. Implementation details about handling complex control structures are discussed in Appendix A.1.

3. Preliminary Study: Can LLMs reason with program traces in natural language?

Before introducing NEXT, we first conduct a preliminary study to explore whether LLMs could reason with execution traces in natural language out-of-box without additional training. Answering this question will motivate our finetuning approach to improve such reasoning skills. Specifically,

[2] We use the `sys.settrace()` hook in Python.

[3] As a comparison, 95% examples in our MBPP-R benchmark can fit into a 2K context window using our inline representation, while only 60% of them can fit into the same window using the Scratchpad trace format in Nye et al. (2021). A more detailed comparison is shown in Tab. 7.

Benchmarks	Prompting Methods	PaLM 2-L	GPT-3.5	GPT-4	Mixtral 8x7B	DeepSeek Coder 33B	StarCoder 15.5B	Avg.
MBPP-R	Vanilla w/ trace	27.5	41.8	62.6	16.1	23.9	13.3	30.9
	+ CoT	26.6	46.4	62.8	21.1	18.2	12.6	$31.3_{+0.4}$
	+ CoT; − trace	19.0	47.1	51.3	18.1	12.9	10.6	$26.5_{-4.8}$
HEFIX+	Vanilla w/ trace	59.1	70.1	88.4	32.9	57.3	29.3	56.2
	+ CoT	48.8	75.6	84.8	34.1	30.5	16.5	$48.4_{-7.8}$
	+ CoT; − trace	43.3	72.0	82.9	25.6	22.6	18.3	$44.1_{-4.3}$

Table 1: Few(3)-shot prompting repair accuracy using greedy decoding. Results worse than the settings specified in the previous row above are underlined in red.

we follow the trace representation in §2 and few-shot prompt an LLM to solve program repair tasks using CoT reasoning.

Models. We evaluate the following general-purpose models: PaLM 2 (Anil et al., 2023), GPT (OpenAI, 2023)[4], and Mixtral (Jiang et al., 2024). We also test two code-specific LLMs: StarCoder (Li et al., 2023) and DeepSeek Coder (Guo et al., 2024). Tab. 1 reports the results on two Python program repair datasets (see §5 for details).

LLMs struggle on CoT reasoning with traces. We observed mixed results when comparing vanilla prompting with traces without CoT (**Vanilla w/ trace** in Tab. 1) and CoT prompting with rationales (**+CoT**). Surprisingly, CoT prompting is even worse on HUMANEVALFIX-PLUS, with an average drop of -7.8% compared to vanilla prompting, especially for code-specific LLMs ($57.3 \mapsto 30.5$ for DeepSeek Coder and $29.3 \mapsto 16.5$ for StarCoder). After inspecting sampled rationales predicted by PaLM 2-L, we observe that the model is subject to strong hallucination issues, such as mentioning exceptions not reflected in the given traces. Indeed, as we later show in §5.2, the overall correctness rate of explaining errors in input programs among these sampled rationales from PaLM 2-L is only around 30%. Moreover, CoT reasoning is even more challenging for those models when we remove execution traces from the inputs (**+CoT;−trace**), resulting in an average performance drop of 4.8% on MBPP-R and 4.3% on HUMANEVALFIX-PLUS. These results suggest that while our trace representation is useful for LLMs to understand and leverage execution information for program repair (since "−trace" leads to worse results), they could still fall short on CoT reasoning using natural language with those program traces. This finding therefore motivates us to improve LLMs in reasoning with execution through finetuning, which we elaborate in §4.

4. NEXT: Naturalized Execution Tuning

We present NEXT, a self-training method to finetune LLMs to reason with program execution using synthetic rationales.

Overview of NEXT. Fig. 1 illustrates NEXT, with its algorithm detailed in Algo. 1. NEXT is based on existing self-trained reasoning approaches (Zelikman et al., 2022; Uesato et al., 2022), which employ expert iteration to improve a base LLM using synthetic rationales sampled from the model. Given a training set $\mathcal{D}$ of repair tasks with execution traces, NEXT first samples candidate NL rationales and fixed code solutions from the LLM. Those candidate solutions are filtered using unit test execution diagnostics, and those that pass all test cases are then used to update the model via finetuning. This sample-filter-train loop is performed for multiple iterations, improving the model's rationales and repair success rate after each iteration.

Sampling rationales and code solutions. For each iteration i, we sample rationales $\hat{r}$ and fixes $\hat{y}$ in tandem from the current model $P_{\theta^{(i)}}$ (Line 5, Algo. 1). We use few-shot prompting (§3) when $i = 0$ and zero-shot prompting with trained models for later iterations. In contrast to existing self-training methods that leverage all training problems, NEXT only samples candidate solutions from the subset of problems in $\mathcal{D}$ that are challenging for the base model $P_{\theta^{(0)}}$ to solve (Line 1). Specifically, given a metric $\mathcal{M}(\cdot)$, we only use problems $d \in \mathcal{D}$ if $P_{\theta^{(0)}}$'s metric on d is below a threshold m. Refer to §5 for more details about the $\mathcal{M}(\cdot)$ and m of our program repair task. Focusing on sampling solutions from those hard problems not only significantly reduces sampling cost, it also improves program repair accuracy, as it helps the model towards learning to solve more challenging problems (Kommrusch et al., 2023). See Appendix C for a more detailed analysis.

Filtering candidate solutions. Given a candidate set of sampled NL rationales and their code fixes, NEXT uses unit test execution results to identify plausible rationales that lead to correct fixes for learning (Line 6). Using test execution diagnostics as a binary reward function is natural for program repair tasks since each repair problem in our dataset comes with unit tests to test the functional correctness of its proposed fixes (Ye et al., 2022). While we remark that this filtering criteria does not directly consider rationale quality, we empirically demonstrate in §5 that the quality of rationales improves as learning continues.[5]

[4] We use `gpt-3.5-turbo-1106` and `gpt-4-1106-preview`.

[5] Note that the rationale and fix quality may plateau at a different

Algorithm 1 Naturalized Execution Tuning (NEXT)

Input: Training set $\mathcal{D} = \{(x_j, \tilde{y}_j, T_j, \epsilon_j)\}_{j=1}^{|\mathcal{D}|}$ (§2); Development set $\mathcal{D}_{dev}$; Base LLM $P_{\theta^{(0)}}$; Number of iterations I; Executor $\mathcal{E}$; Evaluation metric $\mathcal{M}$ and threshold m

1: $\mathcal{D}_H \leftarrow \{d \mid d \in \mathcal{D}, \mathcal{M}(P_{\theta^{(0)}}, d) < m\}$ *// Identify hard problems $\mathcal{D}_H$ with metric $\mathcal{M}(\cdot) < m$*
2: **for** $i = 0$ **to** I **do**
3: $\quad \mathcal{B}^{(i)} \leftarrow \{\}$
4: $\quad$ **for** $(x_j, \tilde{y}_j, T_j, \epsilon_j)$ **in** $\mathcal{D}_H$ **do**
5: $\quad\quad S_j^{(i)} \sim P_{\theta^{(i)}}(r, y \mid x_j, \tilde{y}_j, T_j, \epsilon_j)$ *// Sample rationales r and fixes y using trace ϵ_j.*
6: $\quad\quad \mathcal{B}^{(i)} \leftarrow \mathcal{B}^{(i)} \cup \{(\hat{r}, \hat{y}) \mid (\hat{r}, \hat{y}) \in S_j^{(i)}, \mathcal{E}(\hat{y}, T_j) = 1\}$ *// Filter with test cases T_j and add to $\mathcal{B}^{(i)}$.*
7: $\quad \theta^{(i+1)} \leftarrow \arg\max_\theta \mathbb{E}_{\mathcal{B}^{(i)}}[P_\theta(\hat{r}, \hat{y} \mid x, \tilde{y}, T, \epsilon)]$ *// Finetune model $P_{\theta^{(0)}}$ with data in $\mathcal{B}^{(i)}$.*
8: $i^* \leftarrow \arg\max_i \sum_{d \sim \mathcal{D}_{dev}} \mathcal{M}(P_{\theta^{(i)}}, d)/|\mathcal{D}_{dev}|$ *// Select the best checkpoint i^**

Output: model $P_{\theta^{(i^*)}}$

Model training. After collecting a set of training examples $\mathcal{B}^{(i)}$, we finetune the model to maximize the probability of generating the target rationales and code fixes given the task input (Line 7). Following Zelikman et al. (2022), we always finetune the model from its initial checkpoint $P_{\theta^{(0)}}$ to avoid over-fitting to instances sampled from early iterations that are potentially of lower-quality.

Discussion. NEXT can be seen as an instantiation of the rationale bootstrapping method proposed in Zelikman et al. (2022) (§ 3.1), which synthesizes latent rationales with correct answers for math and logical reasoning tasks. However, NEXT focuses on program comprehension by reasoning with execution traces, which is critical for solving challenging coding tasks that require understanding execution information, such as program repair (§5). Besides, NEXT models both rationales and programs (code fixes) as latent variables. Using unit test execution results as weak supervision, NEXT is able to explore possible strategies to reason with execution and discover plausible rationales catered towards solving the specific downstream task. As we show in Appendix D, rationales generated by NEXT employ a variety of reasoning patterns to locate and explain bugs in our repair dataset. Finally, while we apply NEXT to program repair, our framework is general and can be extended to other programming tasks that require reasoning about execution, such as code generation with partial execution contexts (Yin et al., 2023) or inferring program execution results (Nye et al., 2021), which we leave as important future work.

5. Experiments

Models. We evaluate NEXT using PaLM 2-L (Unicorn) as the base LLM (Anil et al., 2023). Its finetuning API is publicly accessible on Google Cloud Vertex AI platform.

Datasets. We use two Python program repair benchmarks, **MBPP-R** and **HUMANEVALFIX-PLUS** (**HEFIX+** hereafter). MBPP-R is a new repair benchmark that we create from MBPP (Austin et al., 2021), a popular function-level Python code generation dataset. We create MBPP-R by collecting LLM-generated incorrect code solutions to MBPP problems, with a total of $10,047$ repair tasks for training and $1,468$ tasks (from a disjoint set of MBPP problems) in the development for evaluation (Appendix B.1). In addition to MBPP-R, we also evaluate on HEFIX+. HEFIX+ is derived from HUMANEVALFIX (Muennighoff et al., 2023) which consists of 164 buggy programs for problems in the HUMANEVAL dataset (Chen et al., 2021a). We further augment HUMANEVALFIX with the more rigorous test suites from EvalPlus (Liu et al., 2023) to obtain HEFIX+. While both original datasets MBPP and HUMANEVAL feature function-level algorithmic code generation problems, problems from the two datasets may still differ in their topics, algorithms or data structures used. Therefore, we use HEFIX+ to measure generalization ability without further finetuning.

Evaluating Code Fixes. We use PASS@k (Kulal et al., 2019; Chen et al., 2021a), defined as the fraction of solved repair tasks using k samples ($k \leq 25$), to measure the end-to-end functional correctness of fixed programs with tests.

Evaluating Rationale Quality. Decoupling the quality of intermediate CoT rationales and downstream task performance (program repair PASS@k) is a non-trivial research question in LLM reasoning (Prasad et al., 2023), with most works on improving CoT reasoning still hill-climbing towards downstream task performance without evaluating intermediate rational quality (*e.g.,* Lightman et al. (2023)). To disentangle the evaluation of rationale quality from end-to-end repair accuracy, we propose an extrinsic **proxy-based evaluation** metric for rationales. Specifically, given a rationale r, we prompt a smaller LLM to solve the original repair task conditioning on r, and use the correctness of the predicted code fix (using greedy decoding) to approximate the quality of r. Intuitively, smaller LLMs would rely more on information from the rationale and could be more sensitive to its errors. Therefore, their performance could be a better indicator of rationale quality. We report averaged scores on two PaLM 2 variants for proxy-based evaluation: 1) a smaller general-purpose language model

iteration i.

Models	End-to-end Fix Rate				Proxy-based Evaluation (PASS@k on smaller LMs)			
	PASS@1	PASS@5	PASS@10	PASS@25	PASS@1	PASS@5	PASS@10	PASS@25
GPT-4; 3-shot	63.2	75.1	78.5	82.7	44.8	66.5	72.5	77.8
GPT-3.5; 3-shot	42.9	65.0	70.7	76.7	26.6	48.8	57.0	66.4
PaLM 2-L; 3-shot	23.2	45.7	54.7	65.0	22.5	43.4	51.9	61.5
PaLM 2-L+NEXT; 0-shot	$49.3_{+26.1}$	$68.1_{+22.4}$	$73.5_{+18.8}$	$79.4_{+14.4}$	$28.8_{+6.3}$	$49.9_{+6.5}$	$57.3_{+5.4}$	$65.5_{+4.0}$

Table 2: Improvements by NEXT on the PaLM 2-L model (in subscripts) on MBPP-R. GPT-3.5/4 results are for reference.

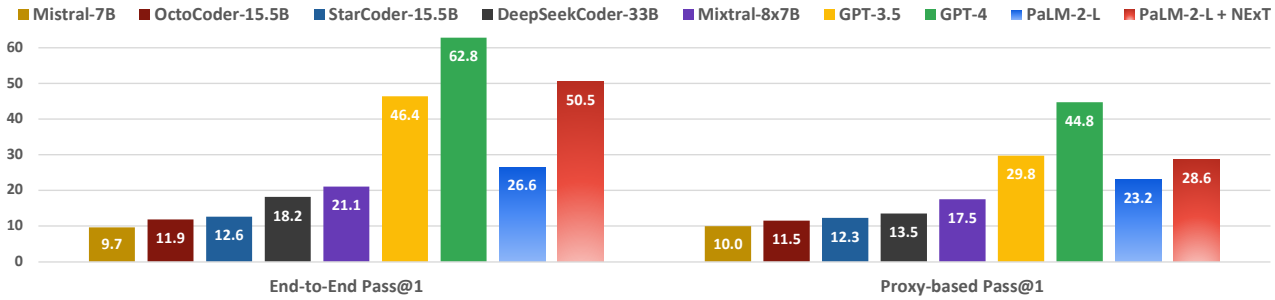


Figure 3: Greedy-decoding results on MBPP-R on PaLM 2-L+NEXT and existing LLMs.

PaLM 2-S; and 2) PaLM 2-S* which is specialized in coding (Anil et al., 2023). Note that while we primarily use proxy-based metrics to evaluate rationales, we also perform human ratings of rationale quality (§5.2), with results in line with our proxy-based evaluation.

Hyperparameters. We perform temperature sampling ($T = 0.8$) with a sample size of 32 for training ($|S_j| = 32$ in Algo. 1) and PASS@k evaluation. In the first iteration in Algo. 1, we use PASS@1 estimated with these 32 samples as the filtering metric $\mathcal{M}(\cdot)$ to find challenging problems whose $\mathcal{M}(\cdot) \leq 10\%$ for training. We perform 10 iterations of NEXT training and pick the best model using PASS@1 on the development set.

5.1. Main Results

In our experiments, we compare our model with strong LLMs (used in §3), analyze the impact of rationales and program traces, and perform generalization experiments on HEFIX+ and human evaluation of rationale quality.

NEXT improves program fix rate. We first compare the end-to-end program repair performance of PaLM 2-L before and after NEXT training (**PaLM 2-L+NExT**) in Tab. 2 (*Left*). NEXT leads to significant improvements on the end-to-end fix rates across the board, with a 26.1% absolute improvement on PASS@1. Interestingly, the gain on PASS@k is generally higher for smaller k. This might suggest that the model becomes more confident about program fixes after NEXT training, while the sample diversity also improves, as indicated by improved PASS@25. For reference, we also include results from GPT models. Notably, PaLM 2-L+NEXT outperforms GPT-3.5 on all PASS@k metrics.

NEXT improves rationale quality. Tab. 2 (*Right*) shows the improvements of PaLM 2-L+NEXT on our proxy-based evaluation, where we approximate rationale quality using the performance of smaller LMs when conditioned on those rationales. Again, NEXT yields consistent improvements across all PASS@k metrics. This suggests that NEXT improves PaLM 2-L's skill in reasoning with execution to solve MBPP-R problems, leading to rationales that are more helpful for smaller LMs. In Appendix D, we present a case study to demonstrate different reasoning strategies PaLM 2-L+NEXT adopts to repair programs using execution information. As we later show in §5.2, our proxy-based metrics are also consistent with human ratings, and rationales from PaLM 2-L+NEXT are strongly preferred by annotators compared to those from PaLM 2-L.

PaLM 2-L+NExT outperforms strong LLMs. We compare PaLM 2-L+NEXT with a series of strong LLMs from the preliminary study (§3) in Fig. 3. PaLM 2-L+NEXT outperforms strong open-source LLMs by a minimum of 29.4% and 11.1% on end-to-end and proxy-based PASS@1 results, respectively, while on par with GPT-3.5. These results show that PaLM 2-L+NEXT is a competitive model on program repair by reasoning with execution.

Learning to reason in natural language improves generalization and sample diversity. To further demonstrate the importance of using CoT reasoning in NEXT self-training, we compare PaLM 2-L+NEXT with a strong self-training-based program repair model implemented in NEXT, which directly generates code fixes using runtime execution information without CoT reasoning. This ablation resembles SelfAPR (Ye et al., 2022), which also adopts self-training to iteratively synthesize data using unit test diagnostics, while

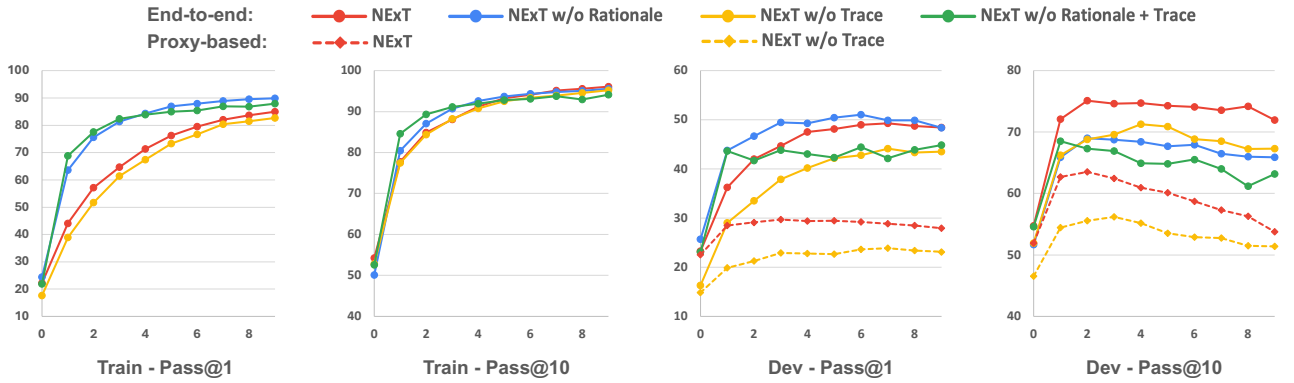


Figure 4: Ablations on removing rationales and/or traces during the iterative training of NEXT. Note that different min/max values are taken for y-axis for clarify among different curves but consistent gridline intervals are used for easier comparison.

our ablation uses traces with richer execution information. Fig. 4 shows model performance w.r.t. NEXT training iterations. When trained without CoT reasoning (**NEXT w/o rationale**), PaLM 2-L converges much faster on the training set, which is not surprising since the model only learns to generate code fixes without additional reasoning tasks such as explaining bugs in NL. However, on the DEV set, PaLM 2-L+NEXT still outperforms this baseline in PASS@10 with comparable PASS@1 accuracy, and the gap on PASS@10 becomes larger with more iterations. This shows that by reasoning in natural language, PaLM 2-L+NEXT generalizes much better to unseen MBPP-R problems with greater sample diversity. In Fig. 6 of Appendix C, we also show that the gain from PaLM 2-L+NEXT against this ablation on PASS@k is even more pronounced for larger $k > 10$, which suggests that learning to reason in CoT rationales improves sample diversity on program repair, similar to the findings on other code generation tasks (Yin et al., 2023).

Reasoning with execution traces is critical. To understand the importance of leveraging program traces to reason with execution, we compare with an ablation of NEXT *without* using program traces, which follows the same procedure in Algo. 1 except that traces ϵ are not used to generate rationales in Line 5 (**NEXT w/o traces**, Fig. 4). This variant can also be seen as a direct application of the rationale generation bootstrapping method in Zelikman et al. (2022), which trains a model on sampled rationales that lead to correct task solutions without relying on additional execution information. Without traces, PaLM 2-L is consistently worse than PaLM 2-L+NEXT on the DEV set across iterations, both in terms of end-to-end fix rate and proxy-based metrics. This suggests that reasoning with execution information is critical for PaLM 2-L on program repair tasks. Interestingly, while the gap on the development set is significant, the two models achieve similar scores on the training set, which suggests that reasoning with pre-existing execution traces also help the model generalize better to unseen tasks at test-time.

	Test w/ Trace		Test w/o Trace	
Methods	E2E	Proxy	E2E	Proxy
PaLM 2-L	23.2	22.5	19.0	14.8
+NEXT (w/ trace)	$49.3_{+26.1}$	$28.8_{+6.3}$	$40.8_{+21.8}$	$19.5_{+4.7}$
+NEXT w/o trace	–	–	$44.1_{+25.1}$	$23.9_{+9.1}$

Table 3: PaLM 2-L+NEXT trained with traces outperforms PaLM 2-L when traces are absent at test time as shown in highlighted results. Results are on MBPP-R; **Test w/ Trace:** results from Tab. 2.

Our model works without traces at test-time. While program traces are crucial for reasoning with execution, such execution information may not always be available at test time (*e.g.,* when execution is prohibitively expensive). To stress-test PaLM 2-L+NEXT in scenarios where execution information is absent, we remove execution traces from its input at test time in Tab. 3. PaLM 2-L+NEXT still yields an end-to-end fix rate of 40.8%, which is an 21.8% improvement over the 3-shot PaLM 2-L baseline and is only 3.3% lower than NEXT trained without traces, for which is tested in-distribution. The results from the proxy-based evaluation of rationales are also consistent with the fix rate.

Our model generalizes to HEFIX+ at test-time. To further evaluate the generalization ability of PaLM 2-L+NEXT, we test our model (trained on MBPP-R) on HEFIX+. Tab. 4 summarizes the results. NEXT achieves reasonable generalization on HEFIX+, outperforming the base PaLM 2-L model by a large margin (*i.e.,* 14.3% on end-to-end fix rate and 6.0% on proxy evaluation). Aligned with our previous findings on MBPP-R in Fig. 4, reasoning with execution traces (*c.f.* w/o traces) improves fix rate and rationale quality. Moreover, we remark that with iterative learning, PaLM 2-L+NEXT is on par with the strong program repair method without CoT reasoning (w/o rationale), similar to the results on MBPP-R. This is in contrast with our preliminary study in §3, where PaLM 2-L with CoT

Models / PASS@1	End-to-End	Proxy-based
Baselines w/ 3-shot prompting		
Mistral-7B*	12.8	16.5
OctoCoder-15.5B*	17.7	17.7
StarCoder-15.5B*	14.6	13.1
DeepSeekCoder-33B*	28.0	18.3
Mixtral-8x7B*	32.3	30.8
GPT-4	77.6	56.6
GPT-3.5	59.4	41.8
PaLM-2-L	32.2	31.9
PaLM-2-L w/o tracing†	30.3	30.4
PaLM 2-L+NEXT	$42.5_{+10.3}$	$38.0_{+6.1}$
w/o tracing†	$38.1_{+7.8}$	$30.6_{+0.2}$
w/o rationale	$44.5_{+12.3}$	–
w/o tracing + rationale†	$31.4_{+1.1}$	–

Table 4: Generalization results on HEFIX+. PaLM 2-L+NEXT models are only trained with MBPP-R. *obtained using greedy decoding; †no traces provided at test time.

prompting is much worse than vanilla prompting without using rationales. Overall, these results indicate that PaLM 2-L+NEXT could robustly generalize to out-of-distribution repair tasks without additional dataset-specific finetuning.

5.2. Human Evaluation of Rationale Quality

Our proxy-based evaluation suggests the extrinsic value of the CoT rationales from PaLM 2-L+NEXT. We further conduct an intrinsic evaluation by manually rating the quality of model-predicted rationales on 104 sampled MBPP-R repair tasks from the DEV set. Specifically, we ask raters to judge the quality of rationales generated by three models (PaLM 2-L+NEXT, PaLM 2-L and GPT-3.5) in a three-way side-by-side setting. Each rationale is rated in two aspects: (1) its helpfulness in explaining bugs (Q_1, *e.g.,* first two paragraphs in $\hat{r}$, Fig. 1), and (2) its helpfulness in suggesting code fixes (Q_2, *e.g., "a fixed version that uses ..."* in $\hat{r}$). Each question has a three-scale answer (✅ Completely correct and very helpful; ☑ Partially correct with minor errors but still helpful; ❌ Incorrect and not helpful). We also compute an **overall score** of rationale quality using numeric values of $\{+1, 0.5, 0\}$ for the three scales and averaged over Q_1 and Q_2. Finally, we ask raters to pick a single **best choice** if there is not a clear tie. More details about our human evaluation pipeline is described in Appendix B.3.

Tab. 5 summarizes the result. Compared to the base PaLM 2 model, PaLM 2-L+NEXT generates significantly more high-quality rationales with correct explanations of bugs and fix suggestions. Additionally, compared to GPT-3.5, PaLM 2-L+NEXT also has more rationales with correct bug explanations, while interestingly, GPT-3.5 generates more rationales with partially correct fix suggestions. We hypothesize that including more exemplars with detailed fix

	Explain bugs?			Suggest fixes?				
	✅	☑	❌	✅	☑	❌	Overall	Best?
GPT-3.5	43	26	35	44	16	44	51.9%	34.6%
PaLM 2-L	27	24	53	31	5	68	34.9%	6.7%
+NEXT	48	24	32	42	6	56	50.5%	32.7%

Table 5: Results for human annotation of rationale quality. Base models use 3-shot prompting. Numbers under the questions are counts of ratings.

suggestions to our few-shot prompts during NEXT training (Appendix E) would help mitigate this issue. Nevertheless, the overall scores and rater-assigned best choice suggest that the rationales predicted by PaLM 2-L+NEXT are of significantly higher quality compared to those from PaLM 2-L, and are on par with the predictions from GPT-3.5. Overall, this finding is in line with the proxy evaluation results in Fig. 3 (GPT 3.5 $\approx$ PaLM 2-L+NEXT $\gg$ PaLM 2-L), suggesting that the latter is a reasonable surrogate metric for rationale quality. In Appendix D, we present example generated rationales that show a variety of reasoning patterns.

6. Related Work

Reasoning about Program Execution Several lines of research has explored learning methods to reason about program execution. Program synthesis systems often leverage the execution states of partially generated programs (Shin et al., 2018; Wang et al., 2018; Chen et al., 2021b; Shi et al., 2022) or the next execution subgoals (Shi et al., 2024) to guide search in sequence-to-sequence models. There has also been work on training neural networks to mimic program execution, like a learned interpreter (Zaremba & Sutskever, 2014; Bieber et al., 2020; Nye et al., 2021; Pi et al., 2022), often with specialized neural architectures to model the data flow of program execution (Graves et al., 2014; Gaunt et al., 2016; Bosnjak et al., 2016; Bieber et al., 2022). Instead of using domain-specific architectures to encode and reason about program execution, our work focuses on teaching LLMs to reason with execution in natural language. In particular, *Scratchpad* (Nye et al., 2021) and *Self-Debugging* (Chen et al., 2023) are two notable works that also models execution traces using LLMs. The core difference is that these methods focus on predicting reasoning chains that contain trace information, such as executed lines with variable states (Nye et al., 2021) or their natural language summaries (Chen et al., 2023). On the other hand, NEXT aims to leverage existing execution traces from a runtime to aid the reasoning process, which often leads to more compact rationales tailored for downstream tasks. We present a more detailed comparison and discussion on NEXT and these related works in Appendix A.3.

Program Repair Several works in program repair have

leveraged execution information such as traces (Gupta et al., 2020; Bouzenia et al., 2023) or test diagnostics (Xia & Zhang, 2023; Ye et al., 2022). Different from Bouzenia et al. (2023) which represents traces by directly pairing unrolled executed lines with their variable states, NEXT inlines indexed variable states as code comments, which is more token efficient while preserving the original code structure. Similar to NEXT, Ye et al. (2022) construct synthetic self-training data using test execution results, while our approach generates both NL rationales and fixed programs with better interpretability. Recently, LLMs have been applied to program repair (Fan et al., 2022; Xia & Zhang, 2022; Xia et al., 2023; Sobania et al., 2023; Paul et al., 2023; Jiang et al., 2023). Among them, Kang et al. (2023) uses a ReAct-style CoT reasoning loop (Yao et al., 2022) to predict repair actions based on interactive feedback from debuggers, while NEXT focuses on tuning LLMs to reason with pre-existing execution information without intermediate feedback. Finally, as a related stream of research, self-improvement methods iteratively refine a model's code solutions using CoT reasoning over self-provided (Madaan et al., 2023) or test-driven feedback (Chen et al., 2023; Olausson et al., 2023). Instead of relying on high-level execution signals like error messages, NEXT trains LLMs to reason with step-wise program traces. Our learnable rationales are also more flexible without following a predefined reasoning template. Besides, since traces already capture rich execution semantics, the resulting rationales could be more succinct and targeted to the downstream task (*e.g.,* explain bugs), without redundant reasoning steps to trace the program by the model itself to recover useful execution information.

Supervised CoT Reasoning LLMs can solve problems more accurately when instructed to work out the answer step by step in a *chain of thought* or a *scratchpad* (Wei et al., 2022a; Nye et al., 2021; Rajani et al., 2019; Shwartz et al., 2020). Improvements on this approach involve finetuning LLMs on chain-of-thought reasoning data. Such CoT data is either manually curated (Chung et al., 2022; Longpre et al., 2023; Lightman et al., 2023), or distilled from more capable teacher models (Gunasekar et al., 2023; Mukherjee et al., 2023; Mitra et al., 2023; Fu et al., 2023). Instead of relying on labeled or distilled data, NEXT uses self-training to iteratively bootstrap a synthetic dataset of high-quality rationales with minimal manual annotation. Our work differs from previous work using bootstrapping (Zelikman et al., 2022; Hoffman et al., 2023) in the type of rationales and the use of execution information; see §4 for more discussion. While we use the correctness of the program fix for filtering the rationales, which is reminiscent of outcome supervision; it is also possible to use process supervision with human annotations (Uesato et al., 2022; Lightman et al., 2023), or obtain such supervision automatically by estimating the quality of each step using Monte Carlo Tree Search (Wang et al., 2024) and by identifying partially-correct program prefixes (Ni et al., 2022). Finally, existing research has investigated finetuning of LLMs to predict the execution information directly, such as predicting line-by-line execution traces (Nye et al., 2021), abstract runtime properties (Pei et al., 2023), or final output (Zaremba & Sutskever, 2014; Bieber et al., 2020). NEXT addresses a different problem; instead of predicting the execution information, NEXT takes it as given, and instead learns to discover flexible task-specific NL rationales that aid a downstream programming task.

7. Conclusion

In this paper we present NEXT, a self-training method to finetune LLMs to reason with program execution given traces. We demonstrate that PaLM 2-L trained using NEXT yields high-quality natural language rationales and achieves stronger success rates on two program repair tasks. As future work, we plan to apply NEXT to a broader range of program understanding tasks while expanding the trace representation to support more programming languages.

Acknowledgements

We would like to express our sincere gratitude to Martín Abadi, Xinyun Chen, Hanjun Dai, Yixin Liu, Sirui Lu, Kexin Pei and members of the Learning for Code team at Google DeepMind for their invaluable feedback. We thank anonymous reviewers for their insightful comments. We are also grateful to Austin Tarango for his support to this work.

Impact Statement

Our work aims to improve the debugging ability of code LLMs. While this may help developers produce better code, the code itself might have any number of positive or negative societal and ethical implications none of which we feel must be specifically highlighted here.

References

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., et al. PaLM 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Anthony, T. W., Tian, Z., and Barber, D. Thinking fast and slow with deep learning and tree search. In *Neural Information Processing Systems (NeurIPS)*, 2017.

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.

Bieber, D., Sutton, C., Larochelle, H., and Tarlow, D. Learning to execute programs with instruction pointer attention graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, October 2020.

Bieber, D., Goel, R., Zheng, D., Larochelle, H., and Tarlow, D. Static prediction of runtime errors by learning to execute programs with external resource descriptions. *ArXiv*, abs/2203.03771, 2022.

Bosnjak, M., Rocktäschel, T., Naradowsky, J., and Riedel, S. Programming with a differentiable Forth interpreter. *ArXiv*, abs/1605.06640, 2016.

Bouzenia, I., Ding, Y., Pei, K., Ray, B., and Pradel, M. TraceFixer: Execution trace-driven program repair. *ArXiv*, abs/2304.12743, 2023.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Chen, X., Song, D. X., and Tian, Y. Latent execution for neural program synthesis beyond domain-specific languages. *ArXiv*, abs/2107.00101, 2021b.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47: 1943–1959, 2018.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Valter, D., Narang, S., Mishra, G., Yu, A. W., Zhao, V., Huang, Y., Dai, A. M., Yu, H., Petrov, S., hsin Chi, E. H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q. V., and Wei, J. Scaling instruction-finetuned language models. *ArXiv*, abs/2210.11416, 2022.

Cito, J., Dillig, I., Murali, V., and Chandra, S. Counterfactual explanations for models of code. *International Conference on Software Engineering (ICSE)*, 2022.

Dasigi, P., Gardner, M., Murty, S., Zettlemoyer, L., and Hovy, E. H. Iterative search for weakly supervised semantic parsing. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

Fakhoury, S., Chakraborty, S., Musuvathi, M., and Lahiri, S. K. Towards generating functionally correct code edits from natural language issue descriptions. *ArXiv*, abs/2304.03816, 2023.

Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., and Tan, S. H. Automated repair of programs from large language models. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1469–1481, 2022.

Fu, Y., Peng, H.-C., Ou, L., Sabharwal, A., and Khot, T. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning (ICML)*, 2023.

Gaunt, A. L., Brockschmidt, M., Kushman, N., and Tarlow, D. Differentiable programs with neural libraries. In *International Conference on Machine Learning (ICML)*, 2016.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *ArXiv*, abs/1410.5401, 2014.

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Giorno, A. D., Gopi, S., Javaheripi, M., Kauffmann, P. C., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., and Li, Y.-F. Textbooks are all you need. *ArXiv*, abs/2306.11644, 2023.

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.

Gupta, K., Christensen, P. E., Chen, X., and Song, D. X. Synthesize, execute and debug: Learning to repair for neural program synthesis. *ArXiv*, abs/2007.08095, 2020.

Heinonen, A., Lehtelä, B., Hellas, A., and Fagerholm, F. Synthesizing research on programmers' mental models of programs, tasks and concepts - a systematic literature review. *Inf. Softw. Technol.*, 164:107300, 2022.

Hoffman, M. D., Phan, D., Dohan, D., Douglas, S., Le, T. A., Parisi, A. T., Sountsov, P., Sutton, C., Vikram, S., and Saurous, R. A. Training Chain-of-Thought via Latent-Variable inference. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. Deep code comment generation. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20010, 2018.

Hunt, A. and Thomas, D. The pragmatic programmer: From journeyman to master. 1999.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Jiang, N., Liu, K., Lutellier, T., and Tan, L. Impact of code language models on automated program repair. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1430–1442, 2023.

Kang, S., Chen, B., Yoo, S., and Lou, J.-G. Explainable automated debugging via large language model-driven scientific debugging. *ArXiv*, abs/2304.02195, 2023.

Kommrusch, S., Monperrus, M., and Pouchet, L.-N. Self-supervised learning to prove equivalence between straight-line programs via rewrite rules. *IEEE Transactions on Software Engineering*, 2023.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. S. SPoC: Search-based pseudocode to code. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

Le Goues, C., Pradel, M., and Roychoudhury, A. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Li, Y., Wang, S., and Nguyen, T. N. DLFix: Context-based code transformation learning for automated program repair. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 602–614, 2020.

Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., and Sundaresan, N. Automating code review activities by large-scale pre-training. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

Liang, C., Norouzi, M., Berant, J., Le, Q. V., and Lao, N. Memory augmented policy optimization for program synthesis and semantic parsing. *Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

Longpre, S., Hou, L., Vu, T., Webson, A., Chung, H. W., Tay, Y., Zhou, D., Le, Q. V., Zoph, B., Wei, J., and Roberts, A. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning (ICML)*, 2023.

Ma, W., Liu, S., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., and Liu, Y. ChatGPT: Understanding code syntax and semantics. 2023.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Welleck, S., Majumder, B. P., Gupta, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback. *ArXiv*, abs/2303.17651, 2023.

Mitra, A., Corro, L. D., Mahajan, S., Codas, A., Simoes, C., Agrawal, S., Chen, X., Razdaibiedina, A., Jones, E., Aggarwal, K., Palangi, H., Zheng, G., Rosset, C., Khanpour, H., and Awadallah, A. Orca 2: Teaching small language models how to reason. *ArXiv*, abs/2311.11045, 2023.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., Von Werra, L., and Longpre, S. OctoPack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

Mukherjee, S., Mitra, A., Jawahar, G., Agarwal, S., Palangi, H., and Awadallah, A. H. Orca: Progressive learning from complex explanation traces of GPT-4. *ArXiv*, abs/2306.02707, 2023.

Ni, A., Yin, P., and Neubig, G. Merging weak and active supervision for semantic parsing. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 34, pp. 8536–8543, 2020.

Ni, A., Inala, J. P., Wang, C., Polozov, A., Meek, C., Radev, D., and Gao, J. Learning math reasoning from self-sampled correct and partially-correct solutions. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2022.

Ni, A., Iyer, S., Radev, D., Stoyanov, V., Yih, W.-t., Wang, S., and Lin, X. V. LEVER: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning (ICML)*, pp. 26106–26128. PMLR, 2023.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., and Solar-Lezama, A. Demystifying GPT self-repair for code generation. *arXiv preprint arXiv:2306.09896*, 2023.

OpenAI. GPT-4 technical report, 2023.

Paul, R., Hossain, M. M., Siddiq, M. L., Hasan, M., Iqbal, A., and Santos, J. C. S. Enhancing automated program repair through fine-tuning and prompt engineering. 2023.

Pei, K., Bieber, D., Shi, K., Sutton, C., and Yin, P. Can large language models reason about program invariants? In *International Conference on Machine Learning (ICML)*, 2023.

Pi, X., Liu, Q., Chen, B., Ziyadi, M., Lin, Z., Fu, Q., Gao, Y., Lou, J.-G., and Chen, W. Reasoning like program executors. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 761–779, 2022.

Prasad, A., Saha, S., Zhou, X., and Bansal, M. Receval: Evaluating reasoning chains via correctness and informativeness. *arXiv preprint arXiv:2304.10703*, 2023.

Rajani, N. F., McCann, B., Xiong, C., and Socher, R. Explain yourself! leveraging language models for commonsense reasoning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 4932–4942, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1487.

Ross, S. I., Martinez, F., Houde, S., Muller, M. J., and Weisz, J. D. The programmer's assistant: Conversational interaction with a large language model for software development. *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Shi, K., Dai, H., Ellis, K., and Sutton, C. CrossBeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations (ICLR)*, 2022.

Shi, K., Hong, J., Deng, Y., Yin, P., Zaheer, M., and Sutton, C. ExeDec: Execution decomposition for compositional generalization in neural program synthesis. In *International Conference on Learning Representations (ICLR)*, 2024.

Shin, E. C., Polosukhin, I., and Song, D. Improving neural program synthesis with inferred execution traces. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31. Curran Associates, Inc., 2018.

Shwartz, V., West, P., Le Bras, R., Bhagavatula, C., and Choi, Y. Unsupervised commonsense question answering with self-talk. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4615–4629, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.373.

Siegmund, B., Perscheid, M., Taeumel, M., and Hirschfeld, R. Studying the advancement in debugging practice of professional software developers. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pp. 269–274, 2014. doi: 10.1109/ISSREW.2014.36.

Sobania, D., Briesch, M., Hanna, C., and Petke, J. An analysis of the automatic bug fixing performance of ChatGPT. *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pp. 23–30, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Uesato, J., Kushman, N., Kumar, R., Song, F., Siegel, N., Wang, L., Creswell, A., Irving, G., and Higgins, I. Solving math word problems with process- and outcome-based feedback. *ArXiv*, abs/2211.14275, 2022.

Wang, C., Tatwawadi, K., Brockschmidt, M., Huang, P.-S., Mao, Y., Polozov, O., and Singh, R. Robust text-to-SQL generation with execution-guided decoding. *arXiv: Computation and Language*, 2018.

Wang, P., Li, L., Shao, Z., Xu, R. X., Dai, D., Li, Y., Chen, D., Wu, Y., and Sui, Z. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *ArXiv*, abs/2312.08935, 2024.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. *"arXiv preprint arXiv:2201.11903"*, January 2022a.

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 (NeurIPS)*, 35: 24824–24837, 2022b.

Xia, C. and Zhang, L. Less training, more repairing please: revisiting automated program repair via zero-shot learning. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

Xia, C. and Zhang, L. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *ArXiv*, abs/2304.00385, 2023.

Xia, C., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494, 2023.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629, 2022.

Ye, H., Martinez, M., Luo, X., Zhang, T., and Monperrus, M. SelfAPR: Self-supervised program repair with test execution diagnostics. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

Yin, P., Li, W.-D., Xiao, K., Rao, A., Wen, Y., Shi, K., Howland, J., Bailey, P., Catasta, M., Michalewski, H., Polozov, O., and Sutton, C. Natural language to code generation in interactive data science notebooks. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 126–173, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.9.

Zaremba, W. and Sutskever, I. Learning to execute. *ArXiv*, abs/1410.4615, 2014.

Zelikman, E., Wu, Y., Mu, J., and Goodman, N. STaR: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems (NeurIPS)*, 35: 15476–15488, 2022.

A. Additional Details of NEXT

A.1. Details for Inline Trace Representation

Definitions. A program $y \in \mathcal{Y}$ consists of a sequence of statements $\{u_1, ..., u_m\}$. And a program state h is a mapping between identifiers (*i.e.,* variable names) to values, *i.e.,* $h \in \{k \mapsto v | k \in \mathcal{K}, v \in \mathcal{V}\}$. Given an input to the program, an execution trace is defined as a sequence of program states, *i.e.,* $\epsilon = \{h_1, ..., h_t\}$, which are the results after executing the statements with the *order of execution*, *i.e.,* $\{u_{e_1}, u_{e_2}, ..., u_{e_t}\}$. In this way, the relation between program statements and execution states can be seen as a function that maps from states to statements, *i.e.,* $h_i \mapsto u_{e_i}$, because each statement could be executed multiple times due to loops or recursion.

Program state representation. For typical programs, most of the variable values will stay the same between two adjacent states h_{i-1} and h_i. Thus to save tokens, we represent a state h_i only by the variables that have changed the value compared with the previous state h_{i-1}. And we use a reified variable state representation, *i.e.,* using the grammar for an init function in Python (*e.g.,* `lst=[1, 2, 3]`). Note that it is possible for a statement to have no effect on any traceable variables (*e.g.,* "`pass`", or "`print`", or "`lst[i]=lst[i]`"). To distinguish this case with unreached statements (*e.g.,* "`else`" branch that next got executed), we append a string "`NO_CHANGE`" instead. In addition to the variable state, we number all the states by the order of execution and prepend the ordinal number to the beginning of the state, *e.g.,* "`(1) odd_list=[]`" in Fig. 2.

Inline trace representation. To obtain the inline trace representation, we first group the program states in a trace ϵ by the corresponding program statements to collect a sequence of states for the same statement u_i as $H_i = \{h_j | u_{e_j} = u_i\}$, and we order the states in H_i by the execution order. For statements inside a loop body, or a function that is called recursively, the number of corresponding states can be very large. In order to further save tokens, if $|H_i| > 3$, we will only incorporate the first two states and the last state, and skip the ones in the middle. After that, we simply concatenate all the state representations with the semicolon ";" as the delimiter, and append it after the statement itself u_i following a hash "`#`" to note it as an inline comment. An example of the resulting representation is "`even_list.append(n) # (4) even_list=[1]; (8) even_list=[1, 3]; ...; (20) even_list=[1, 3, 5, 7, 9];`", as shown in Fig. 2.

Limitations. First of all, our tracing framework currently do not extend beyond native Python programs, thus it can not trace code that is not written in Python (*e.g.,* C code in `numpy`). One other limitation of our tracing representation is that for "`if`" conditions, though it would be better to leave traces of "`(1) True; (2) True; (3); False;`", currently our tracing framework that based on the "`sys.settrace()`" hook of Python does not capture this. However, since we labeled all the states by the execution order, the LLMs can infer the conditions by the fact that certain branch is taken. Another limitation is the representation of Collections. Currently we still present all the elements in a collection, and empirically it works well with benchmarks as MBPP-R and HEFIX+. However, certain heuristics may be needed to skip certain elements (*e.g.,* like the one we use to skip certain states in a loop) to be more token efficient. For more complex objects (*e.g.,* Tensors, DataFrames), while we can define heuristics to represent key properties of those objects in traces (*e.g.,* "a float tensor of shape 128 x 64","a Dataframe with columns Name, Math, ..."), perhaps a more interesting idea would be to let the models decide which properties they would inspect and generate relevant code (*e.g.,* "`tensor.shape`" or "`df.head(3)`") to inspect them in a debugger or interpreter (*e.g.,* pdb). The same idea can be applied to longer programs, as the model can selectively decide which lines of code to inspect and create traces for, similar to how human developers debug programs. We will leave these as exciting future directions.

A.2. Details for Iterative Self-Training

Bootstrapping rationales and fixes via temperature sampling. To avoid potential "cold start" problem (Liang et al., 2018; Ni et al., 2020), for the first iteration, we use few-shot prompting with three exemplars (shown in Appendix E) and set the sample size to 96. For all later iterations, we use zero-shot prompting as the model is already adapted to the style of the rationales and fixes after the first round of finetuning, and we set the sample size to 32. We set the sampling temperature $T = 0.8$ for all iterations.

Filtering rationales and fixes. Given the inputs in the prompt, we sample the rationale and fixes in tandem. To separate the natural language rationale and the program fix, we use an regular expression in Python to extract the content between two

Methods	Use of Trace	Rationale Format	Model Fine-tuning
NEXT	Input	Natural Language	Yes
Scratchpad (Nye et al., 2021)	Output	Scratchpad Repr.	Yes
Self-Debugging (Chen et al., 2023)	Output	Natural Language	No

Table 6: Comparison between the methods proposed in NEXT, Scratchpad, and Self-Debugging.

Trace Repr.	Length Cutoff (# Tokens)							
	128	256	512	1,024	2,048	4,096	8,192	16,384
Inline (ours)	0.1%	7.3%	37.5%	78.9%	95.1%	98.5%	99.2%	99.5%
Scratchpad	0.0%	0.2%	15.1%	38.2%	60.1%	76.1%	85.1%	92.1%

Table 7: Percentage of MBPP-R examples that can be fit into different context windows using different trace representations (*i.e.,* ours and Nye et al. (2021)). Traces of all three tests are included.

sets of three backticks (```), which is commonly used to note code blocks in markdown.[6] After we filter out the rationales and fixes that are incorrect using the test cases, we create the training set by sub-sampling correct "(rationale, fix)" pairs to allow a maximum of 3 correct fixes with their rationales for each problem in MBPP-R. This is to balance the number of rationales and fixes for each problem and avoid examples from certain examples (typically easier ones) being overly represented in the training set.

A.3. Discussion with Previous Work

Here we discuss NEXT in the context of two important previous work in the domain of reasoning about program execution, namely *Scratchpad* (Nye et al., 2021) and *Self-Debugging* (Chen et al., 2023). Such comparison is also characterized by Tab. 6.

***Scratchpad* and NEXT.** Similarly to NEXT, Nye et al. (2021) also proposed to use execution traces to help the LLMs to reason about program execution. However, Nye et al. (2021) aimed to generate these traces as intermediate reasoning steps at inference time, either via few-shot prompting or model fine-tuning. Yet in NEXT, we use execution traces as part of the input to the LLMs, so they can directly use the execution states to ground the generated natural language rationales. Moreover, we choose to use natural language as the primary format for reasoning, which is more flexible and easier to be understood by the human programmers. We also perform a length comparison of our proposed inline trace representation with the scratchpad representation proposed in Tab. 7, and results show that our proposed inline trace representation is much more compact than scratchpad.

***Self-Debugging* and NEXT.** Self-Debugging (Chen et al., 2023) is a seminal approach that also performs CoT reasoning over program execution to identify errors in code solutions. Different from NEXT, Self-Debugging can optionally leverages high-level execution error messages to bootstrap CoT reasoning, while our method trains LLMs to reason with concrete step-wise execution traces. In addition, Self-Debugging also introduced a particular form of CoT rationales that resemble step-by-step traces in natural language. Notably, such rationales are generated by LLMs to aid the model in locating bugs by simulating execution in a step-by-step fashion. They are not the ground-truth execution traces generated by actually running the program. As we discussed in §6, in contrast, our model relies on existing traces from program execution. Since those traces already capture rich execution information, intuitively, the resulting CoT rationales in NEXT could be more succinct and "to the point" without redundant reasoning steps to "trace" the program step-by-step by the model itself in order to recover useful execution information.

Finally, we remark that our "Test w/o Trace" setting in §5.1 shares similar spirits with the setup in Self-Debugging, as both methods perform CoT reasoning about execution without gold execution traces. From the results in Tab. 3, NEXT also greatly improves the model's ability to repair programs even without using gold execution traces at test time. This may suggest that NEXT can potentially improve the self-debugging skills of LLMs through iterative training, for which we leave

[6] For the strong LLMs that we used in this work, we did not observe any issue for following this style, which is specified in the few-shot prompt. The only exceptions are with GPT models, where they typically append the language (*i.e.,* "python") after the first set of backticks (*e.g.,* ```python), which we also handled with `regex`.

as exciting future work to explore.

B. Experiment Setup Details

B.1. Creating MBPP-R

The original MBPP dataset (Austin et al., 2021) consists of three splits, *i.e.,* train/dev/test sets of 374/90/500 Python programming problems. To increase the number of training example, we first perform a re-split of the original MBPP dataset, by moving half of the test data into the training split, resulting in 624/90/250 problems in the re-split dataset. Then for each MBPP problem in the re-split train and dev set, we collect a set of failed solutions from the released model outputs in Ni et al. (2023). More specifically, we take the 100 samples for each problems, filter out those correct solutions, and keep the ones that do not pass all the tests. As different problems have various number of buggy solutions, we balance this out by keeping at most 20 buggy solutions for each MBPP problem.[7] This yields the MBPP-R dataset, with $10,047$ repair tasks in the training set and $1,468$ examples in the dev set.

B.2. Use of test cases.

For each program repair task, there is typically a set of open test cases that are used for debugging purposes, as well as a set of hidden test cases that are only used for evaluation of correctness. When we generate traces using test cases, we use only the open test cases and only feed the open test cases to the model as part of the prompt. Then when we evaluate the generated fix, we resort to all test cases (*i.e.,* open + hidden tests) and only regard a fix as correct when it passes all test cases. While the HUMANEVAL dataset makes this distinction between open and test cases, the MBPP dataset does not make such distinction. Thus for MBPP-R, we use all test cases both as the inputs and during evaluation. While this may lead to false positives when the fixes are overfit to the test cases, and we did find such case during human annotations.

B.3. Details of Human Annotation of Rationale Quality

We annotated model predictions on 104 sampled MBPP-R repair tasks from the DEV set. Those fix tasks are randomly sampled while ensuring that they cover all the 90 dev MBPP problems. All the tasks are pre-screened to be valid program repair problems. Annotation is performed in a three-way side-by-side setting. Models are anonymized and their order is randomized. Raters are asked to judge the quality of rationales from three models (PaLM 2-L+NEXT, PaLM 2-L and GPT-3.5) on the same MBPP-R problem. Each rationale is rated from two aspects: (1) its helpfulness in explaining bugs (Q_1 : *Does the rationale correctly explain bugs in the original code? e.g.,* first two paragraphs in $\hat{r}$, Fig. 1), and (2) its helpfulness in suggesting code fixes (Q_2 : *Does the rationale suggest a correct and helpful fix? e.g.,* *"a fixed version that uses ..."* in $\hat{r}$, Fig. 1).[8] Each question has a three-scale answer (✅ Completely correct and very helpful ; ☑ Partially correct with minor errors but still helpful; ❌ Incorrect and not helpful). In a pilot study, we find that fix suggestions could often be redundant if the rationale already contains detailed explanation of bugs such that a developer could easily correct the code without an explicit fix suggestion (*e.g.,* Example 2, Appendix D). Therefore, for Q_2, we also consider such cases as correct (✅) if a model didn't suggest a fix in its rationale but the fix is obvious after bug explanations. We list our annotation guideline in Fig. 5. Note that for Q_2, both answers `(1)` and `(3)` are counted as correct (✅) answers.

C. Additional Experiment Results

Here we show the learning curve of NEXT and all its ablations in Fig. 6. We also show the full results for MBPP-R and HEFIX+ in Tab. 8 and Tab. 9, respectively.

Learning CoT rationales further improves PASS@25. From §5.1, we mention that learning to reason in natural language improves sample diversity, registering higher PASS@10 than the baseline of finetuning for generating fixes only (**NEXT w/o Rationale**). From Tab. 8 and Tab. 9, we can observe that such performance advantage is even larger with PASS@25, with 7.6% improvements on MBPP-R and 6.8% improvements on HEFIX+.

[7] This actually biased the dataset towards harder problems as easier problems may not have more than 20 buggy solutions from 100 samples, thus it might be one of the reasons for repairing solutions in MBPP-R to be more challenging than generating code for the original MBPP dataset.

[8] We only rate the quality of rationales (not the fixed code), while we still show the predicted fixed code to raters for reference.

For each task guid that you are going to work on:

1. Search for its entry in `annotation_data.html`. You will see a prompt and three model predictions (explanations and fixed code). In this task, **you will only rate the quality of the natural language explanations.** We also provided a correctness label of the fixed code. Please only use that for reference. Sometimes, the natural language explanation is correct even if the fixed code is wrong.
2. For each of the three models, please check its natural language explanation by answering the following questions. The three models we compare are: GPT3.5, NExT (ours) and PaLM-2-Large. Their order is randomized in each annotation task.

A typical explanation contains (1) explanation of why the code is wrong, and (2) how to fix the buggy code. you are going to rate the quality of these two parts separately. Here's an example explanation along with the code fix:

The issue with the provided code is that the string "element" is being inserted into the list instead of the value of the `element` variable. To fix this, you should use the `element` variable directly in the `insert` method. Here's the corrected code:

```
# Code fix, please ignore in this annotation
def add_str(tup, element):
    res = list(tup)
    for i in range(1, len(res)*2-1, 2):
        res.insert(i, element)
    return res
```

Questions:

Does the explanation correctly identify the error(s) in the original code?

1. Yes. It identifies all the errors in the code and there is no factual error in its explanation.
2. Partially correct. It only identifies some errors in the code, or the explanation has some factual errors. But overall it is still helpful.
3. No. The explanation has significant issues and is not helpful.

Does the explanation suggest a correct and helpful fix to the original code?

1. Yes. It gives a correct and helpful suggestion to fix the original code
2. Somewhat. It gives suggestions to fix some but not all the errors in the original code, or the suggestion has some errors but it's still helpful.
3. No, but Okay. It didn't suggest how to fix the code, but a developer should be able to easily correct the code given the explanation of the code error.
4. No. The suggestion is not correct at all, or there isn't any suggestion and it's not clear how to come up with a fix.

Figure 5: Instructions for the human annotators when annotating the quality of the model generated rationales.

Training on hard-only examples. One part of our data filtering pipeline is to only perform sampling and train on the samples from hard problems (§4). Here we discuss more about the benefits and potential issues of doing so, by presenting results on a "*w/o hard-only*" ablation, where the model learns from rationales and fixes from both hard and easy examples. Efficiency-wise, by only sampling on the hard example, which is around half of the problems, we greatly can accelerate the sampling process. And from results in Fig. 6, only training with hard example also comes with performance benefits under the iterative self-training framework. More specifically, we notice a non-trivial gap between the training curve of this "w/o hard-only" baseline and the rest of the ablations, especially for PASS@10 and PASS@25 performance on the training set. This means that the model trained on both easy and hard examples leads to more problems in the training set unsolved (*i.e.,* none of the samples are correct), and no learning signal can come from such problems. This also reflects on the dev set

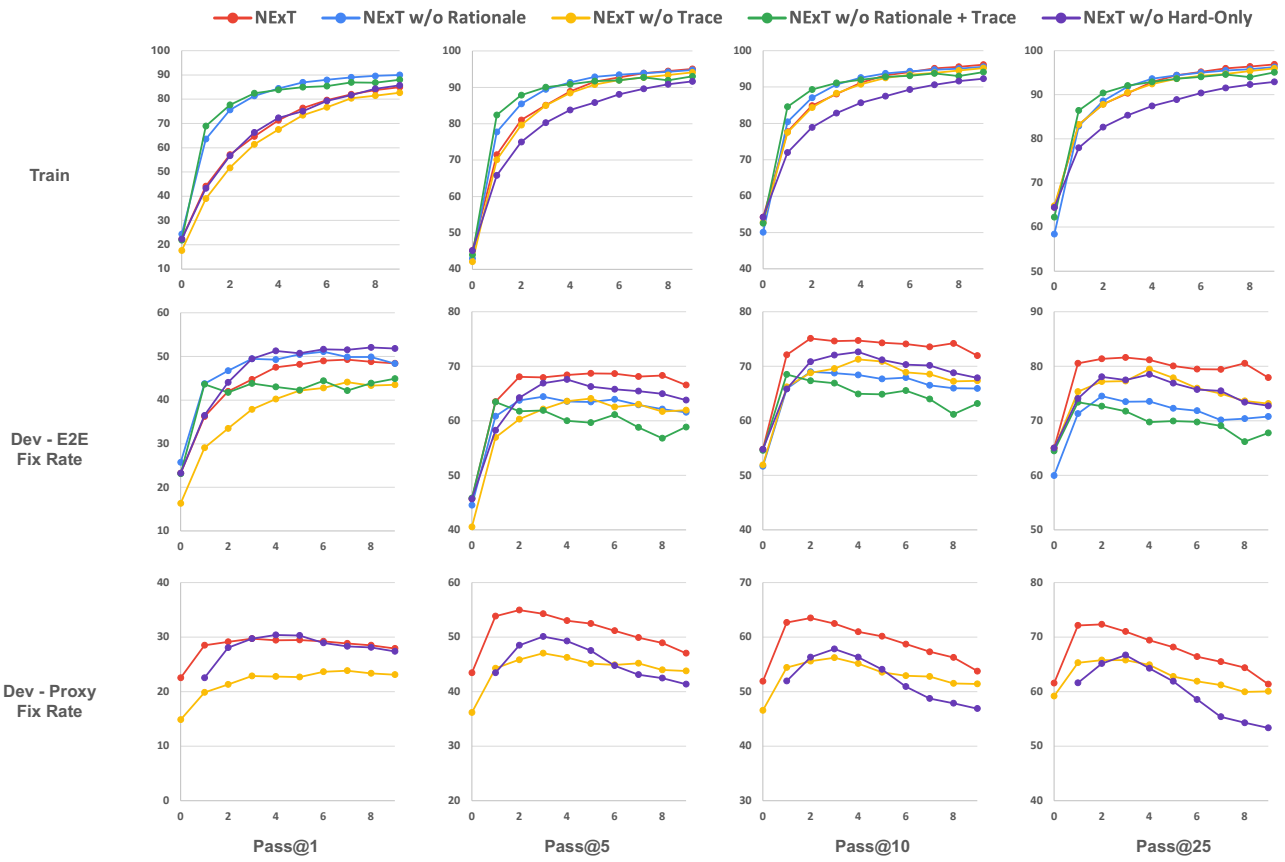


Figure 6: PASS@k performance on the train and dev sets of MBPP-R for NEXT and all its ablations.

performance. While it is worth noticing that the end-to-end PASS@1 performance for "w/o hard-only" is slightly better than NEXT trained only trained on hard examples, it performs worse in all other evaluations, with the trend of larger gaps with higher k values for PASS@k, especially for the proxy-based evaluation. This suggests that training on hard examples not only improves sample efficiency, but also improves the general fix rate as well as the quality of the generated rationales.

Proxy-based evaluation results are consistent with different proxy models. In the previous proxy-based evaluation §5.1, we report the proxy-based fix rates by averaging over the performance using PaLM 2-S and PaLM 2-S* as the proxy models. In Tab. 8 and Tab. 9, we show the separated results for different proxy models. From these results, we can observe that the relative rationale quality evaluated by different proxy models are largely consistent, with the stronger proxy model (PaLM 2-S*) having better proxy-based fix rates. In addition to the consistency we show with human annotations, this shows the robustness of our proposed proxy-based evaluation method for measuring CoT rationale quality.

D. Case Study

In this section we present a set of examples to showcase how PaLM 2-L+NEXT reasons with program execution to solve MBPP-R problems. We discover several reasoning patterns the model exhibits that leverage trace information to identify and explain bugs in programs. First, as shown in **Example 1**, the model could refer to exceptions or error messages in the trace (eg in `Trace 2`) to explain bugs in the code. Next, **Example 2** shows that the model could also leverage variable states in the trace (*e.g.,* in `Trace 2`) and compare them with the expected values to locate the cause of bugs. Besides, the `NO_CHANGE` annotations for variables whose values are preserved after execution of a step could also help the model explain the execution process in the rationale (*e.g.,* `(3) NO_CHANGE` $\mapsto$ *"the first sublist is already sorted"*). Perhaps a more interesting scenario is when the model reasons over multiple steps of computation to track down the cause of a bug. In **Example 3**, the model attempts to trace the computation of steps 2 - 4 in `Trace 1` to explain why the sum is a float instead of an integer. Another example is **Example 4**, where the model summarizes the loop iterations in steps 2 - 9 of `Trace 1` to explain the cause of

Models	End-to-End Fix Rate					Proxy-based Fix Rate (PaLM 2–S)					Proxy-based Fix Rate (PaLM 2–S*)				
	GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling			
	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25
GPT–3.5	46.4	42.9	65.0	70.7	76.7	27.9	24.7	46.1	54.5	64.6	31.8	28.5	51.5	59.5	68.2
GPT–3.5 w/o trace	47.1	46.8	65.9	70.7	75.7	27.2	25.6	47.0	55.5	64.7	30.9	30.2	53.0	60.7	68.8
GPT–4	62.8	63.2	75.1	78.5	82.7	41.8	42.2	64.5	71.0	76.6	47.8	47.4	68.5	73.9	79.0
GPT–4 w/o trace	51.3	44.8	68.5	73.4	78.5	29.4	27.1	54.2	63.4	72.2	34.9	32.0	60.3	68.5	75.7
PaLM 2-L	26.6	23.2	45.7	54.7	65.0	21.5	21.1	41.1	49.5	59.2	24.9	23.9	45.8	54.3	63.8
PaLM 2-L w/o trace	19.0	16.3	42.1	52.8	64.8	14.7	13.7	33.9	44.1	56.7	17.4	15.9	38.3	48.9	61.6
PaLM 2-L w/o rationale	27.5	25.7	44.5	51.7	60.0	–	–	–	–	–	–	–	–	–	–
PaLM 2-L w/o rationale + trace	23.8	23.1	45.8	54.6	64.5	–	–	–	–	–	–	–	–	–	–
NEXT	50.5	49.3	68.1	73.5	79.4	25.3	26.1	46.8	54.4	62.9	31.8	31.6	53.0	60.2	68.1
test w/o trace	41.1	40.8	61.8	68.9	76.4	17.6	17.5	35.6	43.5	53.4	21.0	21.5	42.2	50.6	60.1
NEXT w/o hard–only	52.9	52.1	65.0	68.8	73.4	23.5	25.1	38.6	44.0	50.9	30.0	29.7	44.1	49.7	55.9
test w/o trace	41.9	42.2	58.1	63.2	69.2	16.3	17.8	32.1	37.9	45.0	18.7	21.0	36.7	43.0	50.5
NEXT w/o rationale	51.8	51.1	63.9	67.9	71.8	–	–	–	–	–	–	–	–	–	–
test w/o trace	43.7	43.0	57.2	61.7	66.3	–	–	–	–	–	–	–	–	–	–
NEXT w/o trace	44.5	44.1	63.0	68.5	75.0	22.3	21.8	42.3	50.1	59.2	25.9	25.9	48.0	55.4	63.2
NEXT w/o rationale w/o trace	46.3	44.9	58.9	63.2	67.8	–	–	–	–	–	–	–	–	–	–

Table 8: Full results on MBPP-R. "GD Acc." denotes PASS@1 evaluated with greedy decoding. All models in the top half are few-shot prompted while the bottom half shows the result of NEXT and its ablations.

Models	End-to-End Fix Rate					Proxy-based Fix Rate (PaLM 2–S)					Proxy-based Fix Rate (PaLM 2–S*)				
	GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling			
	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25
GPT-3.5	68.9	59.4	84.5	89.2	93.0	42.1	39.0	66.1	73.4	80.2	46.3	44.6	71.6	78.8	86.8
GPT-3.5 w/o trace	65.2	65.4	85.3	89.2	92.6	45.7	41.7	68.2	76.3	84.5	50.0	47.2	73.8	81.1	88.6
GPT-4	79.9	77.6	89.3	91.1	92.9	56.1	55.4	75.7	80.8	85.8	61.0	57.7	77.5	82.7	87.4
GPT-4 w/o trace	79.3	68.9	88.3	90.7	92.9	54.9	46.1	72.3	79.0	86.1	59.8	48.7	74.4	80.8	87.5
PaLM 2-L	43.3	32.2	64.3	73.8	81.5	32.9	28.9	59.0	69.2	79.1	43.3	34.9	65.8	74.3	82.9
PaLM 2-L w/o trace	38.4	30.3	61.9	72.9	83.3	25.6	27.8	56.2	66.0	76.6	31.1	33.0	63.5	72.7	81.8
PaLM 2-L w/o rationale	53.0	45.3	71.5	78.9	85.4	–	–	–	–	–	–	–	–	–	–
PaLM 2-L w/o rationale + trace	48.2	43.2	71.4	80.0	87.7	–	–	–	–	–	–	–	–	–	–
NEXT	46.3	42.5	62.6	69.1	76.5	31.7	34.8	54.8	62.4	70.2	40.9	41.3	61.8	68.9	76.4
test w/o trace	42.7	41.2	62.9	70.6	79.5	26.8	26.4	48.0	56.1	64.2	36.0	32.6	55.7	64.4	72.8
NEXT w/o hard-only	48.8	47.7	64.8	70.4	76.6	32.9	37.2	50.8	55.5	61.9	41.5	42.4	56.3	60.8	66.9
test w/o trace	47.6	44.2	64.4	70.4	75.5	31.7	33.3	46.9	51.4	57.3	38.4	38.5	54.6	59.2	63.9
NEXT w/o rationale	47.6	44.5	58.9	63.7	69.4	–	–	–	–	–	–	–	–	–	–
test w/o trace	46.3	44.7	60.4	65.2	70.2	–	–	–	–	–	–	–	–	–	–
NEXT w/o trace	40.9	38.1	59.1	65.3	71.5	29.3	26.9	52.1	61.1	71.5	33.5	34.4	63.1	70.8	77.4
NEXT w/o rationale w/o trace	30.5	31.4	44.6	49.0	54.1	–	–	–	–	–	–	–	–	–	–

Table 9: Full results on HEFIX+. Same notations from Tab. 8 apply.

the missing last element in the result list. Interestingly, while the model is able to reason over multiple steps of execution in its rationales, as the reasoning chain becomes longer, it is more likely to contain minor logical errors, as highlighted in the rationales of Examples 3 and 4.

Besides minor logical errors in CoT reasoning, we also identify several more pronounced error modes. First, even if the NL rationale looks valid, the actual code fix might be incorrect (**Example 4**). Second, for cases where there are multiple bugs in the program, the model might only focus on the more obvious bugs during reasoning (*e.g.,* explaining the reason of `IndexError` in `Trace 2` of **Example 5**), while ignoring the other more nuanced bugs that are buried in the trace (*e.g.,* Incorrect outputs in `Trace 1` and `Trace 3` caused by zero-initialized `freq_arr`). This could be especially problematic when there are hidden bugs *not* captured by the trace, which could happen when a buggy program crashes in the middle of execution and the rest the of the code is not covered in the trace. Since the model could be more reliant on trace information after fine-tuning, it might be more likely to miss such implicit bugs. As an example, in **Example 6** the model only noticed the `TypeError` in the trace, without spotting that the code that computes the end index in the output tuple is off by one.

Example 1: An example where our model explains exceptions in the trace

Task: Write a function to remove leading zeroes from an ip address

Buggy Code:

```
def removezero_ip(ip):
    ip_list = ip.split(".")
    for i in range(4):
        ip_list[i] = str(int(ip_list[i]))
    return ".".join(ip_list)
```

Test Cases:

```
assert removezero_ip("216.08.094.196")==('216.8.94.196')
assert removezero_ip("12.01.024")==('12.1.24')
assert removezero_ip("216.08.094.0196")==('216.8.94.196')
```

Traces:

```
# Trace 1:
def removezero_ip(ip):  # (0) ip='216.08.094.196';
    ip_list = ip.split(".")  # (1) ip_list=['216', '08', '094', '196'];
    for i in range(4):  # (2) i=0; (4) i=1; (6) i=2; ...; (10) NO_CHANGE;
        ip_list[i] = str(int(ip_list[i]))  # (3) NO_CHANGE; (5) ip_list=['216', '8', '094', '196']; (7) ip_list=['216', '8', '94',
'196']; (9) NO_CHANGE;
    return ".".join(ip_list)  # (11) __return__='216.8.94.196';

assert removezero_ip("216.08.094.196")==('216.8.94.196')

# Trace 2:
def removezero_ip(ip):  # (0) ip='12.01.024';
    ip_list = ip.split(".")  # (1) ip_list=['12', '01', '024'];
    for i in range(4):  # (2) i=0; (4) i=1; (6) i=2; (8) i=3;
        ip_list[i] = str(int(ip_list[i]))  # (3) NO_CHANGE; (5) ip_list=['12', '1', '024']; (7) ip_list=['12', '1', '24']; (9)
__exception__ = IndexError('list index out of range')
    return ".".join(ip_list)

assert removezero_ip("12.01.024")==('12.1.24')

# Trace 3:
def removezero_ip(ip):  # (0) ip='216.08.094.0196';
    ip_list = ip.split(".")  # (1) ip_list=['216', '08', '094', '0196'];
    for i in range(4):  # (2) i=0; (4) i=1; (6) i=2; ...; (10) NO_CHANGE;
        ip_list[i] = str(int(ip_list[i]))  # (3) NO_CHANGE; (5) ip_list=['216', '8', '094', '0196']; (7) ip_list=['216', '8', '94',
'0196']; (9) ip_list=['216', '8', '94', '196'];
    return ".".join(ip_list)  # (11) __return__='216.8.94.196';

assert removezero_ip("216.08.094.0196")==('216.8.94.196')
```

Rationale and Code Solution: (code fix is correct ✅)

The problem spec asks to write a function to remove leading zeroes from an IP address.

Now let's look at the program. The program first splits the IP address into a list of strings, and then iterates through the list to convert each string to an integer, and then back to a string. This will remove leading zeroes.
However, the problem is that it iterates through the list for 4 times, but the length of the list might not be 4. For example, when `ip="12.01.024"`, the length of the list is only 3. Thus when `i=3`, it will throw an `IndexError` .

Here is a simple fix:

```
def removezero_ip(ip):
    ip_list = ip.split(".")
    for i in range(len(ip_list)):
        ip_list[i] = str(int(ip_list[i])).lstrip('0')
    return ".".join(ip_list)
```

Example 2: An example where our model uses variable states to solve the task

Task: Write a function to sort each sublist of strings in a given list of lists

Buggy Code:

```
def sort_sublists(words):
    words.sort(key = len)
    for w in words:
        w.sort()
    return words
```

Test Cases:

```
assert sort_sublists([['green', 'orange'], ['black', 'white'], ['white', 'black', 'orange']])==[['green', 'orange'], ['black', '
white'], ['black', 'orange', 'white']]
assert sort_sublists([['green', 'orange'], ['black'], ['green', 'orange'], ['white']])==[['green', 'orange'], ['black'], ['green',
'orange'], ['white']]
assert sort_sublists([['a','b'],['d','c'],['g','h'] , ['f','e']])==[['a', 'b'], ['c', 'd'], ['g', 'h'], ['e', 'f']]
```

Traces:

```
# Trace 1:
def sort_sublists(words):  # (0) words=[['green', 'orange'], ['black', 'white'], ['white', 'black', 'orange']];
    words.sort(key = len)  # (1) NO_CHANGE;
    for w in words:  # (2) w=['green', 'orange']; (4) w=['black', 'white']; (6) w=['white', 'black', 'orange']; (8) NO_CHANGE;
        w.sort()  # (3) NO_CHANGE; (5) NO_CHANGE; (7) w=['black', 'orange', 'white'], words=[['green', 'orange'], ['black', 'white
'], ['black', 'orange', 'white']];
    return words  # (9) __return__=[['green', 'orange'], ['black', 'white'], ['black', 'orange', 'white']];

assert sort_sublists([['green', 'orange'], ['black', 'white'], ['white', 'black', 'orange']])==[['green', 'orange'], ['black', '
white'], ['black', 'orange', 'white']]

# Trace 2:
def sort_sublists(words):  # (0) words=[['green', 'orange'], ['black'], ['green', 'orange'], ['white']];
    words.sort(key = len)  # (1) words=[['black'], ['white'], ['green', 'orange'], ['green', 'orange']];
    for w in words:  # (2) w=['black']; (4) w=['white']; (6) w=['green', 'orange']; ...; (10) NO_CHANGE;
        w.sort()  # (3) NO_CHANGE; (5) NO_CHANGE; (7) NO_CHANGE; (9) NO_CHANGE;
    return words  # (11) __return__=[['black'], ['white'], ['green', 'orange'], ['green', 'orange']];

assert sort_sublists([['green', 'orange'], ['black'], ['green', 'orange'], ['white']])==[['green', 'orange'], ['black'], ['green',
'orange'], ['white']]  # (12)  __exception__ = AssertionError()

# Trace 3:
def sort_sublists(words):  # (0) words=[['a', 'b'], ['d', 'c'], ['g', 'h'], ['f', 'e']];
    words.sort(key = len)  # (1) NO_CHANGE;
    for w in words:  # (2) w=['a', 'b']; (4) w=['d', 'c']; (6) w=['g', 'h']; ...; (10) NO_CHANGE;
        w.sort()  # (3) NO_CHANGE; (5) w=['c', 'd'], words=[['a', 'b'], ['c', 'd'], ['g', 'h'], ['f', 'e']]; (7) NO_CHANGE; (9) w
=['e', 'f'], words=[['a', 'b'], ['c', 'd'], ['g', 'h'], ['e', 'f']];
    return words  # (11) __return__=[['a', 'b'], ['c', 'd'], ['g', 'h'], ['e', 'f']];

assert sort_sublists([['a','b'],['d','c'],['g','h'] , ['f','e']])==[['a', 'b'], ['c', 'd'], ['g', 'h'], ['e', 'f']]
```

Rationale and Code Solution: (code fix is correct ✅)

The problem spec asks to sort each sublist of strings in a given list of lists.

Now let's look at the program. The program first sorts the list of lists by the length of each sublist, and then sorts each sublist. The first assertion passes as the first sublist is already sorted, and the second sublist is also sorted, and the third sublist is sorted in the for loop. The second assertion fails as the list of lists is sorted by the length of each sublist, and thus the return value is not the same as the input. The third assertion passes as the first and the last sublists are already sorted, and the second and the third sublists are sorted in the for loop.

The problem is that the program sorts the list of lists by the length of each sublist, which is not required by the problem spec. Thus we can simply remove the first line of the function.

Here is the corrected code:

```
def sort_sublists(words):
    for w in words:
        w.sort()
    return words
```

Example 3: An example where PaLM 2-L+NEXT reasons over multiple steps of execution

Task: Write a function to get the sum of a non-negative integer

Buggy Code:

```
def sum_digits(number):
    sum = 0
    while(number):
        sum += number %
        number = number / 10
    return sum
```

Test Cases:

```
assert sum_digits(345)==12
assert sum_digits(12)==3
assert sum_digits(97)==16
```

Traces:

```
# Trace 1:
def sum_digits(number):  # (0) number=345;
    sum = 0  # (1) sum=0;
    while(number):
        sum += number %
        number = number / 10  # (3) number=34.5; (5) number=3.45; (7) number=0.34500000000000003; ...; (655) number=0.0;
    return sum  # (656) __return__=13.333333333333333;

assert sum_digits(345)==12  # (657)  __exception__ = AssertionError()

# Trace 2:
def sum_digits(number):  # (0) number=12;
    sum = 0  # (1) sum=0;
    while(number):
        sum += number %
        number = number / 10  # (3) number=1.2; (5) number=0.12; (7) number=0.012; ...; (651) number=0.0;
    return sum  # (652) __return__=3.3333333333333335;

assert sum_digits(12)==3  # (653)  __exception__ = AssertionError()

# Trace 3:
def sum_digits(number):  # (0) number=97;
    sum = 0  # (1) sum=0;
    while(number):
        sum += number %
        number = number / 10  # (3) number=9.7; (5) number=0.97; (7) number=0.097; ...; (653) number=0.0;
    return sum  # (654) __return__=17.77777777777777;

assert sum_digits(97)==16  # (655)  __exception__ = AssertionError()
```

Rationale and Code Solution: (code fix is correct ✅)

The problem spec asks to write a function to get the sum of all digits in a non−negative integer.

Now let's look at the program. The program seems to be correct at the first glance. It first initializes the sum to 0, and then it uses a while loop to get each digit in the number and add it to the sum. However, if we look at the output of the program, we will find that the sum is not an integer.
This is because the program does not convert the number to an integer before it gets the digits. For example, for the first assertion, when ‘number=345‘, the first digit ‘5‘ is added to the sum and ‘sum=5‘. Then ‘number‘ is updated to ‘34.5‘, and the second digit ‘4‘ is added to the sum and ‘sum=9.5‘. And so on.

Here is a fixed version:

```
def sum_digits(number):
    number = int(number)
    sum = 0
    while(number):
        sum += number %
        number = number // 10
    return sum
```

Example 4: An example where PaLM 2-L+NEXT reasons over multiple steps of execution

Task: Write a function to convert tuple into list by adding the given string after every element.

Buggy Code:

```
def add_str(test_tuple, add_str):
    res = list(test_tuple)
    for i in range(1,len(test_tuple)):
        res.insert(i*2−1,add_str)
    return res
```

Test Cases:

```
assert add_str((5, 6, 7, 4, 9) , "FDF") == [5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9, 'FDF']
assert add_str((7, 8, 9, 10) , "PF") == [7, 'PF', 8, 'PF', 9, 'PF', 10, 'PF']
assert add_str((11, 14, 12, 1, 4) , "JH") == [11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH', 4, 'JH']
```

Traces:

```
# Trace 1:
def add_str(test_tuple, add_str):
    # Write your code here
    res = list(test_tuple)  # (1) res=[5, 6, 7, 4, 9];
    for i in range(1,len(test_tuple)):  # (2) i=1; (4) i=2; (6) i=3; ...; (10) NO_CHANGE;
        res.insert(i*2-1,add_str)  # (3) res=[5, 'FDF', 6, 7, 4, 9]; (5) res=[5, 'FDF', 6, 'FDF', 7, 4, 9];
        (7) res=[5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 9]; (9) res=[5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9];
    return res  # (11) __return__=[5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9];

assert add_str((5, 6, 7, 4, 9) , "FDF") == [5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9, 'FDF']  # (12)  __exception__ =
AssertionError()

# Trace 2:
def add_str(test_tuple, add_str):
    # Write your code here
    res = list(test_tuple)  # (1) res=[7, 8, 9, 10];
    for i in range(1,len(test_tuple)):  # (2) i=1; (4) i=2; (6) i=3; (8) NO_CHANGE;
        res.insert(i*2-1,add_str)  # (3) res=[7, 'PF', 8, 9, 10]; (5) res=[7, 'PF', 8, 'PF', 9, 10]; (7) res=[7, 'PF', 8, 'PF', 9,
 'PF', 10];
    return res  # (9) __return__=[7, 'PF', 8, 'PF', 9, 'PF', 10];

assert add_str((7, 8, 9, 10) , "PF") == [7, 'PF', 8, 'PF', 9, 'PF', 10, 'PF']  # (10)  __exception__ = AssertionError()

# Trace 3:
def add_str(test_tuple, add_str):
    # Write your code here
    res = list(test_tuple)  # (1) res=[11, 14, 12, 1, 4];
    for i in range(1,len(test_tuple)):  # (2) i=1; (4) i=2; (6) i=3; ...; (10) NO_CHANGE;
        res.insert(i*2-1,add_str)  # (3) res=[11, 'JH', 14, 12, 1, 4]; (5) res=[11, 'JH', 14, 'JH', 12, 1, 4]; (7) res=[11, 'JH',
14, 'JH', 12, 'JH', 1, 4]; (9) res=[11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH', 4];
    return res  # (11) __return__=[11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH', 4];

assert add_str((11, 14, 12, 1, 4) , "JH") == [11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH', 4, 'JH']  # (12)  __exception__ =
AssertionError()
```

Rationale and Code Solution: (code fix is wrong ✗)

The problem spec asks to convert a tuple to a list and add a given string after every element.

Now let's look at the program. The program first converts the tuple to a list, which is good. Then it iterates from 1 to the length of the list, and inserts the given string at index `i*2-1`. This seems correct at the first glance, but the problem is that it will not add the string after the last element. For example, when ‘test_tuple=(5, 6, 7, 4, 9)‘, the length of the list is 5, and the for loop will iterate from 1 to 4. Thus the string will be inserted at index 1, 3, 5, and 7, which corresponds to after the second, fourth, and sixth element. The string will not be inserted after the last element (i.e., the ninth element).

Here is a simple fix:

```
def add_str(test_tuple, add_str):
    res = list(test_tuple)
    for i in range(len(test_tuple)):
        res.insert(i*2,add_str)
    return res
```

Example 5: An example where PaLM 2-L+NEXT failed to identify all the bugs in the program

Task: Write a python function to find the difference between highest and least frequencies in a given array.

Buggy Code:

```
def find_Diff(nums, n):
    freq_arr = [0 for i in range(n)]
    for i in range(len(nums)):
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1
    return max(freq_arr) - min(freq_arr)
```

Test Cases:

```
assert find_Diff([1,1,2,2,7,8,4,5,1,4],10) == 2
assert find_Diff([1,7,9,2,3,3,1,3,3],9) == 3
assert find_Diff([1,2,1,2],4) == 0
```

Traces:

```
# Trace 1:
def find_Diff(nums, n):  # (0) n=10, nums=[1, 1, 2, 2, 7, 8, 4, 5, 1, 4]; (1) .0=<range_iterator object>;
    freq_arr = [0 for i in range(n)]  # (2) i=0; (3) i=1; (4) i=2; ...; (13) freq_arr=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
    for i in range(len(nums)):  # (14) i=0; (16) i=1; (18) i=2; ...; (34) NO_CHANGE;
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1  # (15) freq_arr=[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]; (17) freq_arr=[0, 2, 0, 0, 0, 0,
0, 0, 0, 0]; (19) freq_arr=[0, 2, 1, 0, 0, 0, 0, 0, 0, 0]; ...; (33) freq_arr=[0, 3, 2, 0, 2, 1, 0, 1, 1, 0];
    return max(freq_arr) - min(freq_arr)  # (35) __return__=3;

assert find_Diff([1,1,2,2,7,8,4,5,1,4],10) == 2  # (36) __exception__ = AssertionError()

# Trace 2:
def find_Diff(nums, n):  # (0) n=9, nums=[1, 7, 9, 2, 3, 3, 1, 3, 3]; (1) .0=<range_iterator object>;
    freq_arr = [0 for i in range(n)]  # (2) i=0; (3) i=1; (4) i=2; ...; (12) freq_arr=[0, 0, 0, 0, 0, 0, 0, 0, 0];
    for i in range(len(nums)):  # (13) i=0; (15) i=1; (17) i=2;
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1  # (14) freq_arr=[0, 1, 0, 0, 0, 0, 0, 0, 0]; (16) freq_arr=[0, 1, 0, 0, 0, 0, 0,
1, 0]; (18)  __exception__ = IndexError('list index out of range')
    return max(freq_arr) - min(freq_arr)

assert find_Diff([1,7,9,2,3,3,1,3,3],9) == 3

# Trace 3:
def find_Diff(nums, n):  # (0) n=4, nums=[1, 2, 1, 2]; (1) .0=<range_iterator object>;
    freq_arr = [0 for i in range(n)]  # (2) i=0; (3) i=1; (4) i=2; ...; (7) freq_arr=[0, 0, 0, 0];
    for i in range(len(nums)):  # (8) i=0; (10) i=1; (12) i=2; ...; (16) NO_CHANGE;
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1  # (9) freq_arr=[0, 1, 0, 0]; (11) freq_arr=[0, 1, 1, 0]; (13) freq_arr=[0, 2, 1,
0]; (15) freq_arr=[0, 2, 2, 0];
    return max(freq_arr) - min(freq_arr)  # (17) __return__=2;

assert find_Diff([1,2,1,2],4) == 0  # (18) __exception__ = AssertionError()
```

Rationale and Code Solution: (code fix is wrong ✗)

The problem spec asks to find the difference between the highest and the least frequencies in a given array.

Now let's look at the program. The program first creates a frequency array of length `n`, and then iterates through the input array to count the frequency of each number. It then returns the difference between the maximum and the minimum frequencies in the frequency array.

The program looks good at the first glance.
However, the problem is that the length of the frequency array is ‘n‘, which is the length of the input array.
However, the numbers in the input array can be larger than ‘n‘, and thus the frequency array will not be able to accommodate all the numbers in the input array. For example, in the second assertion, the input array is `[1, 7, 9, 2, 3, 3, 1, 3, 3]`, and the length of the frequency array is also `9`. When `nums[i]=9`, it will try to access `freq_arr[9]`, which is out of bound, and thus an `IndexError` is thrown.

Here is a simple fix:

```
def find_Diff(nums, n):
    freq_arr = {}
    for i in range(len(nums)):
        if nums[i] in freq_arr:
            freq_arr[nums[i]] += 1
        else:
            freq_arr[nums[i]] = 1
    max_freq = 0
    min_freq = 0
    for num, freq in freq_arr.items():
        if freq > max_freq:
            max_freq = freq
        if freq < min_freq or min_freq == 0:
            min_freq = freq
    return max_freq - min_freq
```

Example 6: An example where PaLM 2-L+NEXT failed to identify a bug not revealed in the trace

Task: Write a function to search a literals string in a string and also find the location within the original string where the pattern occurs.

Buggy Code:

```
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)

    return match
```

Test Cases:

```
assert search_literal('python','python programming language')==(0,6)
assert search_literal('programming','python programming language')==(7,18)
assert search_literal('language','python programming language')==(19,27)
```

Traces:

```
# Trace 1:
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)  # (1)  __exception__ = TypeError('tuple expected at most 1 argument, got 2')

    return match

assert search_literal('python','python programming language')==(0,6)

# Trace 2:
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)  # (1)  __exception__ = TypeError('tuple expected at most 1 argument, got 2')

    return match

assert search_literal('programming','python programming language')==(7,18)

# Trace 3:
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)  # (1)  __exception__ = TypeError('tuple expected at most 1 argument, got 2')

    return match

assert search_literal('language','python programming language')==(19,27)
```

Rationale and Code Solution: (code fix is wrong ✗)

Now let's look at the program. The program looks good at the first glance as `text.find(pattern)` is the correct way to find the location of a literal string in a given string. But the problem is that it does not use the `tuple()` function correctly. The `tuple()` function takes a single iterable as input, and returns a tuple with the elements from the iterable.
But in the program, it takes two arguments, which will throw a ‘TypeError‘ of ‘tuple expected at most 1 argument, got 2‘.

Here is a simple fix:

```
def search_literal(pattern, text):
    match = tuple(text.find(pattern),)
    match = match + (text.find(pattern) + len(pattern) - 1,)

    return match
```

E. Full Prompt

Here we show the instruction and all three few-shot examples used in this work. The rationales and fixes in the few-shot examples are hand annotated by the authors. Each exemplar has three unit tests and their traces.

````
We are doing pair programming, I will give you the problem definition and my implementation, which does not pass all test cases. And you need to help me to fix my code.

# Example 1

Here is the specification in natural language and the program that I wrote:
```
# Write a function to find the similar elements from the given two tuple lists.
def similar_elements(t1, t2):
        s1 = set(t1)
        s2 = set(t2)
        return s1.intersection(s2)

```
Here are the assertions I need to pass:
```
assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)
assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)
```

Here is a *cheatsheet* for the gold execution trace for each assertions, use the information here but pretend you did all the reasoning about execution yourself.

#################### cheatsheet starts ####################
def similar_elements(t1, t2):   # (0) t1=(3, 4, 5, 6), t2=(5, 7, 4, 10);
        s1 = set(t1)    # (1) s1={3, 4, 5, 6};
        s2 = set(t2)    # (2) s2={10, 4, 5, 7};
        return s1.intersection(s2)  # (3) __return__={4, 5}


assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)  #
--------------------
def similar_elements(t1, t2):   # (0) t1=(1, 2, 3, 4), t2=(5, 4, 3, 7);
        s1 = set(t1)    # (1) s1={1, 2, 3, 4};
        s2 = set(t2)    # (2) s2={3, 4, 5, 7};
        return s1.intersection(s2)  # (3) __return__={3, 4}


assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)  #
--------------------
def similar_elements(t1, t2):   # (0) t1=(11, 12, 14, 13), t2=(17, 15, 14, 13);
        s1 = set(t1)    # (1) s1={11, 12, 13, 14};
        s2 = set(t2)    # (2) s2={17, 13, 14, 15};
        return s1.intersection(s2)  # (3) __return__={13, 14}


assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)  #
#################### cheatsheet ends ####################

Given the above information, can you tell me why my code is wrong, and also provide a fixed version?

The problem spec asks to find the similar elements in the two tuples, from the assertion we can see that it essentially wants the common elements from the two tuples. It also seems that the common elements should be returned in a tuple with ascending order.

Now let's look at the program. While it is correct and efficient to use `intersection()` to find the common elements between two sets, it will also return a set. Take the first assertion as an example, `s1` will first be converted to a set like `{3, 4, 5, 6}` and similarly for `s2`, then the return value will be `{4, 5}`. Though the elements are right, the expected return value is a tuple and not a set. Thus we only need to further convert the sets to tuples using `tuple()`.

Here is the corrected code:
```
def similar_elements(t1, t2):
        s1 = set(t1)
        s2 = set(t2)
        return tuple(s1.intersection(s2))
```

# Example 2

Here is the specification in natural language and the program that I wrote:
```
# Write a python function to identify non-prime numbers.
````
````

````
def is_not_prime(n):
  if n==1:
    return False
  else:
    for i in range(2,n//2+1):
      if n%i==0:
        return False
      else:
        return True
```
Here are the assertions I need to pass:
```
assert is_not_prime(2) == False
assert is_not_prime(10) == True
assert is_not_prime(35) == True
```

Here is a *cheatsheet* for the gold execution trace for each assertions, use the information here but pretend you did all the reasoning about execution yourself.

################### cheatsheet starts ###################
def is_not_prime(n):  # (0) n=2;
  if n==1:  #
    return False
  else:
    for i in range(2,n//2+1):  # (1) __return__=None
      if n%i==0:
        return False
      else:
        return True


assert is_not_prime(2) == False  #
--------------------
def is_not_prime(n):  # (0) n=10;
  if n==1:  #
    return False
  else:
    for i in range(2,n//2+1):  # (1) i=2;
      if n%i==0:  #
        return False  # (2) __return__=False
      else:
        return True


assert is_not_prime(10) == True  #
--------------------
def is_not_prime(n):  # (0) n=35;
  if n==1:  #
    return False
  else:
    for i in range(2,n//2+1):  # (1) i=2;
      if n%i==0:  #
        return False
      else:
        return True  # (2) __return__=True


assert is_not_prime(35) == True  #
################### cheatsheet ends ###################

Given the above information, can you tell me why my code is wrong, and also provide a fixed version?

The problem spec asks to write a Python function that identifies non-prime numbers. Non-prime numbers are numbers that are divisible by numbers other than 1 and itself. Note that 0 and 1 are also non-prime numbers.

Now let's check the program. Let's run the program through some examples. For the first assertion, when `n=2`, it will enter the `else` branch, and then in the `for` loop, since the condition is `i in range(2, n//2+1)`, which is `range(2, 2)`, it will actually not enter the loop since the condition is not met in the first place. And then it will reach the end of the function, and the `return` value will be `None`. And `for` the third assertion, when `n=35`, it also enters the `else` branch, and it then enters the loop with `i=2` as `n//2+1=18 > 2`. Then in the next step, since `n%i==0` is false, it enters the else branch and returns `True`. This somehow matches the expected output, but it is by chance.

While it is not obvious how to correct this program, another way to implement the `is_not_prime()` function is just to first create a helper function to identify prime numbers as `is_prime()`, and then simply `return not is_prime(x)`.

Here is a re-implementation:
```
````

````
def is_prime(x):
  if x <= 1:
    return False
  else:
    for i in range(2,x):
      if x%i == 0:
        return False
    return True

def is_not_prime(x):
  return not is_prime(x)
```

# Example 3

Here is the specification in natural language and the program that I wrote:
```
# Write a function to find the largest integers from a given list of numbers using heap queue algorithm.
def heap_queue_largest(list1,n):
  largest=heapq.nlargest(n,list1)
  return largest

```
Here are the assertions I need to pass:
```
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],2)==[85, 75]
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],5)==[85, 75, 65, 58, 35]
```

Here is a *cheatsheet* for the gold execution trace for each assertions, use the information here but pretend you did all the reasoning about execution yourself.

#################### cheatsheet starts ####################
def heap_queue_largest(list1,n):  # (0) list1=[25, 35, 22, 85, 14, 65, 75, 22, 58], n=3;
  largest=heapq.nlargest(n,list1)  # (1)  __exception__ = NameError("name 'heapq' is not defined")
  return largest


assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]
--------------------
def heap_queue_largest(list1,n):  # (0) list1=[25, 35, 22, 85, 14, 65, 75, 22, 58], n=2;
  largest=heapq.nlargest(n,list1)  # (1)  __exception__ = NameError("name 'heapq' is not defined")
  return largest


assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],2)==[85, 75]
--------------------
def heap_queue_largest(list1,n):  # (0) list1=[25, 35, 22, 85, 14, 65, 75, 22, 58], n=5;
  largest=heapq.nlargest(n,list1)  # (1)  __exception__ = NameError("name 'heapq' is not defined")
  return largest


assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],5)==[85, 75, 65, 58, 35]
#################### cheatsheet ends ####################

Given the above information, can you tell me why my code is wrong, and also provide a fixed version?

The problem spec asks to find `n` largest integers in a given list. It also suggests that the heap queue shall be used.

Now let's look at the program. The program looks good at the first glance as `heapq.nlargest()` is the correct way to get the `n` largest integers from an iterable (e.g., a list). But the problem is that it does not import the `heapq` package first. Thus all three assertions will fail at the first line of the function, and throw a `NameError` of `name 'heapq' is not defined`.

Here is a simple fix:
```
import heapq
def heap_queue_largest(list1,n):
  largest=heapq.nlargest(n,list1)
  return largest
```
````
````