

CRUXEVAL-X: A Benchmark for Multilingual Code Reasoning, Understanding and Execution

Ruiyang Xu^{1,2*}, Jialun Cao^{3*}, Yaojie Lu¹, Hongyu Lin¹,
Xianpei Han¹, Ben He^{1,2}, Shing-Chi Cheung³, Le Sun¹

¹Chinese Information Processing Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing, China

²University of Chinese Academy of Sciences, Beijing, China

³The Hong Kong University of Science and Technology, Hong Kong, China

{xuruiyang2022,hongyu,luyaojie,xianpei,sunle}@iscas.ac.cn

benhe@ucas.edu.cn

{jcaoap, scc}@cse.ust.hk

Abstract

Code benchmarks such as HumanEval are widely adopted to evaluate Large Language Models' (LLMs) coding capabilities. However, there is an unignorable **programming language bias** in existing code benchmarks – over 95% code generation benchmarks are dominated by Python, leaving the LLMs' capabilities in other programming languages such as Java and C/C++ unknown. Moreover, **coding task bias** is also crucial. Most benchmarks focus on code generation capability, while benchmarks for *code reasoning* (given input, reasoning output; and given output, reasoning input), an essential coding capability, are insufficient. Yet, constructing multi-lingual benchmarks can be expensive and labor-intensive, and codes in contest websites such as Leetcode suffer from data contamination during training. To fill this gap, we propose CRUXEVAL-X, a **multi-lingual code reasoning benchmark** that contains 19 programming languages. It comprises at least 600 subjects for each language, along with 19K content-consistent tests in total. In particular, the construction pipeline of CRUXEVAL-X works in a fully automated and test-guided manner, which iteratively generates and repairs based on execution feedback. Also, to cross language barriers (e.g., dynamic/static type systems in Python/C++), we formulated various transition rules between language pairs to facilitate translation. Our intensive evaluation of 24 representative LLMs reveals the correlation between language pairs. For example, TypeScript and JavaScript show a significant positive correlation, while Racket has less correlation with other languages. More interestingly, even a model trained solely on Python can achieve at most 34.4% Pass@1 in other languages, revealing the cross-language generalization of LLMs. The leaderboard is available at <https://cruxeval-x.github.io/leaderboard.html>.

Introduction

Large language models (LLMs) have shown advanced proficiency in various domains, including code generation (Liu et al. 2024; Du et al. 2024), defect detection (Yang et al. 2024b) and program repair (Xia and Zhang 2023; Zhong,

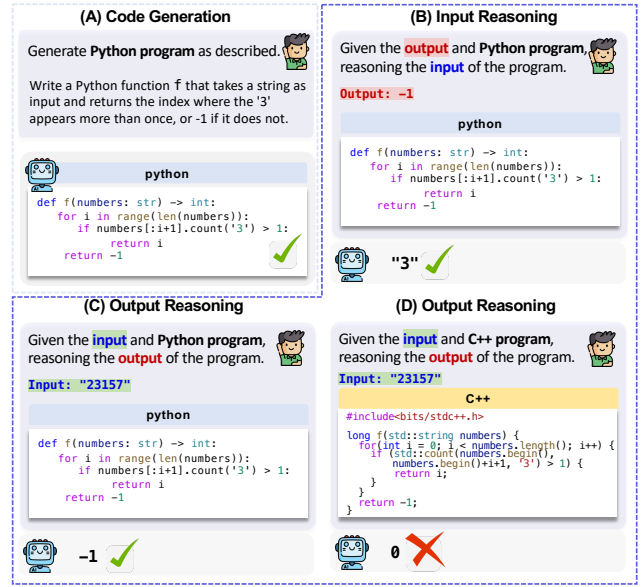


Figure 1: Code Generation vs. Code Reasoning

Wang, and Shang 2024; Hu et al. 2024). Benchmarks such as HumanEval (Chen et al. 2021) and SWE-bench (Jimenez et al. 2023) were introduced to measure LLMs' capabilities, providing insights into LLMs' strengths and weaknesses.

Recent studies (Cao et al. 2024a; Chai et al. 2024; Chen et al. 2024) have spotted two significant biases in current benchmarks. First, **Programming language bias**. As pointed out by prior studies (Cao et al. 2024a; Chai et al. 2024; Chen et al. 2021; Austin et al. 2021), *Python* dominates code generation benchmarks with over 95% involvement. Other programming languages (PLs) such as Java and C/C++, despite their popularity and availability, gain less exploration. Second, **Coding task bias**. Most coding benchmarks focus on code generation tasks (*i.e.*, giving descriptions in natural language and generating the program, as shown in Figure 1 (A)), while **code reasoning** (*i.e.*, given the program, reasoning the input or output of the program,

*These authors contributed equally.

as shown in Figure 1 (B - D)), as an essential coding capability of LLMs, is seldom evaluated (Chen et al. 2024). A recent work introduced a code reasoning benchmark (Gu et al. 2024; Chen et al. 2024), while it is only in Python. Figure 1 (C - D) shows that simply changing PLs from Python to C++ can turn a correct reasoning into an incorrect one.

However, constructing multi-lingual benchmarks is not a trivial task. First, *human annotation can be expensive*. As reported by recent work (Chai et al. 2024), they spent a total of \$12,000 US dollars for human annotators, providing the working environment, free meals, souvenirs, and free GPT-4 interface usage to construct their multi-lingual benchmark. Second, *automated translation does not perform well*. The latest studies (Yin et al. 2024) show that even the best LLM (*i.e.*, ChatGPT) can only achieve an average of 64% success translation rate, which is far from practice. Rule-based translation (Cassano et al. 2023; Ling et al. 2022) usually suffers from generalizability issues, making them limited in handling prescribed code structures. Additionally, multi-lingual solutions from contest websites such as LeetCode and Codeforces were included in most LLMs training sources, thus suffering from *data contamination issue* (Cao et al. 2024b).

To fill the research gaps, we introduce CRUXEVAL-X, a multi-lingual code reasoning benchmark that contains 19 popular PLs, including C++, Rust, Java, *etc.*, expanded from CruxEval (Gu et al. 2024), a code reasoning benchmark written in Python. For each PL in CRUXEVAL-X, there are at least 600+ functions. In total, there are 12,660 subjects along with 19K test cases for input/output reasoning.

Noteworthy that the pipeline of constructing CRUXEVAL-X works in a fully automated manner. It first translates the test cases by transition rules adapted from prior work (Cassano et al. 2023), then iterates the generation-and-repair process intensively. In particular, the transition rules are formulated to cross the language barriers. For example, Python employs a dynamically typed system where types are determined at runtime, whereas C++ uses a statically typed system requiring explicit type declarations at compile time. The rules facilitate the translation of the test cases. Additionally, inspired by prior work (Yin et al. 2024; Rozière et al. 2022), we employ a test-guided manner (Rozière et al. 2022) to generate the translation and iteratively repair the generated code using execution feedback (*e.g.*, compilation error, runtime error) (Yin et al. 2024).

Through intensive experiments on 24 mainstream LLMs, we observe several interesting findings. First, in multiple PLs, the input reasoning and output reasoning capabilities of LLMs are comparable. Also, there is a noticeable correlation between certain PLs (*e.g.*, JavaScript and TypeScript show a positive correlation, while Racket consistently yields the worst results). More interestingly, we observe that even if a model is only trained on Python (*e.g.*, phi-1 and phi-1.5), it still can reach a 16% ~ 26% output reasoning success rate in other PLs, compared with 25.6% in Python. The finding indicates the cross-language generalization of LLMs.

The contributions can be summarized as follows: (1) We introduce CRUXEVAL-X, a multi-lingual code reasoning benchmark that contains 19 popular PLs. (2) We introduce an automated code translation pipeline that adopts a test-

guided and iterative generate-and-repair practice. (3) We evaluate 20+ LLMs against CRUXEVAL-X and yield inspiring findings.

Benchmark Construction

In this section, we detailed the construction process of CRUXEVAL-X in Figure 2. It can be divided into three main steps. First, we translate the function signature via mapping variable type annotations (Step 1 in Figure 2). Then we employ a rule-based approach to translate Python test cases into other PLs (Step 2 in Figure 2). Finally, we integrate multiple LLMs to translate the code by iterating the generation-and-repair process (Step 3 in Figure 2).

Step I. Function Signature Translation

To enhance the accuracy and standardization of function translation results, we first translate the function signatures and dependencies. Note that Python does not require an explicit type annotation, which may confuse the translation for the function signature. For example, as shown in Figure 2 Step I, the types of two input parameters (*i.e.*, `s1` and `s2`) are unclear. So we extract the input variables from the function signature using the syntax tree and match them with the tests. Based on the variable types in the tests, we annotate the input and output variables in the function signature.

Then, we adopt the rules as prior work (Cassano et al. 2023) to map the types from Python to other PLs. In particular, we identify the data types in the annotated Python signature (*e.g.*, parameter types, return types), mapping the types from Python to other PLs according to the rules, then structuring the signature in the corresponding PLs. Take the example in Figure 2 Step I, the Python signature `def f(s1:str, s2:str) -> str` is translated into that in C (`std::string f(std::string s1, std::string s2)`). After translating the tests, all 800 subjects in Python can be translated, as shown in Table 1.

Step II. Test Suites Translation

We employ a test-guided approach to ensure the correctness of the translation results, which necessitates test cases in various PLs. Prior works (Athiwaratkun et al. 2022; Cassano et al. 2023) provided various rules for mapping Python test cases to other PLs. We adopt the mapping rules from MutiPL-E (Cassano et al. 2023) to assist the transition of our test suites.

However, their rules have limited support for type handling (*e.g.*, they cannot handle a list with hybrid types). Thus, to maximize the success rate, we made two improvements to enhance the rules. First, we enhance handling structured types such as `List` and `Dict`. For example, when handling C#, we add an equality function to check whether two `Dict` types are equal. Second, when dealing with variables that have complex types that are not as well-supported in some other PLs, we transform these variables into more generic types without significantly altering the original function’s functionality. For example, we change type `List[Union(str, int)]` into `List(str)` if the function keeps the same functionality. A small portion

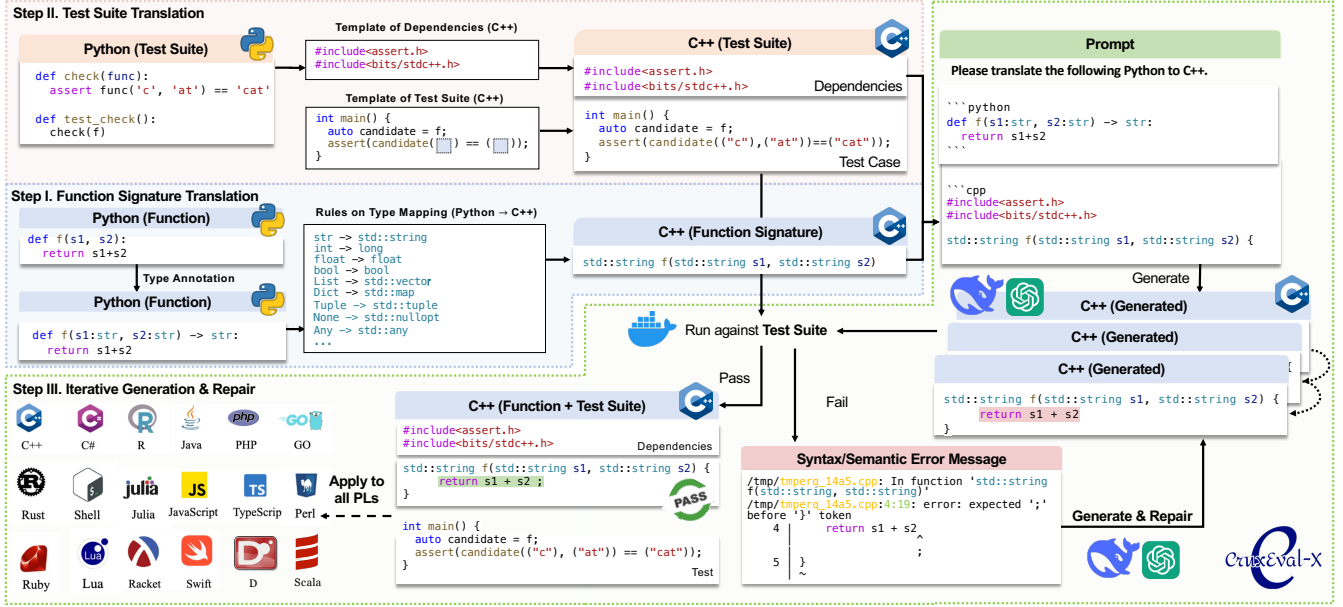


Figure 2: The Pipeline of CruxEval-X Construction.

of the data that cannot be converted is discarded. The result of Step II is shown in Table 1. Further details can be found in the Appendix.

Step III. Iterative Generation & Repair

After the test, dependencies, and signatures are properly transited, we then use multiple LLMs to iteratively translate the original Python reference code into corresponding PLs. In particular, each LLM undergoes two substeps: generation and repair. These substeps are based on the results of the previous LLM. Once a problem successfully passes the test cases, it will not appear in the next round of iteration. Below, we will elaborate on these two substeps in detail.

Generation. Relying solely on a single LLM’s limited number of generations is unlikely to achieve high accuracy in translation tasks (Yin et al. 2024). Therefore, we propose a multi-round generation method, which involves interaction with the testing environment to determine whether to proceed with the next round of iteration.

Let A_0 represent the number of translated codes that pass the tests and the number of total problems is U . For all questions that are not correctly answered, we used LLM M to generate results through multiple rounds. We denote the number of correct results in the i -th round as A_i , with the maximum number of generation rounds set as N . For A_i , if the increase in the number of correct questions compared to the results k rounds before A_{i-k} is less than the threshold δ , the process will stop early. Here, k is a fixed constant, and the formula is as follows:

$$A_i = \text{Correct}(P(O_i | U - A_{i-1}; M)) + A_{i-1} \quad (1)$$

for $i \in \{1, 2, \dots, N\}$

Stop if $(i > k)$ and $(A_i - A_{i-k} < \delta)$

We obtain the i -th round code result O_i from the code generation distribution $P(O_i | U - A_{i-1}; M)$. Then we calculate the correct results using $\text{Correct}(\cdot)$ as the function.

To fully leverage the strengths of various LLMs, we select the closed-source LLM GPT3.5-Turbo and the open-source LLM DeepseekCoder-33B-Instruct for iterative translation tasks. We initially use GPT3.5-Turbo for preliminary generation. Given the higher usage cost of closed-source LLMs, we set N to 5, k to 5, δA to 0, and the temperature to 0.2. The result is shown in Table 1 under the column “w/o Iter”. We also use this as a baseline for the effectiveness of single LLM’s generation within a limited number of iterations to compare with the final results of our pipeline. Subsequently, we employ Deepseekcoder-33B-Instruct on top of the foundation laid by GPT3.5-Turbo to conduct further generation. We set N to 50, k to 5, δA to 0, and the temperature to 0.8.

Repair Simply generating code will still result in many errors the LLM cannot solve. Therefore, after the generation step of each LLM, we provide them error messages for error correction. We observed that the cost of multiple iterative error corrections is high and the benefits are relatively low (Chen et al. 2023), so we only perform error correction once after each of the two LLMs.

After the generation of GPT3.5-Turbo, we directly provide the LLM with the erroneous code along with the error messages for correction. After the Iterating Generation of DeepseekCoder-33B-Instruct, since the untranslated code can produce numerous incorrect code snippets after multiple rounds of iteration, which may contain the same errors, we first use simhash to deduplicate the erroneous code and then proceed with error correction on the deduplicated code. Error correction in different phases uses the LLM employed in that specific phase, with a temperature setting of 0.

Multiturn Repair based on overlap After completing the steps above, we first calculate the intersection of correctly answered questions across different PLs. To our surprise, there were only 333 questions that all PLs answered correctly. However, 563 problems have been successfully translated correctly by at least 16 PLs. Upon analyzing the questions our LLM failed to solve, we find that each PL has its difficulties in translating from python. For example, in Julia, the index for arrays and other collection types starts from 1, which differs from Python. The details of the difficulties can be found in Appendix.

Based on these observations, we conduct final generation and iterative error correction on the questions that are correctly translated by more than 15 PLs. Due to the small number of questions requiring translation, we utilized GPT-4o for generation and error correction. We set the temperature to 0, generate once, and correct errors three times. During the generation process, we provide the LLM with three corresponding typical examples based on the difficulties we find. The overlap is increased to 462 after repair of GPT-4o.

We manually modified 38 questions that GPT-4o almost got right, expanding our dataset to 500 entries. We determined that 500 entries are sufficient to distinguish the effectiveness of the LLMs and most of the questions that failed to be translated cannot be expressed well in other PLs. Therefore, we use these 500 entries as our CRUXEVAL-X benchmark. The final result of our pipeline is shown in Table 1 under the column “w/ Iter”. The prompt of each step can be found in Appendix.

Languages	Step I	Step II	Step III	
			w/o Iter	w/ Iter
C# (cs)	800	774	380	670
C++ (cpp)	800	800	549	733
D (d)	800	754	95	629
GO (go)	800	752	293	699
Java (java)	800	774	541	698
JavaScript (js)	800	800	634	743
Julia (jl)	800	774	410	680
Lua (lua)	800	800	582	741
Perl (pl)	800	799	591	728
PHP (php)	800	800	622	755
R (r)	800	800	542	699
Racket (rkt)	800	800	264	681
Ruby (rb)	800	800	658	748
Rust (rs)	800	754	449	690
Scala (scala)	800	799	462	712
Shell (sh)	800	763	528	674
Swift (swift)	800	796	415	654
TypeScript (ts)	800	774	592	726

Table 1: The result of each step, The portion within parentheses in the “Language” column represents the abbreviations for various languages. Due to the constraints of page size, these abbreviations are used to better display certain charts or tables.

Experiments

Experiment Setup

LLMs for evaluation We select 24 LLMs across 4 types for evaluation, including *general* LLMs (GPT-3.5-Turbo, GPT-4o-mini, GPT-4o (Brown et al. 2020; Achiam et al. 2023), Llama3 (AI 2024), Qwen2 (Yang et al. 2024a), phi-3-instruct (Abdin et al. 2024)), *multilingual code* LLMs (Deepseekcoder-V2 (Zhu et al. 2024), Deepseekcoder-V1 (Guo et al. 2024), CodeLlama (Roziere et al. 2023), Starcoder (Li et al. 2023a), Starcoder2 (Lozhkov et al. 2024), CodeQwen1.5-Chat (Bai et al. 2023)), *instruction-tuned multilingual* LLMs (Deepseekcoder-instruct-V1 (Guo et al. 2024), WizardCoder (Luo et al. 2023), CodeLlama-Python, CodeLlama-Instruct (Roziere et al. 2023)), and *single or few-language code* LLMs (CodeGen (Nijkamp et al. 2022), phi-1 (Gunasekar et al. 2023), phi-1.5 (Li et al. 2023b)).

Evaluation task We adopt the task settings from prior work (Gu et al. 2024), dividing the tasks into output reasoning and input reasoning. For any PLs dataset $L^k \in \{L_i\}_{i=1}^K$, where $K = 19$, representing the total number of PLs. We provide a function f^{L_k} and the corresponding test. The input of this test example is i^{L_k} , and the output is o^{L_k} .

The output reasoning task can be expressed as:

$$r^{L_k} = I(P(o^{L_k} | f^{L_k}, i^{L_k}, M)) \quad (2)$$

The input reasoning task can be expressed as:

$$r^{L_k} = I(P(i^{L_k} | f^{L_k}, o^{L_k}, M)) \quad (3)$$

where M is any LLMs. We get the input or output from the code generation distribution $P(\cdot)$ and compose test cases $(i, o)^{L_k}$. $I(\cdot)$ is the indicator function by executing this test case with function f^{L_k} . If f^{L_k} passes the test, the evaluation result is 1, otherwise 0.

Evaluation method. We use pass@1 (Kulal et al. 2019; Chen et al. 2021) as the evaluation metric to assess both the task of output reasoning and input reasoning. We set the temperature to 0 and employ greedy decoding for generation as prior work (Cao et al. 2024a). For closed-source LLMs, we generate outputs by calling OpenAI’s API. The version of the api is *gpt-3.5-turbo*, *gpt-4o-mini*, *gpt-4o*.

Overall Result.

Figure 3 shows Pass@1 evaluation results of various LLMs arranged in descending order of the LLMs’ parameter size. It can be observed that LLMs with a larger number of parameters tend to perform better. The closed-source LLMs GPT-4o achieved the best results among all evaluated LLMs. Notably, the result of the open-source LLMs Deepseekcoder-V2 is better than GPT-4o-mini, with an average Pass@1 of 62.8% on input reasoning and 65.0% on output reasoning. From the perspective of average values, it can be observed that under different types of PLs, the LLMs’ capabilities on input and output reasoning are quite similar, which echos the observation made by prior work (Gu et al. 2024) in Python. More interestingly, during the evaluation, we introduced the few-language LLMs CodeGen-muti, which was only trained on C, C++, GO, Java, JavaScript, and Python, as well as the

Input Reasoning Performance																				
Models	Size	cs	cpp	d	go	java	js	jl	lua	pl	php	py	r	rkt	rb	rs	scala	sh	swift	ts
GPT-4o	-	70.2	64.6	71.6	75.4	69.8	73.2	67.0	73.0	70.2	74.8	70.6	74.4	67.4	72.0	73.6	65.4	70.6	74.2	74.0
GPT-4o-mini	-	58.8	52.2	60.6	62.0	57.2	59.6	56.2	63.4	57.4	61.0	59.6	60.4	51.2	61.6	61.2	52.6	57.2	63.4	61.2
GPT-3.5 Turbo	-	52.2	39.2	50.2	53.4	55.4	50.0	47.0	53.2	47.6	52.2	51.6	48.6	45.4	49.6	53.0	54.2	47.6	58.2	48.4
Deepseekcoder-V2	236B	63.8	57.0	66.6	64.0	64.8	67.0	58.4	62.0	61.4	64.2	64.0	65.8	58.0	63.2	63.6	58.2	62.4	62.6	66.6
Qwen2-Instruct	72B	52.0	54.2	49.6	55.4	50.0	51.6	51.0	51.2	47.8	55.2	52.4	53.2	47.8	54.4	57.2	50.6	52.4	51.6	52.0
CodeLlama-Python	34B	38.8	40.0	39.2	39.0	41.4	45.8	44.8	45.0	43.2	48.0	46.8	42.2	38.8	44.0	44.2	43.0	44.6	45.0	44.0
CodeLlama-Instruct	34B	44.6	48.4	43.8	46.0	44.4	52.6	50.4	49.4	46.0	52.0	51.2	48.4	42.4	48.2	48.6	48.0	46.2	49.4	53.2
CodeLlama	34B	40.4	44.6	45.6	41.2	39.0	50.0	49.0	47.0	46.6	48.8	49.8	47.6	39.8	46.6	46.8	44.6	44.4	50.0	48.6
WizardCoder-V1.1	33B	44.8	25.4	46.4	47.6	48.4	45.6	49.2	48.8	44.6	50.0	50.0	45.0	42.4	49.2	48.2	48.2	45.4	51.0	46.4
Deepseekcoder-instruct	33B	46.0	43.6	49.8	49.0	46.8	48.8	47.0	50.0	46.8	52.0	51.8	48.2	41.6	52.0	48.4	47.0	48.2	52.2	49.6
Deepseekcoder-base	33B	41.2	42.8	43.2	45.6	43.8	46.0	47.6	47.4	47.2	48.6	49.2	50.6	42.8	47.4	46.8	44.0	46.4	48.2	45.0
Starcoder2	15B	41.4	43.8	51.6	45.2	42.6	44.0	48.2	44.6	44.8	49.8	46.6	45.8	45.0	49.0	46.6	37.0	47.4	52.2	46.2
WizardCoder-V1.0	15B	29.2	30.0	30.6	28.6	29.6	33.0	34.8	33.6	36.2	36.8	33.2	33.4	36.4	33.6	33.0	29.0	35.0	34.0	32.4
Starcoder	15B	28.2	30.0	33.0	33.2	33.4	35.2	34.4	31.6	34.0	36.4	34.8	33.4	36.6	35.0	34.8	27.4	37.0	30.8	33.2
phi-3-instruct	14B	31.8	26.0	38.8	36.4	37.2	42.4	36.2	37.2	35.6	41.2	43.4	39.2	24.4	36.0	36.8	38.0	33.6	41.2	42.8
Llama-3-Instruct	8B	37.0	36.4	35.0	38.6	36.2	38.4	39.6	40.0	36.2	36.6	38.4	42.2	24.2	35.8	37.6	38.0	31.6	42.2	38.2
CodeQwen1.5-Chat	7B	42.8	42.0	43.0	46.4	44.6	43.8	42.2	42.8	41.6	44.8	43.0	43.4	38.2	43.6	42.0	39.4	46.6	45.8	43.6
CodeLlama-Instruct	7B	38.6	36.0	38.4	38.4	38.2	39.6	42.2	43.4	36.4	40.4	41.0	41.0	38.8	41.6	37.6	42.6	39.6	40.2	41.0
CodeLlama	7B	36.4	36.2	36.8	34.6	36.4	36.6	40.2	39.6	36.0	39.4	40.2	40.0	36.6	39.2	35.4	37.8	36.8	39.2	38.8
Deepseekcoder-instruct	6.7B	35.0	37.0	35.6	40.4	35.0	36.6	39.2	38.8	39.4	42.2	38.2	42.0	37.2	40.2	37.4	36.8	42.8	40.8	34.2
Deepseekcoder-base	6.7B	38.8	42.4	41.2	43.2	40.4	43.6	42.6	42.8	41.6	46.4	41.4	46.2	43.0	44.6	41.6	40.8	44.8	43.4	41.8
CodeGen-multi	6B	28.8	25.4	6.2	25.6	36.2	25.2	17.4	24.4	38.4	22.8	22.6	27.2	16.2	6.4	18.8	31.0	48.6	32.4	25.2
phi-1.5	1.3B	29.2	16.0	13.2	25.8	26.8	9.8	30.4	26.6	17.8	26.6	25.8	8.4	6.6	1.4	25.2	30.4	34.4	26.6	30.8
phi-1	1.3B	0.2	7.0	9.6	3.6	2.8	17.0	19.0	17.4	23.6	9.2	11.8	9.4	11.2	6.8	5.4	1.8	19.8	14.0	14.0
Average		40.4	38.3	40.8	42.4	41.7	43.1	43.1	43.9	42.5	45.0	44.1	43.2	38.0	41.7	42.7	41.1	44.3	45.4	43.8

Output Reasoning Performance																				
Models	Size	cs	cpp	d	go	java	js	jl	lua	pl	php	py	r	rkt	rb	rs	scala	sh	swift	ts
GPT-4o	-	75.0	74.8	71.4	77.0	73.2	77.6	73.6	74.8	74.0	75.4	75.4	72.0	70.8	74.0	74.4	71.8	71.6	76.0	76.4
GPT-4o-mini	-	63.0	63.0	61.4	63.4	54.0	61.8	57.8	60.0	57.4	64.2	61.6	59.6	56.6	61.2	61.8	61.2	56.2	63.0	61.2
GPT-3.5 Turbo	-	54.2	43.2	56.0	53.2	43.6	56.2	54.2	54.6	51.8	55.2	57.2	49.4	48.0	56.4	54.6	56.4	51.0	57.8	53.6
Deepseekcoder-V2	236B	66.6	66.2	63.4	68.0	67.6	65.4	64.8	63.6	63.0	67.4	66.8	63.0	62.2	65.2	65.8	63.2	58.8	67.8	66.4
Qwen2-Instruct	72B	51.2	50.2	51.6	53.6	38.2	52.0	51.0	49.0	45.8	50.8	51.2	45.0	46.8	50.8	51.0	51.0	45.6	50.4	53.2
CodeLlama-Python	34B	41.4	44.8	45.6	41.8	41.4	45.4	45.2	42.8	43.6	43.8	43.8	42.4	38.6	42.8	46.6	43.8	42.0	44.4	44.8
CodeLlama-Instruct	34B	44.4	46.2	45.8	46.8	40.6	47.4	45.6	42.8	44.0	44.8	44.0	40.2	38.2	44.2	46.4	43.8	40.6	45.2	45.0
CodeLlama	34B	44.6	47.8	44.2	45.2	38.4	47.0	45.8	42.8	43.8	46.4	46.4	38.8	38.4	45.4	47.2	47.4	43.8	47.6	47.4
WizardCoder-V1.1	33B	47.0	46.8	45.8	44.2	50.8	50.0	47.0	46.0	45.2	51.4	49.6	44.0	42.4	48.2	47.8	45.0	44.4	48.0	49.8
Deepseekcoder-instruct	33B	52.0	51.4	49.0	48.8	53.2	55.0	50.4	50.4	50.0	53.0	52.2	48.2	46.6	52.8	50.6	48.0	49.4	52.8	53.6
Deepseekcoder-base	33B	48.2	50.0	46.0	48.6	49.2	51.4	46.8	48.0	48.4	52.0	49.8	45.2	46.4	49.0	46.2	47.6	46.0	49.2	51.2
Starcoder2	15B	46.0	47.4	47.2	49.0	48.4	50.0	49.2	44.8	49.4	48.4	48.4	47.2	45.0	51.0	48.8	45.2	45.8	49.6	48.6
WizardCoder-V1.0	15B	25.2	30.0	30.6	33.2	26.8	33.6	30.2	30.2	31.0	33.0	34.0	31.6	29.6	32.8	31.2	31.2	29.8	34.2	34.0
Starcoder	15B	20.4	31.6	31.8	31.0	18.4	33.4	32.2	31.8	29.8	32.6	32.6	30.0	29.2	33.4	32.6	30.0	30.2	33.0	33.0
phi-3-instruct	14B	34.2	37.6	39.0	31.0	34.2	41.6	41.2	34.4	35.8	37.8	42.4	36.6	24.6	42.2	37.4	36.2	37.2	41.4	43.0
Llama-3-Instruct	8B	32.0	30.8	31.2	31.4	25.0	35.0	31.4	34.0	29.6	27.0	33.6	27.2	28.0	31.8	34.4	33.8	32.0	36.4	33.8
CodeQwen1.5-Chat	7B	37.8	40.2	40.2	40.6	35.4	43.6	42.6	40.4	39.6	43.0	41.4	38.2	39.0	44.6	42.0	35.0	38.2	43.8	42.2
CodeLlama-Instruct	7B	32.2	35.6	34.4	35.0	24.4	38.2	35.2	32.2	34.2	36.0	35.4	32.0	29.6	37.0	37.4	33.0	33.0	34.6	38.8
CodeLlama	7B	32.6	34.4	33.8	33.4	28.4	38.0	35.2	34.4	35.2	38.0	34.4	32.6	30.8	34.8	36.8	33.4	31.0	35.0	38.2
Deepseekcoder-instruct	6.7B	34.8	41.8	40.4	39.4	32.8	47.6	42.6	38.8	42.0	43.8	43.6	40.8	39.2	43.2	41.8	40.6	37.8	43.2	44.0
Deepseekcoder-base	6.7B	41.2	46.2	43.2	42.8	42.6	44.8	46.0	41.0	40.4	41.8	44.8	42.8	43.0	42.6	42.0	43.2	40.6	47.6	45.4
CodeGen-multi	6B	21.4	23.6	25.0	26.4	21.6	22.8	22.8	23.8	20.4	25.2	24.8	23.4	17.8	24.0	25.2	22.0	22.2	25.0	21.4
phi-1.5	1.3B	16.0	26.0	24.8	22.6	15.8	23.0	23.6	21.2	22.0	22.2	25.6	21.8	16.8	19.6	22.0	21.6	17.6	25.6	25.2
phi-1	1.3B	5.8	9.0	13.2	14.8	4.6	20.8	19.2	15.8	15.6	18.6	22.4	17.6	10.4	18.0	16.4	11.0	16.4	19.2	19.0
Average		40.3	42.4	42.3	42.6	37.9	45.1	43.1	41.6	41.3	43.8	44.2	40.4	38.3	43.5	43.4	41.5	40.1	44.6	44.6

Figure 3: The result of each LLM in CRUXEVAL-X. Each result is shaded with a background color from blue to white based on the Pass@1. The bluer, the larger. Deepseekcoder-V2 is a MOE LLM; the parameter activated during inference is 21B.

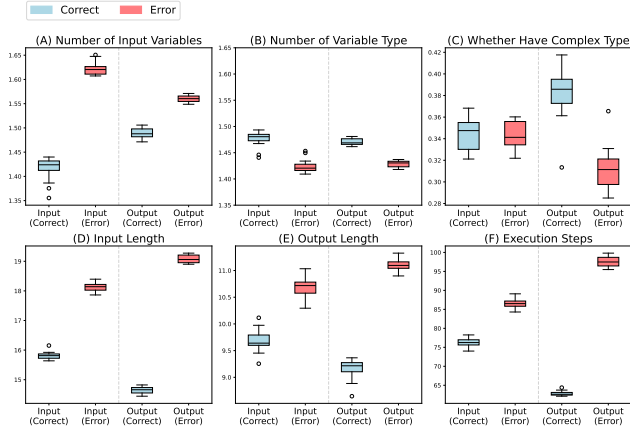


Figure 4: Key Factors for LLM Code Reasoning Capability.

single-language LLMs *phi-1.5* and *phi-1*, which were pre-trained only on Python. However, they achieve similar results in the input and output reasoning tasks across 19 PLs. Notably, *phi-1*, having only seen Python, achieved a Pass@1 of 11.8% on Python input prediction, but scored 23.6% on Perl input prediction. We will provide a more detailed analysis of this phenomenon in Analysis part.

Analysis on Key Factors for LLM Code Reasoning

To get more insight into what factors in code affect LLMs’ code reasoning ability, we explore six factors (*e.g.*, average number of input variables, average input length) and statistics their correlation with correct/incorrect reasoning. The result is shown in Figure 4. Each column of the box plot displays the distribution of 19 PLs. In particular, the columns “Num of Input Variable”, “Num of Variable Type” are counted by averaging the number of types of input parameters in method signatures, “Input/Output Length” is the average string length of the input/output. For instance, in the test case where `f("a", 123) == 123`, the input length is 4, and the output length is 3. “Whether Have Complex Type” checks whether there are List, Dict, Tuple, Set types in input and output. “Execution Steps” calculates the average execution steps in Python bytecode operations, following prior work (Gu et al. 2024).

From Figure 4, we can see from Sub-figures A-C that *the number/types of input variables have little impact* on the code reasoning, especially Sub-figure B, which shows that the reasoning capability is slightly better when more types of variables are involved. A more counter-intuitive observation is made from sub-figures D-E. They indicate that the reasoning capability *is negatively correlated with the length of input/output strings* instead of the number of data types.

Furthermore, regarding input reasoning capability, the more input variables, the more challenging it is for LLMs to reason about the correct inputs, thus the worse the input reasoning performance (Sub-figures A and D). Similarly, the longer the outputs, the harder the output reasoning, resulting in worse performance (Sub-figure E).

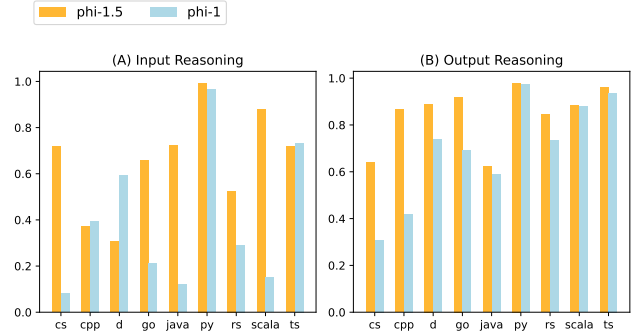


Figure 5: The number of syntax error of each LLM.

Analysis on Cross-language Generalization

To investigate the cross-language generalizability of LLMs, we investigate the reasoning ability of *phi-1* and *phi-1.5*, which are trained on English and Python only. To get a better understanding, we analyze the capability in terms of *syntax* and *semantics* in 9 PLs because they provide clear error messages to distinct syntactic/semantic errors.

Syntactic Correctness. To generalize to other PLs, it is critical to ensure syntactic correctness. Figure 5 (A)-(B) show the number of syntactic-correct cases made by these two LLMs in both tasks. It is clear that *Python* has the highest syntactic correctness in both tasks, followed by *Go* and *TypeScript*. On the contrary, C++, C#, and Java witness the most syntactic errors for three LLMs. Interestingly, even though *phi-1* and *phi-1.5* have not trained on PLs other than Python, they can still achieve an average of 49.1% and 72.0% syntactically correctness rate in other PLs, respectively, compared with 97.0% and 98.7% achieved on Python. It indicates *the cross-language generalizability* of LLMs.

Semantic Correctness. Beyond syntactic correctness, semantic correctness poses higher requirements, *i.e.*, passing the tests. The results are shown in the last two rows in Figure 3. Similar results can be observed in both tasks and both LLMs. In particular, *phi-1.5* reaches 25.8% input reasoning performance on Python, while on other PLs, an average of 19.0% can also be reached. The observation further consolidates *the cross-language generalizability* of LLMs.

Cross-NL and Cross-PL Generalization. From Figure 5 and Figure 3, there is a noticeable increase from *phi-1* to *phi-1.5* (an average of 10.7% vs. 21.7% on input reasoning, and 15.1% vs. 21.7% on output reasoning). According to the description (Abdin et al. 2024), *phi-1.5* is further fine-tuned with more synthetic texts in *natural language* (NL). Considering the dramatic improvement in code reasoning, it is highly likely that the improvement in NL reasoning positively impacts code reasoning.

Analysis on Programming Language Correlation

To further investigate the correlations between these 19 PLs in CRUXEVAL-X, we calculate each PL pair’s correlation (*i.e.*, cosine similarity, ranging from -1 to 1), visualized in

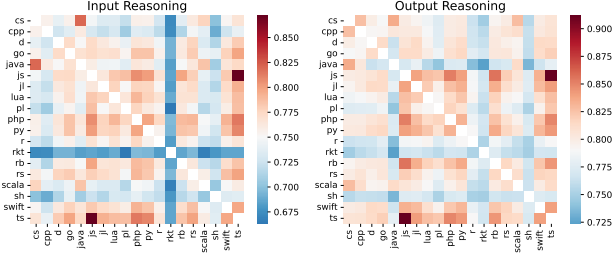


Figure 6: The Correlation between PL Pairs.

Figure 6. In particular, for each PL, we flatten the results of LLMs as a feature vector and calculate the cosine similarities for each pair of PLs.

Overall, Figure 6 shows that the correlation between PL pairs is generally similar, with an average of 0.7+ cosine similarities. Among all PL pairs, **JavaScript and TypeScript correlate the most strongly** (0.87 and 0.91 on both tasks). It indicates that the *code reasoning capabilities on different PLs are highly correlated*. Also, the correlation in output reasoning is slightly higher than in input reasoning, with an average of 0.79 vs. 0.75.

It is also noteworthy that Racket has the most minor correlation with all the other PLs. It may be because of its *distinct syntax*. A case study can be found in Listing 3.

Case Study

After identifying the phi-series-LLMs (*i.e.*, phi-1, phi-1.5) exhibit cross-language generalization and the correlation across PLs, we further analyze the predictions of phi-1.5 to get a deeper understanding. We noticed that out of the 128 correct instances in Python by phi-1.5, 61.7% (79/128) are also correct in PHP, while only 39.8% (51/128) are correct in Racket. Therefore, we use one example in these three PLs (Python, PHP, and Racket) to understand its rationale.

Analysis on Subject 106. Listing 1-3 demonstrates an instance where phi-1.5 generalizes Python (Listing 1)’s reasoning capabilities to other languages. Each example’s check function has been uniformly simplified to `assert f() == ()`. From a grammar structure perspective, their respective function definitions, indentation formats, and the functions they invoke exhibit significant differences. However, overall, PHP and Python share a more similar structure, both utilizing `sort` for sorting and `return` for returning output values. Therefore, phi-1.5 is able to generalize its code reasoning abilities to PHP, but fails to comprehend the sorting command in Racket, leading to incorrect predictions.

Upon analyzing these 128 questions, we observe that excluding those where the output could be directly derived from the input, such as `assert f("zej", "owc") == "zej"`, which accounted for approximately 40% of the cases, there are still numerous examples demonstrating that phi-1.5 has developed a certain level of cross-language capabilities. From these examples, we can observe that the multilingual generalization capability of the model is positively correlated with the grammar structural similarity between

languages. Even Racket, a language significantly different from others, maintains certain logical similarities in aspects such as function definitions, loops, and conditional branches. This is a key reason why Phi-1.5 can achieve considerable effectiveness across multiple languages.

Listing 1: Subject-106 (Python)

```
1 def f(lst: List[int]) -> List[int]:
2     lst.sort()
3     return lst[0:3]
4 assert f([5,8,1,3,0])==????
5 # phi-1.5 answer: [0, 1, 3]
```

Listing 2: Subject-106 (PHP)

```
1 <?php
2 function f($lst) {
3     sort($lst);
4     return array_slice($lst, 0, 3);
5 }
6 assert f(array(5,8,1,3,0)) == ???
7 // phi-1.5 answer: array(0, 1, 3)
```

Listing 3: Subject-106 (Racket)

```
1 (define (f lst)
2   (define sorted-list (sort lst <))
3   (take sorted-list 3))
4 (require rackunit)
5 assert f(list 5 8 1 3 0) == ???
6 # phi-1.5 answer: list 5 8 1
```

Related Work

Multi-Task Code Benchmark Recently, there has been an increasing number of tasks related to code that are used to evaluate the various capabilities of LLMs in the field of coding, including code generation (Chen et al. 2021; Austin et al. 2021), code repair (Jimenez et al. 2023; Tian et al. 2024), and code description (Chai et al. 2024). However, datasets that assess the reasoning abilities of code are relatively limited, and the currently proposed reasoning datasets are confined to the Python language (Gu et al. 2024; Chen et al. 2024). In this work, we have expanded the Python language reasoning dataset CRUXEVAL (Gu et al. 2024) to encompass 19 PLs, thereby addressing the deficiency in reasoning datasets at the multilingual level.

Multi-Language Code Benchmark Multilingual evaluation datasets are an important method for assessing the comprehensive coding capabilities of code LLMs. In the early stages, multilingual code datasets were mainly used for code translation tasks (Elnaggar et al. 2021; Ahmad et al. 2021; Roziere et al. 2020, 2021; Zhu, Suresh, and Reddy 2022; Yan et al. 2023; Zhu et al. 2022). These datasets often consist of problem solutions in different languages extracted from algorithm competition-related websites, thus suffering from data contamination issue. Benchmark like McEval (Chai et al. 2024), which relies on human annotation, requires a

high cost. In this work, we provide a process using LLMs for multilingual code translation, which can achieve a high accuracy and low cost in creating a multilingual dataset.

Conclusion

In this work, we provide a fully automated process for constructing a multilingual dataset based on a Python code language dataset. Through this process, we successfully transform the CRUXEval dataset into a multilingual dataset containing 19 PLs and test its effectiveness on 24 LLMs, demonstrating the validity of the dataset. Furthermore, we find that models trained on only a few languages exhibit the ability to transfer their prediction capabilities to other languages in input/output reasoning tasks, and this ability is influenced by the model's own reasoning capabilities.

References

- Abdin, M.; Jacobs, S. A.; Awan, A. A.; Aneja, J.; Awadallah, A.; Awadalla, H.; Bach, N.; Bahree, A.; Bakhtiari, A.; Behl, H.; et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.
- Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Ahmad, W. U.; Tushar, M. G. R.; Chakraborty, S.; and Chang, K.-W. 2021. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*.
- AI, M. 2024. Introducing meta llama 3: The most capable openly available llm to date. Blog. Online; accessed 15-January-2024.
- Athiwaratkun, B.; Gouda, S. K.; Wang, Z.; Li, X.; Tian, Y.; Tan, M.; Ahmad, W. U.; Wang, S.; Sun, Q.; Shang, M.; et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.
- Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.
- Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.
- Cao, J.; Chen, Z.; Wu, J.; chi Cheung, S.; and Xu, C. 2024a. Can AI Beat Undergraduates in Entry-level Java Assignments? Benchmarking Large Language Models on JavaBench. *arXiv:2406.12902*.
- Cao, J.; Zhang, W.; Cheung, S.-C.; and on, S. 2024b. Concerned with Data Contamination? Assessing Countermeasures in Code Language Model. *arXiv:2403.16898*.
- Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.-H.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7): 3675–3691.
- Chai, L.; Liu, S.; Yang, J.; Yin, Y.; Jin, K.; Liu, J.; Sun, T.; Zhang, G.; Ren, C.; Guo, H.; et al. 2024. McEval: Massively Multilingual Code Evaluation. *arXiv preprint arXiv:2406.07436*.
- Chen, J.; Pan, Z.; Hu, X.; Li, Z.; Li, G.; and Xia, X. 2024. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? *arXiv:2403.16437*.
- 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Chen, X.; Lin, M.; Schärli, N.; and Zhou, D. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.
- Du, X.; Liu, M.; Wang, K.; Wang, H.; Liu, J.; Chen, Y.; Feng, J.; Sha, C.; Peng, X.; and Lou, Y. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.
- Elnaggar, A.; Ding, W.; Jones, L.; Gibbs, T.; Feher, T.; Angerer, C.; Severini, S.; Matthes, F.; and Rost, B. 2021. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.
- Gu, A.; Rozière, B.; Leather, H.; Solar-Lezama, A.; Synnaeve, G.; and Wang, S. I. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.
- Gunasekar, S.; Zhang, Y.; Aneja, J.; Mendes, C. C. T.; Del Gorno, A.; Gopi, S.; Javaheripi, M.; Kauffmann, P.; de Rosa, G.; Saarikivi, O.; et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.
- Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.
- Hu, X.; Kuang, K.; Sun, J.; Yang, H.; and Wu, F. 2024. Leveraging print debugging to improve code generation in large language models. *arXiv preprint arXiv:2401.05319*.
- Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Kulal, S.; Pasupat, P.; Chandra, K.; Lee, M.; Padon, O.; Aiken, A.; and Liang, P. S. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.
- Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al.

- 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.
- Li, Y.; Bubeck, S.; Eldan, R.; Del Giorno, A.; Gunasekar, S.; and Lee, Y. T. 2023b. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*.
- Ling, M.; Yu, Y.; Wu, H.; Wang, Y.; Cordy, J. R.; and Hassan, A. E. 2022. In Rust We Trust – A Transpiler from Unsafe C to Safer Rust. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 354–355.
- Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.
- Lozhkov, A.; Li, R.; Allal, L. B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.
- Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.
- Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.
- Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Roziere, B.; Lachaux, M.-A.; Chatussot, L.; and Lample, G. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33: 20601–20611.
- Rozière, B.; Zhang, J.; Charton, F.; Harman, M.; Synnaeve, G.; and Lample, G. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.
- Roziere, B.; Zhang, J. M.; Charton, F.; Harman, M.; Synnaeve, G.; and Lample, G. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*.
- Tian, R.; Ye, Y.; Qin, Y.; Cong, X.; Lin, Y.; Liu, Z.; and Sun, M. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*.
- Xia, C. S.; and Zhang, L. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385*.
- Yan, W.; Tian, Y.; Li, Y.; Chen, Q.; and Wang, W. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint arXiv:2310.04951*.
- Yang, A.; Yang, B.; Hui, B.; Zheng, B.; Yu, B.; Zhou, C.; Li, C.; Li, C.; Liu, D.; Huang, F.; et al. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.
- Yang, A. Z.; Le Goues, C.; Martins, R.; and Hellendoorn, V. 2024b. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–12.
- Yin, X.; Ni, C.; Nguyen, T. N.; Wang, S.; and Yang, X. 2024. Rectifier: Code Translation with Corrector via LLMs. *arXiv:2407.07472*.
- Zhong, L.; Wang, Z.; and Shang, J. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.
- Zhu, M.; Jain, A.; Suresh, K.; Ravindran, R.; Tipirneni, S.; and Reddy, C. K. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.
- Zhu, M.; Suresh, K.; and Reddy, C. K. 2022. Multilingual code snippets training for program translation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, 11783–11790.
- Zhu, Q.; Guo, D.; Shao, Z.; Yang, D.; Wang, P.; Xu, R.; Wu, Y.; Li, Y.; Gao, H.; Ma, S.; et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931*.

Appendix

The improvements of MutiPL-E pipeline

enhance of the pipeline (1) For C# we enhance the check function to include the ability to judge if the List and Dict types are equal. (2) For Julia we add the data type to empty dict, for example change input Dict() to Dict{String, String}(). Otherwise, the function can not accept the input (3) For JavaScript we change String which Contains special characters, such as "example\n" to `example\n `.

transform complex types (1) For inputs and outputs that include functions, such as "bfrerat".split("-"), we will replace the input or output with the result after the function execution. (2) When the input is of Callable type, such as lambda x: x.reverse(), we will remove the parameter and incorporate the Callable type into the main function internally. (3) For complex variables that contain multiple types, if we can convert them into a simpler type without altering the function's functionality, we will preserve such functions. For instance, as illustrated in Listing 4, consider a dictionary d: Dict[str, Union[int, str]]. If converting all its values to the str type does not alter the function's behavior, we will retain it; otherwise, we will discard them.

Listing 4: example with complex type

```
1 from typing import Dict, Union, Tuple
2
3 def f(d: Dict[str, Union[int, str]]) ->
  Tuple[bool, bool]:
4     r = {
5         "c": d.copy(),
6         "d": d.copy()
7     }
8     return (r["c"] is r["d"], r["c"] ==
              r["d"])
9
10 def check(candidate):
11     assert candidate({"i": 1, "love": "
    parakeets"}) == (False, True)
12
13 def test_check():
14     check(f)
```

The difficulties for translating

(1) For indexing functions such as 'index', the starting position is not 0 but 1, with Julia being a typical language that exhibits this behavior. (2) For the conversion between Python's 'str' type and its own 'char' and 'string' types, D language is a typical case where this issue arises. (3) For the transformation and comparison of dictionary types, C# is a typical language where such errors are common.

Prompt of Benchmark Construction

In Figures 7 and 8, we include the prompts we use for our benchmark construction. We use a few-shot prompt for all models. For Generation step of each model, the prompt is

show in Figure 7. The example in this Figure is used for GPT3.5-turbo. After the Generation and Repair of GPT3.5-turbo, we choose three examples from the correct generated problems which include str, List, Dict type respectively. We use these examples for Deepseekcoder-instruct-33b Generation. For GPT-4o, based on the summarized difficulties in translation, we provide three examples, as shown in Listing 5, 6, and 7. we present these examples in three distinct languages. For each specific language translation, we employ the corresponding language version of the three examples.

<pre>===== System ===== You are a helpful programming assistant designed to translate code and complete code snippets. ===== User ===== Please translate the python function to cpp function: '''python def add(a, b): return a + b ''' The function starts as follows, and your task is to complete it so that its semantics are the same as the python code above. Note that the number of packages called at the beginning of the given function cannot be reduced, but can only be increased. '''cpp #include<assert.h> #include<bits/stdc++.h> long add(long x, long y) { ''' ===== Assistant ===== '''cpp #include<assert.h> #include<bits/stdc++.h> long add(long x, long y) { return x + y; } ''' ===== User ===== Please translate the python function to cpp function: '''python {the python code} ''' The function starts as follows, and your task is to complete it so that its semantics are the same as the python code above. Note that the number of packages called at the beginning of the given function cannot be reduced, but can only be increased. '''cpp {the cpp function head} '''</pre>
--

Figure 7: Prompt of Generation

```

===== System =====
You are an expert programming assistant.

===== User =====
Please translate the language of the function from python to
cpp

```python
{the python code}
```

===== Assistant =====
```cpp
{the error cpp code}
```

===== User =====
The code you translated has the following {error type} error:

{error message}

Please analyze the cause of the error and then return the
repaired code in cpp.

```

Figure 8: Prompt of Repair

As illustrated in Figure 8, the prompt for Repair is shown, which depicts a single round of error correction. The compiler's returned error messages are provided to the model for correction. For multiple rounds of error correction, subsequent error messages are appended to the dialogue after the model encounters errors again.

Listing 5: example1 for GPT-4o (D)

```

1 import std.math;
2 import std.typecons;
3 import std.conv;
4 import std.algorithm;
5 import std.array;
6 import std.string;
7
8 string f(string x, string y) {
9     char[] yMutable = y.dup;
10    yMutable.reverse();
11    string tmp = yMutable.map!(c => c ==
        "9" ? "0" : "9").array.map!(c =>
        c.to!string).array.join("");
12    if (x.isNumeric && tmp.isNumeric) {
13        return x ~ tmp;
14    } else {
15        return x;
16    }
17 }
18 unittest
19 {
20     alias candidate = f;
21     assert(candidate("", "sdasdnakjsda80
        ") == "");
22 }
23 void main(){}

```

Listing 6: example2 for GPT-4o (Swift)

```

1 import Foundation
2
3 func f(strand: String, zmnc: String) ->
    Int {
4     var strand = strand
5     var poz = strand.range(of: zmnc)
6     while poz != nil {
7         strand.removeSubrange(poz!)
8         poz = strand.range(of: zmnc)
9     }
10    let lastIndex = strand.range(of:
        zmnc, options: [], range: nil,
        locale: nil)?.lowerBound.
        utf16Offset(in: strand)
11    return lastIndex ?? -1
12
13 func ==(left: [(Int, Int)], right: [(Int,
    Int)]) -> Bool {
14     if left.count != right.count {
15         return false
16     }
17     for (l, r) in zip(left, right) {
18         if l != r {
19             return false
20         }
21     }
22     return true
23 }
24
25 assert(f(strand: "", zmnc: "abc") == -1)

```

Listing 7: example3 for GPT-4o (Python)

```

1 from typing import Dict, Tuple
2
3 def f(d: Dict[str, int]) -> Tuple[int,
    int]:
4     if "x" in d:
5         x = d["x"]
6     if "y" in d:
7         y = d["y"]
8     return x, y
9
10 def check(candidate):
11     assert candidate({"x": 5, "y": 12})
        == (5, 12)
12
13 def test_check():
14     check(f)

```

Programming Language Correlation in Translation

We observed that after a four-step translation process, the intersection of the 18 programming languages contained only 333 entries. This indicates that each programming language has its unique subset of correctly translated parts. Therefore, we constructed a Venn diagram to study the correlation between the sets of correct translations among different languages.

Specifically, the results are shown in Figure 9. Due to the limitation of the number of sets that a Venn diagram

can clearly represent, we explored by two methods, each selecting five representative languages. First, we chose the five languages with the largest union of results from the 18 languages to construct the first Venn diagram, which is the left half of Figure 9. Subsequently, we selected the top five most widely used languages, excluding Python, based on data from GitHub 2.0, to construct the second Venn diagram, which is the right half of Figure 9.

From Figure 9, we can observe that mainstream programming languages often have more similar syntax structures, and the model’s generation capability is stronger for these languages. Therefore, the intersection of the generation results for these five languages is relatively large, with 632 entries, while the union is relatively small, totaling 776 entries. Lua, PHP, R, Ruby, and JavaScript are among the languages with the broadest correct translation entries across all programming languages, with their union totaling 798 entries.

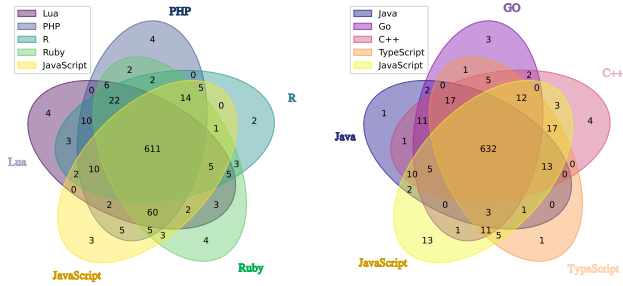


Figure 9: The communitie of each language in code translation.

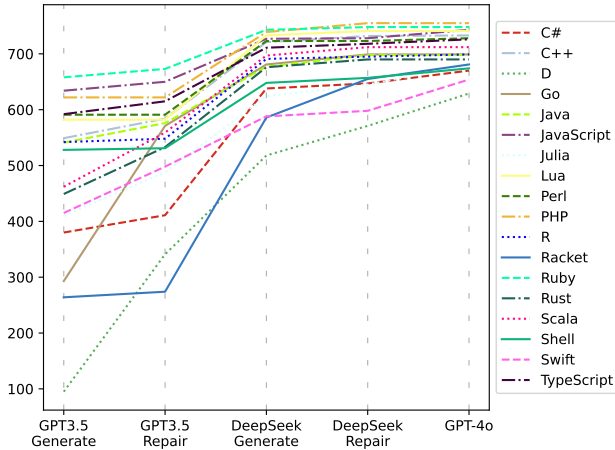


Figure 10: The result of each step in translation

The result of each Step in Translation

Figure 10 shows the improvement brought by each step for every language during the construction of the benchmark. It can be observed that each step leads to an overall enhancement in translation performance. For mainstream languages such as C++ and Java, the number of correctly translated

items can exceed 600 after one or two steps. For lower-frequency languages like D and Racket, the effect is gradually improved, eventually resulting in all languages having more than 600 correct translations.

GPU Usage and Total Cost of Translation

The total cost of GPT3.5-Turbo and GPT-4o is about \$60 US dollars. For Deepseekcoder-Instruct-33b, we use 1 NVIDIA A100-80GB GPU and the generation and repair takes about 72 hours.

Prompt of Input/Output Reasoning

In Figures 11, 12, 13, 14, we include the prompts we use for our benchmark construction. We use a few-shot prompt for all models. The examples of few-shot is shown in Listings 8, 9, 10. All the prompts and examples are demonstrated in the C++ language.

You will be given a cpp function f and a check function, where you only know the output of the test case. Find any input such that executing f on the input leads to the given output. There may be multiple answers, but you should only output one. Think step by step before arriving at an answer. Finally, surround the answer, with no additional words, with [ANSWER] and [/ANSWER] tags. Express your answer as a function call that when executed will give the output.

```

'''cpp
{example1}
'''

[ANSWER]
{answer1}
[/ANSWER]
'''cpp
{example2}
'''

[ANSWER]
{answer2}
[/ANSWER]
'''cpp
{example3}
'''

[ANSWER]
{answer3}
[/ANSWER]
'''cpp
{function}
'''

[ANSWER]
{function answer}
[/ANSWER]

```

Figure 11: Prompt of input reasoning (non-GPT)

Based on the given code, which may contain errors, complete the "???" in assert statement with the output when executing the cpp code on the given test case. Do NOT output any extra information, even if the function is incorrect or incomplete. Do NOT output a description for the assert.

```
...
'''cpp
{function}
'''
[ANSWER]
{function answer}
[/ANSWER]
```

Figure 12: Prompt of output reasoning (non-GPT)

You will be given a cpp function f and a check function, where you only know the output of the test case. Output the completion of the check function so that the code will run without errors by finding any input such that executing f on the input leads to the given output. There may be multiple answers, and you can output any one. Do NOT output any additional information.

```
...
'''cpp
{function}
'''
[ANSWER]
{function answer}
[/ANSWER]
```

Figure 13: Prompt of input reasoning (GPT)

Based on the given code, which may contain errors, complete the "???" in assert statement with the output when executing the cpp code on the given test case. Do not output any extra information, even if the function is incorrect or incomplete

```
...
'''cpp
{function}
'''
[ANSWER]
{function answer}
[/ANSWER]
```

Figure 14: Prompt of output reasoning (GPT)

Listing 8: Input/Output Reasoning example1 (C++)

```
1 #include<assert.h>
2 #include<bits/stdc++.h>
3 long f(std::vector<std::string> my_list)
4 {
5     long count = 0;
6     for (std::string i : my_list) {
7         if (i.size() % 2 == 0) {
8             count += 1;
9         }
10    }
11    return count;
12 }
13 int main() {
14     auto candidate = f;
15     assert(candidate((std::vector<std::string>({(std::string)"mq", (std::string)"px", (std::string)"zy"
16     }))) == (3)));
```

Listing 9: Input/Output Reasoning example2 (C++)

```
1 #include<assert.h>
2 #include<bits/stdc++.h>
3 std::string f(std::string s1, std::string s2) {
4     return s1 + s2;
5 }
6 int main() {
7     auto candidate = f;
8     assert(candidate(("ba"), ("nana"))
9     == ("banana"));
```

Listing 10: Input/Output Reasoning example3 (C++)

```
1 #include<assert.h>
2 #include<bits/stdc++.h>
3 std::tuple<long, long> f(std::map<std::string, long> d) {
4     long x = 0, y = 0;
5     if(d.find("x") != d.end()){
6         x = d["x"];
7     }
8     if(d.find("y") != d.end()){
9         y = d["y"];
10    }
11    return std::make_tuple(x, y);
12 }
13 int main() {
14     auto candidate = f;
15     assert(candidate((std::map<std::string, long>({{"x", 5}, {"y", 12}}))) == (std::make_tuple(5, 12)));
16 }
```