Exploring Multi-Lingual Bias of Large Code Models in Code Generation

Chaozheng Wang[†], Zongjie Li[‡], Cuiyun Gao[†*], Wenxuan Wang[†]
Ting Peng[§], Hailiang Huang[§], Yuetang Deng[§], Shuai Wang[‡], Michael R. Lyu[†]
[†] The Chinese University of Hong Kong, Hong Kong, China
[‡] Hong Kong University of Science and Technology, Hong Kong, China
[§] Tencent Inc., Guangzhou, China
{czwang23, wxwang, lyu}@cse.cuhk.edu.hk,{zligo, shuaiw}@cse.ust.hk, cuiyungao@outlook.com,

Abstract—Code generation aims to synthesize code and fulfill functional requirements based on natural language (NL) specifications, which can greatly improve development efficiency. In the era of large language models (LLMs), large code models (LCMs) have been recently proposed to generate source code. LCMs can generate highly feasible solutions for programming problems described in natural language. Despite the effectiveness, we observe a noticeable multilingual bias in the generation performance of LCMs. Specifically, LCMs demonstrate proficiency in generating solutions when provided with instructions in English, yet may falter when faced with semantically equivalent instructions in other NLs such as Chinese. Moreover, the ability of LCMs to generate code exhibits variety across different programming languages (PLs), such as Python and C++. The observed phenomenon indicates the presence of multi-lingual bias within the generative capabilities of LCMs, which has remained unexplored.

In this paper, we aim to investigate the multi-lingual bias that exists in current LCMs. First, we initiate our investigation by constructing the first multi-lingual evaluation benchmark *X-HumanEval-X*, enabling us to systematically evaluate the extent of multi-lingual bias that exists in current LCMs. In our large-scale experiments on nine popular LCMs, we observe a pronounced multi-lingual bias of LCMs in code generation, including multi-NL and multi-PL bias. Specifically, when using Chinese instructions, the code generation capabilities of LCMs decrease by at least 13% in terms of the Pass@1 metric. Furthermore, LCMs perform variously across different programming languages, e.g., the performance gap between Python and C++ reaches as high as 20.9%. Then we explore the bias in the prompting phase and find that prompting LCMs through one-step and multi-step translation aids in mitigating such bias. We further explore the impact of instruction tuning based on a self-constructed multi-lingual dataset Multi-EvolInstruct-Code (MEIC) that contains two natural languages (i.e., English and Chinese) and more than twenty programming languages. Experiments on the nine popular LCMs demonstrate that the instruction tuning substantially reduces the multi-lingual bias (e.g., decreasing the multi-NL bias and multi-PL bias by up to 84% and 40%, respectively), while enhancing the efficacy of LCMs in code generation (e.g., increasing the Pass@1 metric by 31%∼46%). We finally provide insights and implications for researchers and developers aimed at mitigating the multi-lingual bias and improving the code generation capabilities of LCMs.

Index Terms—code generation, multi-lingual, large language models

* Cuiyun Gao is the corresponding author.

I. INTRODUCTION

Code generation is a fundamental task within the domain of code intelligence, designed to interpret natural language instructions and produce corresponding code snippets that fulfill specifications. With the development of large language models (LLMs), the field of code intelligence has witnessed the emergence of large code models (LCMs) that are specifically tailored for programming tasks including code generation [13, 20, 26, 47, 60]. These models, trained on extensive datasets of source code, have substantially enhanced the efficiency of code generation, thereby considerably reducing the coding burden for developers. The remarkable capabilities of LCMs have garnered attention from users globally, spanning diverse roles within the software development process. This diversity highlights the wide range of purposes and ways in which users engage with LCMs, including the use of multiple natural languages (NLs) and the expectation for LCMs to produce code snippets across different programming languages (PLs).

In the domain of code generation, the broad user base of LCMs introduces a great demand for multi-lingual competencies, encompassing both multi-NL and multi-PL support. From the perspective of multi-NL, users expect these models to understand and generate content across multiple languages. According to the report [58], a large population of users (around 80%) are from non-English speaking regions, including but not limited to Japan and China, thereby underscoring the global demand for multi-lingual support. From the multi-PL perspective, the demand for multi-PL in LCMs arises from the distinct design principles and exclusive features of each language, which have contributed to their popularity within specific communities and development [10]. For instance, Copilot [5], an LCM-based intelligence programming assistant, receives and processes user requests to generate source code in a variety of programming languages, including Python, C++, and JavaScript [12].

Despite the remarkable demands of users in using LLMs to understand and generate different languages, they are still faced with multi-lingual bias that undermines their universal effectiveness [17, 32, 43, 50, 56]. Existing research [32] highlights that LLMs tend to exhibit superior performance in

(a) English Instruction

```python
def below_zero(operations: List[int]) -> bool:
    """ You're given a list of deposit and withdrawal operations on a bank
    account that starts with
    zero balance. Your task is to detect if at any point the balance of account
    falls below zero, and
    at that point function should return True. Otherwise it should return
    False.
    >>> below_zero([1, 2, 3])
    False
    >>> below_zero([1, 2, -4, 5])
    True
    """
    balance = 0
    for operation in operations:
        balance += operation
            if balance < 0:
                return True
    return False
```

✓ YES

Pass All Unit Tests!

(b) Chinese Instruction

```python
def below_zero(operations: List[int]) -> bool:
    """给定一个以零余额开始的银行账户上的存款和取款操作列表。
    您的任务是检测账户余额是否在任何时候低于零，并在该点返回True。否则，应返回False。
    >>> below_zero([1, 2, 3])
    False
    >>> below_zero([1, 2, -4, 5])
    True
    """
    return sum(operations) < 0
```

Input: [1,2,-4,5]
→
Expected: True
Output: False

✗ NO

Fail!

(c) Python Solution

```python
def string_sequence(n: int) -> str:
    """ Return a string containing space-delimited numbers
    starting from 0 upto n inclusive.
    >>> string_sequence(0)
    '0'
    >>> string_sequence(5)
    '0 1 2 3 4 5'
    """
    return ' '.join([str(i) for i in range(n + 1)])
```

✓ YES

Pass All Unit Tests!

(d) C++ Solution

```cpp
/*
Return a string containing space-delimited numbers
starting from 0 upto n inclusive.
>>> string_sequence(0)
"0"
>>> string_sequence(5)
"0 1 2 3 4 5"
*/
#include<stdio.h>
#include<string>
using namespace std;
string string_sequence(int n){
    string s="";
    for(int i=0;i<=n;i++) {
        s+=to_string(i);
        s+=" ";
    }
    return s;
}
```

input: 0
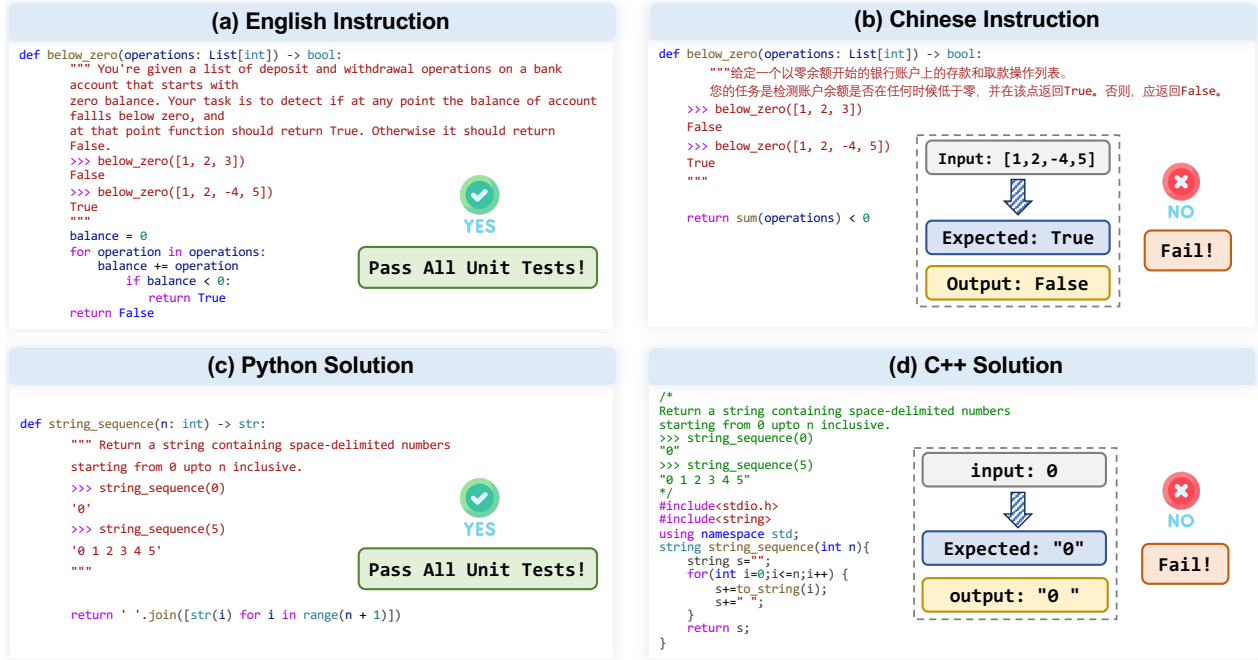→
Expected: "0"
output: "0 "

✗ NO

Fail!

Fig. 1. Motivation example of the bias that exists among multiple natural languages ((a) and (b)) and programming languages ((c) and (d)). The experimented LCM is DeepSeek-Coder 33B.

English compared to other languages, a discrepancy attributed primarily to the predominance of English in the training datasets. Similarly, such a multi-lingual bias is also observed in current LCMs. As the example shown in Figure 1 (a) and (b), DeepSeek-Coder 33B [26] generates the correct solutions in response to the English instructions but fails with the semantically equivalent Chinese instructions. Additionally, the performance of LCMs varies across different programming languages. For instance, when given the same English instruction, DeepSeek-Coder 33B effectively addresses the task in Python (c) but fails to account for a crucial condition when generating code in C++ (d), demonstrating the model's bias in handling various programming languages. However, the multi-lingual bias in the domain of code generation has remained under-explored, which necessitates a comprehensive investigation on systematically assessing and mitigating such bias.

In this paper, we aim to investigate the multi-lingual bias of LCMs in the code generation task. To facilitate this investigation, we first construct a benchmark *X-HumanEval-X*, which contains instructions of two NLs (i.e., English and Chinese due to their popularity [11]) as well as corresponding solutions in three PLs (including Python, Java, and C++). Based on the *X-HumanEval-X*, we evaluate the code generation performance of nine popular LCMs across different NLs and PLs, aiming to analyze the multi-lingual bias inherent in LCMs. Then we proceed to investigate the potential for mitigating this bias via training-free methods (e.g., in the prompting phase) and training-based methods. Specifically, for training-free methods, we explore mitigating the bias by translating the Chinese instructions into English and analyze three prompting strategies including one-step translation, multi-step translation, and self-translation. For training-based methods, we explore the mitigation of bias during the training phase, specifically focusing on the instruction tuning phase. Instruction tuning, also called supervised fine-tuning, is a pivotal step in adapting language models, aiming at refining LCMs with high-quality examples of instructions paired with their corresponding answers [61]. Therefore, we construct a multi-lingual dataset named Multi-EvolInstruct-Code (MEIC) based on EvolInstruct [40]. MEIC comprises 91,766 training instances related to code generation, encompassing two natural languages (i.e., English and Chinese) and over twenty programming languages. By instruction-tuning the LCMs with MEIC, we explore how the choice of languages and training methods affect the multi-lingual bias of LCMs in the code generation task.

Through extensive experiments on nine popular LCMs including StarCoder [33], CodeLlama [48], and DeepSeek-Coder [26], we achieve the following findings.

1) **Current LCMs exhibit substantial multi-lingual bias in code generation.** We uncover two dimensions of the multi-lingual bias within current LCMs including (a) **Multi-natural language bias.** When presented with instructions in English and Chinese that convey the same intent, we observe that the average Pass@1 rate on the *X-HumanEval-X* dataset experiences a minimum decline of 13% when switching from English to Chinese instructions. This highlights the presence of bias in the multi-natural language understanding capabilities of LCMs. (b) **Multi-programming language bias.** Given the same instruction, LCMs may successfully generate accurate solutions in widely-used programming languages such as Python and Java but struggle with more intricate languages,

such as C++. Specifically, the average code generation performance gap across different PLs reaches as high as 23.7% in terms of the average Pass@1, thereby substantiating the existence of the notable multi-programming language bias.

2) **Prompting LCMs through translation mitigates the multi-NL bias**. In addition, our exploration of instruction translation reveals that translation-based prompting strategies can mitigate multi-NL bias. Specifically, for one-step and multi-step translation strategies that employ third-party translation tools (e.g., Google Translation) to convert instructions from Chinese into English diminish the multi-NL bias, reducing the bias from 17.2% to as low as 3.8% in terms of the averaged Pass@1. However, we observe that the code generation performance based on the self-translation prompting experiences a drastic decrease at the ratio of 62.3%, even enlarging the multi-NL bias.

3) **Instruction tuning aids in mitigating the multi-lingual bias while enhancing the performance of LCMs in code generation**. Experiments on the nine popular LCMs demonstrate that the instruction tuning substantially reduces the multi-lingual bias (e.g., decreasing the multi-NL bias and multi-PL bias by up to 84% and 40%, respectively), and enhances the efficacy of LCMs in code generation (e.g., increasing the Pass@1 metric by 31%~46%). The results indicate that increasing the diversity of both NLs and PLs in the training data is beneficial for boosting the overall performance of LCMs in code generation and reducing the multi-lingual bias at the same time.

In summary, this paper makes the following contributions:

- To the best of our knowledge, we are the first to dive into the multi-lingual bias of LCMs from both the natural language and programming language perspectives in the code generation task.
- We construct the first multi-lingual benchmark *X-HumanEval-X* and one instruction tuning dataset MEIC that contains two natural languages and more than twenty programming languages.
- We conduct extensive experiments on nine popular LCMs and provide findings for researchers and developers that aim to mitigate the multi-lingual bias and improve the code generation capabilities of LCMs.

II. OVERVIEW

A. Research Questions

In this paper, we mainly investigate the following research questions through experiments.

RQ1: To what extent does the multi-lingual bias exist in LCMs in the code generation task?

RQ2: Whether the multi-lingual bias can be mitigated in the prompting phase?

RQ3: Whether the multi-lingual bias can be mitigated in the instruction tuning phase? Specifically, we explore two sub-questions, including 1) **RQ3.1:** How do different natural languages used in the instruction tuning phase affect the LCMs' multi-NL understanding performance? 2) **RQ3.2:** How do different programming languages affect LCMs' multi-PL generation performance?

In the following part of this section, we introduce the overview of our exploration in details.

B. Benchmark Construction

To initiate our investigation and answer the first research question, we construct the first multi-lingual evaluation benchmark *X-HumnEval-X* and study the multi-lingual bias from two aspects including multi-NL understanding bias and multi-PL generation bias, respectively.

For assessing multi-programming language (multi-PL) generation bias, we select three representative PLs including Python, Java, and C++ by utilizing the HumanEval-X benchmark [63]. HumanEval-X serves as a variation of the original HumanEval benchmark [20], specifically adapted to include not just Python but also other PLs. This choice is motivated by the widespread use and popularity of these programming languages [10]. Our goal is to explore and quantify the performance gaps when generating solutions across varied programming languages.

For the bias in multi-natural language understanding, we choose to employ ChatGPT [2] (GPT-3.5-turbo version) to translate the benchmark's original English instructions into Chinese due to its promising translation performance [30]. This procedure is taken to construct a Chinese version of the benchmark, given that the initial set of instructions was exclusively in English. To ensure precision, the first two authors independently review every translated result and make corrections when necessary. Then the two authors discuss the corrections and reach a consensus, confirming the accuracy of the corrected version. This ensures that the instructions are correctly represented and can be effectively used for evaluating the performance of the models with Chinese instructions.

C. Exploration on Mitigating Multi-Lingual Bias in Prompting

Given the observed bias in the understanding capabilities of LCMs, a straightforward approach to mitigate this bias involves translating Chinese prompts into English prior to prompting the LCMs. Thus, to answer RQ2, which explores the potential for reducing multilingual bias during the prompting phase, we preceding experiments that involve translating Chinese instructions into English. Specifically, we explore the following methods: 1) Employing the LCMs themselves for self-translating the Chinese instructions [27, 50], denoted as **Self-Translation**, and 2) Utilizing external translation tools, such as Google Translation [7], to convert Chinese instructions into English. Moreover, we apply translation tools in two manners which include **One-Step Translation** and **Multi-Step Translation**. One-Step Translation utilizes translation tools to directly translate all Chinese instructions into English, including the example cases. For Multi-Step Translation, individual statements are translated into English separately, aiming for potentially greater accuracy or context preservation in each translation step.

D. Mitigating Multi-Lingual Bias in Instruction Tuning

In RQ3, we explore mitigating the multi-lingual bias of LCMs in the instruction tuning phase. Specifically, we delve into how the choices of the data and training methods of instruction tuning affect the multi-lingual bias, with the exploration details as below.

1) Instruction Tuning Dataset Construction: To study the multi-lingual bias from both natural language and programming language perspectives, we first construct an instruction tuning dataset that contains code generation instructions and their corresponding output answers named MEIC. Specifically, we utilize the Evol-Instruct [40, 59] technique to construct the dataset. Following previous work [40], we start from an open-source dataset Code-Alpaca [3] that contains 20K instruction-output pairs as seed instructions, and extend their depth and width via ChatGPT [2]. The details of how we extend instructions are shown in our anonymous repository. For the extended instructions, we also feed them into GPT4 (with the API version of GPT-4-1106) to obtain output answers. Given that the generated dataset is purely in English, we remark the dataset as D_{Eng}. In summary, the dataset contains 91,766 training instances (i.e., an input instruction and output answer pair) in more than 20 programming languages such as Python, JavaScript, and SQL.

After getting the instruction tuning dataset through Evol-Instruct, we also translate the dataset into Chinese via Chat-GPT, which we remark as D_{Chi}. The first two authors also randomly sample 1,000 instances of the dataset, achieving a 99% confidence level with a confidence interval of 0.88%. Upon reviewing the translation results, we concur that 988 instances are accurately translated. Hence, it can be asserted that the dataset, translated using ChatGPT, maintains an accuracy rate of 98%, indicating the quality of ChatGPT's translation. Totally, D_{Eng} and D_{Chi} contain 49.0 and 55.8 million tokens, respectively (obtained by the tokenizer of CodeLlama). The statistics of the dataset can be accessed in Table I. For the reasons for choosing Chinese as the studied natural language, readers can refer to Section V-C.

2) Exploration on Mitigating Multi-NL Understanding Bias: To study the impact of used data and training methods of instruction tuning on the multi-NL bias of LCMs, we explore four instruction tuning methods based on the constructed instruction tuning dataset.

1) **English-Based Tuning.** We use the English version of our MEIC D_{Eng} and construct the training instance as the format "[INS]\n[ANS]" to train LCMs, where *[INS]* and *[ANS]* represent the instruction and corresponding answer, respectively.
2) **Chinese-Based Tuning.** We use the translated dataset D_{Chi} to train LCMs in the same way as English-based tuning.
3) **Mixed-NL-Based Tuning.** We randomly sample 50% training instances from D_{Eng} and D_{Chi} with the same data construction.

4) **Translation-Aware Tuning.** We propose to take both the English and Chinese instructions into account. Specifically, we construct the training data through a translation-aware template that provides both English instruction, translated Chinese instruction, and the corresponding Chinese answer. The details of the template for translation-aware tuning are shown in our anonymous repository.

3) Exploration on Mitigating Multi-PL Generation Bias: We further investigate the impact of programming languages used for instruction tuning on the code generation performance and multi-PL bias of LCMs.

Specifically, we split the dataset into two parts according to different PLs including Python and Other-PLs due to the imbalanced PL distribution in our MEIC. Then we conduct instruction tuning in three methods including 1) **Python-based Tuning**, 2) **Other-PLs-based Tuning**, and 3) **Full data-based Tuning**, respectively. In methods 1) and 2), we utilize the Python and Other-PLs parts to tune LCMs, respectively. In the method 3), we conduct instruction tuning with all training instances in MEIC.

III. EXPERIMENTAL SETUP

A. Selected LCMs

In this paper, we select three kinds of popular and state-of-the-art LCMs with their versions in different sizes. In specific, our selected LCMs are:

- **StarCoder** [33] is a large language model trained on the mixture of source code and natural language texts. Its training data incorporate more than 80 different programming languages as well as text extracted from GitHub issues and commits and from notebooks. The total account of training tokens exceeds 1T. We select its 3B, 7B, and 16B versions in our experiments.
- **CodeLlama** [48] is a family of large language models for code based on LLama 2 [51] with state-of-the-art code generation, blank infilling, and long-context processing capabilities. In this paper, we choose CodeLlama's base model (i.e., CodeLlama Base) in three different sizes including 7B, 13B, and 34B for instruction tuning.
- **DeepSeek-Coder** [26] is a series of large code models that have an identical architecture to CodeLlama. DeepSeek-Coder is trained from 2T tokens from scratch, which comprises 87% code and 13% natural language in both English and Chinese. DeepSeek-Coder achieves state-of-the-art performance in a variety of code intelligence tasks. Specifically, we choose DeepSeek-Coder Base in sizes of 1.3B, 6.7B, and 33B in this paper.

B. Evaluation Metrics

Following the prior studies [33, 48, 49], we use the Pass@k metric to evaluate the accuracy of LCMs in solving programming problems, examining whether LCMs can pass all unit tests within k solutions. The metric can be described as the following

$$pass@k = \frac{\sum_{i=1}^{n} \prod_{j=1}^{k} (\mathbb{1}(pass_{s_i^j})}{n} \tag{1}$$

Version	#Tokens	Python	JavaScript	SQL	Java	C++	HTML	CSS	PhP	Bash	Others	Total
D_{Eng}	49.0M	49,346	10,291	8,105	6,976	3,597	1,879	1,800	1,286	814	7,672	91,766
D_{Chi}	55.8M											

TABLE II
HYPER-PARAMETER SETTINGS.

Hyperparameter	Value		Hyperparameter	Value
Optimizer	AdamW [29]		Warm-up steps	100
Learning rate	5e-6		Training batch size	512
LR scheduler	Cosine Scheduler [38]		Validation batch size	32
Sequence Len.	2,048		Adam epsilon	1e-8
Max. gradient norm	1.0		Precision	BF16
Max Gen. Tokens	512		Top-P	0.95

where n and k denote the number of problems and the number of generated solutions, respectively. s_i^j indicates the j-th solution for the i-th problem. The function $\mathbb{1}(x)$ returns 1 if x is True and otherwise returns 0, and $pass(s)$ returns True if the solution s can pass all unit tests. In this paper, following previous work [33, 40, 57], we choose Pass@1 as our metric, i.e., $k = 1$.

C. Implementation Details

All the experiments are run on a server with 8*A100 GPU with 80GB graphic memory. Specifically, we utilize Optimizer State Sharding (ZeRO 3) techniques in DeepSpeed [45, 46] to save GPU memory and improve training efficacy. For larger models such as CodeLlama 34B and DeepSeek-Coder 33B, we additionally offload the optimizer state into CPU memory to further avoid out-of-CUDA memory issues. The training hyper-parameters are listed in Table II following previous work [33, 40, 57]. We randomly select 5% of the training samples as a validation set and use the checkpoint with the lowest validation loss for inference.

For fast inference, we utilize vLLM [31] based on PagedAttention to improve efficiency. The inference hyper-parameter is also listed in Table II. After generation, we conduct post-processing (e.g., truncating the content beyond the solution function) to ensure the generated code can be correctly evaluated following previous work [14, 33].

IV. EXPERIMENT ANALYSIS

In this section, we elaborate on the answers to the proposed research questions based on the experimental results.

A. RQ1: Existence of Multi-Lingual Bias in LCMs

Given a programming problem, we expect LCMs to generate correct solutions no matter the language that instructions are described in such as English and Chinese. In addition, we also expect LCMs to be able to solve programming problems in different PLs. Therefore, in RQ1, we evaluate the code generation capabilities of LCMs and the potential multi-lingual bias across different PLs including Python, C++, and Java

in different NLs, including English and Chinese. To answer RQ1, we utilize our constructed *X-HumanEval-X* to evaluate the performance of the selected nine LCMs and compare their performance across different NLs and PLs. Besides base models (i.e., versions after pre-training), their corresponding instruction-tuned models (i.e., versions after instruction tuning) are also evaluated. The results are shown in Table III. From the results, we can achieve the following observations.

1) **LCMs exhibit bias in multi-NL understanding.** All the experimented LCMs exhibit a notable multi-NL bias, i.e., the performance gap of LCMs when generating code with instructions in English and Chinese, across all programming languages. More precisely, when instructions are presented in Chinese, the average Pass@1 rate for the LCMs under study drops by 17.2% and 14.3% for the base and instruction-tuned model versions in Python, respectively. In the case of the base model version of CodeLlama 34B, its capability to generate code in Java experiences an even more pronounced decrease, with a performance reduction of 37.8%. The results reveal a pronounced bias in LCMs regarding their ability to understand different natural languages when tasked with code generation.

2) **LCMs have biased abilities in the multi-PL generation.** From the multi-PL perspective, we observe that LCMs perform variously in generating solutions in different programming languages. For instance, the base model versions of LCMs achieve the best Pass@1 rate in the Python language, which is 5.7% and 11.3% higher than that in C++ and Java, respectively. The code generation performance in the Java language is the lowest among the three experimented programming languages with both English and Chinese instructions. Compared to Python, the average Pass@1 rates for Java programming problems, when instructions are provided in English and Chinese, are lower by 11.5% and 23.7%, respectively. We attribute the bias of different PLs on base models to the training objective of base models, which is focused on auto-aggressive decoding. Without instruction tuning, base models present unsatisfactory capabilities to follow instructions and lead to a scenario where base models struggle with determining "when to stop" during code generation. This problem is specifically severe when generating solutions in Java, primarily because solutions in Java are typically organized as classes rather than functions. Thus, base models achieve the worst performance in Java on average.

However, the models after instruction tuning (the lower part of the table) exhibit their poorest performance in C++ (average 46.77% in English instructions), which presents the maximum performance gap at 13.04%. This discrepancy can be ascribed to the enhanced capability of LCMs to learn to

TABLE III
CODE GENERATION RESULTS (PASS@1) OF THE MULTI-NL AND MULTI-PL BENCHMARKS ON *X-HumanEval-X*, WHERE SC, CL, AND DSC INDICATE STARCODER, CODELLAMA, AND DEEPSEEK-CODER, RESPECTIVELY. THE COLUMN **NL-BIAS** AND **PL-BIAS** DENOTE THE MAXIMUM PERFORMANCE GAP OF THE AVERAGE PASS@1 AMONG DIFFERENT NLS AND PLS. "-" DENOTES THAT THE INSTRUCTION TUNING VERSION OF THE MODELS IS NOT RELEASED.

Benchmark-PL	Benchmark-NL	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg	NL-Bias	PL-Bias
						Base Model							
Python	English	22.56	26.82	31.70	31.09	35.36	54.87	28.65	49.39	55.48	37.32	Δ17.25%	
	Chinese	18.29	23.78	26.21	29.26	28.04	41.46	25.61	46.34	47.56	31.84		
C++	English	19.51	24.39	29.87	28.04	34.75	50.60	31.70	43.90	51.21	34.88	Δ13.65%	Δ23.70%
	Chinese	18.29	22.56	26.21	21.95	30.49	40.24	31.09	39.02	46.34	30.69		
Java	English	20.73	22.56	24.39	25.00	30.49	50.00	30.49	46.95	50.60	33.47	Δ30.03%	
	Chinese	15.85	20.12	19.51	18.90	23.78	31.09	22.56	36.58	43.29	25.74		
						Instruction Tuned Model							
Python	English	-	-	34.15	35.97	42.68	53.04	59.14	71.95	73.17	52.87	Δ14.31%	
	Chinese	-	-	29.26	31.09	37.19	45.12	54.87	62.19	64.02	46.25		
C++	English	-	-	28.66	32.92	42.07	47.56	48.17	62.80	65.24	46.77	Δ13.05%	Δ13.04%
	Chinese	-	-	25.61	30.49	30.49	44.51	45.12	55.48	57.92	41.37		
Java	English	-	-	29.26	39.63	37.19	52.49	56.70	71.34	73.17	51.40	Δ16.39%	
	Chinese	-	-	25.00	31.09	28.04	39.63	50.60	65.85	68.90	44.16		

follow instructions and generate solutions with the proper format, obtaining larger improvements in Java (achieving the average Pass@1 at 51.4% in English). We suppose that the inherent complexities associated with C++ programming result in LCMs exhibiting inferior performance when compared to their counterparts in Java and Python.

> **Finding 1:** Current LCMs exhibit pronounced bias in both multi-natural language understanding and multi-programming language generation. Specifically, when transitioning from English to Chinese instructions, the average Pass@1 rate experiences a minimum decrease of 13%. When generating source code across different programming languages, multi-PL bias reaches as high as 23.7%.

B. RQ2: Mitigating Multi-Lingual Bias in Prompting

To answer the second RQ, we investigate the effectiveness of three prompting strategies including self-translation, one-step translation, and multi-step translation in mitigating multi-NL bias. The experiment results are shown in Table IV.

From the table, we can find that in the case of self-translation, a notable decline in the performance is observed. Specifically, the average Pass@1 of code generation through self-translation stands at merely 12.0%, marking a pronounced decrease of 62.3% compared to performance with original Chinese instructions. These unfavorable outcomes indicate that current LCMs, primarily trained on source code, exhibit inadequate translation capabilities, making self-translation even enlarge the multi-NL bias.

For one-step and multi-step translation, these strategies achieve a substantial improvement in mitigating the multi-NL bias. Specifically, the average Pass@1 rate increases by 5.9% and 13.0% compared to the original Chinese instructions, reducing the multi-NL bias from the original 17.2% to 10.6% and 3.8%, respectively. Such results demonstrate that effectively translating Chinese instructions into English helps mitigate the bias in multi-NL understanding. Furthermore, our observations indicate that multi-step translation substantially outperforms one-step translation, effectively narrowing the performance gap and approximating the performance in the original English instructions. This disparity can be attributed to the limitations of current translation tools in handling content that intertwines textual and symbolic elements, leading to a compromise in both the fidelity of translation and the efficacy of mitigating multi-NL bias.

> **Finding 2:** Prompting LCMs through one-step and multi-step translation can mitigate the multi-NL bias, reducing the bias from 17.2% to as low as 3.8%. However, for self-translation, due to the unsatisfactory translation capabilities of LCMs, the average Pass@1 drops by 62.3%.

C. RQ3: Mitigating Multi-Lingual Bias in Instruction Tuning

In RQ3, we opt to focus on the base model versions of the LCMs because the specifics of their instruction tuning phase, including the data and training strategies employed, are not accessible to us. Using base models allows us to concentrate solely on understanding how the choice of data and training methods used for instruction tuning influences the model performance.

1) RQ3.1: Multi-NL Understanding Bias: We conduct instruction tuning on the selected LCMs through the proposed four training methods in Section II and the results are shown in Table V.

For **English-based Tuning**, we can observe that instruction tuning with pure English training instances can notably improve the code generation performance with both English and Chinese instructions. Specifically, compared to the original base models, instruction tuning by English data achieves an average of 28.2% and 34.5% improvement on the Pass@1 metric with English and Chinese instructions, respectively. The results reveal inter-dependencies within the knowledge representations of various natural languages embedded in LCMs. The inter-dependencies render that instruction tuning LCMs in one natural language can concurrently enhance their performance across different languages.

TABLE IV
CODE GENERATION RESULTS (PASS@1) OF DIFFERENT PROMPTING STRATEGIES FOR MITIGATING MULTI-NL BIAS.

Methods	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg
Chinese (w/o Trans.)	18.29	23.78	26.21	29.26	28.04	41.46	25.61	46.34	47.56	31.84
Self Trans.	1.83	2.44	8.54	7.32	11.59	29.26	7.32	13.41	26.21	11.99
One-Step Trans.	20.12	25.61	26.82	28.04	32.31	48.17	28.62	45.73	48.17	33.73
Multi-Step Trans.	21.95	25.61	28.66	29.26	32.92	52.44	30.49	47.56	54.87	35.97
Original English	22.56	26.82	31.70	31.09	35.36	54.87	28.65	49.39	55.48	37.32

However, despite the noticeable performance improvement, an obvious disparity persists between the models' treatment of English and Chinese instructions. After English-based tuning, the average Pass@1 rate of Chinese instructions is 11.75% lower than that of English, implying that relying solely on English data cannot well address the underlying bias towards non-English languages.

> **Finding 3:** Instruction tuning LCMs with English dataset can bring substantial performance improvement on code generation in both English and Chinese instructions, i.e., the Pass@1 rate improves 28.2% and 34.5%, respectively. However, the bias between English and Chinese benchmarks is still severe, indicated by the obvious performance gap at 11.75%.

For **Chinese-based Tuning**, similar to purely English tuning, we also observe a considerable improvement in the performance of the LCMs, i.e., the Pass@1 rates obtain an average relative improvement of 26.6% and 44.3% in English and Chinese instructions, respectively. This observation aligns well with the English-based tuning. The difference is that by utilizing Chinese data, LCMs register a 7.3% larger improvement on the Chinese benchmark while suffering a decrease of 1.3% on the English benchmark compared to English-based tuning. For the overall performance of the two languages, Chinese-based tuning outperforms English-based tuning by 2.8% in terms of the average Pass@1 rates of the two benchmarks. The results indicate that compared with English-based tuning, Chinese-based tuning can achieve competitive performance on the English benchmark while performing substantially better on the Chinese benchmark.

From the perspective of the performance gap between English and Chinese, even though the average performance in English is still better than that in Chinese, we observe that tuning with Chinese leads to a gap of 2.79%, which is 76% lower than that of English-based tuning. Specifically, for DeepSeek-Coder after Chinese-based tuning, the average Pass@1 rates of the Chinese benchmark even outperform that of the English benchmark (e.g., 37.19 v.s. 34.14 in DeepSeek-Coder 1.3B). We attribute this superior performance under Chinese instructions to the deliberate inclusion of a non-trivial amount of Chinese corpora during the pre-training phase of DeepSeek-Coder [26] for improving the models' comprehension of Chinese.

> **Finding 4:** Compared to English-based tuning, Chinese-based tuning improves the Pass@1 rate by 7.3 in Chinese instructions while sacrificing 1.3% in English instructions, achieving a lower multi-NL bias at 2.79%.

For **Mixed-NL-based Tuning**, we construct a mixed-NL dataset by randomly selecting 50% training instances from D_{eng} and D_{chi}, respectively. From the table, we observe that in the English benchmark, mixed-NL-based tuning performs better than both English and Chinese-based tuning, obtaining an average of 1.0% and 2.2% improvement on overall Pass@1, respectively. Such improvements indicate that the capabilities of LCMs in English can be further boosted by involving Chinese data. For the Chinese benchmark, the average Pass@1 of mixed-NL-based tuning is slightly lower than that of purely Chinese tuning (0.9%), which still outperforms English-based tuning by 6.4%. The results demonstrate that mixing English and Chinese can further boost LCMs in English instructions and obtain promising capability in Chinese instruction understanding, leading to a multi-NL bias at 6.11%. This dual-language training method yields superior overall performance compared to training exclusively in a single language.

Through **Translation-Aware Tuning**, the average multi-NL bias achieves 6.09%, which is close to that of mixed-NL-based tuning. In addition, LCMs achieve the best performance in both English and Chinese benchmarks. Specifically, translation-aware tuning improves the average Pass@1 by 2.38% and 2.95% over mixed-NL tuning in the English and Chinese benchmarks, respectively. The notable performance improvement can be ascribed to the LCMs' ability to discern and learn the intricate triplet relationships among English instructions, Chinese instructions, and their corresponding responses during the instruction tuning phase.

> **Finding 5:** For mixed-NL-based tuning and translation-aware tuning, LCMs present similar multi-NL bias at 6.1%. From the perspective of code generation capabilities, LCMs achieve better performance over tuning with exclusively English or Chinese data.

2) RQ3.2: Multi-PL Generation Bias: In this research question, we conduct a quantitative analysis to explore the influence of instruction tuning with various programming languages on the performance of LCMs. We split the training dataset based on the associated programming languages and their training instances, specifically categorizing it into

TABLE V

COMPARISON OF CODE GENERATION RESULTS (PASS@1) OF ENGLISH AND CHINESE BENCHMARKS ON THE *X-HumanEval-X*. SC, CL, AND DSC INDICATE STARCODER, CODELLAMA, AND DEEPSEEK-CODER, RESPECTIVELY. ALL EXPERIMENTS ARE CONDUCTED IN PYTHON. THE COLUMN **NL-BIAS** DENOTES THE MAXIMUM PERFORMANCE GAP OF THE AVERAGE PASS@1 AMONG DIFFERENT NLS. **BOLD** AND <u>UNDERLINE</u> RESULTS REPRESENT THE BEST PERFORMANCE IN ENGLISH AND CHINESE BENCHMARKS, RESPECTIVELY.

Training Method	Benchmark-NL	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg	NL-Bias
Base Models	English	22.56	26.82	31.70	31.09	35.36	54.87	28.65	49.39	55.48	37.32	Δ17.25%
	Chinese	18.29	23.78	26.21	29.26	28.04	41.46	25.61	46.34	47.56	31.83	
English-based	English	30.49	34.14	44.51	44.51	52.49	62.80	38.41	**59.14**	64.02	47.83	Δ11.75%
	Chinese	28.65	30.49	35.36	39.02	46.34	56.70	34.14	54.78	59.75	42.80	
Chinese-based	English	30.49	36.58	**45.12**	43.29	54.26	62.80	34.14	55.48	62.80	47.22	Δ2.79%
	Chinese	<u>32.31</u>	33.63	38.41	41.46	49.35	59.14	<u>37.19</u>	<u>57.92</u>	64.02	45.94	
Mixed-NL-based	English	32.31	36.58	43.29	**46.34**	54.26	63.41	35.36	**59.14**	64.02	48.30	Δ6.11%
	Chinese	31.70	<u>34.14</u>	<u>39.63</u>	41.46	47.56	58.53	35.97	57.31	63.41	45.52	
Translation-aware	English	**34.13**	**38.41**	**45.12**	45.12	**55.48**	**64.02**	**39.02**	58.53	**65.24**	**49.45**	Δ6.09%
	Chinese	<u>32.31</u>	<u>34.14</u>	<u>39.63</u>	<u>42.07</u>	<u>51.21</u>	<u>60.97</u>	36.58	57.31	<u>65.24</u>	<u>46.61</u>	

TABLE VI

COMPARISON OF CODE GENERATION RESULTS (PASS@1) IN PYTHON, C++, AND JAVA BENCHMARKS. SC, CL, AND DSC INDICATE STARCODER, CODELLAMA, AND DEEPSEEK-CODER, RESPECTIVELY. ALL EXPERIMENTS ARE CONDUCTED IN ENGLISH. THE COLUMN **PL-BIAS** DENOTES THE MAXIMUM PERFORMANCE GAP OF THE AVERAGE PASS@1 AMONG DIFFERENT PLS. **BOLD**, *italic* AND <u>UNDERLINE</u> RESULTS REPRESENT THE BEST PERFORMANCE IN PYTHON, C++, AND JAVA BENCHMARKS, RESPECTIVELY.

Training Method	Benchmark-PL	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg	PL-Bias
Base Models	Python	22.56	26.82	31.70	31.09	35.36	54.87	28.65	49.39	55.48	37.32	Δ11.50%
	C++	19.51	24.39	29.87	28.04	34.75	50.60	31.70	43.90	51.21	34.88	
	Java	20.73	22.56	24.39	25.00	30.49	50.00	30.49	46.95	50.60	33.47	
Python	Python	**30.49**	33.53	43.29	**44.51**	51.21	**62.80**	**38.41**	**59.75**	63.41	47.49	Δ10.05%
	C++	21.95	31.70	39.02	37.80	46.95	57.92	*37.80*	*57.31*	57.92	43.15	
	Java	28.04	31.70	39.63	39.63	52.43	57.92	37.80	54.26	58.53	44.44	
Other-PLs	Python	29.26	32.92	37.80	34.75	47.56	60.97	37.80	57.92	61.58	44.51	Δ6.73%
	C++	23.78	33.53	39.02	37.80	50.00	56.70	35.36	54.87	57.92	43.22	
	Java	29.87	33.53	41.46	40.24	49.35	58.53	40.85	<u>57.92</u>	<u>63.41</u>	46.13	
Full data	Python	**30.49**	**34.14**	**44.51**	**44.51**	**52.49**	**62.80**	**38.41**	59.14	**64.02**	**47.83**	Δ5.97%
	C++	*26.21*	*34.14*	*41.46*	*41.46*	*51.52*	*59.14*	*37.80*	56.70	*59.14*	45.29	
	Java	<u>32.31</u>	<u>35.36</u>	<u>43.29</u>	<u>44.51</u>	<u>55.48</u>	<u>60.97</u>	<u>41.46</u>	57.31	62.80	48.17	

Python and Other-Programming Languages (Other-PLs) following previous work [57], comprising 49,346 and 42,420 instances respectively. Subsequently, these segmented datasets are employed to train LCMs. Then we assess the LCMs' code generation performance across various programming languages, including Python, Java, and C++. We present our results in Table VI.

For **Python-based Tuning**, from the table, we can observe that training on the data of a single Python language brings remarkable performance improvement among the benchmarks. Specifically, the average Pass@1 of LCMs increases by 27.3%, 23.7%, and 32.8% in Python, C++, and Java, respectively. This observation about programming languages aligns with the natural languages, indicating an implicit relationship among the knowledge about programming languages embedded in LCMs. Through instruction tuning LCMs with even a single language, the capabilities in other languages will also be exploited (no matter a natural language or programming language). This improvement suggests that the instruction tuning process, even when it is concentrated on a specific language, can enhance the multi-lingual generalization, boosting the models' overall ability to understand instructions and generate code.

From the perspective of multi-PL bias, despite the substantial improvement, multi-PL bias is still obvious, i.e., a 10.05% performance gap between Python and C++ benchmarks is observed.

After **Other-PLs-based Tuning**, LCMs enhance their code generation capabilities across multiple programming languages, although the degree of improvement varies when compared to training exclusively on the Python language dataset. Specifically, when the training is conducted on programming languages other than Python (Other-PLs), there is an observed increase in the Pass@1 metric by 19.3%, 23.9%, and 37.8% for the three benchmarks, respectively. Compared to Python-based tuning, training on other PLs improves the performance in Java by 3.8% while suffering a 6.5% reduction for Python. In addition, the method utilizes more PLs for instruction tuning, mitigating the multi-PL bias compared to original base models and Python-based tuning by at least 30% (i.e., the performance gap between C++ and Java is 6.7%). These outcomes suggest that the selection of PLs can bring greater improvement in LCMs' capabilities to generate the corresponding PL, thus, the limited diversity of PLs used for instruction tuning tends to be insufficient to mitigate multi-PL bias.

Finding 6: The code generation performance in multi-programming languages can be improved by instruction tuning, regardless of the specific programming language used for training. Notably, more substantial improvements are observed when LCMs are tuned with datasets corresponding to the target programming language, resulting in a multi-PL bias larger than 6.7%.

For **Full Data-based Tuning** with the mixture of the above-mentioned PLs, LCMs achieve the best average Pass@1 in all PLs including Python, C++, and Java among experimented methods. Compared to the base models, full data-based tuning increases the overall Pass@1 by 27.5%, 25.7%, and 39.1% in the three experimented benchmarks, respectively. Such superior performance of full data-based training demonstrates that increasing the diversity from the aspect of programming languages can further bring enhancement in the code generation capabilities of multiple programming languages.

In terms of the bias among experimented PLs, we observe that increasing the diversity of PLs for instruction tuning aids in mitigating bias in multi-PL generation. For instance, when training is conducted exclusively with Python, the widest performance gap observed is 10.05% (between Python and C++). However, when tuning is performed using a full dataset that encompasses multiple programming languages, this gap narrows by 40% (i.e., a bias of 5.97% between Java and C++).

Finding 7: Full data-based tuning that further enhances the diversity of PLs mitigates the average multi-PL bias by 40% compared to Python-based tuning. In addition, the average Pass@1 rate obtains the largest improvement at 27.5%, 25.7%, and 39.1% in Python, C++, and Java, respectively.

V. Discussion

A. Implication of Findings

In this section, we discuss the implications of our work for researchers and developers.

For researchers. Our research demonstrates that current large code models present obvious multi-lingual bias in the code generation task. With well-designed training methods and dataset construction, instruction tuning substantially improves LCMs' performance of code generation and mitigates the multi-lingual bias. However, as shown in the results of RQ3, the multi-lingual bias still persists in current LCMs. Our results also reveal the potential research directions in the era of LCM for the community. Specifically:

- **Exploring to collect high-quality multi-lingual data.** The users in current software communities such as StackOverflow mainly concentrate on certain NLs (i.e., English) or PLs (e.g., Python and JavaScript) [1], which potentially leads to a degraded diversity when collecting data from these communities. Therefore, exploring effective methods for gathering, evaluating, and integrating diverse language data for training LCMs is crucial.
- **Exploring to amplify underrepresented languages.** In addition, the highly imbalanced multi-lingual data used for training, e.g., the imbalanced distribution of programming languages as reported in [39], result in the under-representation of some languages. Thus, developing strategies that can effectively amplify the presence and influence of underrepresented languages in the training data (e.g., weighted training [53] and language-agnostic representations [64]) needs to be further investigated.
- **Exploring more sophisticated training methods.** This paper aims to mitigate the multi-lingual bias from the angle of instruction tuning. Besides instruction tuning, the exploration of more sophisticated fine-tuning methodologies (e.g., multi-lingual preference learning [21, 44]) presents a promising avenue for future research.

For developers. Instruction tuning enables the pre-trained LCMs to follow human instructions and better exploit the knowledge embedded in the model. Our findings indicate that the multi-lingual data and training methods have a substantial impact on the performance of LCMs. Based on our findings, we conclude the following insights and takeaways for developers to conduct instruction tuning and adapt LCMs into practice in their work.

- The presence of multi-lingual bias in current LCMs can affect their performance in generating source code from instructions given in languages other than English. This bias may lead developers, facing unsatisfactory outcomes with their native or preferred languages, to resort to using English when interacting with LCMs.
- Instruction tuning LCMs with a single language (no matter NL and PL) improves the performance of LCMs in code generation across other languages.
- Instruction tuning helps to mitigate the multi-lingual bias. However, tuning with a purely English dataset still results in a notable degree of bias, specifically measured at 11.75%.
- Enhancing the diversity from both NL and PL perspectives of the instruction tuning dataset further improves the generation capabilities of LCMs and mitigates multi-lingual bias, making LCMs more accessible and useful to global developers.

B. Prompting or Instruction Tuning

In our experiments in RQ2 and RQ3, we demonstrate that the multi-NL bias can be potentially mitigated in both the prompting and instruction tuning phases. In this section, we discuss the advantages and limitations of the two methods.

For **prompting**, we observe that one-step translation and multi-step translation aid in mitigating the multi-NL bias by 40% and 76%, respectively. Despite the contribution to mitigating multi-NL bias, these methods have the following limitations: 1) The benefits come with auxiliary translation tools and non-trivial costs. For example, in the multi-step translation process, we averagely translate 3.9 statements for each programming problem. 2) Translation helps to mitigate the bias by narrowing the performance gap between Chinese

and English instructions; however, the essential code generation capabilities of LCMs are not improved. 3) LCMs typically generate responses such as code comments and explanations in the same language as the input. Consequently, when users who are not proficient in English opt to translate their queries into English, they may find the English responses difficult to comprehend, resulting in additional costs to translate back.

For **instruction tuning**, as noted in RQ3, we observe that instruction tuning not only effectively mitigates the multi-NL bias by as high as 84% but also substantially improves the code generation capabilities of LCMs, as evidenced by an overall 39% increase in the Pass@1 rate. However, instruction tuning involves the creation of a multi-lingual dataset and demands computational resources for training the LCMs. This requirement could pose practical challenges and resource constraints in implementation.

Drawing upon our empirical findings, developers can choose proper methods to mitigate the multi-lingual bias according to their resources and expectations in practice.

C. Threats to Validity

We have identified the following major threats to validity:

Limited LCMs. The experiments are based on open-source popular LCMs, which may bring bias in the results. The reason we refrain from utilizing close-source models such as ChatGPT [2] and Gemini [6] stems from the fact despite their superior capabilities, closed-source models are prohibited within certain companies due to privacy concerns [54]. Consequently, instruction tuning open-source LCMs as an internal programming assistant presents a viable solution, which is the point that this paper focuses on. To mitigate this issue, we select three types of popular LCMs of varying sizes, ranging from 1.3 billion to 34 billion parameters to control the threat.

Limited natural languages. In this paper, we specifically focus on English and Chinese for conducting experiments on multi-NL understanding. This choice is grounded in the fact that English and Chinese are recognized as the two most prevalent languages globally [11]. Additionally, according to existing studies [56], Chinese and English present a high imbalance from the perspective of training data. In addition, despite the prevalence of Chinese, a substantial performance disparity with English is still observed. Thus, we believe the findings obtained by investigating the multi-NL bias in Chinese and English can also be generalized to other languages.

VI. RELATED WORK

A. Large Code Models

The use of large language models in natural language processing inspires researchers and companies to develop LCMs for programming tasks. Generally, LCMs can be obtained through two approaches: continuing the training of foundation LLMs or pre-training from scratch [4, 14, 26, 33, 48, 63]. CodeLlama [48] is an example of the former, which is based on the LLaMA2 foundation model [52]. On the other hand, Starcoder2 [39] is an example of the latter, trained from scratch with over 600 programming languages. In addition to these open-source models, big companies also develop their coding products with LCMs, such as GitHub Copilot [5] and Tabnine [9]. LCMs show promising results in various code tasks, including code completion and summarization [41]. However, they face challenges similar to LLMs, such as safety output [35] and intellectual property concerns [36, 37].

Apart from source-level code, LCMs have also been designed for low-level code, such as decompilation and LLVM IR code [34]. For instance, Cummins et al. [22] proposed a 7B LCM to optimize LLVM assembly code, while 01.AI [8] developed a "machine language model" to analyze executable programs and binary code.

B. Multi-Lingual Inconsistency in Large Language Models

The performance gap of LLMs between high-resourced languages and underrepresented languages has been studied recently in various natural language processing tasks [17, 32, 43, 50, 56], such as logical reasoning [24], nature language understanding [28], and nature language generation [23]. Specifically, [50] revealed that GPT-3 [18] and PaLM [16] perform worse when answering the math questions in underrepresented languages, compared with the performance of answering the same questions in English. The work [56] found that LLMs produce significantly more unsafe responses for non-English queries than English ones, indicating the unsatisfied safety alignment for non-English languages. These performance gaps can be attributed to the language-imbalanced nature of training data and alignment data: The majority of training and alignment data of current LLMs, such as GPT-4, are in English [15, 17, 42, 62]. Meanwhile, previous works have proposed several methods to transfer knowledge from high-resource languages to underrepresented languages. Additionally, the work [24] proposes an attention mechanism that uses a dedicated set of parameters to encourage cross-lingual attention in code-switched sequences. Another thread of work adopts prompting strategies. [28] design a generic template prompt that stimulates cross-lingual and logical reasoning skills to enhance task performance across languages. In addition, [23] utilizes the LLM to generate multilingual training data, which is then used for fine-tuning the LLM, to alleviate the performance gap without any human labeling effort.

Different from previous works, this paper focuses on code generation tasks, which have yet to be explored, and investigates the two aspects of multi-lingual bias, including multi-NL understanding and multi-PL generation. Existing work in the code intelligence field only reports the performance in different PLs [19, 25, 55], ignoring the bias among PLs. Contrarily, our work conducts extensive experiments to investigate how to improve the code generation performance and multi-lingual bias of LCMs.

VII. CONCLUSION

In this paper, we have identified the multi-NL and multi-PL bias of LCMs in the code generation task. In addition, we have explored to mitigate the bias in the prompting and

instruction tuning phase. Extensive experiments demonstrate that both prompting and instruction tuning aid in mitigating the multi-lingual bias, and instruction tuning can further boost the code generation capabilities of LCMs. Moreover, we provide insights and implications for researchers and developers aimed at mitigating the multi-lingual bias and improving the performance of LCMs.

REFERENCES

[1] 2023 Developer Survey. https://survey.stackoverflow.co/2023/#most-popular-technologies-language-other.

[2] ChatGPT. https://chat.openai.com/.

[3] CodeAlpaca. https://github.com/sahil280114/codealpaca.

[4] CodeX. https://openai.com/blog/openai-codex/.

[5] Copilot. https://github.com/features/copilot.

[6] Gemini. https://deepmind.google/technologies/gemini/#introduction.

[7] Google Translation. https://translate.google.com/.

[8] mlmproject. https://mlm.lingyiwanmu.com/.

[9] tabnine. https://www.tabnine.com/.

[10] The top programming languages. https://octoverse.github.com/2022/top-programming-languages.

[11] Top 10 most spoken languages in the world in 2024. https://www.forbesindia.com/article/explainers/most-spoken-languages-world/91687/1.

[12] I. Akvelon, "Github copilot efficiency explored: Key takeaways from akvelon's survey," https://medium.com/@akvelonsocialmedia/github-copilot-efficiency-explored-key-takeaways-from-akvelons-survey-1b46e2391f0e, 2023.

[13] Z. Ali, H. Darwis, L. B. Ilmawan, S. R. Jabir, A. R. Manga *et al.*, "Memory efficient with parameter efficient fine-tuning for code generation using quantization," in *2024 18th International Conference on Ubiquitous Information Management and Communication (IMCOM)*. IEEE, 2024, pp. 1–6.

[14] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey *et al.*, "Santacoder: don't reach for the stars!" in *Deep Learning for Code (DL4C) Workshop*, 2023.

[15] D. M. Alves, J. Pombal, N. M. Guerreiro, P. H. Martins, J. Alves, A. Farajian, B. Peters, R. Rei, P. Fernandes, S. Agrawal *et al.*, "Tower: An open multilingual large language model for translation-related tasks," *arXiv preprint arXiv:2402.17733*, 2024.

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

[17] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, Q. V. Do, Y. Xu, and P. Fung, "A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity," *IJCNLP*, 2023.

[18] T. B. Brown and et al., "Language models are few-shot learners," *ArXiv*, vol. abs/2005.14165, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:218971783

[19] W. Chaozheng, Y. Yuanhang, G. Cuiyun, P. Yun, Z. Hongyu, and M. R. Lyu, "Prompt tuning in code intelligence: An experimental evaluation," *IEEE Transactions on Software Engineering*, 2023.

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

[21] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[22] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve *et al.*, "Large language models for compiler optimization," *arXiv preprint arXiv:2309.07062*, 2023.

[23] Y. Deng, W. Zhang, S. J. Pan, and L. Bing, "Multilingual jailbreak challenges in large language models," *ICLR*, 2023.

[24] N. Foroutan, M. Banaei, K. Aberer, and A. Bosselut, "Breaking the language barrier: Improving cross-lingual reasoning with structured self-attention," in *Conference on Empirical Methods in Natural Language Processing*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:264439333

[25] S. Gao, C. Gao, C. Wang, J. Sun, D. Lo, and Y. Yu, "Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1933–1945.

[26] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[27] J. Hu, S. Ruder, A. Siddhant, G. Neubig, O. Firat, and M. Johnson, "Xtreme: A massively multilingual multi-task benchmark for evaluating cross-lingual generalisation," in *International Conference on Machine Learning*. PMLR, 2020, pp. 4411–4421.

[28] H. Huang, T. Tang, D. Zhang, W. X. Zhao, T. Song, Y. Xia, and F. Wei, "Not all languages are created equal in llms: Improving multilingual capability by cross-lingual-thought prompting," *EMNLP*, 2023.

[29] L. Ilya and H. Frank, "Decoupled weight decay regularization," *International Conference on Learning Representations, ICLR*, 2018.

[30] W. Jiao, W. Wang, J.-t. Huang, X. Wang, S. Shi, and Z. Tu, "Is chatgpt a good translator? yes with gpt-4 as the engine," *arXiv preprint arXiv:2301.08745*, 2023.

[31] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[32] V. D. Lai, N. T. Ngo, A. P. B. Veyseh, H. Man, F. Dernoncourt, T. Bui, and T. H. Nguyen, "Chatgpt beyond english: Towards a comprehensive evaluation of large language models in multilingual learning," *Findings of EMNLP*, 2023.

[33] R. Li, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, L. Jia, J. Chim, Q. Liu *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research*, 2023.

[34] Z. Li, P. Ma, H. Wang, S. Wang, Q. Tang, S. Nie, and S. Wu, "Unleashing the power of compiler intermediate representation to enhance neural program embeddings," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2253–2265.

[35] Z. Li, C. Wang, Z. Liu, H. Wang, D. Chen, S. Wang, and C. Gao, "Cctest: Testing and repairing code completion systems," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, 2023, p. 1238–1250.

[36] Z. Li, C. Wang, P. Ma, C. Liu, S. Wang, D. Wu, and C. Gao, "On the feasibility of specialized ability stealing for large language code models," 2023.

[37] Z. Li, C. Wang, S. Wang, and G. Cuiyun, "Protecting intellectual property of large language model-based code generation apis via watermarks," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. ACM, 2023.

[38] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," in *International Conference on Learning Representations*, 2016.

[39] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[40] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations*, 2023.

[41] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[42] OpenAI, "Gpt-4 technical report," 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257532815

[43] K. Peng, L. Ding, Q. Zhong, L. Shen, X. Liu, M. Zhang, Y. Ouyang, and D. Tao, "Towards making the most of chatgpt for machine translation," in *Conference on Empirical Methods in Natural Language Processing*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257704711

[44] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[45] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20:*

International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020, pp. 1–16.

[46] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "{Zero-offload}: Democratizing {billion-scale} model training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 551–564.

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

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

[49] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao *et al.*, "Pangu-coder2: Boosting large language models for code with ranking feedback," *arXiv preprint arXiv:2307.14936*, 2023.

[50] F. Shi, M. Suzgun, M. Freitag, X. Wang, S. Srivats, S. Vosoughi, H. W. Chung, Y. Tay, S. Ruder, D. Zhou, D. Das, and J. Wei, "Language models are multilingual chain-of-thought reasoners," *ICLR*, 2022.

[51] H. Touvron and et al., "Llama 2: Open foundation and fine-tuned chat models," *ArXiv*, vol. abs/2307.09288, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259950998

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

[53] C. Wang, S. Gao, P. Wang, C. Gao, W. Pei, L. Pan, and Z. Xu, "Label-aware distribution calibration for long-tailed classification," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[54] C. Wang, J. Hu, C. Gao, Y. Jin, T. Xie, H. Huang, Z. Lei, and Y. Deng, "How practitioners expect code completion?" in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1294–1306.

[55] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[56] W. Wang, Z. Tu, C. Chen, Y. Yuan, J. tse Huang, W. Jiao, and M. R. Lyu, "All languages matter: On the multilingual safety of large language models," *ArXiv*, 2023.

[57] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.

[58] B. Wodecki, "Chatgpt passes 1 billion page views," https://aibusiness.com/nlp/chatgpt-passes-1b-page-views, 2023, accessed: 2024-03-01.

[59] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang, "Wizardlm: Empowering large language models to follow complex instructions," *arXiv preprint arXiv:2304.12244*, 2023.

[60] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Unveiling memorization in code models," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 856–856.

[61] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu *et al.*, "Instruction tuning for large language models: A survey," *arXiv preprint arXiv:2308.10792*, 2023.

[62] J. Zhao, Z. Zhang, Q. Zhang, T. Gui, and X. Huang, "Llama beyond english: An empirical study on language capability transfer," *arXiv preprint arXiv:2401.01055*, 2024.

[63] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[64] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," in *International Conference on Learning Representations*, 2020.