Unraveling the Potential of Large Language Models in Code Translation: How Far Are We?

Qingxiao Tao*, Tingrui Yu*, Xiaodong Gu, Beijun Shen†

School of Software, Shanghai Jiao Tong University, Shanghai, China
{tao_qingxiao, hzfsls, xiaodong.gu, bjshen}@sjtu.edu.cn

***Abstract*—While large language models (LLMs) exhibit state-of-the-art performance in various tasks, recent studies have revealed their struggle for code translation. This is because they haven't been extensively pre-trained with parallel multilingual code, which code translation heavily depends on. Moreover, existing benchmarks only cover a limited subset of common programming languages, and thus cannot reflect the full potential of LLMs in code translation. In this paper, we conduct a large-scale empirical study to exploit the capabilities and incapabilities of LLMs in code translation tasks. We first craft a novel benchmark called PolyHumanEval by extending HumanEval to a multilingual benchmark of 14 languages. With PolyHumanEval, we then perform over 110,000 translations with bleeding-edge code LLMs. The result shows LLMs' suboptimal performance on Python to other languages and the negligible impact of widely adopted LLM optimization techniques such as conventional pre-training and instruction tuning on code translation. To further uncover the potential of LLMs in code translation, we propose two methods: (1) intermediary translation which selects an intermediary language between the source and target ones; and (2) self-training which fine-tunes LLMs on self-generated parallel data. Evaluated with CodeLlama-13B, our approach yields an average improvement of 11.7% computation accuracy on Python-to-other translations. Notably, we interestingly find that Go can serve as a lingua franca for translating between any two studied languages.**

***Index Terms*—Code translation, large language models, intermediary translation, self-training.**

I. Introduction

Code translation, which aims at migrating source code written in one programming language (PL) to another, has emerged as a rapidly growing technology in software engineering [1]. Software applications are often developed for multiple platforms, requiring the same product to be written in multiple languages. From the maintenance perspective, it is often demanded to migrate software to a modern language to meet the ever-growing requirements. Automatic code translation can significantly reduce developers' effort in software maintenance, improving code reliability, safety, and efficiency.

Recently, large language models (LLMs) for code such as StarCoder [2] have demonstrated promising performance in code translation [1, 3]. Through extensive pre-training on ultra-large code corpora, LLMs acquire the ability of zero-shot translation between arbitrary languages. These properties make LLMs a promising direction for code translation.

Despite the superb advances in many code comprehension and generation tasks, the potential of LLMs for code translation remains to be fully explored. Unlike other code tasks [4, 5], LLMs demonstrate to be more struggling with code translation. This is because they have not been extensively pre-trained on parallel multilingual code, which code translation heavily depends on. Recent research [3] has revealed that LLMs suffer from numerous kinds of translation errors, such as API mismatch, incorrect data type, and missing library imports. These errors suggest that there is still ample room for improving LLMs in the code translation task.

In this paper, we conduct a large-scale and comprehensive study to exploit the potential of LLMs in code translation. First, we build a new code translation benchmark *PolyHumanEval* by extending HumanEval [5] to 14 programming languages. We also design a rule-based tool to generate equivalent test programs for all PLs and ensure all our canonical solutions have passed those tests. Based on PolyHumanEval, we perform over 110,000 translations with bleeding-edge code LLMs. We find that LLMs exhibit asymmetrical capability in code translation with more proficiency in translating other languages to Python while struggling with Python-to-other translation. It also demonstrates asymmetrical capability in comprehending and generating code for the same language. Moreover, conventional pre-training and instruction tuning techniques have a marginal effect on LLM's capability in code translation. Our results also indicate that prompting LLMs with the function signature (obtained by simple rule-based conversion) of the target language enhances the code translation performance considerably.

To further uncover the potential of LLMs in code translation, we propose two novel methods, namely, intermediary translation and self-training. The intermediary translation takes Go as a lingua franca between the source and target languages. The self-training fine-tunes LLMs on self-generated parallel code, which undergoes self-checking to ensure high quality. Evaluation results show that both methods enhance LLMs' ability in code translation considerably, yielding an average improvement of 11.7% computation accuracy on Python-to-other translations with CodeLlama-13B.

Our contributions can be summarized as follows:

- We build the PolyHumanEval benchmark by extending

* Both authors contributed equally to this research.
† Beijun Shen is the corresponding author.

TABLE I: Statistics of The Studied Code LLMs.

Model	Type[1]	Training Tasks	Training Data
CodeLlama-(7B/13B/34B)	base	Completion + Infilling + Long-Context-Tuning	Closed
CodeLlama-(7B/13B/34B) -Python	tuned	w/ Python-Training	Closed
CodeLlama-(7B/13B/34B) -Instruct	tuned	w/ Instruction-Tuning + Self-Instruct	Closed
StarCoderBase	base	Completion + Infilling	StarCoderData
StarCoder	tuned	w/ Python-Training	StarCoderData(Python)
StarChat-α	tuned	w/ Instruction-Tuning	OpenAssistant[12]
OctoCoder	tuned	w/ Python-Training + Instruction-Tuning	StarCoderData(Python) + OpenAssistant + CommitFT[5]
StarCoderPlus	tuned	w/ Python-Training + NL-Training	StarCoderData(Python) + RefinedWeb[13]
StarChat-β	tuned	w/ Python-Training + NL-Training + Instruction-Tuning	StarCoderData(Python) + RefinedWeb + OpenAssistant
StarCoderBase-7B	base	Completion + Infilling	StarCoderData
CodeGen2.5-7B-Multi	base	Completion + Infilling	StarCoderData
CodeGen2.5-7B-Mono	tuned	w/ Python-Training	StarCoderData(Python)
CodeGen2.5-7B-Instruct	tuned	w/ Python-Training + Instruction-Tuning	StarCoderData(Python) + Unknown[2]
CodeGeeX2-6B	base	Completion + Infilling + Cross-File Completion	Closed

1 All "tuned" models are further trained on the "base" model in the same block.
2 The technical report only mentioned that CodeGen2.5-7B-Instruct was fine-tuned on public instruction dataset but did not specify which one.

HumanEval to 14 popular PLs, and conduct a large-scale empirical study on the performance of state-of-the-art code LLM series on code translation tasks across multilingual PLs.
- We propose novel methods of intermediary translation and self-training to unravel LLMs' potential in code translation. The results demonstrate significant improvements in the code translation capabilities of LLMs.

II. Experimental Setup

Our study aims to answer the following research questions:

RQ1. How effective are code LLMs in performing code translation?

RQ2. How to efficiently prompt LLMs to translate source code?

RQ3. What are effective ways to improve the performance of LLMs on code translation?

A. Studied LLMs

We select four representative open-source code LLMs, as summarized in Table I:

- **CodeLlama** [6] is a family of code LLMs further trained upon Llama2 [7] using grouped query attention. We study its 7B, 13B and 34B variants.
- **StarCoder** [2] is a code LLM built on GPT-2 structure with multi-query attention. The LLM has been trained on StarCoderData, a multilingual code dataset extracted from The Stack [8].
- **CodeGen** [9] is a well-known family of code LLMs developed by Salesforce. We examine its latest version CodeGen2.5, which is built on Llama-7B and has been pre-trained on StarCoderData.
- **CodeGeeX2** [10] is a code LLM with multi-query attention. The model was initialized from ChatGLM2 [11].

We provide the configurations for LLMs in Table II. All models are trained and evaluated on 2×NVIDIA GeForce RTX 4090 and undergo int-8 quantization [14].

TABLE II: Configurations and Hyperparameters for LLMs.

LLM Generation Settings		**LoRA Fine-tuning Settings**	
temperature	0.01	lora_r	16
top_p	0.9	lora_alpha	32
top_k	50	lora_dropout	0.05
repetition_penalty	1.0	learning_rate	5e-6
max_new_tokens	2048	batch_size	16

B. Studied Programming Languages

We select PLs according to their popularity and data availability from The Stack [8], which is a large code corpus for pertaining LLMs. The popularity of a language is gauged by the volume of its code data. We focus on the most popular programming languages in the corpus, including 14 PLs: C++, C#, Dart, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust, Scala, Swift, and TypeScript.

C. Benchmark

Existing code translation benchmarks [1, 15, 16] mainly focus on long-standing PLs (e.g., Python and Java), while paying little attention to the emerging ones (e.g., Rust and Kotlin). Though there have been previous benchmarks like HumanEvalPack [5] that provide HumanEval solutions in different PLs, they do not ensure the equivalence of their solutions.

To tackle these issues, we construct a new code translation benchmark *PolyHumanEval*, a multilingual benchmark of 14 PLs. PolyHumanEval is extended from HumanEval [17], a widely-used benchmark for LLM evaluation. For each programming problem in the HumanEval dataset, we handcraft solutions in different PLs and validate their semantic consistency with both peer-reviewing and tests. To generate completely equivalent tests for different languages, we develop a rule-based tool [18] to automatically generate test programs in all PLs based on problem metadata. This metadata is manually defined, including a function signature that employs standardized data types crafted uniformly across all languages, and the associated test cases specified by the inputs and

TABLE III: Translation Results of LLMs.

Model	C++	C#	Dart	Go	Java	JS	Kotlin	PHP	Ruby	Rust	Scala	Swift	TS	Avg.
							X→Python							
CodeLlama-7B	70.12	78.66	75.00	79.27	79.27	76.22	78.66	78.05	75.00	81.10	72.56	78.05	76.83	76.83
CodeLlama-13B	81.10	**91.46**	**82.32**	90.85	**87.80**	**83.54**	**85.37**	**82.32**	**90.24**	87.20	**88.41**	84.76	**84.76**	**86.16**
CodeLlama-34B	**87.20**	86.59	80.49	**91.46**	84.76	82.93	80.49	80.49	84.15	**87.80**	82.93	**85.37**	81.71	84.34
StarCoderBase-15B	67.68	77.44	71.34	75.00	78.66	81.71	77.44	76.22	75.00	73.78	77.44	80.49	79.27	76.27
StarCoderBase-7B	74.39	75.00	70.73	68.29	74.39	76.83	71.34	78.05	67.07	64.02	67.07	70.73	76.83	71.90
CodeGen2.5-7B-Multi	62.20	71.95	69.51	69.51	73.17	77.44	78.05	75.00	72.56	71.95	75.00	73.17	74.39	72.61
CodeGeeX2-6B	73.17	75.00	76.22	77.44	71.95	76.83	79.27	78.05	70.73	80.49	73.17	79.88	81.10	76.41
							Python→X							
CodeLlama-7B	62.20	71.95	53.66	57.32	68.29	70.12	75.00	57.93	64.02	51.83	68.29	60.37	67.07	63.70
CodeLlama-13B	67.68	70.73	58.54	58.54	71.34	72.56	77.44	66.46	63.41	59.15	71.95	62.20	70.12	66.93
CodeLlama-34B	**75.00**	**84.76**	**70.12**	**67.07**	**79.88**	**76.22**	**78.05**	**70.12**	**66.46**	**66.46**	**76.22**	**67.07**	**73.78**	**73.17**
StarCoderBase-15B	55.49	67.68	53.05	53.05	60.37	67.68	66.46	63.41	57.93	48.17	64.02	34.15	68.29	58.44
StarCoderBase-7B	57.32	73.17	52.44	47.56	72.56	68.29	69.51	59.76	47.56	45.12	56.71	28.05	70.12	57.55
CodeGen2.5-7B-Multi	53.05	73.17	56.10	51.22	64.63	71.34	63.41	62.20	59.15	51.83	59.15	35.37	70.73	59.33
CodeGeeX2-6B	51.83	59.76	48.17	53.66	61.59	59.15	51.22	54.27	52.44	37.20	51.83	36.59	58.54	52.02

* The best scores are in bold.

their expected results. Consequently, our benchmark encompasses a comprehensive collection of 2296 solutions across 14 languages for 164 problems, each solution accompanied by numerous test cases.

D. Evaluation Metrics

Following the approach of TransCoder[19], we use computational accuracy (CA) as the evaluation metric for code translation. CA assesses whether the source and generated code produce the same output when given identical inputs. Specifically, only the first generated result is considered for evaluation (i.e., pass@1).

III. Answer to RQ1: Performance

In this section, we conduct a comprehensive study of over 110,000 translations to evaluate the capabilities of LLMs. Our study investigates 7 base models and 13 tuned LLMs across 26 translation tasks. It also takes CodeLlama-13B as a representative to delve into the LLMs' capacity for understanding and generating diverse languages on 182 tasks.

A. Overall Performance

We first explore how effectively code LLMs perform on code translation. To save computation resources while showcasing a wide array of languages, we select Python as the pivot language and investigate all its translations from and to other PLs, since Python is the most extensively studied PL in LLM-based code generation. For each translation task, we construct a prompt that comprises the task intent (i.e., "`Translate X code to Y`"), code in the source language, the function signature of the target language, and two in-context examples, as shown in Figure 1 (b).

The results are summarized in Table III. Overall, LLMs exhibit fairly high CA scores on translating code. For example, CodeLlama-13B achieves an average CA score of 76.55 on Python translation tasks. This demonstrates the vast potential of LLMs in code translation.

However, we notice that the translation performance of LLMs is highly biased: while LLMs exhibit high proficiency in translating from other languages to Python, they encounter challenges when translating from Python to other languages. For example, CodeLlama-13B achieves an average CA score of 86.16 when translating other languages to Python, while decreases to 66.93 when it comes to Python→X translations. We will perform a more fine-grained analysis of this biased phenomenon in Section III-B.

Across the studied LLMs, the CodeLlama series exhibits the best performance. Despite having fewer parameters, CodeLlama-13B significantly outperforms StarCoderBase-15B on all tasks. CodeGeeX2-6B obtains comparable results with CodeLlama-7B on X→Python translation but encounters difficulties in the opposite direction. Comparing models with similar sizes, the StarCoderBase-7B and CodeGen2.5-7B-Multi exhibit comparable performance. But both models fall behind CodeLlama-7B regarding overall performance across all tasks. We conjecture that such a difference is mainly caused by different training data. Particularly, both StarCoderBase-7B and CodeGen2.5-7B-Multi have been pre-trained on StarCoderData (311GB), which is much smaller than CodeLlama's dataset (859GB). In addition, the absence of Swift data in the StarCoderData results in subpar performance in Swift generation tasks.

Finding 1: LLMs exhibit asymmetrical capability in code translation: it demonstrates proficiency on X→Python tasks while struggling with Python→X tasks.

B. Comparing across Different Programming Languages

To further investigate LLMs' capability of comprehending and generating different PLs, we take CodeLlama-13B as a representative model to perform translation tasks between all language pairs. Table IV summarizes the results. The comprehending (generation) score of each language is denoted

TABLE IV: Translation Results by CodeLlama-13B between All Language Pairs.

Source\Target	C++	C#	Dart	Go	Java	JS	Kotlin	PHP	Python	Ruby	Rust	Scala	Swift	TS	Und. Avg.
C++	-	80.49	70.12	66.46	83.54	66.46	76.22	78.66	81.10	67.68	60.37	68.90	62.80	79.27	72.46
C#	71.34	-	64.02	67.07	73.78	79.27	79.88	72.56	91.46	78.05	57.93	76.83	64.02	82.32	73.73
Dart	71.34	68.90	-	66.46	64.63	78.66	72.56	72.56	82.32	85.37	53.66	67.68	59.15	81.10	71.11
Go	79.27	85.37	70.73	-	92.07	79.27	81.10	82.93	90.85	75.61	72.56	69.51	67.07	81.71	**79.08**
Java	69.51	82.93	61.59	68.29	-	79.88	71.95	70.12	87.80	82.93	58.54	75.61	61.59	81.71	73.27
JS	75.00	78.66	60.98	68.90	66.46	-	72.56	67.07	83.54	76.22	53.66	60.98	56.10	98.78	70.69
Kotlin	71.34	78.66	62.20	62.20	65.24	71.34	-	71.34	85.37	74.39	62.20	75.00	65.24	71.34	70.45
PHP	71.95	74.39	65.85	61.59	75.61	79.88	79.27	-	82.32	81.10	56.10	68.29	62.80	81.10	72.33
Python	67.68	70.73	58.54	58.54	71.34	72.56	77.44	66.46	-	63.41	59.15	71.95	62.20	70.12	<u>66.93</u>
Ruby	71.95	78.66	62.80	65.24	75.61	80.49	79.27	76.22	90.24	-	53.66	73.17	61.59	77.44	72.80
Rust	73.17	81.71	58.54	69.51	77.44	76.22	80.49	75.61	87.20	70.73	-	72.56	60.98	75.00	73.78
Scala	70.73	81.10	67.07	66.46	78.05	76.83	82.32	78.05	88.41	78.66	60.37	-	60.98	75.00	74.16
Swift	69.51	78.05	59.15	60.37	71.34	78.66	74.39	69.51	84.76	67.07	62.20	64.63	-	75.61	70.40
TS	66.46	78.66	59.76	67.07	68.29	98.78	76.22	70.12	84.76	76.83	55.49	64.63	58.54	-	71.20
Gen. Avg.	71.48	78.33	63.18	65.24	74.11	78.33	77.21	73.17	**86.02**	75.23	<u>58.91</u>	69.98	61.77	79.27	-

* The scores in bold denote the highest across rows or columns, while the underlined scores denote the lowest.

as the average CA score of all translation tasks where it is the source (target).

In terms of comprehending capability, we figure out that CodeLlama-13B attains the highest score on Go, while surprisingly gaining the lowest score on Python. One possible explanation for this counter-intuitive phenomenon is that Python has distinctive linguistic features (e.g., list comprehension) that make it difficult to understand. In comparison, Go is a procedural language, emphasizing the use of basic conditional and loop statements, which makes it easier to understand.

We surprisingly note that LLMs can understand unseen languages. For example, StarCoderBase and CodeGen2.5 models are not trained with Swift code, and thus they are incapable of translating Python to Swift. However, they achieve a competitive CA score in translating Swift to Python. Combined with Finding #1 (biased capability), we believe that LLM's capabilities of comprehending and generating a specific language are imbalanced. LLMs could be skilled at generating a certain language while poor at comprehending it, and vice versa.

In terms of generation capability, however, CodeLlama-13B achieves a substantially higher score on Python than other languages, in alignment with the previous finding that LLMs are proficient at generating Python code. Meanwhile, it has the lowest score on Rust. We attribute such a difference to the linguistic features. For example, Rust has a strict syntax on ownership and mutability, which is difficult for LLMs to comprehend and generate, while Python has a freer and more human-like syntax.

Finding 2: LLMs exhibit asymmetrical capabilities in comprehending and generating code for the same language. Among source languages to translate, Go is the easiest while Python is the hardest to understand. Among target languages, Python is the easiest while Rust is the hardest to generate.

C. Comparing across Base and Tuned LLMs

The releases of an LLM are often accompanied by their continual pre-trained or instruction-tuned variants, which often show better performance on certain tasks like code generation. To verify whether they work in code translation, we compare the performance of all 13 tuned variants to that of their base models. The results are summarized in Table V.

TABLE V: CA Differences between Base and Tuned LLMs.

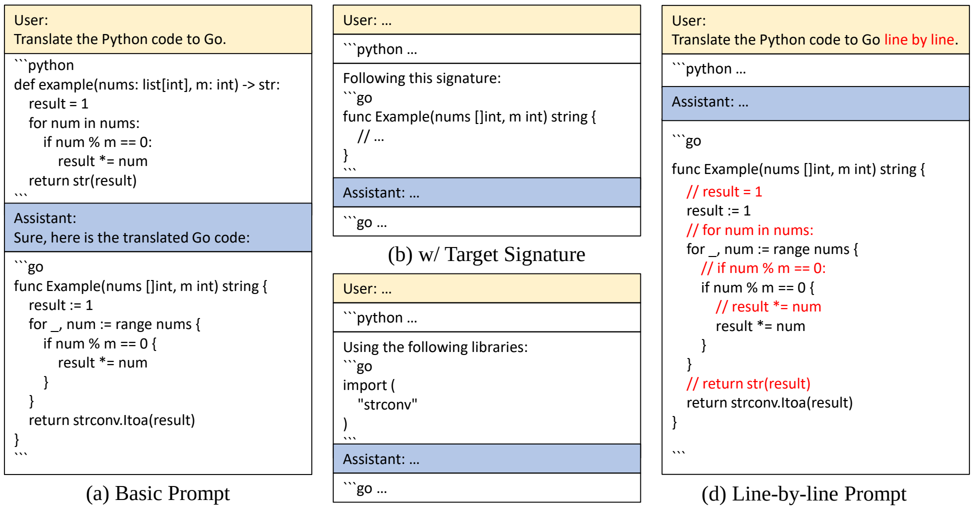
Model	Tuning[1]	X→Python		Python→X		Total	
		Avg.	Δ	Avg.	Δ	Avg.	Δ
CodeLlama-7B	-	76.83	-	63.70	-	70.26	-
CodeLlama-7B-Python	Py	82.69	+5.86	58.11	-5.58	70.40	+0.14
CodeLlama-7B-Instruct	In	80.40	+3.57	63.46	-0.23	71.93	+1.67
CodeLlama-13B	-	86.16	-	66.93	-	76.55	-
CodeLlama-13B-Python	Py	86.77	+0.61	62.43	-4.50	74.60	-1.95
CodeLlama-13B-Instruct	In	85.84	-0.33	66.84	-0.09	76.34	-0.21
CodeLlama-34B	-	84.34	-	73.17	-	78.75	-
CodeLlama-34B-Python	Py	82.32	-2.02	69.14	-4.03	75.73	-3.02
CodeLlama-34B-Instruct	In	83.26	-1.08	68.57	-4.60	75.92	-2.84
StarCoderBase	-	76.27	-	58.44	-	67.35	-
StarCoder	Py	78.71	+2.44	55.16	-3.28	66.93	-0.42
StarChat-α	In	73.50	-2.77	59.01	+0.56	66.25	-1.10
OctoCoder	Py, In	77.49	+1.22	50.61	-7.83	64.05	-3.31
StarCoderPlus	Py, NL	68.53	-7.74	44.23	-14.21	56.38	-10.98
StarChat-β	Py, NL, In	69.18	-7.08	47.80	-10.65	58.49	-8.87
CodeGen2.5-7B-Multi	-	72.61	-	59.33	-	65.97	-
CodeGen2.5-7B-Mono	Py	68.43	-4.18	24.30	-35.04	46.36	-19.61
CodeGen2.5-7B-Instruct	Py, In	75.00	+2.39	31.43	-27.91	53.21	-12.76

1 'Py', 'In', and 'NL' are abbreviations for further Python pre-training, instruction tuning, and further NL pre-training.

1) Effect of Further Python Pre-training: We find that further pre-training on Python is effective on smaller LLMs, such as CodeLlama-7B-Python and StarCoder, on X→Python tasks. Comparatively, they have a negative effect on Python→X tasks. A possible reason is that further training in Python intensifies LLMs' knowledge of Python while weakening other languages, thus harming the polyglot capability of LLMs.

2) Effect of Instruction Tuning: Instruction tuning also hampers code translation. For example, instruction tuning on both CodeLlama-13B and CodeLlama-34B leads to lower scores on all tasks. This is possibly caused by the discrepancy of objectives between fine-tuning and the final translation task. Instruction tuning aims at teaching LLMs to follow natural language instructions, which are closer to NL-to-code than code-to-code tasks.

3) Effect of Further NL Pre-training: LLMs (e.g., StarCoderPlus, StarChat-β) further pre-trained on NL corpora demonstrate significantly lower performance compared with



~~~
User:
Translate the Python code to Go.
```python
def example(nums: list[int], m: int) -> str:
    result = 1
    for num in nums:
        if num % m == 0:
            result *= num
    return str(result)
```
Assistant:
Sure, here is the translated Go code:
```go
func Example(nums []int, m int) string {
    result := 1
    for _, num := range nums {
        if num % m == 0 {
            result *= num
        }
    }
    return strconv.Itoa(result)
}
```
~~~

(a) Basic Prompt

~~~
User: ...
```python ...
Following this signature:
```go
func Example(nums []int, m int) string {
    // ...
}
```
Assistant: ...
```go ...
~~~

(b) w/ Target Signature

~~~
User: ...
```python ...
Using the following libraries:
```go
import (
    "strconv"
)
```
Assistant: ...
```go ...
~~~

(c) w/ Required Libraries

~~~
User:
Translate the Python code to Go line by line.
```python ...
Assistant: ...
```go
func Example(nums []int, m int) string {
    // result = 1
    result := 1
    // for num in nums:
    for _, num := range nums {
        // if num % m == 0:
        if num % m == 0 {
            // result *= num
            result *= num
        }
    }
    // return str(result)
    return strconv.Itoa(result)
}
```
~~~

(d) Line-by-line Prompt

Fig. 1: An Illustration of the Four Prompt Designs in Our Experiments.

TABLE VI: Results of Different Prompts@CodeLlama-13B.

Prompt	C++	C#	Dart	Go	Java	JS	Kotlin	PHP	Ruby	Rust	Scala	Swift	TS	Avg.
X→Python														
basic	78.05	85.98	75.61	87.80	85.98	82.32	81.71	81.10	82.93	82.93	85.37	80.49	81.71	82.46
w/ target signature	81.10	**91.46**	82.32	**90.85**	**87.80**	**83.54**	85.37	**82.32**	**90.24**	87.20	**88.41**	**84.76**	**84.76**	**86.16**
w/ required libraries	**81.71**	86.59	**85.37**	90.24	**87.80**	81.71	**85.98**	80.49	87.20	**89.63**	85.98	81.71	**84.76**	85.32
line-by-line	77.44	88.41	69.51	85.37	82.32	79.88	79.27	78.05	81.71	72.56	80.49	81.10	77.44	79.50
Python→X														
basic	59.76	66.46	57.32	54.88	68.90	70.12	74.39	64.63	59.15	**59.15**	68.29	56.10	65.85	63.46
w/ target signature	**67.68**	70.73	**58.54**	58.54	**71.34**	**72.56**	**77.44**	**66.46**	**63.41**	**59.15**	**71.95**	**62.2**	**70.12**	**66.93**
w/ required libraries	64.63	**77.44**	50.61	**59.15**	64.63	**72.56**	71.34	65.24	57.32	56.10	**71.95**	54.27	64.02	63.79
line-by-line	60.37	67.07	57.32	48.78	66.46	70.12	64.02	61.59	60.98	53.05	67.68	55.49	69.51	61.73

* The best scores are in bold.

their base models. This is probably because LLMs gradually forget PL knowledge during the process of NL pre-training.

Finding 3: The widely-used LLM optimization techniques, such as instruction tuning, show negative impacts on code translation, presumably due to the divergence between their objectives.

IV. Answer to RQ2: Prompting LLMs

Given that prompting is the most straightforward way of engaging with LLMs, we explore the design of prompts to harness the potential of LLMs for code translation. We focus on two key factors in prompt design, namely, the content of the prompt and the number of demonstration examples in the prompt, to figure out the optimal setting.

As illustrated in Figure 1, we design a straight prompt (a) that merely contains an intention: "`translate the X code to Y`" followed by code in the source language. Based on the straight prompt, we augment with three variants (b-d) using two popular prompt engineering techniques:

- *Prompt priming* provides additional context information to LLMs, including the function signature of the target code (Figure 1.b) and required libraries (Figure 1.c), which have been shown to be effective in code generation tasks [20]. We obtain the function signature by rule-based conversion and extracting library import statements of each question from our handcrafted solutions.
- *Chain-of-thought prompting* [21] asks an LLM to solve a problem via multiple intermediate steps. We prompt LLMs with examples of line-to-line alignment between the source and target languages (Figure 1.d), and ask them to translate the source code line by line.

We conduct experiments on CodeLlama-13B under a 2-shot setting. The results are presented in Table VI. We observe that providing more contextual hints to LLMs can greatly enhance the performance. Particularly, function signature in the target language plays a critical role in LLM-based code translation. The average CA score increases by 3.70 (3.47) in X→Python (Python→X) tasks. Prompts with required libraries demonstrate efficacy on X→Python while having a marginal impact on Python→X tasks. Meanwhile, line-by-line prompting leads to a decrease in the CA score.

To understand the impact of demonstration examples, we evaluate LLM performance under 0-shot, 1-shot, and 2-shot

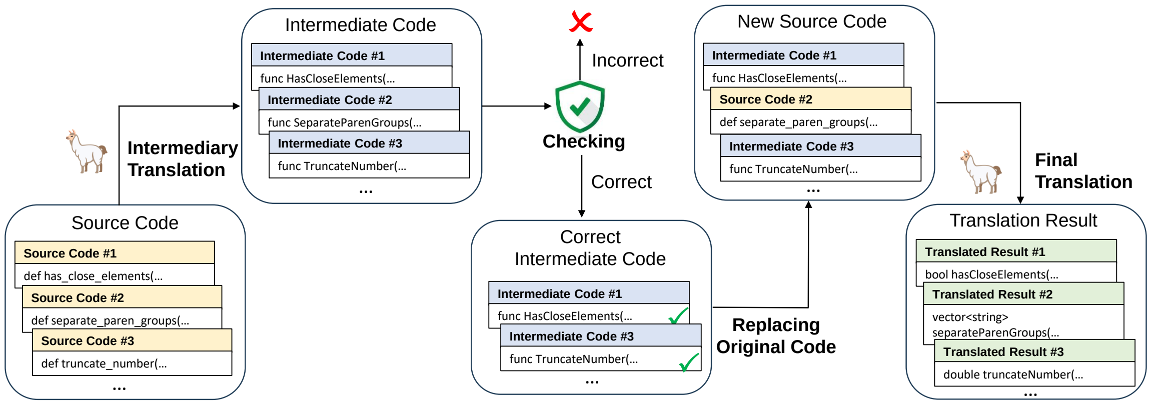


Fig. 2: An Illustration of Intermediary Translation.

TABLE VII: Results under Different N-shot Settings.

Task	Model	0-shot	1-shot	2-shot
X→Python Average	CodeLlama-7B	76.17	**79.04**	76.83
	CodeLlama-13B	82.36	85.60	**86.16**
	CodeLlama-34B	67.68	80.86	**84.34**
Python→X Average	CodeLlama-7B	60.27	63.41	**63.70**
	CodeLlama-13B	63.46	64.54	**66.93**
	CodeLlama-34B	50.98	69.56	**73.17**

* The best scores are in bold.

settings, respectively, using the best prompt template, i.e., prompt with target signature. We adopt CodeLlama (7B, 13B, and 34B) to perform the translation tasks. Table VII shows that in most cases, 2-shot translation achieves the best performance. Especially, CodeLlama-34B gains 43.5% (24.6%) higher CA with 2-shot settings than 0-shot on Python→X (X→Python) tasks.

Finding 4: Prompt priming and examples are the effective ways to unlock LLM capacity on code translation. Among various priming techniques, providing the function signature (obtained by simple rule-based conversion) of the target language yields the optimal translation.

V. Answer to RQ3: Improvement

To further uncover the potential of LLMs, we propose two novel LLM optimization methods: intermediary translation and self-training.

A. Intermediary Translation

Inspired by prior studies on multi-step LLM inferences [21, 22], we propose a two-step translation method, as illustrated in Figure 2. For a given code snippet to be translated, we first ask the LLM to transform it into an intermediate translation result. The translation then undergoes a test using the corresponding test cases. If correct, the LLM then translates the intermediate result to the target language.

We design two techniques for generating the intermediate translation:

1) *Style transfer*, which involves instructing the LLM to transform code written in the source language into code that adopts a comparable style to the target language. For example, when LLM performs Python→Java translation, it first translates the source code into a pure procedural style, e.g., converting the list comprehension and functional programming APIs into simple '`if`' and '`for`' statements.

2) *Intermediary language*, which asks the LLM to translate the code to a third-party language that acts as a lingua franca in the programming world. The idea is inspired by natural language translation scenarios where we often use English for communication across different natural languages.

B. Self-Training

To alleviate the data-hungry issue of parallel multilingual code, we propose a self-training method for code translation. We harness the capabilities of the LLM to generate source code, test cases, and parallel code on target languages for fine-tuning itself. Take Python→Go translation in Figure 3 as an example. To enhance the variety of code data, we first extract a range of APIs from Python documentation, including built-in APIs and those from standard libraries. Next, for each API, we ask the LLM to generate functions that invoke this API. We also ask the LLM to construct 5 corresponding test cases for each function and filter out functions that fail to pass the test. The remaining functions are translated to Go by prompting the LLM. The translated Go functions are also filtered through test cases. We convert the Python test cases in the previous step to Go with our rule-based test case generation tool. Finally, we obtain an augmented data set with verified Python-Go code. Subsequently, LLMs are fine-tuned on the augmented dataset.

C. Evaluation

1) Effect of Intermediary Translation: We conduct experiments with CodeLlama-13B on Python→X, which have been shown to be the most challenging translation tasks. During the process of style transfer, we construct 2-shot prompts to instruct LLMs to convert Python source code into a pure procedural style.

As shown in Table VIII, intermediary translation techniques, including style transfer and translation via intermediary lan-

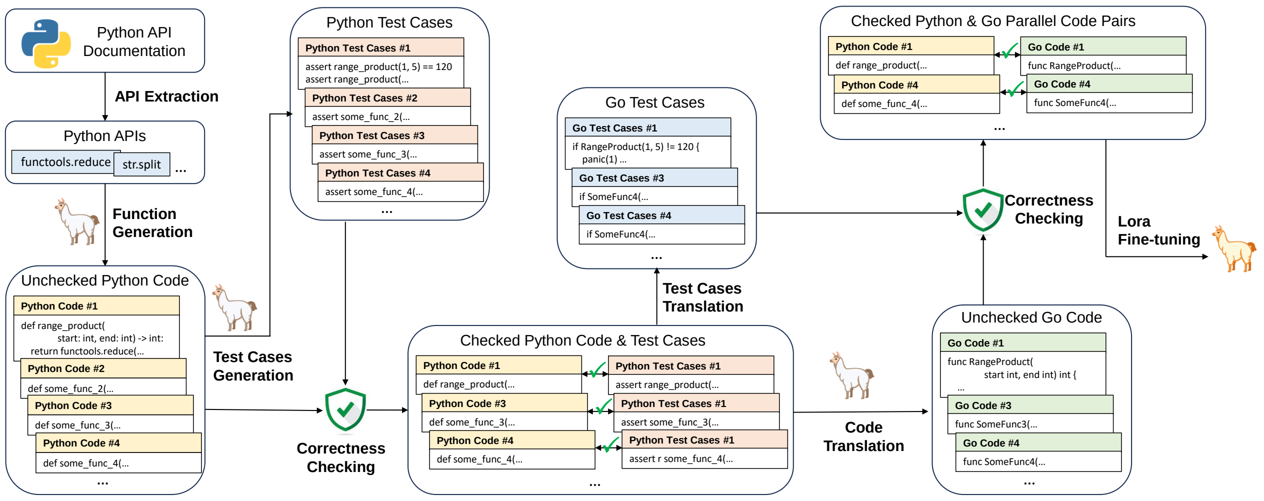


Fig. 3: An Illustration of Self-training.

guages, have achieved remarkable improvements overall. This can be attributed to their ability to assist LLMs to better understand the source language. C++, C#, Go and Java are effective intermediary languages that could improve performance. Among them, using Go as the intermediary language gets the best result, which is identical to the conclusion that Go is the easiest to understand (Section III-B).

We then conduct experiments on three CodeLlama models with style transfer and intermediary translation via Go, and the results are shown in Table IX. Both methods are effective across all LLMs, improving CA by 2.58∼4.73 points. While intermediary translation via Go consistently enhances the performance across all tasks, the impact of style transfer varies depending on the specific task. Languages such as Ruby and Scala do not reap the benefits of procedural style transfer, potentially because they are more aligned with the functional programming paradigm. This underscores the profound impact of source code style on the complexity of translation.

Finding 5: Intermediary translation effectively enhances the code translation capabilities of LLMs, and Go can serve as a lingua franca for translating between any two studied PLs.

2) Effect of Self-Training: We generate 20,000 Python functions among which 3,915 are verified to be correct, and then fine-tune CodeLlama-13B on three parallel code datasets:

1) *Verified Pass@1 Data*: We generate 1 translation for each Python function with temperature = 0.01 and obtain 2,407 verified Python-Go code pairs.
2) *Verified Pass@5 Data*: We generate 5 translations for each Python function with temperature = 0.8 and obtain 3,224 verified Python-Go code pairs. When there are multiple valid results, a random one is selected.
3) *Unchecked Data*: We generate 1 translation for each Python function with temperature = 0.01 *without* verification and obtain 3,915 Python-Go code pairs.

Table X summarizes the evaluation results. For the Python→Go task, fine-tuning on the self-generated data gains remarkable improvement: the verified Pass@5 (Pass@1) data achieves 14.57% (6.25%) more CA score. It shows the effectiveness of self-training with validated data. Such enhancement can also be transferred to generating other PLs, that is, fine-tuning on Python→Go advances all of Python→X tasks.

For the Go→Python task, however, fine-tuning leads to worse performance. It indicates that fine-tuning on A→B task may in turn degrade the performance of B→A translation. A possible explanation is the different characteristics between the source code and target code generated by LLM: the source code aims for diversity and contains rich APIs, while the target code only aims for correctness and thus tends to be simple. Therefore, it eventually harms the comprehension of target PLs. Surprisingly, translating other PLs to Python benefits from all fine-tuning procedures, as all the fine-tuning datasets include verified Python code.

Finding 6: Self-training significantly optimizes LLM's performance on the targeted translation task, and also boosts other translation tasks due to LLM's transfer learning capability.

3) Effect of Combined Approach: Finally, we assess the effectiveness when adopting both optimization methods. As shown in Table XI, combining self-training and intermediary translation has the best performance of 74.77 average CA on Python→X tasks. It gains 4.9% higher CA than using intermediary translation only, 4.7% more than using self-training only, and 11.7% more than the original CodeLlama-13B.

D. Case Study

Figure 4 shows two cases of Python→Java translations by various approaches. In both cases, the baseline fails to adhere to the original semantics. Adopting intermediary translation

TABLE VIII: Results of Intermediary Translations@CodeLlama-13B on Python→X Tasks.

Approach	C++	C#	Dart	Go	Java	JS	Kotlin	PHP	Ruby	Rust	Scala	Swift	TS	Avg.
Baseline	67.68	70.73	58.54	58.54	71.34	72.56	77.44	66.46	63.41	59.15	71.95	62.20	70.12	66.93
w/ ST	67.07	78.66	61.59	61.59	78.05	76.22	76.22	**75.61**	61.59	62.20	69.51	65.24	72.56	69.70
w/ IL(C++)	-	75.61	63.41	**67.07**	**78.66**	75.61	72.56	73.78	67.68	**62.80**	67.68	61.59	75.00	69.93
w/ IL(C#)	67.07	-	61.59	63.41	66.46	75.00	80.49	71.95	75.00	60.37	**76.22**	65.85	75.61	69.98
w/ IL(Dart)	63.41	69.51	-	61.59	64.63	72.56	70.73	68.90	75.61	53.66	60.37	57.32	67.68	64.96
w/ IL(Go)	**74.39**	76.22	**70.12**	-	**78.66**	72.56	**81.10**	71.95	70.12	59.76	71.95	**67.07**	74.39	**71.29**
w/ IL(Java)	65.85	73.17	62.80	66.46	-	**78.05**	73.78	73.17	**76.22**	60.98	71.34	61.59	**76.22**	70.07
w/ IL(JavaScript)	67.07	66.46	56.10	57.32	67.07	-	70.12	68.29	68.90	56.71	66.46	59.15	75.00	65.48
w/ IL(Kotlin)	69.51	68.29	57.32	59.76	64.02	75.00	-	68.29	64.02	55.49	64.63	55.49	72.56	65.52
w/ IL(PHP)	60.37	66.46	58.54	57.93	70.73	75.61	73.78	-	**76.22**	52.44	68.29	59.15	70.12	65.85
w/ IL(Ruby)	67.07	73.17	57.93	62.20	70.73	73.78	71.34	68.90	-	55.49	70.12	60.37	70.73	66.56
w/ IL(Rust)	73.17	71.95	58.54	59.76	72.56	71.34	71.95	70.73	67.68	-	68.29	57.32	70.73	67.17
w/ IL(Scala)	68.29	**79.88**	52.44	58.54	72.56	75.00	68.29	69.51	70.73	57.32	-	59.76	70.73	67.31
w/ IL(Swift)	67.07	71.34	57.93	60.98	71.34	72.56	75.00	63.41	60.98	55.49	64.63	-	67.68	65.43
w/ IL(TypeScript)	66.46	64.02	53.05	57.32	65.85	75.61	71.34	65.85	67.07	55.49	67.68	56.10	-	64.30

* The best scores are in bold. ST stands for style transfer, and IL(X) stands for translation with intermediary language X.

TABLE IX: Improved CA Scores by Intermediary Translation on Python→X Tasks.

Model	Appoach	C++	C#	Dart	Go	Java	JS	Kotlin	PHP	Ruby	Rust	Scala	Swift	TS	Avg.
CodeLlama-7B	w/ ST	0.00	+3.05	+7.93	+3.66	+0.61	+0.61	+3.05	+9.75	-4.26	+5.49	-0.61	+1.22	+3.05	+2.58
	w/ IL(Go)	+1.82	+1.83	+9.14	-	+2.44	+3.05	+3.66	+8.53	+2.44	0.00	+1.22	+1.22	+3.05	+3.20
CodeLlama-13B	w/ ST	-0.61	+7.93	+3.05	+3.05	+6.71	+3.66	-1.22	+9.15	-1.82	+3.05	-2.44	+3.04	+2.44	+2.77
	w/ IL(Go)	+6.71	+5.49	+11.58	-	+7.32	0.00	+3.66	+5.49	+6.71	+0.61	0.00	+4.87	+4.27	+4.73
CodeLlama-34B	w/ ST	+2.44	+1.83	+6.71	+4.27	+6.71	+4.88	+1.83	+10.98	-11.58	+4.88	0.00	+3.05	+3.05	+3.00
	w/ IL(Go)	+1.22	+1.83	+7.93	-	+1.22	+2.44	-1.83	+8.54	-1.22	+4.27	+3.66	+3.66	+4.27	+3.00

* ST stands for style transfer, and IL(Go) stands for intermediary translation via Go.

TABLE X: Results of Self-Training with Different Fine-tuning Data@CodeLlama-13B.

Fine-tuning Data	C++	C#	Dart	Go	Java	JS	Kotlin	PHP	Ruby	Rust	Scala	Swift	TS	Avg.
X→Python														
None	81.10	**91.46**	82.32	**90.85**	87.80	83.54	85.37	82.32	90.24	87.20	88.41	84.76	84.76	86.16
Unchecked Data	84.76	87.20	82.32	89.63	87.20	87.20	87.20	87.80	**90.85**	90.85	**90.85**	86.59	86.59	87.62
Checked Pass@1 Data	87.20	89.02	**84.15**	87.20	**88.41**	**89.02**	86.59	87.80	**90.85**	**91.46**	90.24	85.98	**88.41**	**88.18**
Checked Pass@5 Data	**89.02**	89.02	**84.15**	87.80	87.80	87.20	**89.63**	**89.02**	90.24	89.63	89.63	**87.20**	85.98	**88.18**
Python→X														
None	67.68	70.73	58.54	58.54	71.34	72.56	77.34	66.46	63.41	59.15	71.95	62.20	70.12	66.93
Unchecked Data	66.46	67.68	57.93	58.54	69.51	74.39	78.05	65.85	68.29	60.37	70.12	60.37	71.95	66.89
Checked Pass@1 Data	**68.90**	71.34	63.41	62.20	71.95	76.22	81.10	68.29	71.34	**63.41**	71.95	64.63	75.61	70.03
Checked Pass@5 Data	**68.90**	**71.95**	**65.85**	**67.07**	**76.22**	**77.44**	**82.32**	**69.51**	**76.22**	57.93	**72.56**	**65.85**	**76.83**	**71.43**

* The best scores are in bold.

via Go leads to an easier style of #1 translation, but can not help example #2. Adopting self-training leads to correct semantics in both cases, but it fails in example #1 due to API misuse. Finally, the combined approach derives correct translation in both cases, achieving the best performance.

These examples demonstrate the effectiveness of our proposed methods: intermediary translation tends to generate easier code and reduces errors related to API usage; self-training makes LLMs better adhere to the original semantics; and these methods are fairly compatible with each other. More translation results are available at our online repository [18].

VI. Discussion

A. *Future Directions*

Despite the effectiveness of our optimization methods, we believe that LLM-based code translation still has a big improvement space. We delineate two future directions.

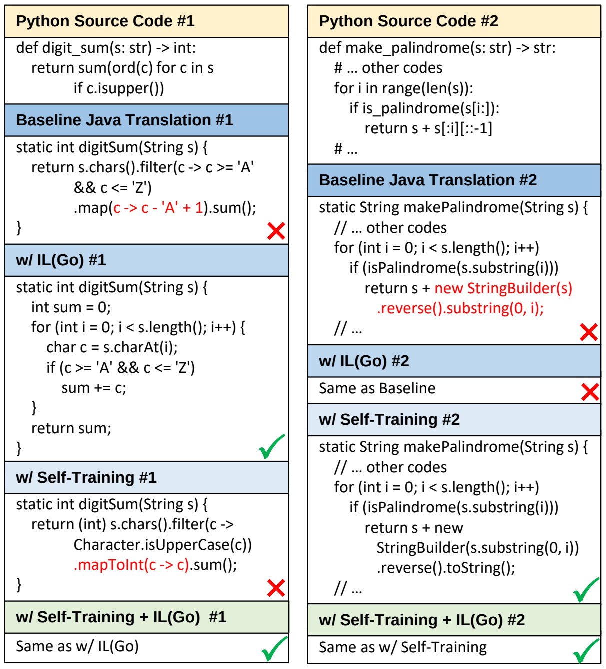
1) Intermediary Language Selection: Although adopting Go as lingua franca has promising results, it might not be the best intermediary language for all language pairs. Therefore, greater attention should be devoted to how to automatically choose the optimal intermediary language, e.g., by leveraging language embedding.

2) High-Quality Parallel Data Generation: Our study has demonstrated the effectiveness of the self-training method, which involves utilizing LLM itself to generate parallel code. However, it only ensures the correctness and API coverage of the generated code data. As for fine-tuning LLMs, the training data should also be diverse and fully reflect the usage and characteristics of programming languages. Further enhancing the quality of generated code could continually improve LLMs' performance in code translation tasks.

TABLE XI: Results of Different Optimization Approaches@CodeLlama-13B on Python→X Tasks.

Approach[1]	C++	C#	Dart	Go	Java	JS	Kotlin	PHP	Ruby	Rust	Scala	Swift	TS	Avg.
CodeLlama-13B	67.68	70.73	58.54	58.54	71.34	72.56	77.44	66.46	63.41	59.15	71.95	62.20	70.12	66.93
w/ ST	67.07	78.66	61.59	61.59	78.05	76.22	76.22	75.61	61.59	62.20	69.51	65.24	72.56	69.70
w/ IL(Go)	**74.39**	76.22	70.12	58.54[2]	78.66	72.56	81.10	71.95	70.12	59.76	71.95	67.07	74.39	71.29
w/ Self-Training	68.90	71.95	65.85	67.07	76.22	77.44	82.32	69.51	76.22	57.93	72.56	65.85	76.83	71.43
w/ Self-Training + ST	70.73	**81.1**	64.63	**68.90**	**80.49**	**81.71**	**84.15**	**76.83**	68.90	**67.68**	69.51	**71.34**	80.49	74.34
w/ Self-Training + IL(Go)	69.51	77.44	**75.00**	67.07[1]	79.27	81.10	82.93	73.17	**76.83**	66.46	**73.78**	68.29	**81.10**	**74.77**

1 The best scores are in bold. ST stands for style transfer, and IL(Go) stands for intermediary translation via Go.
2 Since IL(Go) can not be applied to Python→Go translation, the original scores are displayed here.



Python Source Code #1
```
def digit_sum(s: str) -> int:
    return sum(ord(c) for c in s
            if c.isupper())
```
Baseline Java Translation #1
```
static int digitSum(String s) {
    return s.chars().filter(c -> c >= 'A'
            && c <= 'Z')
            .map(c -> c - 'A' + 1).sum();
}
```
✗

w/ IL(Go) #1
```
static int digitSum(String s) {
    int sum = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c >= 'A' && c <= 'Z')
            sum += c;
    }
    return sum;
}
```
✓

w/ Self-Training #1
```
static int digitSum(String s) {
    return (int) s.chars().filter(c ->
            Character.isUpperCase(c))
            .mapToInt(c -> c).sum();
}
```
✗

w/ Self-Training + IL(Go) #1

Same as w/ IL(Go) ✓

Python Source Code #2
```
def make_palindrome(s: str) -> str:
    # ... other codes
    for i in range(len(s)):
        if is_palindrome(s[i:]):
            return s + s[:i][::-1]
    # ...
```
Baseline Java Translation #2
```
static String makePalindrome(String s) {
    // ... other codes
    for (int i = 0; i < s.length(); i++)
        if (isPalindrome(s.substring(i)))
            return s + new StringBuilder(s)
              .reverse().substring(0, i);
    // ...
```
✗

w/ IL(Go) #2

Same as Baseline ✗

w/ Self-Training #2
```
static String makePalindrome(String s) {
    // ... other codes
    for (int i = 0; i < s.length(); i++)
        if (isPalindrome(s.substring(i)))
            return s + new
              StringBuilder(s.substring(0, i))
              .reverse().toString();
    // ...
```
✓

w/ Self-Training + IL(Go) #2

Same as w/ Self-Training ✓

Fig. 4: Examples of Python→Java Translations.

B. Threat to Validity

Internal Validity. The code snippets examined in our study may differ from that in real-world projects. While correctness is ensured, the handcrafted code might not reflect the programming habits of real developers, which might affect the final translation results. In the future, we will investigate the translation of real-world software projects.

External Validity. This paper has studied four families of code LLMs, which might not represent all LLMs. Furthermore, our optimization methods are only evaluated on CodeLlama. Although these methods are applicable to any LLM, other LLMs such as GPT-4 may exhibit different performances. We leave code translation studies on other LLMs for our future work.

VII. Related Work

Prior work has proposed rule-based, supervised, and unsupervised approaches for code translation tasks. Rule-based approaches, including CxGo [23] and C2Rust [24], rely on conversion rules and usually produce code that lacks readability. Supervised approaches learn from existing parallel code data and produce code closer to the real-world style. It is supported by statistical translation approaches [25], tree-based neural networks [26], and pre-trained language models [27]. However, their performance and generalizability are restricted by the scarcity and quality of parallel data.

Unsupervised approaches alleviate such limitations. Lachaux et al. [19] presented TransCoder, which leverages back-translation to train models without parallel data. Their later work [28, 29] leveraged unit tests and compiler representations for further optimization. However, they still require specialized training for each PL pair and thus demand substantial computation resources for generalization.

Due to the extraordinary efficacy of LLMs [7, 9, 10, 30], researchers have started to empirically study LLMs' code translation capability. Jiao et al. [1] proposed a 4-type taxonomy for code translation and evaluated LLMs' performance on each type of translation task. Pan et al. [3] generalized 15 bug categories from LLMs' unsuccessful code translations. Yang et al. [31] proposed UniTrans, a code translation framework leveraging LLMs by generating test cases and repairing. However, they studied a very limited number of PLs and only proposed prompt-crafting solutions.

Correspondingly, our work conducts a comprehensive study on code translation, covering both common and emerging PLs such as Rust, Kotlin, and Swift. We are also the first to propose systematical optimization solutions for LLMs on code translation tasks, including intermediary translation and self-training.

VIII. Conclusion

This paper conducts a large-scale empirical study to exploit the capabilities and limitations of LLMs in code translation. By investigating 4 popular LLM families on code translation tasks between 14 programming languages, we draw the conclusion that LLMs have biased and sub-optimal code translation capabilities. Further study on LLM variants proves that widely-used LLM optimization techniques are not effective in improving code translation tasks. To that end, we propose two optimization methods, namely, intermediary translation and self-training. Our evaluation on a variety of translation tasks proves their effectiveness and identifies Go as the best intermediary language.

We have released our benchmarks, source code, experimental results, and test generation tool online [18].

Acknowledgments

This research is supported by National Key R&D Program of China (Grant No. 2023YFB4503802) and National Natural Science Foundation of China (Grant No. 62102244, 62232003).

References

[1] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, "On the evaluation of neural code translation: Taxonomy and benchmark," in *ASE*, 2023, pp. 1529–1541.

[2] R. Li, L. B. Allal, Y. Zi, and et al., "StarCoder: may the source be with you!" *CoRR*, vol. abs/2305.06161, 2023.

[3] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *ICSE*, 2024.

[4] S. Chaudhary, "Code Alpaca: An instruction-following LLaMA model for code generation," https://github.com/sahil280114/codealpaca, 2023.

[5] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," *CoRR*, vol. abs/2308.07124, 2023.

[6] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, and et al., "Code Llama: Open foundation models for code," *CoRR*, vol. abs/2308.12950, 2023.

[7] H. Touvron, L. Martin, K. Stone, P. Albert, and et al., "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023.

[8] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The Stack: 3 TB of permissively licensed source code," *CoRR*, vol. abs/2211.15533, 2022.

[9] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "CodeGen2: Lessons for training LLMs on programming and natural languages," *CoRR*, vol. abs/2305.02309, 2023.

[10] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "CodeGeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *KDD*, 2023, pp. 5673–5684.

[11] A. Zeng, X. Liu, Z. Du, Z. Wang, H. Lai, M. Ding, Z. Yang, Y. Xu, W. Zheng, X. Xia, W. L. Tam, Z. Ma, Y. Xue, J. Zhai, W. Chen, Z. Liu, P. Zhang, Y. Dong, and J. Tang, "GLM-130B: an open bilingual pre-trained model," in *ICLR*, 2023.

[12] A. Köpf, Y. Kilcher, D. von Rütte, and et al., "Openassistant conversations - democratizing large language model alignment," in *NeurIPS*, 2023.

[13] G. Penedo, Q. Malartic, D. Hesslow, R. Cojocaru, H. Alobeidli, A. Cappelli, B. Pannier, E. Almazrouei, and J. Launay, "The RefinedWeb dataset for falcon LLM: outperforming curated corpora with web data only," in *NeurIPS*, 2023.

[14] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "GPT3.int8(): 8-bit matrix multiplication for transformers at scale," in *NeurIPS*, 2022.

[15] S. Lu, D. Guo, S. Ren, and et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *NeurIPS Datasets and Benchmarks*, 2021.

[16] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K. Chang, "AVATAR: A parallel corpus for Java-Python program translation," in *Findings of ACL*, 2023, pp. 2268–2281.

[17] M. Chen, J. Tworek, H. Jun, Q. Yuan, and et al., "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[18] Q. Tao, T. Yu, X. Gu, and B. Shen. Project webpage: Unraveling the potential of large language models in code translation. [Online]. Available: https://github.com/q4x3/poly-humaneval

[19] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *NeurIPS*, 2020.

[20] X. Gu, M. Chen, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *TOSEM*, 2024.

[21] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, 2022.

[22] Y. Fu, H. Peng, A. Sabharwal, P. Clark, and T. Khot, "Complexity-based prompting for multi-step reasoning," in *ICLR*, 2023.

[23] Tool for transpiling C to Go. [Online]. Available: https://github.com/gotranspile/cxgo

[24] Migrate C code to Rust. [Online]. Available: https://github.com/immunant/c2rust

[25] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-Conquer approach for multi-phase statistical migration for source code (T)," in *ASE*, 2015, pp. 585–596.

[26] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *NeurIPS*, 2018, pp. 2552–2562.

[27] X. Li, S. Yuan, X. Gu, Y. Chen, and B. Shen, "Few-shot code translation via task-adapted prompt learning," *J. Syst. Softw.*, vol. 212, p. 112002, 2024.

[28] B. Rozière, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," in *ICLR*, 2022.

[29] M. Szafraniec, B. Rozière, H. Leather, P. Labatut, F. Charton, and G. Synnaeve, "Code translation with compiler representations," in *ICLR*, 2023.

[30] A. Josh, A. Steven, A. Sandhini *et al.*, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023.

[31] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and

unleashing the power of large language models in automated code translation," *CoRR*, vol. abs/2404.14646, 2024.