ExeCoder: Empowering Large Language Models with Executability Representation for Code Translation

Minghua He[* †1]**, Yue Chen**[*†1]**, Fangkai Yang**[‡2]**, Pu Zhao**[2]**, Wenjie Yin**[3]**,**
Yu Kang[2]**, Qingwei Lin**[2]**, Saravan Rajmohan**[2]**, Dongmei Zhang**[2]
[1]Peking University, [2]Microsoft, [3]KTH Royal Institute of Technology
hemh2120@stu.pku.edu.cn, fangkai.yang@microsoft.com

Abstract

Code translation is a crucial activity in the software development and maintenance process, and researchers have recently begun to focus on using pre-trained large language models (LLMs) for code translation. However, existing LLMs only learn the contextual semantics of code during pre-training, neglecting executability information closely related to the execution state of the code, which results in unguaranteed code executability and unreliable automated code translation. To address this issue, we propose ExeCoder, an LLM specifically designed for code translation, aimed at utilizing executability representations such as functional semantics, syntax structures, and variable dependencies to enhance the capabilities of LLMs in code translation. To evaluate the effectiveness of ExeCoder, we manually enhanced the widely used benchmark TransCoder-test, resulting in a benchmark called TransCoder-test-X that serves LLMs. Evaluation of TransCoder-test-X indicates that ExeCoder achieves state-of-the-art performance in code translation, surpassing existing open-source code LLMs by over 10.88% to 38.78% and over 27.44% to 42.97% on two metrics, and even outperforms the renowned closed-source LLM GPT-4o. Code is available at https://aka.ms/execoder

1 Introduction

Code translation aims to convert code written in one programming language into another. Translation between different languages can assist developers in adapting applications to new business and environments, demonstrating significant demand and value in real industrial contexts. For example, the Commonwealth Bank of Australia spent around $750 million and five years converting its platform from COBOL to Java (Roziere et al., 2020).

---

[*]These authors contributed equally to this work.
[†]This work was done during the internship at Microsoft.
[‡]Corresponding author.

Input Function

```
# C++
int findMaxAverage(int arr[], int n, int k) {
    if (k > n) return -1;
    int sum = arr[0];
    for (int i = 1; i < k; i++) sum += arr[i];
    int max_sum = sum, max_end = k - 1;
}
```

Translation w/o Executability Representation

```
# Python
def findMaxAverage(arr, n, k):
    if k > n:
        return -1
    sum = sum(arr[:k])
    max_sum = sum
    max_end = k - 1
```

Translation with Executability Representation

```
# Python
def findMaxAverage ( arr , n , k ) :
    if ( k > n ) : return - 1
    sum = arr [ 0 ]
    for i in range ( 1 , k ) : sum += arr [ i ]
    max_sum = sum
    max_end = k - 1
```

Figure 1: **Executability Representation for Code Translation.** Existing models simply copy variable name from the source code, ignoring conflicts with called built-in function, which leads to TypeError exception. The model that considers code executability learn the syntactic structure of source code and avoid call conflicts through explicit loop summation.

Given the high labor costs associated with code translation, the development of automated translation tools has been extensively researched. Recent approaches (Huang et al., 2023; Roziere et al., 2021; Wen et al., 2022; Zhu et al., 2022b) utilize the paradigm of neural machine translation (NMT), aiming to learn the mapping from source language sequences to target language sequences. Despite achieving some success, their effectiveness heavily relies on the availability of cross-language parallel data (Ahmad et al., 2021a). For most languages, parallel resources are scarce or entirely absent. To overcome the limitations of NMT-based approaches, some studies (Yin et al., 2024; Lu et al., 2025; Mei et al., 2025; Macedo et al., 2024; Yang et al., 2024) have explored the use of large language models (LLMs) for code translation. These LLMs

are pre-trained on a vast array of open-source code, generating code by learning the contextual semantics of the code and demonstrating excellent understanding across various programming languages. Nevertheless, existing research indicates that the correct translation rate of advanced LLMs ranges from 2.1% to 47.3%, resulting in 15 different types of execution errors (Pan et al., 2024). Leveraging the potential of LLMs for code translation poses significant challenges.

In our view, the fundamental reason for this issue is the discrepancy between the pre-training tasks of existing LLMs and the requirements of code translation, as illustrated in Figure 1. Unlike natural languages, programming languages possess additional information that indicates the execution state of the code, including more complex functionalities, syntax, and variables, referred to as executability information (Jiao et al., 2023). Executability information pertains to the implementation logic of the source code, the acquisition of which relies on code analysis tools designed by programming language experts and cannot be directly inferred from the context of the source code. Code translation requires that the generated code executes correctly as intended. However, existing LLMs only learn the contextual semantics of code during pre-training, neglecting the executability information closely related to the execution state of the code, which compromises the executability of the generated code and prevents reliable automated code translation.

To address these issues and harness the potential of LLMs in code translation, we propose ExeCoder. This is an LLM specifically designed for code translation, aimed at enhancing the ability of LLMs by leveraging executability representations such as functional semantics, syntactic structures, and variable dependencies within the code. To extract executability knowledge from the code, ExeCoder first devises a representation strategy for executability knowledge. ExeCoder employs external code analysis tools to acquire three types of executability knowledge from the source code and meticulously designs encoding strategies to convert this knowledge into text that is easily interpretable by LLMs. To learn executability knowledge from the code, ExeCoder introduces a Progressive Executability Representation Learning strategy (PERL). The key idea is that the functional semantics, syntactic structures, and variable dependencies of the source code represent progressively refined executability information, and this staged, progressive learning aligns

with the learning theories of programming experts, facilitating representation learning.

To evaluate the effectiveness of ExeCoder, we conducted evaluation on the widely used code translation benchmark, TransCoder-test (Roziere et al., 2020). However, TransCoder-test can only evaluate specific implementations of the code. To address this limitation, we enhanced the TransCoder-test benchmark, resulting in a new benchmark called TransCoder-test-X, capable of evaluating the code translation capabilities of LLMs. Evaluation on TransCoder-test-X indicates that ExeCoder achieves SOTA performance in code translation tasks, surpassing existing open-source large models by over 10.88% to 38.78% and over 27.44% to 42.97% on two metrics, respectively. Notably, the ExeCoder outperforms renowned closed-source LLMs, including GPT-3.5, GPT-4, and GPT-4o, highlighting the significant role of executability representations in code translation.

In summary, our contributions are as follows:

- We developed ExeCoder, a LLM specifically designed for code translation, which significantly outperforms all other open-source code LLMs, achieving SOTA performance. Notably, the ExeCoder surpasses well-known the renowned closed-source LLM GPT-4o.

- We propose a Progressive Executability Representation Learning strategy that aligns with the learning theory of programming experts and effectively learns executability representations of code.

- We enhanced the widely used code translation benchmark, TransCoder-test, resulting in a new benchmark called TransCoder-test-X, which is capable of evaluating the code translation abilities of LLMs.

- We conducted a preliminary study that emphasizes the critical role of executability representations of code in achieving excellent code translation performance.

2 Related Work

2.1 Translation of Programming Languages

The translation between programming languages has been an active research field (Liu et al., 2024; Eniser et al., 2024; Khan et al., 2024; Zhu et al., 2024; Luo et al., 2024), recent work has been
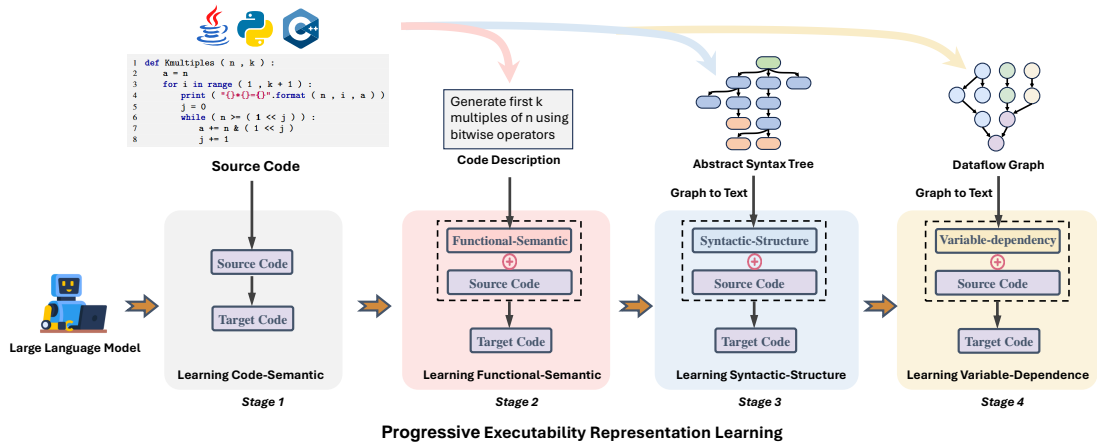
Figure 2: **The ExeCoder's pipeline.** The ExeCoder first utilizes a text that describes the code functionality to encode functional semantic information. Subsequently, the ExeCoder uses an abstract syntax tree (AST) to encode the syntactic structure information. Finally, the ExeCoder employs a data flow graph (DFG) to encode the variable dependency information. With the benefit of the well-designed Progressive Executability Representation Learning (PERL) strategy, the ExeCoder can fully leverage the executability representation to enhance the cross-language understanding capabilities of LLMs.

based on the paradigm of neural machine translation. TransCoder (Roziere et al., 2020) is the most classic unsupervised machine translation method, specifically designed with pre-training tasks to learn the semantics of source code. Other works aim to leverage code analysis tools to enhance the capabilities of code translation. SDA-Trans (Liu et al., 2023) parses the program's syntax tree from the source code to acquire knowledge of the syntactic structure. TransCoder-IR (Szafraniec et al., 2023) parses its low-level representation, LLVM IR, from the source code and designs corresponding pre-training tasks to improve code representation. As LLMs demonstrate exceptional capabilities, their potential in code translation has also been investigated. TRANSAGENT (Yuan et al., 2024) utilizes LLMs as agents to rectify syntax and semantic errors in code translation. LASSI (Dearing et al., 2024) proposes a self-enhancement method, providing feedback during compilation to the LLM through guided debugging and refactoring prompts. Inspired by these works, ExeCoder utilizes code analysis tools to acquire knowledge related to code executability and injects this knowledge into LLMs to enhance their code translation capability.

2.2 Large Language Models for Code

With the development of LLMs, language models tailored for code have garnered significant attention in the community. Representative LLMs include CodeLlama (Roziere et al., 2023), Qwen-Coder (Hui et al., 2024), StarCoder (Li et al., 2023), CodeT5+ (Wang et al., 2023), and Deepseek-Coder

(Guo et al., 2024). These LLMs benefit from extensive pre-training on large code corpora, resulting in a strong understanding of the semantics of source code. Nevertheless, compared to state-of-the-art closed-source LLMs such as GPT-4 (Achiam et al., 2023), these models often lag in capability. To further enhance the ability of code LLMs to address specific coding issues, other works have focused on fine-tuning of pre-trained LLMs. Recent efforts include InstructCoder (Li et al., 2024), WizardCoder (Luo et al., 2023), PanGu-Coder2 (Shen et al., 2023), Magicoder (Wei et al., 2024), WaveCoder (Yu et al., 2024),. These LLMs are built upon pre-trained code LLMs and undergo post-training to further enhance model capabilities. However, these code LLMs are not specifically tailored for code translation, they merely learn from source code while neglecting the requirements for executability inherent in code translation. Unlike these works, ExeCoder encodes executability representation of source code and then employs instruction-tuning to compel LLMs to learn these executability representations, improving their code translation abilities.

3 Methodology

3.1 Overview

In this paper, we introduce ExeCoder, a LLM tailored for code translation task, which seeks to improve the capabilities of LLMs in code translation by leveraging executability representations. Executability representation refers to information indicating the execution state of the source code, which

Input Code Snippet	w/o Functional-semantic Representation	with Functional-semantic Representation

```cpp
# C++
int smallest ( int x, int y, int z ) {
  if ( ! ( y / x ) )
    return ( ! ( y / z ) ) ? y : z;
  return ( ! ( x / z ) ) ? x : z;
}
```

```python
# Python
def smallest(x, y, z):
    if not (y % x):
        return y if not (y % z) else z
    return x if not (x % z) else z
```

```python
# Python
def smallest ( x , y , z ) :
    if ( not ( y / x ) ) :
        return y if not ( y / z ) else z
    return x if not ( x / z ) else z
```

Input Code Snippet	w/o Syntactic-structure Representation	with Syntactic-structure Representation

```cpp
# C++
for (int i = num.size() - 1; i >= 0; i--) {
    int digit = num[i] - '0';
    result += digit * series[series_index];
    series_index = (series_index + 1) % 6;
    result %= 7;
}
```

```python
# Python
for i in range(len(num) - 1, -1, -1):
    digit = int(num[i])
    result += digit * series[series_index]
    series_index = (series_index + 1) % 6
    result %= 7
```

```python
# Python
for i in range ( len ( num ) - 1 , - 1 , - 1 ) :
    digit = ord ( num [ i ] ) - ord ( '0' )
    result += digit * series [ series_index ]
    series_index = ( series_index + 1 ) % 6
    result %= 7
```

Input Code Snippet	w/o Variable-dependency Representation	with Variable-dependency Representation

```cpp
# C++
string minLexRotation ( string st ) {
    int n = st . length ( );
    string arr [ n ];
    string concat = st + st;
    for ( int i = 0; i < n; i ++ )
        arr[i] = concat.substr(i, n);
    sort ( arr, arr + n );
    return arr [ 0 ];
}
```

```python
# Python
def minLexRotation ( st ):
    n = len ( st )
    arr = [ st [ i : ] for i in range ( n ) ]
    arr.sort ( )
    return arr [ 0 ]
```

```python
# Python
def minLexRotation ( st ) :
    n = len ( st )
    arr = [ None ] * n
    concat = st + st
    for i in range ( n ) :
        arr [ i ] = concat [ i : i + n ]
    arr.sort ( )
    return arr [ 0 ]
```

Figure 3: **Three types of Executability Representations.** The first example illustrates an error related to functional semantic, where the baseline model is not informed of the function's role in obtaining the minimum number, substituting a similar modulus symbol for the division operator. The second example highlights an error in syntactic structure, where the baseline model uses forced type conversion to convert characters to numbers, which raises a ValueError exception when non-numeric characters are included in the input. The third example presents an error regarding variable dependency, where the baseline model has not learned the transmission of variables, thereby neglecting to create a concatenation of string with itself. These errors result in a minimal edit distance but have a significant impact on the execution. Learning the executability representation can indicate the execution status, aiding in resolving these issues.

relies on code analysis tools designed by programming language experts and cannot be derived directly from the source code. The automatic translation between different programming languages necessitates that the translated code executes correctly. However, current LLMs are pre-trained only on source code, neglecting executability representations closely associated with execution states, leading to inadequate cross-programming language understanding and unreliable automated code translation. To address this problem, ExeCoder first encodes the executability representations of the source code, and then employs instruction fine-tuning to compel LLMs to learn these executability representations, thereby enhancing their cross-programming language understanding and achieving more reliable automated code translation. Figure 2 illustrates the pipeline of ExeCoder.

3.2 Executability Representation for Code Translation

In order to achieve more reliable code translation, the ExeCoder first customizes an executability knowledge representation strategy specifically for LLMs, encoding three types of executability representations, including functional semantic, syntactic structure, and variable dependency. Then, the ExeCoder constructs a specialized instruction fine-tuning dataset, XLCoST-Instruct, based on

the cross-programming language alignment dataset XLCoST (Zhu et al., 2022a). XLCoST-Instruct compels LLMs to learn the executability representations of the source code, enhancing the cross-language understanding of LLMs.

Functional-semantic Representation. In different programming languages, code with the same functionality may exhibit significant differences in form. Therefore, learning solely from the semantics of the source code may lead to misunderstandings of the code's functionality, which may result in erroneous code translation and execution failures, as shown in Figure 3. To address this, the ExeCoder encodes the functional-semantic representation of the source code, in order to align the functionalities of the source and target code. Functional semantic refers to the effect of the source code once it has been executed, and aligning the code functionalities of the source and target languages guarantees that their execution results remain consistent.

ExeCoder encodes the functional semantics of source code using a natural language description of its functionality, as shown in Figure 11. LLMs are pre-trained on a large amount of natural language data, which enables them to have a good understanding of natural language, facilitating their learning of functional semantics (Zhao et al., 2023). In fact, natural language descriptions that articulate the functional semantics of code are readily avail-

able. A substantial amount of public code data resources is collected from open-source repositories (such as GitHub), where comments accompanying commits serve as high-quality functional semantics of code (Ahmad et al., 2021b).

Syntactic-structure Representation. Compared to natural languages, programming languages have clear syntactic structures and strict grammatical rules. Merely learning the context of source code while neglecting its syntactic structure may result in translated code using incorrect syntax, leading to unexpected execution results or compilation errors, as shown in Figure 3. To address this issue, ExeCoder encodes the syntactic structure information of source code using Abstract Syntax Tree (AST) to enhance LLMs' understanding of the syntax of different programming languages. An AST is a tree that represents the abstract syntactic structure of source code, where each subtree represents a continuous range of subword tokens, and each leaf node represents a single token; this structure has been shown to effectively encode the syntactic structure of source code (Gong et al., 2024). To construct the AST, ExeCoder utilizes the lightweight multilingual parser tree-sitter (Brunsfeld et al., 2025) to parse the source code.

However, AST is structured graph data, and LLMs are pre-trained only on unstructured text, resulting in limited understanding of graph structure (Tang et al., 2024), making it difficult to learn the syntactic structure of source code. To address this, inspired by (Fatemi et al., 2024), ExeCoder further encodes the AST into unstructured text that is easier for LLMs to understand, serving as a representation of the syntactic structure, as shown in Figure 12. ExeCoder simplifies ASTs by keeping only leaf node tokens. It then indexes the AST as a graph and describes nodes and edges in natural language, enhancing LLM understanding of code structure (Fatemi et al., 2024). Further details are provided in Appendix D.2.

Variable-dependency Representation. In different programming languages, the same variable often has different semantics due to variations in programmer preferences and naming conventions (Cheng et al., 2024). The diversity of variable semantics may lead to incorrect variable dependency relationships when solely learning the semantics of source code, further resulting in anomalous or unexpected execution outcomes, as shown in Figure 3. To address this issue, ExeCoder encodes the variable dependency information of the source

code using Data Flow Graph (DFG) to enhance LLMs' understanding of variable dependencies in code. A DFG is a graph that describes the dependencies and interactions between variables, where each node represents a variable and each edge indicates the source of these variables. To construct the DFG, ExeCoder utilizes the lightweight multilingual parser tree-sitter (Brunsfeld et al., 2025) to parse the source code.

Given that a DFG is structured graph data, similar to the handling of AST, ExeCoder also encodes the DFG into unstructured text that is easier for LLMs to understand, serving as a representation of variable dependencies, as shown in Figure 13. ExeCoder directly assigns a numerical index to each node in the DFG and then uses natural language to represent the node information and edge information of the graph. For the node information, ExeCoder describes the token content of each node. For the edge information, ExeCoder describes the neighboring nodes of each node.

3.3 XLCoST-Instruct

To enable LLMs to learn the executability representation and enhance their cross-language understanding capabilities, ExeCoder developed a specially designed instruction fine-tuning dataset XLCoST-Instruct for code translation. Details of XLCoST-Instruct are in Appendix E and F.

3.4 Progressive Executability Representation Learning

ExeCoder aims to utilize executability representations such as functional semantics, syntactic structures, and variable dependencies in code to enhance the capabilities of LLMs in code translation tasks. However, the acquisition of executability representations relies on code analysis tools, which, unlike source code, are domain-specific high-level knowledge that is difficult to comprehend directly, facing two significant challenges in the learning process. First, how can we learn both low-level code semantics and high-level executability representations simultaneously? Second, how can we design learning strategies to enhance the understanding of high-level executability representations? To address these issues, ExeCoder proposes a Progressive Executability Representation Learning (PERL) to leverage progressively refined executability representations of code to enhance LLMs' cross-language understanding capabilities.

Specifically, to address the first issue, PERL incorporates executability representations as auxiliary knowledge during instruction fine-tuning, as illustrated in Appendix F. PERL concatenates the source code with its corresponding executability representation and then prompts LLMs to translate code based on the respective executability representations. This strategy has been shown to effectively enable LLMs to learn high-level knowledge (Yin et al., 2023).

To address the second issue, PERL has designed a phased fine-tuning strategy. Research (Robins et al., 2003) indicates that the optimal learning of programming skills is progressive, beginning with an understanding of the high-level intentions of a program, followed by the implementation of low-level code structures. Inspired by this, PERL has designed a phased fine-tuning strategy where each fine-tuning stage independently learns different executability representations. The key idea is that the functional semantics, syntactic structures, and variable dependencies of source code are executability representations that are refined progressively. Phased progressive learning aligns with the learning theories of programming experts, facilitating representational learning. Specifically, ExeCoder divides the instruction fine-tuning process into four stages: source code, functional semantics, syntactic structures, and variable dependencies, with each stage fine-tuning using only one type of data. Once the fine-tuning process of one stage converges, the next stage of fine-tuning is initiated.

4 Experimental Evaluation

4.1 Experimental Setup

Dataset. We conducted comprehensive experiments on the widely used code translation public dataset TransCoder-test (Roziere et al., 2020) to demonstrate the effectiveness of the ExeCoder. TransCoder-test contains solutions to programming problems collected from GeeksForGeeks, with each problem implemented in C++, Java, and Python, totaling 948 parallel samples. TransCoder-test is equipped with predefined unit test templates for over half of the translation pairs to evaluate whether the generated functions return the same output as the reference functions given the same input. Each unit test consists of ten test cases.

However, these unit test templates establish fixed parameter passing methods or return types that can only evaluate specific implementations. When the generated functions do not conform to the predefined implementations, the unit tests cannot execute as expected, even if the generated functions are functionally equivalent to the reference functions, resulting in unreliable evaluation. To address this, we enhanced TransCoder-test to ensure that the unit test results are capable of assessing the code translation ability. The enhanced test set is referred to as TransCoder-test-X, with detailed processing cases provided in Appendix G.

Match-based Metrics. The matching-based metrics aim to evaluate the quality of generated code through static code analysis. Following existing work (Huang et al., 2023), we employed three metrics: Exact Match (EM), BLEU (Papineni et al., 2002), and CodeBLEU (Ren et al., 2020) for evaluation. These metrics can measure the n-gram overlap, syntactic structure, and semantic equivalence of the generated code.

Execution-based Metrics. The execution-based metrics aim to assess the quality of generated code through the execution status of the code. Following existing work (Roziere et al., 2020), we utilize test computational accuracy (TCA) as a metric, which measures the proportion of code that passes unit tests. Furthermore, to further reflect the fine-grained execution status of the code, we have designed two additional metrics: Compilation Accuracy (CA) and Case Computational Accuracy (CCA), the former measuring the proportion of code that successfully compiles, and the latter measuring the proportion of code that passes case tests.

Baselines. To assess the effectiveness of ExeCoder, we compare it with state-of-the-art LLMs. Specifically, we use three advanced closed-source models as baselines: OpenAI's GPT-3.5, GPT-4, and GPT-4-o (Achiam et al., 2023). Additionally, we select five leading open-source code LLMs as baselines: Deepseek-Coder-6.7b-instruct (our base model) (Guo et al., 2024), CodeLLama-7B (Roziere et al., 2023), Magicoder-S-DS-6.7B (Wei et al., 2024), Qwen2.5-Coder-7B (Hui et al., 2024), and Wavecoder-ultra-6.7b (Yu et al., 2024).

4.2 Experimental Results

Effectiveness of ExeCoder. To assess the effectiveness of ExeCoder in code translation task, we evaluated ExeCoder on the TransCoder-test-X, which includes translations among the three most commonly used programming languages: C++, Python, and Java. Tables 1 and 9 present the comparison

Compilation Accuracy (CA)									
Model	Deepseek-Coder	CodeLLama	Magicoder	Qwen2.5-Coder	WaveCoder	GPT-3.5	GPT-4	GPT-4o	Ours
From C++	88.143	58.836	80.481	66.106	72.586	85.330	**92.698**	91.458	91.559
To C++	91.328	37.901	87.259	54.711	92.077	86.724	94.218	90.792	**97.002**
From Python	84.548	29.810	82.407	46.344	86.235	86.342	87.702	88.018	**90.995**
To Python	70.582	81.466	54.957	53.987	49.138	84.591	89.763	91.272	**93.534**
From Java	72.760	70.930	63.182	48.956	68.019	84.433	91.498	89.150	**95.482**
To Java	83.542	40.208	83.854	52.708	85.625	84.792	**87.917**	86.563	87.500
Average	81.817	53.192	75.357	53.802	75.613	85.369	90.633	89.542	**92.679**
Case Computational Accuracy (CCA)									
Model	Deepseek-Coder	CodeLLama	Magicoder	Qwen2.5-Coder	WaveCoder	GPT-3.5	GPT-4	GPT-4o	Ours
From C++	83.119	53.488	76.133	63.335	68.346	80.633	**88.141**	86.912	87.221
To C++	85.535	35.557	81.488	52.045	86.456	81.724	90.064	86.403	**91.970**
From Python	79.340	28.059	76.641	44.251	80.348	81.509	83.540	83.340	**85.891**
To Python	64.250	71.510	49.932	50.586	43.847	77.844	82.723	84.391	**86.590**
From Java	67.721	64.135	58.896	46.513	63.849	79.051	86.513	83.937	**89.948**
To Java	80.396	38.615	80.250	51.469	82.240	81.625	**85.406**	83.396	84.500
Average	76.727	48.560	70.557	51.367	70.848	80.397	86.064	84.730	**87.687**
Test Computational Accuracy (TCA)									
Model	Deepseek-Coder	CodeLLama	Magicoder	Qwen2.5-Coder	WaveCoder	GPT-3.5	GPT-4	GPT-4o	Ours
From C++	78.434	50.467	71.861	60.873	64.494	77.126	**84.145**	82.813	83.226
To C++	80.086	32.548	76.017	49.465	80.835	76.981	84.904	80.942	**86.617**
From Python	74.580	26.326	71.495	41.986	74.886	76.695	78.275	77.537	**80.717**
To Python	59.267	65.948	46.013	47.414	40.086	73.276	77.478	79.203	**80.927**
From Java	63.631	59.100	55.237	44.228	60.500	75.081	82.254	79.691	**85.163**
To Java	77.292	37.396	76.563	50.208	78.958	78.646	**82.292**	79.896	81.563
Average	72.215	45.297	66.198	49.029	66.627	76.301	81.558	80.014	**83.035**

Table 1: **Execution-based evaluation** of ExeCoder and baseline models on TransCoder-test-X.

results with baseline models.

Generally speaking, ExeCoder achieved state-of-the-art performance in code translation task. Compared to the state-of-the-art closed-source model GPT-4, ExeCoder outperformed it by 2.05%, 1.62%, and 1.48% in execution-based metrics, and by 3.78%, 58.72%, and 29.24% in match-based metrics. This evidence demonstrates ExeCoder's strong effectiveness in code translation task. Compared to open-source models, ExeCoder achieved even more significant advantages, outperforming by up to 39.49%, 39.13%, and 37.74% in execution-based metrics, and by 4.38%, 68.55%, and 61.78% in match-based metrics. This significant leap indicates that ExeCoder has reached state-of-the-art performance in code translation task. Furthermore, compared to the foundation model Deepseek-Coder-6.7b-instruct, ExeCoder also attained significant advantages, outperforming by 10.86%, 10.96%, and 10.82% in execution-based metrics, and by 3.78%, 58.72%, and 29.24% in match-based metrics. These advantages highlight that the executability representation facilitates enhanced cross-programming language understanding of LLMs, achieving precise and reliable automated code translation.

Effectiveness of Executability Representation. We conduct an ablation study to evaluate the effectiveness of the executability representation. Specifically, we examine the following variants of ExeCoder, where the fine-tuning process for each variant maintains the same step.
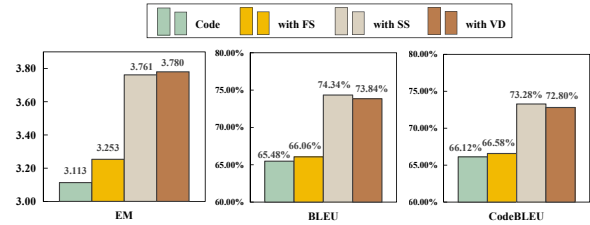

Figure 4: Match-based evaluation of different variants of ExeCoder on TransCoder-test-X.

- **Code**: Fine-tuning using only the source code.

- **with FS**: Fine-tuning using only the source code and functional semantic representation.

- **with SS**: Fine-tuning using only the source code and syntactic structure representation.

- **with VD**: Fine-tuning using only the source code and variable dependency representation.

Tables 2, 3, 4 and Figure 4 present our results, which indicate that the learning from each executability representation contributes to varying degrees of improvement. For execution-based metrics, the inclusion of the three different types of executability representations resulted in an average increase of 0.41%, 1.64%, and 1.21%. For match-based metrics, the integration of the three different types of executability representations led to an average increase of 0.39%, 5.56%, and 5.24%. These advantages indicate that executability representations can provide execution status beyond code semantics, enhancing the cross-language understanding capability of LLMs.

Model	Code	with FS	with SS	with VD
From C++	91.13	91.56 (+0.43)	91.66 (+0.53)	91.14 (+0.01)
To C++	96.36	96.47 (+0.11)	96.90 (+0.54)	97.11 (+0.75)
From Python	89.21	89.52 (+0.31)	91.20 (+1.99)	90.78 (+1.57)
To Python	90.41	91.59 (+1.18)	93.00 (+2.59)	92.89 (+2.48)
From Java	92.68	93.54 (+0.86)	95.05 (+2.37)	95.27 (+2.59)
To Java	86.25	86.56 (+0.31)	88.02 (+1.77)	87.19 (+0.94)
Average	91.01	91.54 (+0.53)	92.64 (+1.63)	92.39 (+1.38)

Table 2: **Compilation Accuracy (CA)** of different variants of ExeCoder.

Model	Code	with FS	with SS	with VD
From C++	86.04	86.48 (+0.44)	87.33 (+1.29)	86.56 (+0.52)
To C++	91.21	90.89 (-0.32)	91.68 (+0.47)	91.78 (+0.57)
From Python	84.27	84.28 (+0.01)	85.37 (+1.10)	84.79 (+0.52)
To Python	83.11	83.96 (+0.85)	85.83 (+2.72)	85.59 (+2.48)
From Java	87.13	87.43 (+0.30)	89.38 (+2.25)	89.38 (+2.25)
To Java	83.11	83.33 (+0.22)	84.57 (+1.46)	83.36 (+0.25)
Average	85.81	86.06 (+0.25)	87.36 (+1.55)	86.91 (+1.10)

Table 3: **Case Computational Accuracy (CCA)** of different variants of ExeCoder.

Model	Code	with FS	with SS	with VD
From C++	81.62	82.17 (+0.55)	82.90 (+1.28)	82.16 (+0.54)
To C++	85.22	85.12 (-0.10)	86.19 (+0.97)	86.08 (+0.86)
From Python	78.28	78.60 (+0.32)	80.08 (+1.80)	78.93 (+0.65)
To Python	76.40	77.59 (+1.19)	79.31 (+2.91)	79.20 (+2.80)
From Java	81.82	82.25 (+0.43)	83.98 (+2.16)	84.09 (+2.27)
To Java	80.10	80.31 (+0.21)	81.46 (+1.36)	79.90 (-0.20)
Average	80.58	81.01 (+0.43)	82.32 (+1.74)	81.73 (+1.15)

Table 4: **Test Computational Accuracy (TCA)** of different variants of ExeCoder.

Model	Code	Code-FS	Code-FS-SS	Code-FS-SS-VD
From C++	91.55	91.13 (-0.42)	91.15 (+0.02)	91.56 (+0.41)
To C++	96.36	96.36 (+0.00)	97.11 (+0.75)	97.00 (-0.11)
From Python	89.73	90.78 (+1.05)	90.58 (-0.20)	91.00 (+0.42)
To Python	93.00	92.56 (-0.44)	93.53 (+0.97)	93.53 (+0.00)
From Java	95.16	94.62 (-0.54)	95.37 (+0.75)	95.48 (+0.11)
To Java	87.08	87.60 (+0.52)	86.46 (-1.14)	87.50 (+1.04)
Average	92.15	92.18 (+0.03)	92.37 (+0.19)	92.68 (+0.31)

Table 5: **Compilation Accuracy (CA)** of various stages within PERL.

Model	Code	Code-FS	Code-FS-SS	Code-FS-SS-VD
From C++	86.52	86.31 (-0.21)	86.63 (+0.32)	87.22 (+0.59)
To C++	90.97	91.08 (+0.11)	92.09 (+1.01)	91.97 (-0.12)
From Python	84.65	85.77 (+1.12)	85.38 (-0.38)	85.89 (+0.51)
To Python	85.62	85.24 (-0.38)	86.11 (+0.87)	86.59 (+0.48)
From Java	89.33	88.42 (-0.91)	89.71 (+1.29)	89.95 (+0.24)
To Java	83.91	84.48 (+0.57)	83.53 (-0.95)	84.50 (+0.97)
Average	86.83	86.88 (+0.05)	87.24 (+0.36)	87.69 (+0.45)

Table 6: **Case Computational Accuracy (CCA)** of various stages within PERL.

Model	Code	Code-FS	Code-FS-SS	Code-FS-SS-VD
From C++	82.37	82.37 (+0.00)	82.38 (+0.01)	83.23 (+0.84)
To C++	85.01	85.87 (+0.86)	87.05 (+1.18)	86.62 (-0.43)
From Python	78.80	80.50 (+1.70)	80.31 (-0.19)	80.72 (+0.41)
To Python	79.31	79.42 (+0.11)	80.17 (+0.75)	80.93 (+0.76)
From Java	84.09	83.87 (-0.21)	84.95 (+1.07)	85.16 (+0.22)
To Java	80.94	81.46 (+0.52)	80.42 (-1.04)	81.56 (+1.15)
Average	81.75	82.25 (+0.50)	82.55 (+0.30)	83.03 (+0.49)

Table 7: **Test Computational Accuracy (TCA)** of various stages within PERL.

5 Conclusion

In this paper, we propose ExeCoder, a large language model specifically designed for code translation. The key idea of ExeCoder is to enhance the capabilities of LLMs in code translation by leveraging executability representations such as functional semantics, syntactic structure, and variable dependencies in code. Additionally, we have enhanced the widely used code translation benchmark TransCoder-test, resulting in a new benchmark called TransCoder-test-X, which is capable of evaluating the code translation abilities of LLMs. Evaluation on TransCoder-test-X indicates that ExeCoder achieves state-of-the-art performance in code translation, surpassing existing open-source large code models by more than 10.88% to 38.78% and more than 27.44% to 42.97% on two different metrics. Notably, the ExeCoder outperforms renowned closed-source LLMs, including GPT-3.5, GPT-4, and GPT-4o, highlighting the significant role of executability representations in code translation. In the future, we plan to incorporate richer executability representations for LLMs using code analysis tools.

Effectiveness of Progressive Executability Representation Learning. We conduct an ablation study to evaluate the effectiveness of the PERL strategy. Specifically, we assess the impact of each fine-tuning phase in PERL on the model's code translation capability. Tables 5, 6, and 7 present our results. We also conducted a more detailed ablation study on PERL, with details provided in Appendix A due to space limitations.

Overall, ExeCoder's meticulously designed PERL strategy significantly enhances the code translation performance through the learning of executability representations. Compared to models fine-tuned solely on source code, the inclusion of executability representations raises the upper limits of model capability, with three execution-based metrics improving by 0.53%, 0.85%, and 1.28%, respectively. Furthermore, the learning of specific executability representations at each stage improves the model's cross-language understanding. Relative to the previous stage, each fine-tuning stage further improved the key execution-based metric TCA by 0.50%, 0.30%, and 0.49%, respectively.

Limitations

A primary limitation of this work is identified. ExeCoder employs static analysis tools to derive executability representations from source code, and these representations often possess extensive contextual information. Although the number of tokens during the fine-tuning process stays within the model's context window, it nonetheless elevates the relevant fine-tuning expenditures, consequently diminishing ExeCoder's scalability to a certain extent. Addressing the reduction of these costs falls outside the purview of the current study and is not the central topic of this paper.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021a. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*.

WU Ahmad, S Chakraborty, B Ray, and KW Chang. 2021b. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Patrick Thomson, ObserverOfTime, Josh Vera, dundargoc, Phil Turnbull, Will Lillis, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Ika, Amin Ya, Tuấn-Anh Nguyen, and 10 others. 2025. tree-sitter/tree-sitter: target/tree-sitter-linux- arm.gz.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782*.

Matthew T Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. 2024. Lassi: An llm-based automated self-correcting pipeline for translating parallel scientific codes. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, pages 136–143. IEEE.

Hasan Ferit Eniser, Valentin Wüstholz, and Maria Christakis. 2024. Automatically testing functional properties of code translation models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 21055–21062.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2024. Talk like a graph: Encoding graphs for large language models. In *The Twelfth International Conference on Learning Representations*.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. Ast-t5: Structure-aware pretraining for code generation and understanding. In *Forty-first International Conference on Machine Learning*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Yufan Huang, Mengnan Qi, Yongqiang Yao, Maoquan Wang, Bin Gu, Colin Clement, and Neel Sundaresan. 2023. Program translation via code distillation. *arXiv preprint arXiv:2310.11476*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE.

Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805.

Kaixin Li, Qisheng Hu, James Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. 2024. Instructcoder: Instruction tuning large language models for code editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 50–70.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and domain aware model for unsupervised program translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 755–767. IEEE.

Jiaqi Liu, Fengming Zhang, Xin Zhang, Zhiwen Yu, Liang Wang, Yao Zhang, and Bin Guo. 2024. hm-codetrans: Human-machine interactive code translation. *IEEE Transactions on Software Engineering*.

Junting Lu, Zhiyang Zhang, Fangkai Yang, Jue Zhang, Lu Wang, Chao Du, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025. Axis: Efficient human-agent-computer interaction with api-first llm-based agents. *Preprint*, arXiv:2409.17140.

Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. 2024. Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2448–2449.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Marcos Macedo, Yuan Tian, Filipe Cogo, and Bram Adams. 2024. Exploring the impact of the output format on the evaluation of large language models for code translation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 57–68.

Lingrui Mei, Shenghua Liu, Yiwei Wang, Baolong Bi, Yuyao Ge, Jun Wan, Yurong Wu, and Xueqi Cheng. 2025. a1: Steep test-time scaling law via environment augmented generation. *Preprint*, arXiv:2504.14597.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611.

Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. In *International Conference on Learning Representations*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, and 1 others. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

Marc Szafraniec, Baptiste Roziere, Hugh James Leather, Patrick Labatut, Francois Charton, and Gabriel Synnaeve. 2023. Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations*.

Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. 2024. Graphgpt: Graph instruction tuning for large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 491–500.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, and 1 others. 2022. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski,

and 1 others. 2023. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 126–173.

Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153.

Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. 2024. Transagent: An llm-based multi-agent system for code translation. *arXiv preprint arXiv:2409.19894*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, and 1 others. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022a. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.

Ming Zhu, Mohimenul Karim, Ismini Lourentzou, and Daphne Yao. 2024. Semi-supervised code translation overcoming the scarcity of parallel code data. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1545–1556.

Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022b. Multilingual code snippets training for program translation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pages 11783–11790.

A Ablation of PERL

we conducted further ablation studies on PERL to demonstrate its necessity and effectiveness. Specifically, we adopted the single-stage mixed training approach and reversed the order of the various representations in PERL, with the evaluation results presented in the table 8. The results show that both training with mixed representations and altering the order of the various representations had varying degrees of impact on the performance of ExeCoder. The training method PERL, which aligns with programming learning theory, achieved optimal performance.

Compilation Accuracy (CA)							
Model	**NL-AST-DFG (ExeCoder)**	**NL-DFG-AST**	**AST-NL-DFG**	**AST-DFG-NL**	**DFG-AST-NL**	**DFG-NL-AST**	**Shuffle**
From C++	91.559	91.354	91.347	91.243	91.782	91.020	91.861
To C++	97.002	96.681	96.681	96.574	96.467	95.824	96.253
From Python	90.995	90.579	89.948	89.844	89.948	89.410	90.460
To Python	93.534	93.534	93.427	93.427	93.103	92.888	93.211
From Java	95.482	95.053	95.375	95.268	94.299	95.053	95.268
To Java	87.500	86.771	86.563	86.354	86.458	86.771	88.125
Average	92.679	92.329	92.223	92.118	92.010	91.828	92.530
Case Computational Accuracy (CCA)							
Model	**NL-AST-DFG (ExeCoder)**	**NL-DFG-AST**	**AST-NL-DFG**	**AST-DFG-NL**	**DFG-AST-NL**	**DFG-NL-AST**	**Shuffle**
From C++	87.221	87.145	86.892	86.548	86.880	86.247	87.616
To C++	91.970	91.842	91.713	91.842	91.028	90.664	90.867
From Python	85.891	85.542	84.940	84.807	84.335	84.334	84.682
To Python	86.590	86.396	86.353	85.868	85.017	85.491	85.847
From Java	89.948	89.186	89.745	89.615	88.184	89.251	89.218
To Java	84.500	83.635	83.510	83.260	83.354	83.677	84.802
Average	87.687	87.291	87.192	86.990	86.466	86.611	87.172
Test Computational Accuracy (TCA)							
Model	**NL-AST-DFG (ExeCoder)**	**NL-DFG-AST**	**AST-NL-DFG**	**AST-DFG-NL**	**DFG-AST-NL**	**DFG-NL-AST**	**Shuffle**
From C++	83.226	82.813	83.226	82.583	82.475	82.148	83.308
To C++	86.617	86.938	86.617	86.831	85.760	85.760	85.439
From Python	80.717	80.630	79.991	79.678	78.935	79.557	79.340
To Python	80.927	80.172	80.603	79.849	78.556	79.095	79.310
From Java	85.163	83.980	84.733	84.732	83.011	84.088	83.872
To Java	81.563	80.313	80.729	80.313	80.104	80.938	81.771
Average	83.035	82.474	82.650	82.331	81.473	81.931	82.173

Table 8: Additional Ablation of PERL.

B Match-based evaluation

Exact Match (EM)									
Model	Deepseek-Coder	CodeLLama	Magicoder	Qwen2.5-Coder	WaveCoder	GPT-3.5	GPT-4	GPT-4o	Ours
From C++	0.000	0.055	0.000	0.000	0.000	0.160	0.000	0.000	**6.120**
To C++	0.000	0.265	0.160	0.000	0.000	1.635	0.055	0.370	**2.325**
From Python	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.055**
To Python	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**10.445**
From Java	0.000	0.265	0.160	0.000	0.000	1.635	0.055	0.370	**6.965**
To Java	0.000	0.055	0.000	0.000	0.000	0.160	0.000	0.000	**0.370**
Average	0.000	0.107	0.053	0.000	0.000	0.598	0.018	0.123	**4.380**
BLEU									
Model	Deepseek-Coder	CodeLLama	Magicoder	Qwen2.5-Coder	WaveCoder	GPT-3.5	GPT-4	GPT-4o	Ours
From C++	9.041	16.316	9.821	4.778	8.696	9.639	7.826	8.846	**72.627**
To C++	11.486	10.305	10.110	3.789	7.820	10.534	8.857	23.432	**80.600**
From Python	7.601	8.926	7.898	4.908	7.565	7.938	7.120	7.628	**62.708**
To Python	9.471	10.443	9.783	3.313	9.417	10.244	9.474	9.989	**80.765**
From Java	12.298	11.429	10.935	1.736	8.706	11.359	9.690	24.439	**81.735**
To Java	7.983	15.923	8.760	4.319	7.730	8.159	6.305	7.493	**55.705**
Average	9.647	12.224	9.551	3.807	8.322	9.646	8.212	13.638	**72.356**
CodeBLEU									
Model	Deepseek-Coder	CodeLLama	Magicoder	Qwen2.5-Coder	WaveCoder	GPT-3.5	GPT-4	GPT-4o	Ours
From C++	37.580	36.493	9.821	27.427	36.075	42.065	41.379	41.966	**72.773**
To C++	36.647	25.350	10.110	21.112	35.228	42.364	38.929	46.186	**74.961**
From Python	34.137	23.232	7.898	22.327	32.635	37.693	38.455	36.742	**64.684**
To Python	38.058	37.066	9.783	22.059	37.976	39.044	38.263	38.726	**75.623**
From Java	39.101	33.405	10.935	17.493	37.552	42.000	40.209	47.541	**76.524**
To Java	36.113	30.714	8.760	24.077	33.058	40.349	42.851	41.337	**63.398**
Average	36.939	31.043	9.551	22.416	35.421	40.586	40.014	42.083	**71.327**

Table 9: **Match-based evaluation** of ExeCoder and baseline models on TransCoder-test-X.

C Implementation Details

We present detailed implementation details. Deepseek-Coder-6.7b-instruct serves as our basic foundation model. To fine-tune the base model, we employed specific configurations, including a batch size of 128, a sequence length of 3076, a learning rate of 2e-6, a cosine learning rate scheduler, and fp16 mixed precision.

D Construction of Executability Representation

In this section, we will present the specific details of encoding syntactic-structure representation and variable-dependency representation, with Figure 5 illustrating an example of C++ code.

Input Function

```cpp
int max(int a, int b)
{
    int x = 0;
    if (b > a)
        x = b;
    else
        x = a;
    return x;
}
```

Figure 5: Example C++ code used for encoding executability representations.

D.1 Variable-dependency Representation

For variable-dependency representation, ExeCoder first utilizes the code analysis tool tree-sitter to parse its data flow graph (DFG), as illustrated in Figure 6. Subsequently, ExeCoder directly assigns a numerical index to each node in the DFG and then uses natural language to represent the node information and edge information of the graph. For the node information, ExeCoder describes the token content of each node. For the edge information, ExeCoder describes the neighboring nodes of each node. The final variable-dependency representation is illustrated in Figure 7.
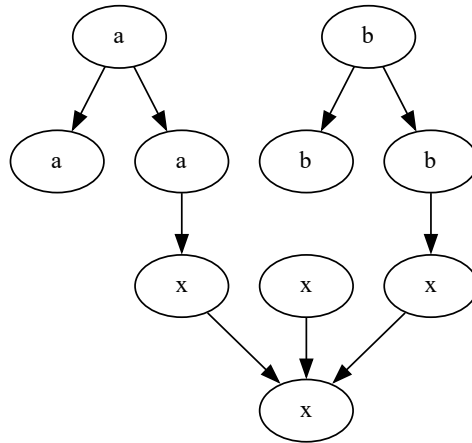
Figure 6: Data flow graph of the example code.

Variable-dependency Representation:

```
G describes a graph among nodes 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
In this graph:
Node 0 represents variable a.
Node 1 represents variable b.
Node 2 represents variable x.
Node 3 represents variable b.
Node 4 represents variable a.
Node 5 represents variable x.
Node 6 represents variable b.
Node 7 represents variable x.
Node 8 represents variable a.
Node 9 represents variable x.
In this graph:
Node 3 is connected to node 1.
Node 4 is connected to node 0.
Node 5 is connected to node 6.
Node 6 is connected to node 1.
Node 7 is connected to node 8.
Node 8 is connected to node 0.
Node 9 is connected to nodes 2, 5, 7.
```

Figure 7: Variable-dependency representation of the example code.

D.2 Syntactic-structure representation

For syntactic-structure representation, ExeCoder first utilizes the code analysis tool tree-sitter to parse its abstract syntax tree (AST) and removes the non-leaf node content, retaining only the tokens at the leaf nodes, as illustrated in Figure 8. Then, similar to the operations on the DFG, ExeCoder assigns a numerical index to each node of the AST and represents the node and edge information of the graph using natural language. The final syntactic-structure representation is illustrated in Figure 9.
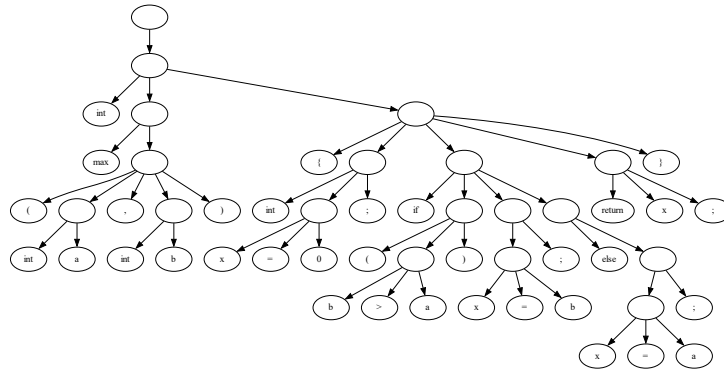
Figure 8: Abstract syntax tree of the example code.

Syntactic-structure Representation:

```
G describes a graph among nodes 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, and 51.
In this graph:
Node 2 represents code int. Node 5 represents code max.
Node 7 represents code {. Node 11 represents code }.
Node 12 represents code (. Node 14 represents code ,.
Node 16 represents code ). Node 17 represents code int.
Node 19 represents code ;. Node 20 represents code if.
Node 24 represents code return. Node 25 represents code x.
Node 26 represents code ;. Node 27 represents code int.
Node 28 represents code a. Node 29 represents code int.
Node 30 represents code b. Node 31 represents code x.
Node 32 represents code =. Node 33 represents code 0.
Node 34 represents code (. Node 36 represents code ).
Node 38 represents code ;. Node 39 represents code else.
Node 41 represents code b. Node 42 represents code >.
Node 43 represents code a. Node 44 represents code x.
Node 45 represents code =. Node 46 represents code b.
Node 48 represents code ;. Node 49 represents code x.
Node 50 represents code =. Node 51 represents code a.
In this graph:
Node 0 is connected to node 1. Node 1 is connected to nodes 2, 3, 4.
Node 3 is connected to nodes 4, 5. Node 4 is connected to nodes 5, 6, 7, 8, 9.
Node 6 is connected to nodes 7, 8, 9, 10, 11. Node 8 is connected to nodes 9, 10, 11.
Node 9 is connected to nodes 10, 11, 12, 13. Node 10 is connected to nodes 11, 12, 13.
Node 13 is connected to nodes 14, 15. Node 15 is connected to nodes 16, 17.
Node 18 is connected to nodes 19, 20, 21. Node 21 is connected to nodes 22, 23, 24.
Node 22 is connected to nodes 23, 24. Node 23 is connected to nodes 24, 25.
Node 35 is connected to nodes 36, 37, 38. Node 37 is connected to nodes 38, 39, 40.
Node 40 is connected to nodes 41, 42. Node 47 is connected to nodes 48, 49, 50.
```

Figure 9: Syntactic-structure representation of the example code.

E   Construction of XLCoST-Instruct

In this section, we will present the construction details of the dataset XLCoST-Instruct, designed for fine-tuning LLMs. XLCoST-Instruct, constructed from the public cross-language code intelligence benchmark dataset XLCoST, aims to enhance the cross-program language understanding capabilities of LLMs by enabling them to learn executability representations of source code through instruction fine-tuning.

ExeCoder selected parallel data from three programming languages: C++, Python, and Java, to construct a fine-tuning dataset, as these are the most widely used programming languages, and their translation tasks can effectively assess the code translation capabilities of LLMs. Here, the source code refers to complete programs rather than mere code snippets. Subsequently, we encoded three types of executability representations from the source code of XLCoST. For functional semantic representation, we encoded the natural language from the natural language-code pairs provided by XLCoST. For syntactic structure representation and variable dependency representation, we followed the methods outlined in Appendix D to encode from the source code. Based on this, a single data instance resulted in four different code representations. Since XLCoST is not specifically designed for instruction fine-tuning of LLMs, ExeCoder manually constructs instructions tailored for translation tasks, with more detailed information provided in Appendix F.

Finally, we conducted similarity checks and deduplication on the resulting instruction fine-tuning dataset. We employed MinHash with a locality-sensitive hashing (LSH) index to eliminate code instances with a Jaccard similarity greater than 0.85, resulting in approximately 55K instances for the instruction fine-tuning dataset.

F   Example in XLCoST-Instruct

In this section, we will present the data instances in the XLCoST-Instruct dataset constructed for fine-tuning LLMs. Figures 10, 11, 12, and 13 illustrate the learning of four different representations of the same source code instance in XLCoST-Instruct, namely code semantics, functional-semantic, syntactic-structure, and variable-dependency.

For the three executability representations, functional-semantic, syntactic-structure, and variable-dependency, ExeCoder utilizes them as auxiliary knowledge during the instruction fine-tuning process. Specifically, ExeCoder concatenates the source code with the corresponding executability representations and then prompts the LLMs to perform code translation tasks based on the respective executability representations, as shown in Figures 11, 12, and 13.

Prompt: Translate the given code from python to cpp. The input Code is marked with <Code>and </Code>. Please note that the code entered is a complete program with main fuction.

Input:

```
<Code>
def Kmultiples ( n , k ) :
    a = n
    for i in range ( 1 , k + 1 ) :
        print ( "{}*{}={}".format ( n , i , a ) )
        j = 0
        while ( n >= ( 1 << j ) ) :
            a += n & ( 1 << j )
            j += 1
</Code>
```

Output:

```cpp
#include <bits/stdc++.h>
using namespace std ;
void Kmultiples ( int n , int k ) {
  int a = n ;
  for ( int i = 1 ;
  i <= k ;
  i ++ ) {
    cout << n << " * " << i << " = " << a << endl ;
    int j = 0 ;
    while ( n >= ( 1 << j ) ) {
      a += n & ( 1 << j ) ;
      j ++ ;
    }
  }
}
```

Figure 10: Example for learning code semantic in XLCoST-Instruct.

Prompt: Translate the given code from python to cpp. The input contains the source code and a description of the code. The input Code is marked with <Code>and </Code>. Please note that the code entered is a complete program with main fuction. The description of the code is marked with <NL>and </NL>.

Input:

```
<Code>
def Kmultiples ( n , k ) :
    a = n
    for i in range ( 1 , k + 1 ) :
        print ( "{}*{}={}".format ( n , i , a ) )
        j = 0
        while ( n >= ( 1 << j ) ) :
            a += n & ( 1 << j )
            j += 1
</Code>

<NL>
Generate first K multiples of N ;
Using Bitwise operators | Function to print the first K multiples of N ;
Print the value of N * i ;
Iterate each bit of N and add pow ( 2 , pos ) , where pos is the index of each set bit ;
Check if current bit at pos j is fixed or not ;
</NL>
```

Output:

```cpp
#include <bits/stdc++.h>
using namespace std ;
void Kmultiples ( int n , int k ) {
  int a = n ;
  for ( int i = 1 ;
  i <= k ;
  i ++ ) {
    cout << n << " * " << i << " = " << a << endl ;
    int j = 0 ;
    while ( n >= ( 1 << j ) ) {
      a += n & ( 1 << j ) ;
      j ++ ;
    }
  }
}
```

Figure 11: Example for learning functional semantic in XLCoST-Instruct.

Prompt: Translate the given code from python to cpp. The input contains the source code and a Abstract Syntax Tree of the code. The input Code is marked with <Code>and </Code>. Please note that the code entered is a complete program with main fuction. The Abstract Syntax Tree of the code is marked with <AST>and </AST>.

Input:

```
<Code>
def Kmultiples ( n , k ) :
    a = n
    for i in range ( 1 , k + 1 ) :
        print ( "{}*{}={}".format ( n , i , a ) )
        j = 0
        while ( n >= ( 1 << j ) ) :
            a += n & ( 1 << j )
            j += 1
</Code>
<AST>
G describes a graph among nodes 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106,
107, 108, 109, 110, 111, 112, 113, 114, and 115.
In this graph:
Node 5 represents code def. Node 6 represents code Kmultiples. Node 8 represents code :.
Node 13 represents code (. Node 14 represents code n. Node 15 represents code ,.
Node 16 represents code k. Node 17 represents code ). Node 20 represents code N.
Node 21 represents code =. Node 22 represents code 16. Node 23 represents code K.
Node 24 represents code =. Node 25 represents code 7. Node 26 represents code Kmultiples.
Node 29 represents code for. Node 30 represents code i. Node 31 represents code in.
Node 33 represents code :. Node 35 represents code (. Node 36 represents code N.
Node 37 represents code ,. Node 38 represents code K. Node 39 represents code ).
Node 40 represents code a. Node 41 represents code =. Node 42 represents code n.
Node 43 represents code range. Node 48 represents code (. Node 49 represents code 1.
Node 50 represents code ,. Node 52 represents code ). Node 55 represents code while.
Node 57 represents code :. Node 59 represents code k. Node 60 represents code +.
Node 61 represents code 1. Node 62 represents code print. Node 64 represents code j.
Node 65 represents code =. Node 66 represents code 0. Node 67 represents code (.
Node 69 represents code ). Node 72 represents code (. Node 74 represents code ).
Node 75 represents code n. Node 76 represents code >=. Node 82 represents code (.
Node 84 represents code ). Node 85 represents code a. Node 86 represents code +=.
Node 88 represents code j. Node 89 represents code +=. Node 90 represents code 1.
Node 92 represents code .. Node 93 represents code format. Node 94 represents code (.
Node 95 represents code n. Node 96 represents code ,. Node 97 represents code i.
Node 98 represents code ,. Node 99 represents code a. Node 100 represents code ).
Node 101 represents code 1. Node 102 represents code <<. Node 103 represents code j.
Node 104 represents code n. Node 105 represents code &. Node 107 represents code ".
Node␣108␣represents␣code␣{}*{}={}.␣Node␣109␣represents␣code␣". Node 110 represents code (.
Node 112 represents code ). Node 113 represents code 1. Node 114 represents code <<.
Node 115 represents code j.
In this graph:
Node 0 is connected to nodes 1, 2, 3, 4. Node 1 is connected to nodes 2, 3, 4, 5, 6.
Node 2 is connected to node 3. Node 3 is connected to node 4.
Node 4 is connected to node 5. Node 7 is connected to nodes 8, 9, 10, 11, 12.
Node 9 is connected to nodes 10, 11. Node 10 is connected to nodes 11, 12, 13.
Node 11 is connected to nodes 12, 13, 14. Node 12 is connected to nodes 13, 14.
Node 18 is connected to node 19. Node 19 is connected to nodes 20, 21, 22, 23, 24, 25.
Node 27 is connected to nodes 28, 29, 30, 31, 32. Node 28 is connected to nodes 29, 30, 31.
Node 32 is connected to nodes 33, 34. Node 34 is connected to nodes 35, 36, 37.
Node 44 is connected to nodes 45, 46, 47, 48, 49. Node 45 is connected to node 46.
Node 46 is connected to node 47. Node 47 is connected to nodes 48, 49, 50, 51.
Node 51 is connected to nodes 52, 53, 54. Node 53 is connected to nodes 54, 55.
Node 54 is connected to nodes 55, 56, 57. Node 56 is connected to nodes 57, 58, 59.
Node 58 is connected to nodes 59, 60. Node 63 is connected to nodes 64, 65, 66.
Node 68 is connected to nodes 69, 70, 71. Node 70 is connected to node 71.
Node 71 is connected to node 72. Node 73 is connected to nodes 74, 75.
Node 77 is connected to nodes 78, 79, 80. Node 78 is connected to nodes 79, 80, 81.
Node 79 is connected to nodes 80, 81, 82. Node 80 is connected to nodes 81, 82, 83.
Node 81 is connected to nodes 82, 83, 84, 85, 86, 87, 88. Node 83 is connected to nodes 84, 85, 86.
Node 87 is connected to nodes 88, 89, 90. Node 91 is connected to nodes 92, 93, 94.
Node 106 is connected to nodes 107, 108, 109. Node 111 is connected to nodes 112, 113, 114.
</AST>
```

Output:

```cpp
#include <bits/stdc++.h>
using namespace std ;
void Kmultiples ( int n , int k ) {
  int a = n ;
  for ( int i = 1 ; i <= k ; i ++ ) {
    cout << n << "␣*␣" << i << "␣=␣" << a << endl ;
    int j = 0 ;
    while ( n >= ( 1 << j ) ) {
      a += n & ( 1 << j ) ;
      j ++ ;}}}
```

Figure 12: Example for learning syntactic structure in XLCoST-Instruct.

Prompt: Translate the given code from python to cpp. The input contains the source code and a Dataflow Graph of the code. The input Code is marked with <Code>and </Code>. Please note that the code entered is a complete program with main fuction. The Dataflow Graph of the code is marked with <DFG>and </DFG>.

Input:

```
<Code>
def Kmultiples ( n , k ) :
    a = n
    for i in range ( 1 , k + 1 ) :
        print ( "{}*{}={}".format ( n , i , a ) )
        j = 0
        while ( n >= ( 1 << j ) ) :
            a += n & ( 1 << j )
            j += 1
</Code>

<DFG>
G describes a graph among nodes 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 29, 30, and 31.
In this graph:
Node 0 represents variable Kmultiples. Node 1 represents variable n.
Node 2 represents variable k. Node 3 represents variable a.
Node 4 represents variable n. Node 5 represents variable i.
Node 6 represents variable range. Node 7 represents variable 1.
Node 8 represents variable k. Node 9 represents variable 1.
Node 10 represents variable print. Node 11 represents variable format.
Node 12 represents variable n. Node 13 represents variable i.
Node 14 represents variable a. Node 15 represents variable j.
Node 16 represents variable 0. Node 17 represents variable n.
Node 18 represents variable j. Node 19 represents variable a.
Node 20 represents variable n. Node 21 represents variable 1.
Node 22 represents variable j. Node 23 represents variable j.
Node 24 represents variable 1. Node 25 represents variable N.
Node 26 represents variable 16. Node 27 represents variable K.
Node 28 represents variable 7. Node 29 represents variable Kmultiples.
Node 30 represents variable N. Node 31 represents variable K.
In this graph:
Node 3 is connected to node 4. Node 4 is connected to node 1.
Node 5 is connected to nodes 6, 7, 8, 9. Node 6 is connected to node 6.
Node 8 is connected to node 2. Node 10 is connected to node 10.
Node 11 is connected to node 11. Node 12 is connected to node 1.
Node 13 is connected to node 5. Node 14 is connected to nodes 3, 19.
Node 15 is connected to node 16. Node 17 is connected to node 1.
Node 18 is connected to nodes 15, 23. Node 19 is connected to nodes 20, 21, 22.
Node 20 is connected to node 1. Node 22 is connected to nodes 15, 23.
Node 23 is connected to node 24. Node 25 is connected to node 26.
Node 27 is connected to node 28. Node 29 is connected to node 0.
Node 30 is connected to node 25. Node 31 is connected to node 27.
</DFG>
```

Output:

```cpp
#include <bits/stdc++.h>
using namespace std ;
void Kmultiples ( int n , int k ) {
  int a = n ;
  for ( int i = 1 ;
  i <= k ;
  i ++ ) {
    cout << n << " * " << i << " = " << a << endl ;
    int j = 0 ;
    while ( n >= ( 1 << j ) ) {
      a += n & ( 1 << j ) ;
      j ++ ;
    }
  }
}
```

Figure 13: Example for learning variable dependency in XLCoST-Instruct.

G Construction of TransCoder-test-X

In this section, we will present the detailed process of enhancing TransCoder-test to obtain TransCoder-test-X.

We first manually added various parameter passing methods under equivalent implementations to the unit test templates, as illustrated in Figures 14 and 15. Specifically, Figure 14 depicts the unit test template in TransCoder-test, which sets specific parameter passing methods for array-type variables in the main function, passing parameters from param0 to the generated function for unit testing. However, this parameter passing method only allows parameters to be passed for array-type variables; when functions are implemented using vector containers, even if the generated function is functionally equivalent to the reference function, the predefined unit test parameters cannot be passed, and the unit tests cannot be executed as expected. To address this, our solution was to manually add various parameter passing methods for equivalent implementations to the unit test templates. Specifically, we manually modified the parameter passing methods in the main function and added template wrapper functions, as shown in Figure 15. The template wrapper function determines the method of parameter passing by assessing the implementation of the function.

We also manually aligned the return types of the parallel data. During the evaluation of TransCoder-test, we found that some return types of the parallel data were misaligned, as shown in Figure 16. Figure 16 illustrates a translation from C++ to Java, where the return type of the source function is int and the return type of the reference function is boolean. The correct translation of the source function should yield a return type of int, rather than the return type of boolean from the reference function. However, in Java, int and boolean types cannot be directly equated, which means that even if the translation is correct, the unit tests cannot execute as intended. To address this issue, we manually aligned the return types of the parallel data.

Furthermore, we corrected inherent errors in the unit test templates present in TransCoder-test, as shown in Figure 17. In Figure 17, the original code in TransCoder-test contains an extraneous comma in the parameters of the main function, preventing it from compiling correctly, which we manually corrected.

Finally, the processed test set is referred to as TransCoder-test-X, where the unit tests are capable of assessing the equivalence of code with different implementation methods and evaluating LLMs' code translation abilities.

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <vector>
#include <fstream>
#include <iomanip>
#include <bits/stdc++.h>
using namespace std;
int f_gold ( int arr [ ], int n, int x ) {
  int i;
  for ( i = 0;
  i < n;
  i ++ ) {
    if ( arr [ i ] == x ) return i;
  }
  return - 1;
}


//TOFILL

int main() {
    int n_success = 0;
    vector<vector<int>> param0
    {{4,5,5,11,13,14,15,19,22,22,23,26,29,29,36,44,48,49,65,65,
    67,68,70,76,79,79,81,85,88,91,91,92,92,97},

    {-24,-78,-32,-48,0,4,-42},
    {0,0,0,0,0,0,0,1,1,1,1},
    {38,14,75,16,91,11,98,43,67,9,21,10,82,72,32,81,48,60,2,91,10,90,12,83},

    {-92,-92,-82,-80,-76,-66,-64,-64,-56,-48,-38,-38,-34,-32,-32,-10,
    -8,-6,-2,0,8,10,18,20,22,22,30,34,38,38,38,44,50,52,56,64,64,66,70,76,88},
    {0,1,1,0,0,1,1,0,0,0,1,1,1,1},

    {1,4,4,4,4,8,12,13,14,14,22,25,25,27,29,33,36,38,40,40,40,
    41,47,47,47,48,48,50,51,52,52,52,55,56,59,59,62,64,66,77,82,84,90,91,91,93},

    {-90,-60,-58,-72,92,54,-32,-70,-94,18,64,-90,-90,-56,82,-14,-74,-96,-90,
    -8,-48,76,-28,10,-52,-8,-46,-32,82,46,58,92,4,48,-96,-66,60,60,62,-68},

    {0,0,0,0,0,0,1,1,1,1},{42,17,77,96,72,36,74,97,7,94,80,7,27,58,49,81,51,9}};

    vector<int> param1 {17,4,6,17,25,11,38,22,8,16};
    vector<int> param2 {5,0,0,75,25,-1,4,22,8,11};
    for(int i = 0; i < param0.size(); ++i)
    {
        if(search(&param0[i].front(),param1[i],param2[i]) ==
        f_gold(&param0[i].front(),param1[i],param2[i]))
        {
            n_success+=1;
        }
    }
    cout << "#Results:" << "␣" << n_success << ",␣" << param0.size();
    return 0;
}
```

Figure 14: Original code in TransCoder-test, evaluating specific code implementations.

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <vector>
#include <fstream>
#include <iomanip>
#include <bits/stdc++.h>
using namespace std;

int f_gold ( int arr [ ], int n, int x ) {
    int i;
    for ( i = 0;
    i < n;
    i ++ ) {
        if ( arr [ i ] == x ) return i;
    }
    return - 1;
}


//TOFILL

template <typename T>
int f_gold(T arr, int n, int x) {
    if constexpr (is_same_v<T, vector<int>>) {
        return f_gold(&arr.front(), n, x);
    } else {
        return f_gold(arr, n, x);
    }
}

template <typename T>
int search(T arr, int n, int x) {
    if constexpr (is_same_v<T, vector<int>>) {
        return search(&arr.front(), n, x);
    } else {
        return search(arr, n, x);
    }
}

int main() {
    int n_success = 0;
    vector<vector<int>> param0
    {{4,5,5,11,13,14,15,19,22,22,23,26,29,29,36,44,48,49,65,65,
    67,68,70,76,79,79,81,85,88,91,91,92,92,97},

    {-24,-78,-32,-48,0,4,-42},
    {0,0,0,0,0,0,0,1,1,1,1},
    {38,14,75,16,91,11,98,43,67,9,21,10,82,72,32,81,48,60,2,91,10,90,12,83},

    {-92,-92,-82,-80,-76,-66,-64,-64,-56,-48,-38,-38,-34,-32,-32,-10,
    -8,-6,-2,0,8,10,18,20,22,22,30,34,38,38,38,44,50,52,56,64,64,66,70,76,88},
    {0,1,1,0,0,1,1,0,0,0,1,1,1,1},

    {1,4,4,4,4,8,12,13,14,14,22,25,25,27,29,33,36,38,40,40,40,
    41,47,47,47,48,48,50,51,52,52,52,55,56,59,59,62,64,66,77,82,84,90,91,91,93},

    {-90,-60,-58,-72,92,54,-32,-70,-94,18,64,-90,-90,-56,82,-14,-74,-96,-90,
    -8,-48,76,-28,10,-52,-8,-46,-32,82,46,58,92,4,48,-96,-66,60,60,62,-68},

    {0,0,0,0,0,0,1,1,1,1},{42,17,77,96,72,36,74,97,7,94,80,7,27,58,49,81,51,9}};

    vector<int> param1 {17,4,6,17,25,11,38,22,8,16};
    vector<int> param2 {5,0,0,75,25,-1,4,22,8,11};
    for(int i = 0; i < param0.size(); ++i)
    {
        if(search(param0[i], param1[i], param2[i]) == f_gold(param0[i], param1[i], param2[i]))
        {
            n_success+=1;
        }
    }
    cout << "#Results:" << " " << n_success << ", " << param0.size();
    return 0;
}
```

Figure 15: Enhanced code in TransCoder-test-X, evaluating various equivalent code implementations.

C++ Code:

```
int f_gold ( int num ) {
  if ( num < 0 ) return f_gold ( - num );
  if ( num == 0 || num == 7 ) return 1;
  if ( num < 10 ) return 0;
  return f_gold ( num / 10 - 2 * ( num - num / 10 * 10 ) );
}
```

Original Java Code:

```
static boolean f_gold ( int num ) {
  if ( num < 0 ) return f_gold ( - num ) ;
  if ( num == 0 || num == 7 ) return true ;
  if ( num < 10 ) return false ;
  return f_gold ( num / 10 - 2 * ( num - num / 10 * 10 ) ) ;
}
```

Enhanced Java Code:

```
static int f_gold ( int num ) {
  if ( num < 0 ) return f_gold ( - num ) ;
  if ( num == 0 || num == 7 ) return 1 ;
  if ( num < 10 ) return 0 ;
  return f_gold ( num / 10 - 2 * ( num - num / 10 * 10 ) ) ;
}
```

Figure 16: Alignment of return types for parallel data in TransCoder-test.

Original Evaluation Code:

```
if __name__ == '__main__':
    param = [
    ([6,7,15,42,47,54,56,59,59,64,68,70,71,75,91,93], 0, 15, 71),
([6,7,15,42,47,56,54,59,59,64,68,71,70, 75,91,93], 0, 15,    71),
([-92,-96,-68,-40,70], 0, 4, ,     -96),
([-92,-86,-68,-40,70], 0, 4,    20),
([-3,-1,0,30,10,45,70,60], 0, 7,     0),
([-3,-1,0,10,5,45,60,50], 0, 7,    12),
([-3,-1,0,10,30,45,60,70], 0, 7,    18),
([0,0,1], 0, 2,     20),
([1,1,1], 0, 2,     17),
([30,2,30,45], 0, 3,    28)
    ]
    n_success = 0
    for i, parameters_set in enumerate(param):
        if binarySearch(*parameters_set) == f_gold(*parameters_set):
            n_success+=1
    print("#Results:␣%i,␣%i" % (n_success, len(param)))
```

Enhanced Evaluaion Code:

```
if __name__ == '__main__':
    param = [
    ([6,7,15,42,47,54,56,59,59,64,68,70,71,75,91,93], 0, 15, 71),
([6,7,15,42,47,56,54,59,59,64,68,71,70, 75,91,93], 0, 15,    71),
([-92,-96,-68,-40,70], 0, 4,    -96),
([-92,-86,-68,-40,70], 0, 4,    20),
([-3,-1,0,30,10,45,70,60], 0, 7,     0),
([-3,-1,0,10,5,45,60,50], 0, 7,    12),
([-3,-1,0,10,30,45,60,70], 0, 7,    18),
([0,0,1], 0, 2,     20),
([1,1,1], 0, 2,     17),
([30,2,30,45], 0, 3,    28)
    ]
    n_success = 0
    for i, parameters_set in enumerate(param):
        if binarySearch(*parameters_set) == f_gold(*parameters_set):
            n_success+=1
    print("#Results:␣%i,␣%i" % (n_success, len(param)))
```

Figure 17: Inherent errors in the unit test template present in TransCoder-test.

7136

H   Example in TransCoder-test-X

In this section, we will present the evaluation instance in TransCoder-test-X and example of ExeCoder's responses to the instance, as shown in Figure 18. In this example, ExeCoder performs the translation from C++ to Python, where the function calculates the number of unique non-zero elements in a given matrix.

Prompt: Translate the given code from cpp to python. The input Code is marked with <Code>and </Code>.

Input:

```
<Code>
int countUnique ( int mat [ ] [ MAX ] , int n , int m ) {
  int rowsum [ n ] , colsum [ m ] ;
  memset ( colsum , 0 , sizeof ( colsum ) ) ;
  memset ( rowsum , 0 , sizeof ( rowsum ) ) ;
  for ( int i = 0 ;
  i < n ;
  i ++ ) for ( int j = 0 ;
  j < m ;
  j ++ ) if ( mat [ i ] [ j ] ) {
    rowsum [ i ] ++ ;
    colsum [ j ] ++ ;
  }
  int uniquecount = 0 ;
  for ( int i = 0 ;
  i < n ;
  i ++ ) for ( int j = 0 ;
  j < m ;
  j ++ ) if ( mat [ i ] [ j ] && rowsum [ i ] == 1 && colsum [ j ] == 1 ) uniquecount ++ ;
  return uniquecount ;
}
</Code>
```

ExeCoder Response:

```python
def countUnique ( mat , n , m ) :
    rowsum = [ 0 ] * n
    colsum = [ 0 ] * m
    for i in range ( n ) :
        for j in range ( m ) :
            if ( mat [ i ] [ j ] ) :
                rowsum [ i ] += 1
                colsum [ j ] += 1
    uniquecount = 0
    for i in range ( n ) :
        for j in range ( m ) :
            if ( mat [ i ] [ j ] and rowsum [ i ] == 1 and colsum [ j ] == 1 ) :
                uniquecount += 1
    return uniquecount
```

Figure 18: Example of ExeCoder's evaluation on TransCoder-test-X.