INTERTRANS: Leveraging Transitive Intermediate Translations to Enhance LLM-based Code Translation

Marcos Macedo
School of Computing
Queen's University
Kingston, ON, Canada
marcos.macedo@queensu.ca

Yuan Tian
School of Computing
Queen's University
Kingston, ON, Canada
y.tian@queensu.ca

Pengyu Nie
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
pynie@uwaterloo.ca

Filipe R. Cogo
Centre for Software Excellence
Huawei Canada
Kingston, ON, Canada
filipecogo@acm.org

Bram Adams
School of Computing
Queen's University
Kingston, ON, Canada
bram.adams@queensu.ca

***Abstract*—Code translation aims to convert a program from one programming language (PL) to another. This long-standing software engineering task is crucial for modernizing legacy systems, ensuring cross-platform compatibility, enhancing performance, and more. However, automating this process remains challenging due to many syntactic and semantic differences between PLs. Recent studies show that even advanced techniques such as large language models (LLMs), especially open-source LLMs, still struggle with the task.**

Currently, code LLMs are trained with source code from multiple programming languages, thus presenting multilingual capabilities. In this paper, we investigate whether such capabilities can be harnessed to enhance code translation. To achieve this goal, we introduce INTERTRANS, an LLM-based automated code translation approach that, in contrast to existing approaches, leverages intermediate translations to bridge the syntactic and semantic gaps between source and target PLs. INTERTRANS contains two stages. It first utilizes a novel Tree of Code Translation (ToCT) algorithm to plan transitive intermediate translation sequences between a given source and target PL, then validates them in a specific order. We evaluate INTERTRANS with three open LLMs on three benchmarks (i.e., CodeNet, HumanEval-X, and TransCoder) involving six PLs. Results show an *absolute improvement* of 18.3% to 43.3% in Computation Accuracy (CA) for INTERTRANS over Direct Translation with 10 attempts. The best-performing variant of INTERTRANS (with the Magicoder LLM) achieved an average CA of 87.3%-95.4% on three benchmarks.

***Index Terms*—automated code translation, large language models, LLM, tree of code translation, intermediate representation**

I. INTRODUCTION

Automatically translating source code between different programming languages (PLs) can significantly reduce the time and effort required for software development teams. Researchers have proposed various automated code translation methods. Data-driven learning-based approaches [34], [35] have shown impressive improvements over traditional rule-based methods [1], [2], [7]. Unlike rule-based approaches, which rely on handcrafted rules and program analysis techniques, learning-based methods can automatically learn syntactic and semantic patterns from large-scale code repositories.

Large language models (LLMs) have shown promising results in various software engineering tasks [16]. Pre-trained on extensive code and text data, LLMs with billions of parameters can perform code translation without requiring task-specific fine-tuning. This eliminates the need for costly and time-consuming processes involved in collecting training datasets and developing specialized models for code translation.

Recent studies have shown that the performance of LLM-based automated code translation, particularly with open-source LLMs, is still far from the production level, with correct translations ranging from 2.1% to 47.3% [30], [38]. These studies found that many errors in LLM-generated code translations stem from the models' lack of understanding of syntactic and semantic discrepancies between source and target languages, which can vary significantly across different pairs. For instance, 80% of the errors in translating from C++ to Go are due to syntactic and semantic differences, while only 23.1% of such errors occur when translating from C++ to C [30]. This variation is intuitive, as certain PLs naturally share more similarities in syntax and semantics than others.

A similar phenomenon has been observed in machine translation for human languages, where translating between certain languages is easier than others [21]. To improve translations for challenging language pairs, a common strategy is to use parallel corpora with a pivot (bridge) language [20]. Traditional statistical machine translation between non-English languages, such as French to German, often involves pivoting through English [37]. This approach remains effective with

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00236

the rise of multilingual neural machine translation models. In a recent work by Meta [15], training language pairs were collected based on linguistic families and bridge languages, facilitating translation across numerous language pairs without exhaustively mining every possible pair.

Inspired by this idea, this paper explores the potential of leveraging transitive intermediate translations from a source PL into other PLs before translating to the desired target PL, an idea not previously explored in the field of automated code translation. For example, to translate a program written in Python to Java, we might first translate it from Python to C++ and then from C++ to Java, as illustrated in Figure 1. This process is done through prompting, without additional training data, thanks to code LLMs that are pre-trained on text and code across multiple PLs and naturally possess multilingual capabilities. While this idea is inspired by machine translation, its potential in the inference stage of LLM-based translation approaches has not been explored.

The idea of utilizing existing PLs as "bridges" is different than earlier work, TransCoder-IR [35], a non-LLM learning-based method that enhances source code pairs by incorporating their corresponding low-level, language-agnostic compiler Intermediate Representations (IR), such as LLVM IRs [24], into the training dataset. Instead of relying on one unified IR to bridge any pair of cross-PL translations, we systematically explore different potential transitive intermediate translations using multiple existing PLs.

INTERTRANS, our novel LLM-based code translation approach that enhances source-target translations via transitive intermediate translations, operates in two stages. In the first stage, a method called Tree of Code Translations (ToCT) generates a *translation tree* containing all potential translation paths for a specific source-target PL pair, conditioned to a set of pre-defined intermediate PLs and the maximum number of intermediate translations to be explored. In the second stage, translation paths are turned into LLM prompts that are executed in a breadth-first order. INTERTRANS then uses a readily available test suite to validate whether the generated translation to the target language is correct, enabling early termination of translation path exploration if a successful path is found before completely exploring the translation tree.

To evaluate the effectiveness of INTERTRANS, we conducted experiments using three code LLMs (Code Llama [32], Magicoder [36], and StarCoder2 [26]) on 4,926 *translation problems* sourced from three datasets, i.e., CodeNet [31], HumanEval-X [39], and TransCoder [33]. Each translation problem aims to translate a program writing in a source PL to a target PL. These problems involve 30 different source-target PL pairs across six languages: C++, JavaScript, Java, Python, Go, and Rust. Our results show that INTERTRANS consistently outperforms direct translation (i.e., without intermediate language translation) with 10 attempts, achieving an absolute Computational Accuracy (CA) improvement of 18.3% to 43.3% (median: 28.6%) across the three LLMs and datasets. Through ablation studies, we analyzed the effects of varying the number and selection of intermediate languages on INTERTRANS's performance. Generally, increasing the number of intermediate translations enhances CA, though the benefits taper off after three translations. Similarly, incorporating more intermediate languages is advantageous, but gains slow after including three languages. The effectiveness of specific intermediate PLs varies across translation pairs, with notable patterns observed in translations from C++/Python to Java via Rust and from Rust to Go via C++. The main contributions of this paper are as follows:

- We present the first study demonstrating that intermediate translations based on existing PLs can enhance the performance of LLM-based code translation.
- We propose ToCT, a novel planning algorithm designed to explore intermediate translations effectively. We also introduce INTERTRANS, an LLM-based code translation approach that uses ToCT and is orthogonal to existing approaches for code translation.
- We conducted a comprehensive empirical study to evaluate INTERTRANS. Our results highlight the effectiveness of INTERTRANS in enhancing LLM-based code translation. We also provide insights for the practical application of INTERTRANS.

The code for implementing INTERTRANS, the datasets, and the notebooks for generating the experiment results are available at: https://github.com/RISElabQueens/InterTrans.

II. INTERTRANS

INTERTRANS translates programs from a source to a target language using an LLM and a series of transitive intermediate translations. The input of INTERTRANS includes: (1) a LLM, (2) a program P_s written in a source language L_s, (3) the target language L_t, (4) a non-empty intermediate PL set L which contains L_s but excludes L_t, (5) a hyper-parameter $maxDepth$, which determines the maximum number of transitive intermediate translations. INTERTRANS utilizes a readily available test suite to evaluate the accuracy of the generated program(s) TP written in the target language, i.e., $TP = \{P_t | P_t \in P_{s,L_t} \wedge s \neq t\}$, where P_{s,L_t} is the set of programs written in L_t that represent translation candidates for P_s.

INTERTRANS operates in two stages. In Stage 1, it constructs all possible *translation (PL) paths* using a novel approach called the *Tree of Code Translations (ToCT)*, which identifies potential sequences of transitive translations from L_s to L_t via intermediate languages from the set L. Stage 2 then uses the source program P_s and the PL paths generated from Stage 1 to perform inferences with an LLM to generate a set of target programs TP written in L_t. These programs, each corresponding to a translation path, are generated and verified sequentially against a test suite. The algorithm terminates when a successful translation is identified, indicated by a P_t that passes the test suite. The following subsections provide detailed descriptions of each stage, accompanied by a running example.

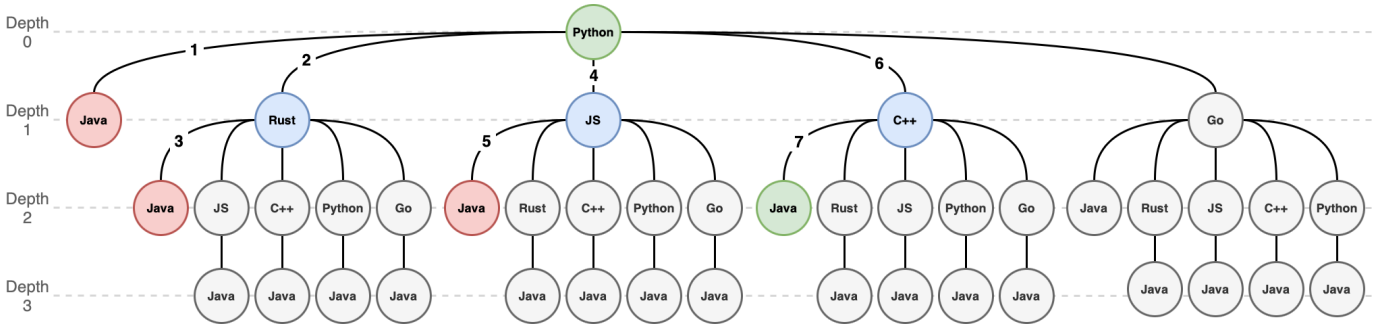

Fig. 1: Running example of INTERTRANS with maxDepth=3 for translating Python to Java, showing a successful translation through C++ after exploring various translation paths. Red nodes represent unsuccessful translations, blue nodes indicate explored translations, green nodes denote successful translations, and grey nodes are skipped translations. The number along with each edge is the execution order of the translations.

A. Stage 1: Generating Tree of Code Translations (ToCT)

Algorithm 1 specifies how ToCT creates (plans) translation PL paths for a given translation PL pair utilizing a set of intermediate languages. Since ToCT operates at the level of translation PL pairs, this planning algorithm only needs to run once for all translation problems involving the same source and target languages.

In ToCT, the intermediate language set L includes the source language L_s but excludes the target language L_t. This is because L_t should be the final target and should not occur as an intermediate step in the translation process, while we should allow L_s to appear in intermediate translations (for cases where a source program can be "simplified" by translating to and from another PL). Below, we use a running example, shown in Figure 1, to illustrate this algorithm. In this example, we aim to translate a Python program to Java, and we consider a maximum depth ($maxDepth$) of 3, meaning that at most three edges can be included in a translation path. The set of intermediate languages includes: Python, Rust, JavaScript, C++, and Go.

ToCT (see Algorithm 1) starts by enqueueing and then dequeueing the source PL, yielding the current path starting from the source PL, i.e., [Python] in our running example. The algorithm continuously explores possible transitions either to an intermediate language or directly to the target language to complete the translation path. This results in the following paths: [Python, Java], [Python, Rust], [Python, JavaScript], [Python, C++], and [Python, Go]. Each of these new paths, along with the incremented depth of 1, is enqueued into Q.

Continuing this process, the algorithm dequeues [Python, Java] and since it ends with the target PL, this path will be added to the final translation PL path output list. Next, the algorithm dequeues [Python, Rust] and explores further transitions, appending each language from the set L to the current path, but excluding Rust to avoid translation between the same PLs. This results in new paths like [Python, Rust, Java], [Python, Rust, JavaScript], etc., which are then enqueued with a depth of 2. This process repeats for all potential paths within the specified maximum depth, ensuring all possible translation paths from Python to Java are explored and recorded.

Algorithm 1 ToCT path generation algorithm

Input: L_s: Source programming language, L_t: Target programming language, $maxDepth$: Maximum depth of the tree, $L = \{L_i\}$: A set of intermediate languages.
Output: All paths from L_s to L_t

```
1:  Initialize an empty list paths
2:  Initialize a queue Q
3:  Enqueue ([L_s], 0) into Q
4:  while Q is not empty do
5:      (currentPath, currentDepth) ← Dequeue Q
6:      currentLang ← last element of currentPath
7:      if currentLang = targetLang then
8:          Append currentPath to paths
9:      else if currentDepth < maxDepth then
10:         for lang ∈ {L_t} ∪ L do
11:             if lang ≠ currentLang then
12:                 newPath ← currentPath + [lang]
13:                 Enqueue (newPath, currentDepth+1) into Q
14: return paths
```

B. Stage 2: Sequential Verification of ToCT

For a specific translation problem, the second stage of the INTERTRANS approach (see Algorithm 2) takes the ToCT-generated plan for the problem's source and target PL, i.e., the list $paths$ from Algorithm 1, to (1) determine the order of the paths that will be verified, (2) generate the translations using an LLM and a prompt template $PromptT$, and (3) evaluate the translations to the target language using the given test suite T. To make INTERTRANS more efficient, an *early-stopping mechanism* is applied (Lines 16-17): as soon as one path successfully translates the code into L_t, Algorithm 2 terminates.

Following the design of ToCT, it is common for multiple paths to share the same initial transitive translation edges. For instance, Path p_1: [Python, Rust, Java] and Path p_2: [Python, Rust, JavaScript] Java share the first translation (edge). To further improve the efficiency of INTERTRANS, we apply memoization within each path to ensure the same edge is

Algorithm 2 Algorithm for executing ToCT-generated plans

Input: P_s: An input source program, $paths$: A list of translation PL paths generated by ToCT, LLM: a LLM that can generate code into $\{L_t\} \cup L$, $PromptT$: A prompt template for the specific LLM, T: a test suite for evaluating the computational accuracy of the generated translation to target PL L_t.

Output: Successful translation, if any, from L_s to L_t for P_s

```
1:  Sort paths by their length in ascending order
2:  for path p ∈ paths do
3:    for edge E_k ∈ p do
4:      if E_k is already processed then
5:        continue with cached output
6:      else
7:        Retrieve extracted source code from E_{k-1}
8:        Create a new prompt using PromptT
9:        Perform translation using LLM and the prompt
10:       Extract source code from inference output
11:       if Failed extracting source code then
12:         break continue with the next path p
13:       Save the extracted code for E_k to cache
14:     if Target language of E_k = L_t then
15:       Verify this translation using the test suite T
16:       if Test suite passes then
17:         return the translation found
18: return the translation failed
```

not computed more than once (Lines 4-5). Note that this optimization requires deterministic output for the same input prompt, which is ensured via a fixed seed in our experiment. Only new translation edges after branching from a shared path are processed. In other words, if p_1 is verified first, then p_2 will reuse the resulting Rust program saved in the memory cache to continue its unique translation to JavaScript.

In Algorithm 2, the input $paths$ are first sorted by length in descending order (Line 1), ensuring that the first explored path is always a direct translation from L_s to L_t. If no direct translation is found, the sorting step following path generation ensures that the algorithm maximizes the number of paths explored relative to the total translations performed. For instance, for our running example, in Figure 1 the numbers along the edges indicate the sequence of steps performed following Algorithm 2 for a specific P_s. The direct translation, i.e., [Python, Java], will be verified first. If the transferred code generated following this path fails, then the path [Python, Rust, Java] will be verified, and so on, until the transferred code generated by path [Python, C++, Java] passes the test suite T, the algorithm stops and returns the successful translation.

For each edge in a translation path, we first generate translated code for the target language of the previous edge (Line 7). Next, we use the given LLM to generate the translation output (Lines 8-9), then extract the source code from this output (Line 10). If the extraction is successful, we then verify if it can pass the test suite T.

III. Experiment Design

We evaluate the effectiveness of InterTrans by answering the following four research questions:

- RQ1: How effective is InterTrans compared to direct translation and other baselines?
- RQ2: How could varying the $maxDepth$ affect the performance of InterTrans?
- RQ3: How could varying the selection of intermediate languages affect InterTrans?
- RQ4: How do semantic errors propagate in InterTrans?

A. Benchmark Dataset Collection and Pre-Processing

Our experiment dataset consists of 4,926 translation problems across 30 source-target translation PL pairs involving six PLs - C++, Go, Java, JavaScript, Python, and Rust. When creating our experiment dataset, we considered three existing datasets. Below, we describe the creation of our experimental datasets from these sources.

TransCoder: The original TransCoder dataset [33] was created by manually collecting coding problems and solutions written in C++, Java, and Python from GeeksforGeeks [4]. Recently, Yang et al. [38] discovered quality issues in this dataset and subsequently conducted a manual verification and curation of the dataset to ensure its correctness. In this study, we reused their curated version, containing a total of 2,826 translation problems and corresponding test suites. We employed the full version of this dataset for comparisons with SOTA learning-based approaches.

HumanEval-X: HumanEval-X [39] extends the python-only code generation evaluation dataset HumanEval [11] with additional canonical solutions and test cases in six PLs: C++, Go, Java, JavaScript, Python, and Rust. We created translation pairs for all 164 tasks in HumanEval-X across the six languages, resulting in 4,920 translation problems. Due to computational constraints (particularly required by the ablation studies performed to understand the impact of varying variables on the performance of InterTrans), we randomly sampled 1,050 translation problems, stratified across the 30 source-target translation pairs, ensuring a 99.9% confidence level.

CodeNet: CodeNet [31] contains programs written in 55 programming languages for learning and evaluating coding tasks and was adopted in a recent empirical study by Pan et al. [30] on LLM introduced translation bugs. Programming tasks in CodeNet are verified by matching the program outputs with the expected results. For our study, we selected tasks with at three test cases to ensure adequate test suite coverage, resulting in 1,112 programming tasks. From these tasks, we generated 15,660 translation problems by concentrating on the six PLs featured in HumanEval-X, removing problems with a file size exceeding 1KB (as a proxy for token length, to prevent inputting into the prompt problems longer than the model's token limit) and ensuring that each translated code snippet could be assessed using three test cases. We created a subset of 1,050 pairs from this dataset using stratified random sampling, ensuring a 99.9% confidence level.

B. Selected Large Language Models

InterTrans relies on an LLM that understands multiple PLs. Almost all recent code LLMs possess this multilingual capability. We have chosen the following three instruct-tuned LLMs over their base models, as instruct-tuned models are fine-tuned to follow prompted instructions more effectively.

Magicoder [36]: An open-source collection of LLMs trained on 75K synthetic instruction-response pairs and includes multiple model variants with different base models. All Magicoder models have around 7B parameters. We use the Magicoder-S-DS variant [6].

StarCoder2 [26]: An open-source collection of LLMs offered by the BigCode project [10]. StarCoder2 has instruction-tuned versions ranging from 1B to 34B parameters. We use the StarCoder2-15B variant [8].

CodeLlama [32]: An open-source collection of LLMs offered by Meta based on Llama 2, specialized in code generation, with 7B, 13B, and 34B parameters. We use the CodeLlama-13B variant [3].

We chose these models because of their proven effectiveness in code generation tasks and their open-source nature, which promotes accessibility and collaborative development.

C. Evaluation Metrics

Similar to recent studies on LLM-based code translation [30], [38], we adopt execution-based evaluation metrics, i.e., Computational Accuracy (CA) [33]. CA assesses whether a transformed target program produces the same outputs as the source function when given identical inputs. CA on a benchmark is the ratio of translation problems that have correctly translated to the target language.

D. Compared Approaches

Direct translation (CA@1 and CA@10): We compare INTERTRANS with direct translation by evaluating performance with a single attempt (CA@1) and multiple attempts (CA@10). For CA@10, a single prompt is used to generate ten translation candidates. The translation is considered successful if any of these ten attempts result in a correct translation. Comparing with CA@1 reveals the additional opportunities INTERTRANS discovers via ToCT. Since INTERTRANS utilizes multiple translation paths, it inherently makes more than one attempt, making a comparison with CA@1 alone insufficient. Hence, to find a fair number of attempts (k) for direct translation, we analyzed how many attempts (paths) INTERTRANS are required to achieve a successful translation across the experiments. On average, 3.9 attempts were needed, with 75% of cases successful within two attempts and less than 0.1% requiring between 59 and 83 attempts. Therefore, we chose CA@10 as a stronger baseline, allowing ten attempts with high temperature setting (0.7) to generate diverse variants and increase the chances of passing the test suite.

Direct translation SFT (CA@10): We also compare INTERTRANS with a version of direct translation that uses fine-tuned LLMs instead of base LLMs. This fine-tuned approach, which we refer to as *Direct Translation SFT (Supervised Fine-Tuning)*, involves training the LLMs on code translation pairs. To maintain consistency, we use the same evaluation dataset as in other settings and utilize the translation tasks excluded from the evaluation dataset for training. Specifically, we use the remaining 3,870 translation problems from the HumanEval-X benchmark that were not part of the evaluation dataset. To ensure that the models are trained on an equal number of PL pairs, we also sample 3,870 translation problems from the CodeNet dataset that were excluded from evaluation. We do not fine-tune models on the TransCoder dataset, as the entire dataset is used for evaluation. Moreover, TransCoder, a fine-tuned model from code translation pairs, has already been included as a baseline for our comparisons. After setting the training and testing datasets for HumanEval-X and CodeNet, we fine-tuned all three LLMs considered in this work and evaluated their performance in terms of CA@10.

Non-LLM SOTA approaches: TransCoder [33] is an unsupervised model pre-trained with cross-lingual language modeling, denoising auto-encoding, and back-translation, leveraging a vast amount of monolingual samples. TransCoder-IR [35], an incremental improvement, introduces the idea of using a low-level compiler Intermediate Representation (IR) to enhance translation performance. In addition to TransCoder's pretraining tasks, TransCoder-IR includes translation language modeling, translation auto-encoding, and IR generation. TransCoder-ST [34] is another enhanced version of TransCoder that uses automatically generated test cases to filter invalid translations, improving performance. These models are trained on only a few PLs, i.e., Python, C++, and Java.

GPT-3.5 and its enhanced version: GPT-3.5 is a powerful closed LLM provided by OpenAI that is capable of code generation. We consider the gpt-3.5-turbo-0613 version. UniTrans with GPT-3.5 is an enhanced version designed for code translation, proposed by Yang et al. [38]. UniTrans generates test cases to aid LLMs in repairing errors by integrating test execution error messages into prompts. Despite UniTrans with GPT-3.5 requiring additional program repair and extra test cases, we include it as a baseline since it represents the state-of-the-art performance on the TransCoder dataset.

E. Implementation

Our scalable reference implementation of the INTERTRANS algorithms is written in Go and implemented as a client (Python) and server (engine written in Go) architecture that communicates over gRPC [17]. The INTERTRANS engine utilizes vLLM [22] as the inference engine, given its performance and dynamic batching capabilities. It queries vLLM endpoints using round-robin to achieve data parallelism during inference and distribute the computational load evenly. The computational infrastructure used for our experiments consists of 6x NVIDIA RTX A6000 GPUs on an AMD EPYC Server with 128 CPU cores.

To ensure deterministic inference results from vLLM across all experiments involving InterTrans, we randomly generated a fixed random seed for inference. We configure the decoder

parameters with top-p set to 0.95, top-k set to 10, and the temperature set to 0.7 for both our approach and the direct translation. When evaluating the baseline performance of direct translation with CA@1 and CA@10, we do not fix the seed to ensure we generate diverse candidates. The selection of top-p, top-k, and temperature aligns with recent studies on code LLMs [14].

During our experiments, even after identifying a successful translation, we still continue to explore and verify all potential translation paths. While one would not do this in practice when using INTERTRANS, it was essential in our empirical study to collect comprehensive data on all translation paths needed for addressing our research questions, particularly RQ3 (impact of removing intermediate PLs). However, this does not impact the reported CA results for INTERTRANS.

For Direct translation SFT, our protocol for fine-tuning the LLMs for code translation is based on that of Liu et al. [25], who performed an empirical study to understand the efficacy of parameter-efficient fine-tuning (PEFT) across five software engineering tasks (defect detection, code clone detection, code summarization, project-specific code summarization and code translation). Across these five tasks, they demonstrate that updating 0.5% of the LLM's parameters leads to performance comparable to complete fine-tuning. Therefore, we use the same protocol. We use QLoRA PEFT approach with LoRA parameters r=16 and alpha=32, and a dropout of 0.1 [13]. We train the models for 30 epochs with early stopping on 4 RTX A6000 GPUs using the HuggingFace Accelerate library.

IV. RESULTS AND ANALYSIS

A. RQ1: Effectiveness of INTERTRANS

Approach: In INTERTRANS, the $maxDepth$ is set to 4, allowing for a maximum of four translations (edges) in a translation PL path. This parameter enables us to explore various translation paths (with 85 maximum attempts). The six PLs of the CodeNet and HumanEval-X benchmarks, i.e., Python, C++, JavaScript, Java, Rust, and Go, serve as intermediate languages. While the TransCoder dataset includes only Python, C++, and Java, additional languages like Rust, JavaScript, and Go can be used as intermediates. This flexibility is possible because INTERTRANS does not verify the correctness of intermediate translations unless they result in a program written in the target language.

Results: As shown in Table I, INTERTRANS consistently surpasses direct translation (CA@1 and CA@10) across all three datasets and all studied LLMs. It achieves an absolute improvement of 18.3% to 43.3% compared to direct CA@10.

Table II displays the comparison of INTERTRANS (with StarCoder2) against non-LLM SOTA approaches, GPT-3.5 and its enhanced version on the TransCoder dataset. The results show that our approach outperforms all others across all six source-target PL pairs. The second best performance is achieved by UniTrans with GPT-3.5. All the LLM-based approaches considered in Table II perform consistently better than the TransCoder models, further showcasing the promising potential of LLMs in automated code translation.

Table III presents a comparison of three approaches, i.e., direct translation, direct translation SFT, and INTERTRANS, on two benchmarks. In Table III, we observe that fine-tuning improved the performance of two LLMs across two datasets compared to their non-fine-tuned counterparts. However, CodeLlama on HumanEval-X showed a decrease in performance after fine-tuning. This result aligns with prior findings by Liu et al. [25], which reported that fine-tuning could negatively impact performance in code translation tasks. Despite these improvements over the original LLMs, INTERTRANS still outperforms the fine-tuned models by 7.4–47%.

B. RQ2: Impact of Varying $maxDepth$

Approach: INTERTRANS utilizes two hyper-parameters, one of which is $maxDepth$. This parameter controls the depth of the translation tree generated by Algorithm 1. In this research question, we investigate how this parameter affects the performance of INTERTRANS. Specifically, we vary $maxDepth$ from 1 (direct translation) to 4. We conducted pairwise comparisons across different depths (1 vs. 2, 1 vs. 3, 1 vs. 4, 2 vs. 3, 2 vs. 4, and 3 vs. 4) to evaluate the significance of the performance changes (i.e., the number of successful and unsuccessful translations) using the Chi-Square statistical test. To account for multiple comparisons across levels within the same model and dataset, we apply the Bonferroni correction to an alpha level of 0.05.

Results: The results of our experiments with varying values for $maxDepth$ are shown in Figure 2. We can observe that as the $maxDepth$ increases, the performance of INTERTRANS consistently improves, although the rate of improvement slows down towards longer paths. For instance, on HumanEval-X, increasing the $maxDepth$ from 1 to 2 results in an absolute improvement of 23.7% for Code Llama, from 2 to 3 results in an improvement of 6.6%, and from 3 to 4, the improvement is 3.2%. Similar patterns are observed across all nine combinations of models and datasets.

Regarding the statistical tests performed, we find that for all datasets and models, there is a statistically significant improvement in terms of CA as the depth increases. Exceptions to this trend are noted for Code Llama and StarCoder2 in the TransCoder dataset, where there is no significant increase in the CA metric when increasing the depth from 3 to 4, and for Code Llama and Magicoder in the HumanEval-X dataset with the same depth change. In other words, out of 54 comparisons (6 depth changes $\times$ 9) conducted, only 4 cases of increasing the depth do not lead to a statistically significant improvement in performance, all involving an increase from depth 3 to 4.

C. RQ3: Impact of Varying the Intermediate Programming Languages

Approach: Besides $maxDepth$, the other hyper-parameter of INTERTRANS is the set of intermediate PLs considered, which determines the width of the translation tree created by ToCT. In this RQ, we investigate the impact of reducing the set and specific types of intermediate PLs by addressing the following two sub-RQs:

TABLE I: Performance of InterTrans compared with Direct Translation. Abs Diff and Rel Diff mean the absolute difference and relative difference compared to Direct (CA@10). The source language column includes all PLs of a dataset. The set of target languages for a given source language includes all PLs of a dataset, except the source language.

Dataset	Source language	Total samples	CA@K (percentage)														
			Code Llama					Magicoder					StarCoder2				
			Direct (CA@1)	Direct (CA@10)	INTER TRANS	Abs. diff.	Rel. diff.	Direct (CA@1)	Direct (CA@10)	INTER TRANS	Abs. diff.	Rel. diff.	Direct (CA@1)	Direct (CA@10)	INTER TRANS	Abs. diff.	Rel. diff.
CodeNet	C++	175	32.0	42.9	61.1	18.3	42.7	50.3	50.9	88.0	37.1	73.0	29.1	40.0	81.7	41.7	104.3
	Go	175	30.3	34.3	61.1	26.9	78.3	50.9	53.1	85.7	32.6	61.3	45.7	50.3	85.1	34.9	69.3
	Java	175	25.7	38.9	55.4	16.6	42.6	45.1	45.7	85.1	39.4	86.2	36.6	41.1	85.7	44.6	108.3
	JavaScript	175	22.3	33.7	64.6	30.9	91.5	50.9	50.9	87.4	36.6	71.9	24.0	25.7	82.9	57.1	222.2
	Python	175	14.3	19.4	57.1	37.7	194.1	41.1	42.3	91.4	49.1	116.2	38.3	44.0	87.4	43.4	98.7
	Rust	175	29.7	38.3	65.1	26.9	70.1	50.9	51.4	86.3	34.9	67.8	36.0	45.1	83.4	38.3	84.8
Total/Average		1,050	25.7	34.6	**60.8**	26.2	75.8	48.2	49.0	**87.3**	38.3	78.1	35.0	41.0	**84.4**	43.3	105.6
HumanEval-X	C++	175	70.3	78.9	91.4	12.6	15.9	73.1	74.3	97.7	23.4	31.5	61.1	66.9	86.3	19.4	29.1
	Go	175	64.0	71.4	90.3	18.9	26.4	62.9	64.0	98.3	34.3	53.6	52.0	55.4	83.4	28.0	50.5
	Java	175	58.3	68.0	87.4	19.4	28.6	65.7	67.4	93.1	25.7	38.1	46.9	48.6	86.3	37.7	77.6
	JavaScript	175	57.1	73.1	93.1	20.0	27.3	60.6	60.6	96.0	35.4	58.5	44.0	44.0	80.6	36.6	83.1
	Python	175	53.7	64.6	82.3	17.7	27.4	61.7	62.9	89.7	26.9	42.7	36.6	36.6	77.1	40.6	110.9
	Rust	175	59.4	72.0	93.7	21.7	30.2	71.4	72.0	97.7	25.7	35.7	52.6	54.3	81.1	26.9	49.5
Total/Average		1,050	60.5	71.3	**89.7**	18.4	25.8	65.9	66.9	**95.4**	28.6	42.7	48.9	51.0	**82.5**	31.5	61.9
TransCoder	C++	946	73.9	75.9	93.2	17.3	22.8	67.9	67.9	92.7	24.8	36.6	63.5	65.2	93.8	28.5	43.8
	Java	931	77.7	79.5	94.8	15.4	19.3	77.4	77.4	91.9	14.5	18.7	79.3	79.9	95.1	15.1	19.0
	Python	949	67.3	69.3	91.6	22.2	32.1	33.5	33.5	87.8	54.3	161.9	73.9	74.6	92.7	18.1	24.3
Total/Average		2,826	72.9	74.9	**93.2**	18.3	24.5	59.5	59.5	**90.8**	31.3	52.6	72.2	73.2	**93.8**	20.6	28.2

TABLE II: CA performance of INTERTRANS and other baselines on TransCoder data set. We adopt the numbers of baseline performance from Yang et al. [38]. A "–" means there is no reported performance on the specific pair.

Models	C++ to Python	Python to C++	Java to C++	C++ to Java	Java to Python	Python to Java	Avg.
TransCoder	36.6	30.4	27.8	49.8	–	–	36.2
TransCoder-IR	–	–	41.0	40.5	–	–	45.8
TransCoder-ST	46.3	47.8	49.7	64.7	–	–	52.2
GPT-3.5	87.1	89.5	92.9	82.2	89.2	74.9	86.0
UniTrans w/ GPT-3.5	88.8	94.2	94.9	85.5	91.2	81.3	87.9
InterTrans w/ StarCoder2	**93.3**	**94.4**	**96.1**	**94.2**	**94.0**	**91.1**	**93.8**

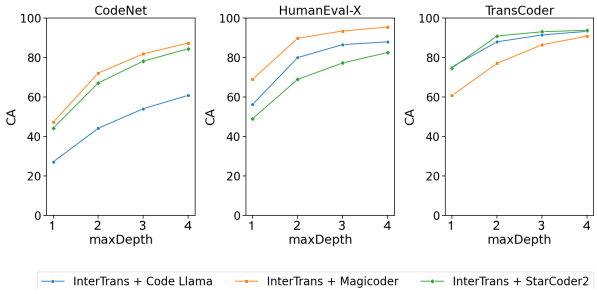

Fig. 2: Performance of INTERTRANS with varying $maxDepth$ on three datasets.

- **RQ3.1:** How does the number of available intermediate PLs influence the performance of INTERTRANS?
- **RQ3.2:** How does the removal of a specific intermediate PL affect the performance of INTERTRANS?

To address the above two sub-RQs, we first conducted an ablation study across all possible combinations of intermediate PLs from the experiments conducted in RQ1 and RQ2, using a $maxDepth$ of 4 with six PLs. Each ablation involves the removal of all translation paths that contain a subset of the set of intermediate PLs. In particular, for each translation, we computed all 31 possible combinations of removing 1 to 5 PLs from the intermediates (i.e. all combinations of intermediate PLs, except those that include the target language). We then removed the edges that involve each individual set and measured whether the translation remained successful (i.e., at least one translation path leads to a correct translation). This ablation was performed for each sample of the nine experiments (3 datasets and 3 LLMs), and we recorded which removed sets caused the translation to be unsuccessful. For this analysis, we leveraged the data we generated during our evaluation described in Section IV-A, where we recorded the execution result of all translation paths in the translation trees.

To answer RQ3.1 in specific, we aggregated the results from the 458,118 translations (4,926 tasks from 3 datasets, each with 31 removal combinations using 3 different models) based on the number of intermediate PLs removed, i.e., the cardinality of the set of removed PLs. This analysis helps us understand the overall impact of the number of intermediate languages on translation success rate. Figure 4 shows the performance of INTERTRANS with 0 (direct translation) to 5 intermediate PLs on three datasets with three base models.

Additionally, in RQ3.2, to investigate whether specific languages are more impactful as intermediates, we analyzed the results from the translations of RQ3.1 that are associated with the removal of a single intermediate PL. We then calculated the mean absolute decrease in translation success for each of the 30 PL pairs in our experiments, caused by the removal of each specific PL. The heatmap in Figure 3 shows the mean absolute decrease in CA when a PL is removed from each of the 30 translation pairs. Darker cells indicate a greater loss in CA, highlighting which PLs are more critical for maintaining high translation accuracy. This heatmap also shows the results of a statistical significance test (Chi-squared Goodness of Fit) we conducted by comparing the the number of successful and unsuccessful translations before (control group) and after (ex-

TABLE III: Comparison of performance on the HumanEval-X and CodeNet datasets for Direct Translation with base LLMs (Direct), Direct Translation with fine-tuned LLMs (Direct SFT), and INTERTRANS.

Dataset	Source language	Total samples	CA@K (percentage)														
			Code Llama					Magicoder					StarCoder2				
			Direct CA@10	Direct SFT CA@10	InterTrans	Abs. diff	Rel. diff.	Direct CA@10	SFT CA@10	InterTrans	Abs. diff.	Rel. diff.	Direct CA@10	SFT CA@10	InterTrans	Abs. diff.	Rel. diff.
CodeNet	All	1050	34.6	56.6	**60.8**	4.2	7.4%	49.0	63.2	**87.3**	24.1	38.1%	41.0	57.4	**84.4**	27	47.0%
HumanEval-X	All	1050	71.3	66.5	**89.7**	23.2	34.9%	66.9	86.1	**95.4**	9.3	10.8%	51.0	76.2	**82.5**	6.3	8.3%

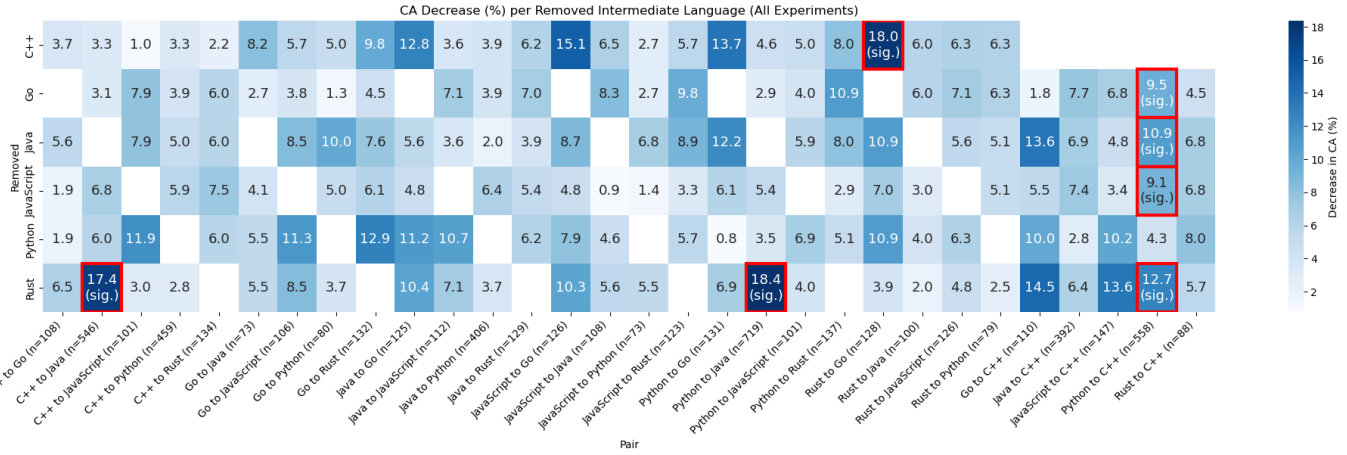

Fig. 3: HeatMap showing the mean absolute decrease in CA (%) when removing a programming language from the intermediates used in our approach, compared to not removing any PL (across all datasets and models). Framed cells annotated with "(sig.)" indicate statistically significant results. The "n" value in the x-axis labels indicates the sample size for each translation pair. For each translation pair, one cell is empty because (by definition) the target PL can not be removed.

perimental group) the removal of a specific PL ($alpha = 0.05$, Bonferroni-corrected). In the heatmap, we highlight the cells associated with statistically significant differences.

Results of RQ3.1: We can observe in Figure 4 that the inclusion of more intermediate PLs consistently improves the translation accuracy of INTERTRANS. For instance, for Magicoder on CodeNet, increasing from zero to one intermediate PL results in a significant improvement of 9.3% in CA (from 47.2% to 56.5%). Similarly, adding a second intermediate PL increases the CA metric by 12.9%, and a third intermediate PL results in a 9.2% increase. However, beyond this point, the incremental gains begin to diminish. Adding a fourth intermediate PL yields a 5.6%, while the addition of a fifth intermediate PL results in a relatively smaller increase of 3.2%. This trend suggests that while the inclusion of intermediate PLs is beneficial for improving translation accuracy, the marginal returns decrease as more intermediate PLs are added. The most substantial gains are observed when moving from zero to three intermediates, after which the improvements become more modest.

Results of RQ3.2: Figure 3 demonstrates that the importance of intermediate PLs varies across different translation pairs. For instance, when translating a program written in C++ to Java (second column of the heatmap), removing Rust as an intermediate PL resulted in a 17.4% decrease in successful translations. In contrast, removing any other PL only led to a decrease ranging from 3.1% to 6.8%. This emphasizes the critical role of certain intermediate PLs in achieving accurate

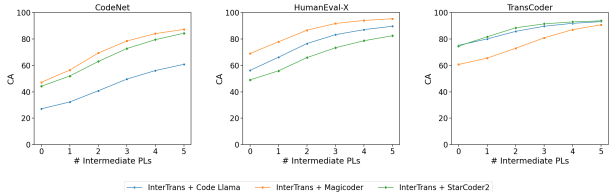

Fig. 4: Average performance of INTERTRANS with varying number of intermediate PLs on three datasets.

translations, yet we could not find any consistent trend across translation pairs.

To better understand the results of Figure 3, we conducted a case study in which we manually examined three translations associated with an absolute decrease in CA higher than 15%. These cases were selected because their absolute decrease in CA were not only substantial but also statistically significant.

Translation from Python to Java via Rust: We observed that direct translation attempted to identify Java APIs and operations that are functionally equivalent to the Python ones, but which may not exist. Moreover, it struggles to handle type requirements in Java and thus often leads to the wrong use of API. Translating from Rust provided a pathway for translating these operations more accurately into Java. Figure 5 illustrates an example. In the source Python code, the expression "`if int(...) in []`" checks whether an integer is present in a list of integers. The direct transla-

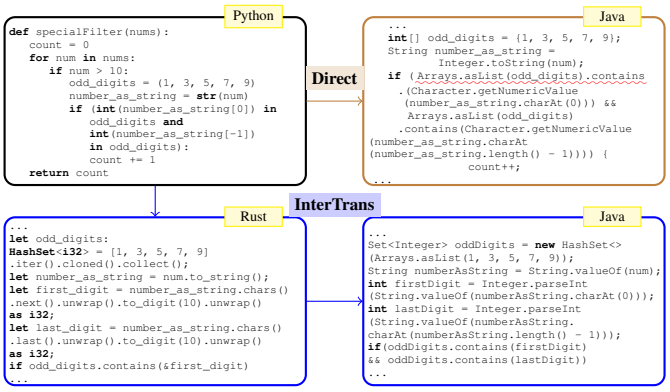

Fig. 5: Example of translation from Python to Java via Rust.

tion uses the `Arrays.asList().contains()` API as an equivalent, but `Arrays.asList()` only accepts reference types, not primitive types. Consequently, passing an `int[]` to `Arrays.asList()` results in a `List<int[]>`, a list of arrays, failing to check for individual integers. In contrast, via the intermediate translation, the Rust code employs a `HashSet` for `odd_digits`, which translates correctly to `Set<Integer>` in Java. This allows for accurate use of the `contains` method to check for individual elements.

Translation from C++ to Java via Rust: We found that direct translation frequently copies code from C++ to Java (since the two languages indeed have similar syntax), but some operations permitted in C++ are not permitted in Java, e.g., using square brackets to access vectors/lists and strings, implicit type conversion to lower precision types, etc. In INTERTRANS, C++ code is first translated into Rust with a more restrictive and distinct syntax, so that the LLM is aware of the syntax differences and avoids directly coping code.

Translation from Rust to Go via C++: Since Rust and Go both support type inference during variable declaration (i.e., developers can declare a variable without specifying the type if it can be inferred from the initialization expression), we observed that direct translation may misunderstand the types of local variables and try to apply invalid APIs on them. For instance, direct translation attempts to translate Rust's `i32::abs` to Go's `Math.Abs`, without noticing that the return value (`w`) has changed from integer to float; this causes a type error three lines later at `w % 10`. In INTERTRANS, the intermediate translation to C++ explicitly annotates the local variable `w` as int, and thus when the C++ is translated to Go, the LLM knows to wrap `w` with `int()` type conversion before performing the remainder operation.

D. RQ4: How do semantic errors propagate in INTERTRANS*?*

Approach: INTERTRANS leverages intermediate translations through intermediate languages, offering new opportunities to enhance code translation accuracy. This approach achieves SOTA performance as demonstrated in Section IV-A. However, using intermediate translations also introduces the potential risk of error propagation. This RQ aims to address this concern.

Manually verifying consecutive translations to determine whether they exhibit the same error would require expertise in all six programming languages we considered, making the process time-consuming, error-prone, and unscalable. Therefore, we propose a qualitative method to capture "error propagation" by observing at least two consecutive runs of the same test case (assertion) failing along a translation path. For example, consider a translation problem with a test suite containing two test cases. Each test case is evaluated four times for a translation path with four translation edges, resulting in two test result sequences: Sequence 1: [P, F, F, P] and Sequence 2: [P, F, P, F]. In Sequence 1, the failure (F) propagates between the second and third intermediate translations. In Sequence 2, no failures propagate. This approach establishes an upper bound on the frequency of semantic error propagation, as consecutive failures of the same test case do not always imply the propagation of the same semantic error. Nonetheless, when a semantic error is propagated, it will be reflected in repeated failures of the same test case across consecutive translations. It is important to note that compilation errors (e.g., syntax issues, API misuse, or missing dependencies) cannot be analyzed using this automated approach, as such errors prevent the execution of all test cases, rendering the test result sequences unsuitable for this analysis. Consequently, our focus in RQ4 is limited to semantic errors.

Following the above design, we further evaluate intermediate translations generated by INTERTRANS using the same test cases applied to the target program. This allows us to create the test result sequences described earlier. We then analyzed these evaluation results for translation paths generated by INTERTRANS with three LLMs on the CodeNet dataset, which provides fine-grained test suites where each test case can be executed independently. In contrast, the HumanEval-X and TransCoder datasets do not support independent validation of test cases, as a single failure causes the program to terminate. Obtaining similar fine-grained feedback for these datasets would require modifying their test suites, which would introduce significant additional implementation and verification work beyond the scope of this study. Thus, we exclude these two datasets in this experiment.

We performed several data preprocessing steps to analyze error propagation in the test result sequences. First, we filtered out direct translation paths from the paths generated by INTERTRANS on CodeNet experiments for the three models, as direct translation paths do not involve intermediate translations and, therefore, cannot exhibit error propagation. Next, we created a balanced subset of paths by randomly sampling an equal number of successful and failed translation paths. This design ensures a fair analysis of error propagation, which can occur in both successful and failed translation paths. The resulting subset includes 4,832 paths, comprising 2,416 successful and 2,416 failed paths, and is representative at a 95% confidence level. For each of the 4,832 paths, we constructed 14,496 test result sequences (3 test cases per suite in CodeNet × 4,832

paths). Since we focus on semantic error propagation rather than compilation errors, we excluded sequences containing at least one compilation error. In the end, we removed 9,726 sequences with compilation errors, resulting in a final subset of 4,770 test result sequences used for further analysis.

Results: Analysis of the 4,770 test result sequences revealed two new findings regarding error propagation in INTERTRANS. First, correct semantics propagate more often, as 76% of the test case sequences (3,630) exhibited no failures throughout their paths. This demonstrates that, in the majority of cases, intermediate translations do not introduce new errors, leaving the final translation unaffected by error propagation. Second, the impact of semantic error propagation on final translation failure is low. Among the remaining 24% of sequences (1,140) where test failures occurred, only 31% (354, or 7.4% of total sequences) showed error propagation between consecutive translations, and just 27% (306, or 6.4% of total sequences) had failures that propagated to the final output. These results indicate that semantic error propagation occurs infrequently in INTERTRANS. When present, it typically manifests near the end of the translation sequence.

V. DISCUSSION

Design Scope of INTERTRANS: Similar to existing studies on LLM-based code translation [30], [38], INTERTRANS focuses on translation between PLs known to the LLM. As such, our results do not apply to scenarios where the source or target PLs are outside the LLM's training data. However, we believe this limitation is mitigated by the fact that most recent LLMs are trained on datasets encompassing hundreds of PLs. For example, StarCoder2, a base LLM used in this work, covers over 600 PLs [26]. Furthermore, INTERTRANS requires no fine-tuning, lowering the entry barrier and enabling practitioners to leverage commodity LLMs effectively.

Computational Cost of INTERTRANS: Despite the performance boost brought by intermediate translation, INTERTRANS comes with overhead computational cost mainly on the inference steps involved in intermediate translation. To mitigate this issue, INTERTRANS is designed to allow flexibility in computational budgets through adjustable parameters, such as the maximum depth (maxDepth) of the Tree of Code Translation (ToCT) and the number of intermediate languages. Our experiments show that even with a shallow ToCT configuration (maxDepth=2), INTERTRANS achieves significant accuracy improvements over Direct Translation (ref. Figure 2).

Additionally, to demonstrate the performance of INTERTRANS under computational cost limit, we conducted an additional experiment comparing INTERTRANS with the baseline, i.e., Direct Translation, under the same inference step budget of K, denoted as Clipped@K. Specifically, we evaluated Clipped@10. Note that, for Direct Translation, the reported Clipped@K value is equivalent to CA@10 as each inference step results in one evaluation candidate. In contrast, INTERTRANS is constrained to terminate after performing K inference steps (some edges are cached, so they do not count), regardless of whether all translation paths are fully explored.

As observed in Table IV, despite this inference budget constraint, INTERTRANS still significantly outperforms Direct Translation. For example, on the HumanEval-X benchmark, INTERTRANS with StarCoder2 achieves a Clipped@10 accuracy of 91.8%, which is still substantially higher than Direct Translation, which achieves an accuracy of 73.2% under the same inference budget.

Beyond the optimizations already implemented, INTERTRANS can benefit from external cost-saving techniques, such as using quantized models or hardware acceleration. Furthermore, algorithmic enhancements like branch prediction or adaptive path pruning present promising opportunities to reduce computational overhead, while maintaining performance.

VI. THREATS TO VALIDITY

Internal Validity: We performed the translation only once for each translation problem, using a fixed random seed for study LLMs when reporting the performance of INTERTRANS. This design reduces the risk of selecting a favorable seed across all nine experiments. However, altering this seed could affect the reported performance. Nonetheless, this does not affect the comparison between INTERTRANS with direct translation (as shown in Table 2, where depth=1 represents direct translation under identical conditions), or the empirical analysis of varying parameters, which are our main goals. Furthermore, we employed a single prompt template for each dataset; changing this template might also alter the reported performance across all models. However, this does not affect comparison results, as we used the same prompt for all models, including direct translation and INTERTRANS. To mitigate the effects of LLMs' sensitivity to prompt templates, we adhered to best practices from the literature. The temperature and top-p, top-k values were set consistently across all LLMs, following established literature. While these may not be the optimal parameters for a specific model, our primary objective is to demonstrate the improvement of INTERTRANS over direct translation, regardless of the LLMs used.

The observed discrepancies between Figure 2 and Table II (an average of 2.8%), particularly in Direct Translation CA@1 results, stem from the non-deterministic nature of the LLM used. Our experiments (on baseline Direct Translation) followed related work [30], [38], where evaluations were conducted once without fixed seed. As we did not fix random seeds during our baseline experiments, slight variations in results were expected. While this introduces a minor threat to validity, using Direct Translation CA@10 as the primary baseline mitigates this concern. By evaluating multiple candidates per translation, CA@10 accounts for inherent randomness and provides a more robust comparison. To further reduce the impact of randomness, we conducted experiments across multiple datasets and 30 translation pairs, ensuring broad coverage and reliable conclusions.

Another threat to internal validity arises from potential data leakage in LLMs, meaning there could be an overlap between the training data of the studied LLMs and the evaluation dataset used in this work. However, this issue would impact

TABLE IV: Comparison of CA@10 in Direct Translation and Clipped InterTrans.

Dataset	Source language	Total samples	CA@K (percentage)											
			Code Llama				Magicoder				StarCoder2			
			Direct CA@10	InterTrans Clipped@10	Abs. diff	Rel. diff.	Direct CA@10	InterTrans Clipped@10	Abs. diff.	Rel. diff.	Direct CA@10	InterTrans Clipped@10	Abs. diff.	Rel. diff.
CodeNet	All	1,050	34.6	**45.8**	11.2	32.5	49.0	**75.5**	26.5	54.0	41.0	**69.9**	28.9	70.3
HumanEval-X	All	1,050	71.3	**82.8**	11.4	16.0	66.9	**91.1**	24.3	36.3	51.0	**71.6**	20.7	40.6
TransCoder	All	2,826	74.9	**88.6**	13.7	18.3	59.5	**79.4**	19.9	33.4	73.2	**91.8**	18.5	25.3

all baseline models, not just INTERTRANS, ensuring that the relative performance comparisons between models in our study remain valid.

External Validity: Potential threats to external validity may arise from the selection of target PLs, LLMs, evaluation datasets, and compared approaches. To mitigate these threats, we selected six popular (ranked among the Top-15 in the TIOBE Index) PLs with varying levels of maturity, encompassing different programming paradigms. Additionally, these languages represent the complete set covered by the three benchmark datasets used in this work. The source-target PL pairs we considered include all those concerned in recent work on LLM-based code generation by Pan et al. [30] and Yang et al. [38]. For dataset selection, our evaluation set is sourced from three well-known benchmarks. Two of these benchmarks were used in the previously mentioned studies, and the third allows for a fair comparison with non-LLM-based models, such as the TransCoder family and GPT-3.5. We selected three popular and recent open-source code LLMs as the base for INTERTRANS. These models are multilingual and have demonstrated strong performance on code generation tasks.

Construct Validity: Similar to prior studies [30], [38], we only consider execution-based evaluation metric, i.e., CA. While execution-based metrics align better with our goal to investigate the capability of LLMs in generating translated code that is functionally equal to the source program, its reliability will be impacted by the effectiveness of output control and the quality of test cases. To mitigate these threats, we applied output control following the best practices suggested by Macedo et al. [27].

For the evaluation datasets, we used the complete, cleaned TransCoder dataset, allowing us to leverage the performance metrics reported by Yang et al. [38] for SOTA approaches, where each translation includes at least one test case. When sampling from the CodeNet dataset, we ensured that each translation problem included three test cases. This threshold is also used in literature [9] to balance computational cost and test suite coverage. The other two datasets provide more test cases. For instance, each translation problem in HumanEval-X contains an average of 7.7 test cases. However, not all translation failures may be captured even with these mitigation approaches.

VII. RELATED WORK

Automated code translation approaches generally fall into two categories: rule-based methods and data-driven learning-based methods. Rule-based automated code translation approaches [1], [2], [5], [7] utilize program analysis techniques and handcrafted rules to translate code between programming languages (PLs). A prominent example is C2Rust [2], which has gained significant attention with 3.8k stars on GitHub as of this writing. However, these tools often produce non-idiomatic translations and are expensive to develop [35]. Learning-based approaches aim to address these limitations by leveraging large-scale data. These methods can be supervised, using parallel code translation pairs, or unsupervised, learning from open-source code. Techniques in this category have evolved significantly, starting with statistical learning techniques [19], [28], [29], progressing to neural network approaches [12], and more recently, to pre-trained model-based [18], [23], [30], [34], [35] and LLM-based methods [30], [38]. Our proposed INTERTRANS is also a LLM-based code translation approach. It is unique among existing methods as it is the first study to explore the potential of leveraging intermediate PLs for code translation.

VIII. CONCLUSION

This work explores the potential of leveraging the multilingual capabilities of LLMs to enhance automated code translation through transitive intermediate translations. We propose INTERTRANS, a novel approach that utilizes a planning algorithm (ToCT) to generate candidate translation paths, which are then evaluated sequentially. Through extensive empirical studies on three benchmarks, our results demonstrate the promise of INTERTRANS with an **absolute improvement boosting of 18.3% to 43.3%** in Computation Accuracy (CA) over direct translation with ten attempts. With only a readily available open-source LLM, e.g., Magicoder, INTERTRANS achieved an average CA of 87.3%-95.4% on three benchmark datasets. INTERTRANS not only enhances translation accuracy, but also provides a new direction for future research in leveraging and interpreting multilingual LLMs for diverse coding tasks. Future work could explore the ordering of intermediate PLs and develop predictive models to identify successful translation paths for specific PL pairs.

ACKNOWLEDGMENTS

We appreciate the support from Natural Sciences and Engineering Research Council of Canada (NSERC)'s Discovery (RGPIN-2019-05071) and Alliance program (ALLRP 597628-24). Additionally, we thank the Vector Institute for its offering of the Vector Scholarship in Artificial Intelligence, which was awarded to the first author. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Huawei and/or its subsidiaries and affiliates.

REFERENCES

[1] C to go translator, 2024. https://github.com/gotranspile/cxgo. Accessed: 2024.

[2] C2rust, 2024. https://github.com/immunant/c2rust. Accessed: 2024.

[3] Codellama, 2024. https://huggingface.co/codellama/CodeLlama-13b-hf/. Accessed: 2024.

[4] Geeksforgeeks, 2024. https://www.geeksforgeeks.org/. Accessed: 2024.

[5] Java 2 csharp translator for eclipse, 2024. https://sourceforge.net/projects/j2cstranslator/. Accessed: 2024.

[6] Magicoder-s-ds, 2024. https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B/. Accessed: 2024.

[7] Sharpen - automated java to c# coversion, 2024. https://github.com/mono/sharpen. Accessed: 2024.

[8] Starcoder2-15b, 2024. https://huggingface.co/bigcode/starcoder2-15b/. Accessed: 2024.

[9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[10] BigCode Project. Bigcode model license agreement, 2024. https://huggingface.co/spaces/bigcode/bigcode-model-license-agreement. Accessed: 2024.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[12] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31, 2018.

[13] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[14] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. Unprecedented code change automation: The fusion of llms and transformation by example. *Proceedings of the ACM on Software Engineering*, 1(FSE):631–653, 2024.

[15] Angela Fan, Shruti Bhosale, Holger Schwenk, Zhiyi Ma, Ahmed El-Kishky, Siddharth Goyal, Mandeep Baines, Onur Celebi, Guillaume Wenzek, Vishrav Chaudhary, et al. Beyond english-centric multilingual machine translation. *Journal of Machine Learning Research*, 22(107):1–48, 2021.

[16] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[17] gRPC. grpc: A high-performance, open-source universal rpc framework, 2024. https://grpc.io. Accessed: 2024.

[18] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 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, 2023.

[19] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184, 2014.

[20] Yunsu Kim, Petre Petrov, Pavel Petrushkov, Shahram Khadivi, and Hermann Ney. Pivot-based transfer learning for neural machine translation between non-english languages. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 866–876, 2019.

[21] David Kolovratnık, Natalia Klyueva, and Ondrej Bojar. Statistical machine translation between related and unrelated languages. In *Proceedings of the Conference on Theory and Practice of Information Technologies (ITAT-09), Kralova Studna, Slovakia, September*, 2009.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[23] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems*, 34:14967–14979, 2021.

[24] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[25] Jiaxing Liu, Chaofeng Sha, and Xin Peng. An empirical study of parameter-efficient fine-tuning methods for pre-trained code models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 397–408. IEEE, 2023.

[26] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[27] Marcos Macedo, Yuan Tian, Filipe Cogo, and Bram Adams. 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, 2024.

[28] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654, 2013.

[29] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 544–547, 2014.

[30] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 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, 2024.

[31] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[32] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[33] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

[34] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.

[35] Marc Szafraniec, Baptiste Roziere, Hugh James Leather, Patrick Labatut, Francois Charton, and Gabriel Synnaeve. Code Translation with Compiler Representations. In *The Eleventh International Conference on Learning Representations*.

[36] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source Code Is All You Need, December 2023. arXiv:2312.02120 [cs].

[37] Hua Wu and Haifeng Wang. Pivot language approach for phrase-based statistical machine translation. *Machine Translation*, 21:165–181, 2007.

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

[39] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, 2023.