INTERTRANS: LEVERAGING TRANSITIVE INTERMEDIATE TRANSLATIONS TO ENHANCE LLM-BASED CODE TRANSLATION

Marcos Macedo[1,†] **Yuan Tian**[1] **Pengyu Nie**[2] **Filipe R. Cogo**[3] **Bram Adams**[1]
[1]School of Computing, Queen's University
[2]Cheriton School of Computer Science, University of Waterloo
[3]Centre for Software Excellence, Huawei Canada

ABSTRACT

Code translation, the process of converting code between programming languages (PLs), is essential for modernizing legacy systems and ensuring cross-platform compatibility. Despite recent advancements, automated code translation, including methods based on large language models (LLMs), still encounters challenges due to syntactic and semantic mismatches between PLs. In this paper, we introduce INTERTRANS, an LLM-based automated code translation approach that, unlike existing methods, leverages intermediate translations to bridge the syntactic and semantic gaps between source and target PLs. INTERTRANS uses 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 involving six PLs. Results demonstrate an *absolute improvement* of 18.3% to 43.3% in Computation Accuracy (CA) for INTERTRANS compared to Direct Translation with 10 attempts. The best-performing variant of INTERTRANS (using the Magicoder LLM) achieved an average CA of 87.3%-95.4% across three benchmarks.

1 INTRODUCTION

Automatically translating source code between different programming languages (PLs) can significantly reduce the time and effort required for software development teams. In the literature, researchers have proposed various automated code translation methods. Data-driven learning-based approaches (Roziere et al., 2021; Szafraniec et al.) have shown impressive improvements over traditional rule-based methods (c2r, 2024; cxg, 2024; sha, 2024).

Large language models (LLMs) have demonstrated strong performance across various software engineering tasks (Fan et al., 2023). However, recent studies reveal that LLM-based automated code translation, particularly with open-source models, remains far from production-ready, achieving correct translations in only 2.1% to 47.3% of cases (Pan et al., 2024; Yang et al., 2024). 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 (Pan et al., 2024). 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 (Kolovratnık et al., 2009). To improve translations for challenging language pairs, a common strategy is to use parallel corpora with a pivot (bridge) language (Kim et al., 2019). In fact, traditional statistical machine translation between

[†]Correspondence to: marcos.macedo@queensu.ca

non-English languages, such as French to German, often involves pivoting through English (Wu & Wang, 2007). This approach remains effective with the rise of multilingual neural machine translation models. For instance, in a recent work by Meta (Fan et al., 2021), 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.

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 (Roziere et al., 2023), Magicoder (Wei et al., 2023), and StarCoder2 (Lozhkov et al., 2024)) on 4,926 *translation problems* sourced from three datasets, i.e., CodeNet (Puri et al., 2021), HumanEval-X (Zheng et al., 2023), and TransCoder (Roziere et al., 2020). 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/tree/paper`.

2 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

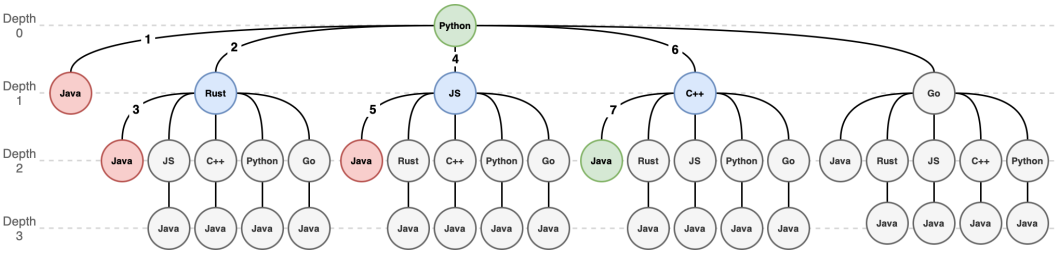

Figure 1: Running example of INTERTRANS with maxDepth=3 for translating Python to Java, showing a successful translation through C++ after exploring various translation paths. Red nodes represent unsuccessful translations, blue nodes indicate explored translations, green nodes denote successful translations,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

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., $TP = \{P_t | P_t \in P_{P_s,L_t} \wedge s \neq t\}$, where P_{P_s,L_t} is the set of programs written in L_t that represent translation candidates for P_s.

Given a translation problem aimed at converting a source program P_s into a target language L_t, 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.

2.1 STAGE 1: GENERATING TREE OF CODE TRANSLATIONS (TOCT)

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

Initialize an empty list $paths$
Initialize a queue Q
Enqueue $([L_s], 0)$ into Q
while Q is not empty **do**
  $(currentPath, currentDepth) \leftarrow$ Dequeue Q
  $currentLang \leftarrow$ last element of $currentPath$
  if $currentLang = targetLang$ **then**
    Append $currentPath$ to $paths$
  else if $currentDepth < maxDepth$ **then**
    for $lang \in \{L_t\} \cup L$ **do**
      if $lang \neq currentLang$ **then**
        $newPath \leftarrow currentPath + [lang]$
        Enqueue $(newPath, currentDepth + 1)$ into Q
return $paths$

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). For example, in Figure 1 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.

2.2 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.

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

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

3 Experimental Setup

3.1 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 (Roziere et al., 2020) was created by manually collecting coding problems and solutions written in C++, Java, and Python from GeeksforGeeks 2024. Recently, Yang et al. 2024 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 (Zheng et al., 2023) extends the python-only code generation evaluation dataset HumanEval (Chen et al., 2021) 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 (Puri et al., 2021) 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. 2024 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 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.

3.2 SELECTED LARGE LANGUAGE MODELS

Magicoder (Wei et al., 2023): 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 2024.

StarCoder2 (Lozhkov et al., 2024): An open-source collection of LLMs offered by the BigCode project (BigCode Project, 2024). StarCoder2 has instruction-tuned versions ranging from 1B to 34B parameters. We use the StarCoder2-15B variant 2024.

CodeLlama (Roziere et al., 2023): 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 2024.

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

3.3 EVALUATION METRICS

Similar to recent studies on LLM-based code translation (Pan et al., 2024; Yang et al., 2024), we adopt execution-based evaluation metrics, i.e., Computational Accuracy (CA) (Roziere et al., 2020). 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.

3.4 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.

Non-LLM SOTA approaches: TransCoder (Roziere et al., 2020) is an unsupervised model pretrained with cross-lingual language modeling, denoising auto-encoding, and back-translation, leveraging a vast amount of monolingual samples. TransCoder-IR (Szafraniec et al.), 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 (Roziere et al., 2021) 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.

Table 1: 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)	InterTrans	Abs. diff.	Rel. diff.	Direct (CA@1)	Direct (CA@10)	InterTrans	Abs. diff.	Rel. diff.	Direct (CA@1)	Direct (CA@10)	InterTrans	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

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. 2024. 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.

3.5 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 2024. The InterTrans engine utilizes vLLM (Kwon et al., 2023) as the inference engine. 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 (Dilhara et al., 2024).

4 Results and Analysis

4.1 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: Table 1 presents the comparison of InterTrans with direct translation (CA@1 and CA@10) across the three datasets, for the three base LLMs. We calculated both absolute and relative differences with CA@10, as the latter serves as a stronger direct translation baseline. As shown in Table 1, InterTrans consistently surpasses direct translation (CA@1 and CA@10) across all three

Table 2: CA performance of INTERTRANS and other baselines on TransCoder data set. We adopt the numbers of baseline performance from Yang et al. 2024. 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**

datasets and all studied LLMs. It achieves an absolute improvement of 18.3% to 43.3% compared to direct CA@10.

Table 2 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 (93.8%). The second best performance (87.9%) is achieved by UniTrans with GPT-3.5. All the LLM-based approaches considered in Table 2 perform consistently better than the TransCoder models, further showcasing the promising potential of LLMs in automated code translation.

4.2 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: We 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.

4.3 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:

- **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?

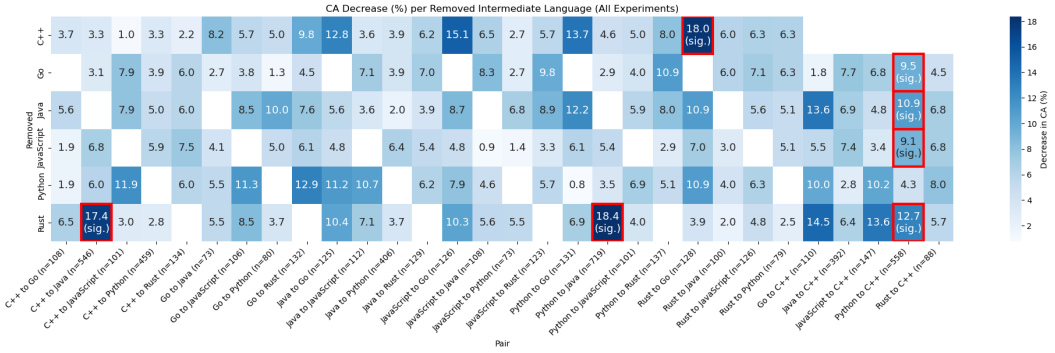

Figure 2: HeatMap showing the mean absolute decrease in CA (%) when removing a programming language from the intermediates used in our approach, compared to not removing any PL (across all datasets and models). Framed cells annotated with "(sig.)" indicate statistically significant results. The "n" value 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.

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 4.1, 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.

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 2 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 (experimental 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 observe 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 2 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 translations.

5 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. Nonetheless, this does not affect the comparison between INTERTRANS with direct translation, or the empirical analysis of varying parameters, which are our main goals. 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 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 The source-target PL pairs we considered include all those concerned in recent work on LLM-based code generation by Pan et al. 2024 and Yang et al. 2024. 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.

Construct Validity: Similar to prior studies (Pan et al., 2024; Yang et al., 2024), 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. 2024. For the evaluation datasets, we ensured that each translation problem included three test cases. This threshold is also used in literature (Austin et al., 2021) 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.

6 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 (c2r, 2024; cxg, 2024; sha, 2024; j2c, 2024) utilize program analysis techniques and handcrafted rules to translate code between programming languages (PLs). However, these tools often produce non-idiomatic translations and are expensive to develop (Szafraniec et al.). Learning-based approaches aim to address these limitations by leveraging large-scale data. Techniques in this category have evolved significantly, starting with statistical learning techniques (Nguyen et al., 2013; 2014; Karaivanov et al., 2014), progressing to neural network approaches (Chen et al., 2018), and more recently, to pre-trained model-based (Lachaux et al., 2021; Roziere et al., 2021; Szafraniec et al.; Jiao et al., 2023) and LLM-based methods (Pan et al., 2024; Yang et al., 2024). 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.

7 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.

8 ACKNOWLEDGEMENTS

We appreciate the support from the Natural Sciences and Engineering Research Council of Canada (NSERC) with funding reference number RGPIN-2019-05071. 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

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

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

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

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

Geeksforgeeks, 2024. https://www.geeksforgeeks.org/. Accessed: 2024.

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

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

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

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.

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

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.

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

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.

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.

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)*, pp. 31–53. IEEE, 2023.

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

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)*, pp. 1529–1541. IEEE, 2023.

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*, pp. 173–184, 2014.

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)*, pp. 866–876, 2019.

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.

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*, pp. 611–626, 2023.

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.

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.

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*, pp. 57–68, 2024.

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*, pp. 651–654, 2013.

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*, pp. 544–547, 2014.

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*, pp. 1–13, 2024.

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.

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.

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.

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.

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*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source Code Is All You Need, December 2023. URL `http://arxiv.org/abs/2312.02120`. arXiv:2312.02120 [cs].

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

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.

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.