Context-aware Intermediate Representations for Cross-language Code Translation

Aranya Venugopal[**]
avenugo3@ncsu.edu
NC State University
Raleigh, NC, USA

Mushtaq Ahmed Shaikh[†*]
mshaikh2@ncsu.edu
NC State University
Raleigh, NC, USA

Abstract

This paper explores leveraging low-level compiler Intermediate Representations (IR) to enhance code translation accuracy and usability. Traditional transpilers rely on syntactic rules, often producing unnatural translations, while Neural Machine Translation (NMT) techniques treat code as token sequences, neglecting semantic distinctions critical for high-quality translation. By augmenting NMT models with LLVM IR, we significantly improve translation accuracy across languages. We develop a novel sequence-to-sequence CodeT5 model for translating code between C and RUST through intermediate representations. The pipeline consists of three stages: C to IR, C IR to Rust IR, and Rust IR to Rust code, each optimized for token constraints and semantic fidelity. Key innovations include IR-based training objectives, IR decompilation strategies, and cross-language IR translation. We showcase the broader applicability of IR-enhanced translation, advancing the state of the art in unsupervised code translation and program synthesis.

CCS Concepts

• **Computing Methodologies** → Machine Translation; • **Software Engineering** → Translators; Compilers; • **Artifical Intelligence** → Neural Networks.

Keywords

Code Translation, LLVM IR, neural machine translation, cross-language code migration, intermediate representations, semantic alignment, neural decompilation

1 Introduction

Automatic code translation plays a vital role in modernizing legacy codebases for new frameworks or transitioning high-level, slower languages to low-level, more efficient ones. Traditional industry solutions, known as transpilers, rely on handcrafted syntactic rules to convert source code between programming languages. While these approaches can provide functional translations, they often fail to produce idiomatic, human-readable code, which poses a significant challenge for maintainability. Developers tasked with modifying or debugging such translations face unnecessary complexity, undermining the utility of these tools.

Recent advancements in Neural Machine Translation (NMT) have opened new avenues for code translation. These models, trained on vast datasets of human-readable code, produce more natural and idiomatic outputs compared to rule-based systems. However, NMT models often struggle to accurately capture the semantics of the source code, leading to errors in the translated program.

To address these limitations, we propose augmenting neural code translation models with Intermediate Representations (IR) generated by compiler toolchains. IR, a language-agnostic, low-level representation of code semantics, provides a robust foundation for bridging the syntactic and semantic gaps between programming languages. By integrating IR into the training pipeline, we aim to enhance the model's ability to produce semantically accurate and context-aware translations. Furthermore, we explore the use of IR as a pivot for translation, as well as its application in IR decompilation to recover source code from compiled representations.

[*]Both authors contributed equally to this research.
[†]Both authors contributed equally to this research.

2 Intermediate Representation in Compilers

Compilers translate programs written in computer language into executable code for a specific machine. Generally, compilers consist of a front-end taking source code as input and a back-end that produces machine binary code. The front-end tokenizes and parses the program. Once the program is parsed, the compiler then produces an Abstract Syntax Tree (AST) and translates the generated AST into Intermediate Representation (IR). The back-end converts the IR into machine-specific executable code.

Modern-day compilers such as LLVM produce Intermediate Representation that is generic across multiple languages. For instance, the IR for statically typed languages such as C, C++, Java, and Python all have similar dialects. The middle end of the compiler which translates the AST to its IR is independent of the front-end and back-end of the compiler. This results in an efficient structure of the compiler. The structure also allows creating new languages by rewriting the front end and back end of the existing compiler.

2.1 LLVM for C IR

Clang, the C front end of the LLVM compiler infrastructure, serves as a versatile and powerful tool for transforming high-level C code into LLVM Intermediate Representation (IR), a platform-independent format that supports optimization and portability across diverse architectures. Designed with modularity and precision in mind, Clang offers detailed diagnostics, making it a valuable resource for both software development and academic research in compiler technology. To generate IR, Clang provides several flexible options. The `-emit-llvm` flag enables the production of IR in human-readable textual format (`.ll`) or compact binary format (`.bc`), which is ideal for subsequent compilation stages. The `-S` flag generates textual IR for debugging and analysis, while the `-c` option outputs bitcode suitable for storage or distribution. Advanced users can leverage the `-Xclang` flag to intercept the IR generation process, enabling custom transformations and integrations with analysis tools.

Clang also supports various optimization levels during IR generation, which can significantly impact the performance of the resulting program. These levels, controlled by flags such as `-O0`, `-O1`, `-O2`, `-O3`, and `-Ofast`, balance compilation speed, run-time performance and code size. For example, `-O0` disables optimizations, preserving the structure of the code for debugging, while `-O3` applies aggressive optimizations to maximize runtime efficiency. The `-Ofast` option further enables optimizations that may violate strict compliance with language standards, which favors performance. By combining its robust IR generation capabilities with customizable optimization levels, Clang seamlessly integrates with LLVM's back-end toolchain to support advanced functionalities like Just-In-Time (JIT) compilation, profiling, and program analysis. This versatility establishes Clang as an indispensable tool for exploring modern compiler architectures and advancing research in high-performance computing.

2.2 Rustc for Rust IR

Rustc, the compiler for the Rust programming language, is a toolchain designed to transform high-level Rust code into executable binaries, leveraging safety and performance as its core principles. Central to its functionality is the generation of an Intermediate Representation (IR) that enables optimizations and portability across architectures. Rustc utilizes LLVM as its back-end, converting Rust code into LLVM IR, which then undergoes further transformations to produce efficient machine code. This multi-stage compilation pipeline ensures a balance between high-level abstractions and low-level performance.

Rustc provides several options to check and manipulate the IR during compilation. The `–emit` flag is particularly versatile, allowing users to control the output of the compilation process. For example, `–emit=llvm-ir` generates LLVM IR in a textual format, suitable for debugging and optimization analysis, while `–emit=llvm-bc` produces LLVM bitcode, a compact binary representation useful for advanced workflows or deferred compilation. For debugging purposes, the `–emit=mir` option exposes the midlevel intermediate representation (MIR), a Rust-specific IR used internally for borrow checking and type analysis before being lowered to the LLVM IR.

Rustc also supports various optimization levels, controlled by the `-C opt-level` flag, which can significantly affect the performance and size of the generated code. The optimization levels range from `0` (no optimizations, favoring fast compilation) to `3` (maximum

optimization for runtime performance). Additionally, the `z` level prioritizes binary size over execution speed, while the `-C codegen-units` flag allows parallel code generation for faster compilation at the expense of some optimizations.

3 Training Objective

Unsupervised machine translation involves learning multilingual sequence embeddings and generating sequences in a target language from these embeddings. This section outlines the objective functions employed for these tasks. In Section 3.1, we describe the three fundamental objectives used by TransCoder, which serves as our baseline NMT system. In Section 3.2, we describe our new functions that leverage LLVM IRs to improve the multilingual representation of source code, and the performance of our translation models. In Section 3.3, we describe the IR decompilation which consists of recovering source code corresponding to a given IR. At inference, the model is only provided with the source code and the IR is not needed.

3.1 Common Objective

TransCoder (Roziere et al., 2020) learns to translate between programming languages by leveraging three unsupervised objectives developed for natural language:

Masked Language Modeling (MLM) trains an encoder to predict randomly masked inputs. It is commonly used to pre-train embeddings for natural and programming languages. MLM allows the model to learn the syntax and semantics of programs. Alternative objectives have been proposed for programming languages. We do not use them here, as MLM remains effective and easy to use on a wide range of programming languages. Denoting $\text{mask}(x)$ the masked version of the code sentence x, and $\text{enc}(t)$ the encoder output, MLM uses the following loss:

$$L_{\text{MLM}} = L_{\text{CE}}(\text{enc}(\text{mask}(x)), x). \tag{1}$$

Denoising Auto Encoding (AE) trains a sequence to sequence (seq2seq) model to retrieve an original sequence from a corrupted version. Corruption is done by masking spans of tokens randomly sampled from a Poisson distribution, as well as removing and shuffling tokens. It uses the following loss ($\text{noise}(x)$ denotes the corrupted version of x):

$$L_{\text{AE}} = L_{\text{MT}}(\text{noise}(x), x). \tag{2}$$

Back-Translation (BT) uses the model to generate a noisy translation of the input sentence, and then trains the model to recover the original input from the translation. It is a simple yet powerful objective for unsupervised machine translation. In practice, it is a required loss to get competitive performance, so it is a staple of all our experiments. Formally, we use the model to translate sequence x into $\hat{y}$ and train the model to reverse the translation process, using the loss:

$$L_{\text{BT}} = L_{\text{MT}}(\hat{y}, x) \tag{3}$$

3.2 IR For Code Representation

Intermediate representations (IR) provide additional information about the code to be translated. We add IR along with the their respective code in the dataset, and use the training objective functions described in Section 3.1, to train the model on Code-IR pairs.

Translation Language Modeling (TLM) works on generating common representations for parallel sentences in different languages. Like the masked language modeling (MLM) objective, it trains an encoder to predict random masked inputs. However, TLM is trained on pairs of parallel sentences, concatenated together and separated by a special token. Here, we concatenate functions in their source language and their corresponding IR, using the source code and IR language embeddings, and train the encoder to predict randomly masked tokens. This allows the model to learn correspondences between the source and the IR. The corresponding loss is ($\oplus$ denotes concatenation):

$$L_{\text{TLM}} = L_{\text{CE}}(\text{mask}(x \oplus z(x)), x \oplus z(x)) \tag{4}$$

Translation Auto-Encoding (TAE) works on transposing the TLM objective into a denoising auto-encoder. The source code and corresponding IR are corrupted and masked, and then concatenated into one sequence (using the language embeddings for code and IR, as previously). TAE is then tasked to recover the original, using the following loss:

$$L_{\text{TAE}} = L_{\text{MT}}(\text{noise}(x) \oplus \text{noise}(z(x)), x \oplus z(x)) \tag{5}$$

IR Generation (MT) trains the model to translate the source code into the corresponding IR. This allows

the encoder to learn source code representations from the semantics of the IR. The loss is:

$$L_{\text{IRGen}} = L_{\text{MT}}(x, z(x)) \tag{6}$$

These three objectives need both the source code and the corresponding IR. However, only a fraction of the functions and files in our dataset could be compiled.

3.3 Cross-Language IR Translation

We propose a novel Intermediate Representation (IR) Translation Model that addresses the challenge of cross-language code translation at the IR level. Unlike previous approaches that focus on source code or high-level translation, our model specifically targets the translation of Intermediate Representations, providing a fine-grained mechanism for translating between programming language IRs. Specifically, we develop a sequence-to-sequence model trained on C IR to Rust IR pairs, capable of transforming computational semantics across language boundaries. The model learns to map the structural and semantic nuances of C's IR to an equivalent Rust IR representation. Formally, given a source IR in C, x_C, our translation function τ aims to generate a target IR in Rust, y_R, by minimizing the following translation loss:

$$L_{\text{IRTrans}} = L_{\text{MT}}(x_C, y_R) \tag{7}$$

This approach enables precise inter-language IR conversion, facilitating more accurate and semantics-preserving code translation at an intermediate representation level.

3.4 IR Decompilation

We introduce a novel approach to Intermediate Representation (IR) Decompilation, a critical task in program understanding and reverse engineering. Our method focuses on reconstructing source code from its corresponding Intermediate Representation, effectively reversing the compilation process. By training a sequence-to-sequence model on IR to source code pairs, we develop a sophisticated decompilation system capable of translating low-level IR back into high-level source code.

The decompilation process is formulated as a supervised machine translation task, where the objective is to learn a mapping from an IR representation $z(x)$ to its original source code x. We leverage pre-training techniques including Masked Language Modeling (MLM) and Auto-Encoding (AE) to enhance the model's understanding of code semantics. Formally, the decompilation loss is defined as:

$$L_{\text{Decomp}} = L_{\text{MT}}(z(x), x) \tag{8}$$

This approach enables precise reconstruction of source code from its intermediate representation, providing a powerful mechanism for program comprehension, legacy code analysis, and cross-language code understanding. By learning the intricate mappings between IR and source code, our model captures the semantic nuances required for accurate decompilation.

4 DATA

4.1 Training Data

ur training data was compiled from two primary sources: CodeNet and SLTrans. From CodeNet, a repository of 14 million competitive programming solutions spanning 55 languages, we carefully selected 783 programming problems that had both C and Rust submissions. For each of these problems, we extracted 5 submissions in C and 5 in Rust, focusing exclusively on solutions marked as "accepted".

SLTrans provided us with C code, from which we used RustC to generate equivalent unsafe Rust codes. From the 683K available C codes, we randomly sampled 10K codes. By leveraging these datasets, our models operate at the function level, which minimizes compilation failures due to missing dependencies while maintaining concise sequence lengths.

For each code pair selected from the CodeNet dataset, we randomly sampled 50 C and Rust equivalent codes and verified that the codes produce identical results by executing them on the problem statement. To a large extent, we manually selected only those code pairs which had semantic and syntactic equivalence. For instance, out of all the submissions for each problem statement, we checked whether the code has an equivalent number of functions and similar function signatures. This enabled us to create a parallel dataset with high-quality, semantically matched code pairs.

The accompanying Table 1 provides a detailed breakdown of our dataset compilation process, highlighting the systematic approach we took in selecting and processing the source code submissions.

Table 1: Dataset Compilation Overview

Dataset	Total Codes	Problems with Dual Submissions	Submissions Selected
CodeNet	14 Million	783 problems	5 C submissions + 5 Rust submissions per problem
SLTrans	683K C codes	N/A	10K C codes randomly selected

4.2 Generating Intermediate Representations

While the LLVM ecosystem provides robust infrastructure for generating intermediate representations (IR), not all front-end compilers produce LLVM IR seamlessly for every language. For this work, we leverage ‘clang‘ for C and ‘rustc‘ for Rust to generate highly optimized IRs using the ‘-Oz‘ flag, which prioritizes size minimization. This optimization facilitates uniformity by reducing extraneous IR variations.

To further standardize the IRs, we strip unnecessary metadata, such as debug information, comments, and header/footer attributes. Block names are canonicalized, and symbol names are demangled to enhance cross-language compatibility. For programs that fail to compile into IR due to unresolved dependencies or compilation errors, we exclude the associated functions from the dataset.

Despite these standardization efforts, the generated IRs for complete source files often exceed the token limit of sequence-to-sequence models. To address this limitation, we adopt a function-level granularity for IR generation. Each function in the C source code is matched to its equivalent function in Rust by analyzing their names and signatures. Functions without equivalent mappings are excluded. This rigorous mapping ensures a high-quality dataset of semantically aligned IRs in both C and Rust.

The dataset statistics are summarized in Table 2, illustrating the number of source code files and functions successfully mapped across both languages.

Table 2: Overview of Intermediate Representation (IR) Dataset Statistics

Language	Source Code Count	Total Functions
C	10,000	6,367
Rust	10,000	6,763

The concise, standardized function-level IRs provide a foundational dataset for analyzing and modeling equivalences between C and Rust. This granular approach not only mitigates issues related to token limits but also enhances the parallelism and semantic fidelity of the dataset.

The dataset statistics, summarized in Table 2, highlight the number of source code files and functions included for both C and Rust. Specifically, we compiled 10,000 source code files for each language, resulting in 6,367 functions for C and 6,763 functions for Rust. This function-level granularity ensures a well-balanced and comprehensive dataset for generating intermediate representations.

4.3 Evaluation

To evaluate the semantic and syntactic equivalence of the generated intermediate representations, we employed the Bilingual Evaluation Understudy (BLEU) metric. BLEU, a widely recognized metric in sequence-to-sequence tasks, measures the degree of correspondence between the generated sequences and reference sequences by analyzing overlapping n-grams. For our analysis, we computed BLEU scores between the IRs of equivalent functions in C and Rust. This allowed us to quantitatively assess the consistency and fidelity of the mappings between the two languages. A higher BLEU score reflects greater alignment, demonstrating the effectiveness of our approach in generating accurate and semantically consistent intermediate representations. This evaluation provides a robust foundation for validating the quality of our dataset and its suitability for downstream tasks such as neural translation and cross-language code synthesis.

5 RESULTS

5.1 Experimental Setup

We designed a three-stage translation pipeline to systematically transform code representations across different intermediate representations (IRs) and programming languages. Our approach leverages the CodeT5 encoder-decoder model [12] to facilitate precise code transformations.

5.2 Dataset Preparation

Our initial dataset comprised code functions from two primary languages, C and Rust. We started with 10,000 source code functions for each language, totaling 6,367 C functions and 6,763 Rust functions.

5.3 Model Architecture and Training

We employed the Salesforce CodeT5 base model [13], a transformer-based architecture specifically designed for code-related tasks. Our pipeline consists of three distinct translation models:

First Model (C to IR): This initial model translates C code functions to their equivalent intermediate representations. Given the ideal token limit of 1024, we preprocessed the dataset to retain functions within this constraint. This preprocessing reduced our function count to 6,364 for both C functions and their corresponding IRs.

Second Model (C IR to Rust IR): Recognizing the verbose nature of Rust IRs, we expanded the token limit to 2,048. We carefully curated the dataset to include only function pairs that adhered to this token limit. This process yielded 4,576 function pairs for C IR and their equivalent Rust IR.

Third Model (Rust IR to Rust Code): Similar to the second model, we maintained a 2,048 token limit, resulting in 4,576 function pairs of Rust IR and corresponding Rust code implementations.

The added special tokens in our Roberta tokenizer (<func>, </*func*>,<*ir*>,</ir>) enable more precise function and intermediate representation demarcation, enhancing the model's understanding of code structure.

5.4 Decompilation Results

The performance of our code translation pipeline can be evaluated through the BLEU scores at each stage.

Table 3: Function Counts Across Translation Models

Model Stage	Token Limit	Final Function Pairs
C to IR	1,024	6,364
C IR to Rust IR	2,048	4,576
Rust IR to Rust Code	2,048	4,576

Table 4 summarizes the BLEU scores for the different model components:

Table 4: BLEU Scores for Code Translation Models

Model	BLEU Score (100)
C to C IR	73.62
C IR to Rust IR	35.05
Rust IR to Rust	31.56

Additionally, we analyzed the validation loss of the models at each stage of the translation pipeline. Figures 1, 3, and 3 show the validation loss plots for the respective models. The validation loss plots provide

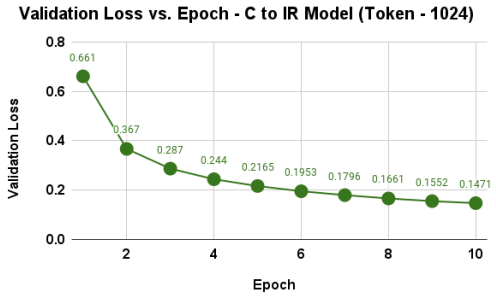

Figure 1: Validation Loss vs. Epoch - C to IR Model (Token Limit: 1024)

insights into the convergence and performance of the models at different stages of the translation pipeline. The results demonstrate the effectiveness of our approach in leveraging the CodeT5 model for precise code transformations.

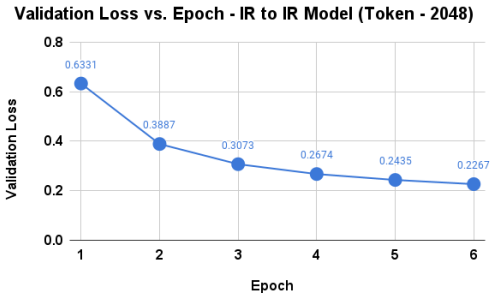

Figure 2: Validation Loss vs. Epoch - IR to IR Model (Token Limit: 2048)

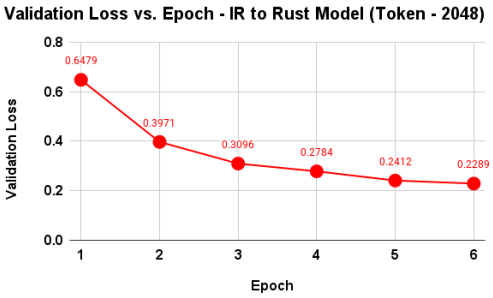

Figure 3: Validation Loss vs. Epoch - IR to Rust Model (Token Limit: 2048)

Instance of how the model at each stage translates the code.

Input C code:

```c
int main(){
    int a,b,c,s = 0,cun=0,week=0;
    scanf("%d%d%d\n",&a,&b,&c);
    while (1) {
        cun++;
        week++;
        s+=a;
        if(week==7){
            s+=b;
            week=0;
        }
        if(s>=c){
            printf("%d\n",cun);
            break;
        }
    }
    return 0;
}
```

Model-1 Output C-IR:

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 0, ptr %4, align 4
  store i32 0, ptr %5, align 4
  store i32 0, ptr %6, align 4
  store i32 0, ptr %7, align 4
  %9 = call i32 (ptr, ...) @scanf(ptr noundef @.str,
    ptr noundef %2, ptr noundef %3, ptr noundef %4)
  br label %10


10: ; preds = %0, %51
  %11 = load i32, ptr %2, align 4
  %12 = add nsw i32 %11, 1
  store i32 %12, ptr %6, align 4
  %13 = load i32, ptr %6, align 4
  %14 = add ns
```

Model-2 Output Rust-IR:

```
i32 @main(i32 %0, ptr %1) unnamed_addr #0 {
top:
  %_8.i = alloca [8 x i8], align 8
  %2 = sext i32 %0 to i64
  call void @llvm.lifetime.start.p0(i64 8, ptr
    nonnull %_8.i)
  store ptr @_ZN9train24484main17h5c5c5c5cE, ptr %_8.
    i, align 8
  ; call std::rt::lang_start_internal
  %3 = call noundef i64 @lang_start_internal(
        ptr noundef nonnull align 1 %_8.i,
        ptr noalias noundef nonnull readonly align 8
        dereferenceable(48) @vtable.0,
        i64 noundef %2,
        ptr noundef %1,
        i8 noundef 0) #6
  call void @llvm.lifetime.end.p0(i64 8, ptr nonnull
    %_8.i)
  %4 = trunc i64 %3 to i32
  ret i32 %4
}

define internal fastcc void @main_0() unnamed_addr #3
```

Table 5: Model Configuration Details

Attribute	Details
Model	Salesforce/codet5-base (CodeT5 variant, Transformer-based)
Tokenizer	Roberta Tokenizer (with added special tokens: <func>, </$func$>,<ir>,</ir>)
Batch Size	4 (with gradient accumulation steps of 4, effective batch size: 16)
Maximum Token Length	Variable 512–2048
Optimizer	AdamW
Learning Rate	2.00×10^{-5}
Weight Decay	0.01
Number of Epochs	6–10

Model 3 - Output Rust Code

```
fn main() {
let mut input = String::new();

io::stdin().read_line(&mut input).unwrap();
let values: Vec<i32> = input
.trim()
.split_whitespace()
.map(|x| x.parse().unwrap())
.collect();

let a = values[0];
let b = values[1];
let c = values[2];

let mut s = 0;
let mut cun = 0;
let mut week = 0;

loop {
cun += 1;
week += 1;
s += a;

if s >= c {
println!("{}", cun);
break;
}
}
}
```

6 DISCUSSIONS

In this work, we demonstrated the effectiveness of leveraging Intermediate Representations (IR) to enhance the accuracy and semantic fidelity of neural code translation models. IRs serve as a robust intermediary for aligning representations across languages, offering distinct advantages depending on how they are utilized—either as pivots for translation or as additional data to improve embeddings.

6.1 Pivot vs. Embedding

The traditional approach, as exemplified by TransCoder, relies solely on source code for training multilingual code representations and performing translation tasks. While this approach has shown promise, it often struggles with semantic misalignment between languages, particularly when source and target languages have significant structural differences. The pivot method, which focuses on translating between IRs rather than source code, mitigates these challenges by leveraging the uniformity of IRs. This method excels in scenarios with limited training data for the source language, as IRs can be generated using rule-based compilers. However, the requirement to compute IRs at test time introduces additional complexity, making it less practical for large-scale or time-sensitive applications.

In contrast, our proposed method integrates IRs into the training objectives to align source code and IR embeddings. This hybrid approach achieves the best of both worlds: the semantic precision of IR-based methods and the practicality of source-code-based approaches. By training the model to utilize both IR and source code, we achieve significant improvements in translation accuracy without the need to compute IRs at inference time. This makes our model as easy to use as TransCoder while providing superior performance.

6.2 Interpreted vs. Compiled Languages

Our approach focuses on compiled languages like C and Rust, which seamlessly integrate with LLVM IR.

However, the applicability of IR-based methods extends to interpreted languages as well. Many modern interpreters generate bytecode, which functions as an IR for the source language. While this paper emphasizes compiled languages, future work could explore leveraging bytecode as a form of IR for interpreted languages, expanding the range of supported languages and improving the universality of neural code translation models.

6.3 Inference Efficiency

An important advantage of our method is its simplicity at inference time. Despite the complexity of integrating IRs into the training pipeline, the resulting model generates translations directly from the source code. The IR-augmented objectives ensure that the model learns rich multilingual embeddings during training, enabling high-quality translations without requiring IR computation during inference. This design choice not only reduces overhead but also makes the model suitable for real-world deployment scenarios where efficiency is critical.

6.4 Limitations and Future Work

While this work demonstrates the effectiveness of leveraging LLVM IR to improve neural machine translation for code, certain limitations remain. One challenge is the token size of the generated IRs, as large functions often exceed the token limits of sequence-to-sequence models. This restricts the model's ability to handle large or complex source code. Additionally, current IRs are not fully optimized for cross-language compatibility; for example, the LLVM IR for Rust could be better aligned with that of C to improve semantic mapping. Another limitation lies in the scalability of the approach, as we primarily focused on function-level granularity for a few languages. Expanding the scope to handle full projects or larger datasets remains an open challenge.

To address these issues, future work could explore techniques to refine IRs and reduce their token sizes, ensuring compatibility with model constraints. For large C codes, modularization strategies could be developed to break code into smaller, token-friendly segments. Creating a custom IR generator for Rust that aligns more effectively with C IR could further enhance cross-language translation accuracy.

7 RELATED WORKS

Recent advancements in neural machine translation (NMT) for code have showcased the potential of leveraging large-scale datasets and sophisticated models to address the complexities of cross-language code translation. Some of the related works are presented below.

TransCoder (Roziere et al., 2020) introduced an unsupervised machine translation framework for code, leveraging large monolingual codebases to translate between C++, Python, and Java with remarkable accuracy.

Building on TransCoder, **DOBF** (Lachaux et al., 2021) improved pre-training methods for NMT of code, further enhancing the model's translation capabilities. DOBF introduced the deobfuscation of function names as a pre-training task, enabling the model to better understand the structure and semantics of code, thus improving translation quality across languages.

Large language models such as **Codex** (Chen et al., 2021) and PALM (Chowdhery et al., 2022) have further advanced the field by training on extensive datasets of code. These models have been employed for unsupervised code translation across multiple languages, leveraging their vast scale and pre-training to achieve significant performance improvements.

8 CONCLUSION

In this paper, we leverage LLVM Intermediate Representations (IR) to enhance neural machine translation for source code. By integrating IRs into the training objectives, we introduced semantically rich representations that significantly improve translation accuracy between C++ and Rust. Our approach resulted in a bleu score of 74.62 for C to C-IR translation, 35.05 for IR to IR translation, 31.56 for Rust-IR to Rust translation. Cross entropy losses turned out to be 14%, 22.6% and 22.8% for the respective models.

While this work focused on LLVM IR, the methodology is applicable to any language pairs sharing a common IR framework. The use of IRs aligns code representations with program semantics, improving both translation quality and cross-language representation. However, the current scale of source and target sequences limits our findings. Future work could explore generating IRs at project-level granularity, enabling more comprehensive training datasets and higher success rates for IR compilation.

References

[1] Roziere, B., Lachaux, M.-A., Chanussot, L., & Lample, G. (2020). TransCoder: Unsupervised Translation of Programming Languages. *arXiv*. Retrieved from https://arxiv.org/pdf/2006.03511

[2] Lu, S., Guo, D., Ren, Z., et al. (2021). CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv*. Retrieved from https://arxiv.org/pdf/2102.04664

[3] Pan, R., et al. (2023). Leveraging Pretrained Models to Improve Code Translation. *arXiv*. Retrieved from https://arxiv.org/pdf/2305.12138

[4] Puri, R., Kung, D., Janssen, G., et al. (2021). CodeNet. *arXiv*. Retrieved from https://arxiv.org/pdf/2105.12655

[5] Pan, R., et al. (2023). IRCoder: Intermediate Representations Make Language Models Robust Multilingual Code Generators. *arXiv*. Retrieved from https://arxiv.org/abs/2403.03894

[6] Yin, P., et al. (2022). Transcoder-enhanced: Multilingual code translation using pretrained models. *arXiv*. Retrieved from https://arxiv.org/abs/2212.10017

[7] Yang, X., et al. (2024). Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *arXiv*. Retrieved from https://arxiv.org/abs/2404.14646

[8] Ahmad, W., et al. (2022). PLBART: A Sequence-to-Sequence Model for Program and Language Understanding and Generation. *arXiv*. Retrieved from https://arxiv.org/abs/2206.05239

[9] Zhou, S., et al. (2023). Towards Robust Multilingual Code Translation with Pretrained Models. *arXiv*. Retrieved from https://arxiv.org/abs/2307.14991

[10] Lemieux, A., et al. (2023). Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. *arXiv*. Retrieved from https://arxiv.org/abs/2308.03109

[11] Jain, M., et al. (2024). Enhanced Cross-Lingual Code Translation with Dual-Representation Learning. *arXiv*. Retrieved from https://arxiv.org/abs/2409.10506

[12] Zhang, Y., et al. (2024). CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv*. Retrieved from https://arxiv.org/abs/2407.07472

[13] Yue Wang, Steven Hoi. "Codet5: The Code-Aware Encoder-Decoder Based Pre-Trained Programming Language Models." Salesforce, 30 Oct. 2024. https://www.salesforce.com/blog/codet5/

Received 20 February 2024; revised 12 March 2024; accepted 5 June 2024