Code to Code Conversion from Java to Python Using T5-Small

Anirudh Dambal[1], Harish Patil[1], Pavan Bhakta[1], and Anusha Adarakatti[1]

[1]School of Computer Science and Engineering, KLE Technological University, Hubballi, Karnataka, India, 580031

This paper introduces a new method of code-to-code translation from Java to Python using the T5-small transformer model, based on a custom-generated dataset. Previous works often depend on language-specific libraries or attributes; however, our dataset prohibits such inbuilt functionalities to promote the creation of fully translatable, algorithmically pure code. The dataset was generated with the help of the Gemini model, where sample text was initially translated into pseudocode, then coded into both Java and Python without language-specific shortcuts. This dataset was next used to train a T5-small model, which learned generalized patterns and structures for code translation. Evaluation results indicate the potential for using the T5-small model to translate Java code into Python with good accuracy in producing semantically equivalent and syntactically correct code. Our approach opens a promising direction for tool-assisted cross-language code translation, with significant implications for software development, maintenance, and education in multilingual coding environments.

Keywords: Code-to-code translation, T5-small transformer, Java to Python, custom dataset, Gemini model, pseudo code, syntactically correct code, cross-language translation.

1 Introduction

DSLs (Domain-specific languages) are widely accepted in having the capability to supply optimizations and abstractions that extend code clarity usability as well as overall performance for certain domains. Latest examples of these DSLs include allotted computing Spark, array processing NumPy, tensor processing TACO, and community packet processing P4. As new DSLs appear for many utility domain names and programming languages, developers are asked to rewrite existing code in order to include those languages in their existing workflows. This code rewriting technique is painful in itself, might also besides introducing bugs into the code and may also fail at preserving the semantics of the original code. This problem of code conversion and compiling from one (higher level) programming language into another (higher level language) is known as transpilation.

Among the recent innovations, Large Language Models Devlin et al. [2019], Brown et al. [2020] have been shown to be promising for taming complicated programming tasks, including code generation, repair, and testing. However, producing dependable code with formal correctness ensures with LLMs still remains challenging. Most papers on LLMs focus solely on code generation without correctness guarantees [Li et al., 2023, 2022, Rozière et al., 2024] or one on generating evidence annotations (including invariants) for given code Pei et al. [2023], Chakraborty et al. [2023]. Lastly, formal verification tools generally have their own specific languages, such as SMT-LIB, and Dafny, to state the encoding of verification problems and specifications. These languages are typically low-resource in the learning datasets of LLMs, which makes it challenging to apply those models directly for code generation in the formal verification languages.

Pre-trained language models with BERT (Devlin et al., 2019), GPT (Radford et al., 2019), and T5 (Raffel et al., 2020) have significantly improved general performance across a wide spectrum of natural language processing activities. Inspired with the help of their success, there are several recent attempts to develop such pre-schooling techniques for programming language (PL) (Svyatkovskiy et al., 2020; Kanade et al., 2020; Feng et al., 2020), holding promising results for code-related tasks.

In this paper, we train the transformer model T5 to transpile Java to Python, two completely different languages in terms of their syntax. We generate a dataset consisting of Java programs and their equivalent Python programs, and then train a T5 model to generate Python programs given Java program inputs.

2 Related work

Compilation using transformers has been done before (Jordi Armengol-Estapé et al., 2021).

Code-to-code models also exist. The model in Drissi et al. [2018] significantly extends on the tree-to-tree encoder/decoder framework advanced by Chen et al. (2018). The base model resorts to using a tree LSTM encoding, as shown by Tai et al. (2015), the source tree. In that process, the input program tree is initially binarized using the Left-Child Right-Sibling representation, then the tree is encoded by an LSTM starting at its leaves as presented by Chen et al. (2018).

[Lachaux et al., 2020] presents a new method of translating functions between programming languages based on monolingual source code. It shows that TransCoder actually captures and translates complex, language-specific patterns across languages. The work also details an experiment to establish the fact that a totally unsupervised technique can, indeed, beat commercial systems that rely on rule-based methods, as well as other programming expertise, to the extent. In addition to that, the authors devised and released a validation and test set containing 852 parallel functions in three languages, accompanied with unit tests, in order to test the quality of translation. Lastly, the code along with the pre-trained models will be released into the public domain.

[Tufano et. al., 2018] explores how NMT can be applied to generate bug-fixing patches by translating the "buggy" code into "fixed" code. To do that, the authors developed a procedure for training the NMT model on real-world examples of bug fixes mined from GitHub repositories. Using GitHub Archive, they harvested commits with bug-fix messages and derived "bug-fix pairs" representing corresponding buggy and fixed code fragments. After preparing these BFPs with Java parsers and abstracting them to a generalized representation, this was helpful in identifying the general patterns that are commonly used by developers when fixing bugs. The model proposed in this work is capable of learning transformations at the AST level with the support of tools such as GumTree for tracking edit actions between buggy and fixed versions. This approach enables the model to closely mimic developer-style fixes, thus achieving over 82% syntactic correctness in the generated patches across different types of bugs. Overall, this work adds a large dataset along with methodology that may further open avenues into automated bug fixing and patch generation. It thus provides a base for potentially robust tools in software maintenance. This paper is open access to any interested person who wishes to reproduce or build upon this approach.

[Katz et. al., 2019] deals with the problem of automatic decompilation, which converts low-level code back into a high-level human-readable programming language. Such a process is significantly important in security research since it's often used to identify vulnerabilities as well as to analyze malware. Traditional decompilers rely on techniques developed from rule-based methods by experts which detect control-flow structures and idioms in low-level representations and lift them to source code. However, propagating such approaches to support other languages or even the emergence of new language constructs comes at great cost. The authors propose a new approach that can learn direct decompilation from a compiler using neural machine translation (NMT). Given a compiler that translates from a source language S to a target language T, their method automatically trains a decompiler that can translate T back to S. The framework has been demonstrated on LLVM IR and x86 assembly code, with high success rates: above 97% on the LLVM IR and 88% on the x86 assembly across benchmark tests, which allows to prove that the method can indeed fulfill decompilation tasks end

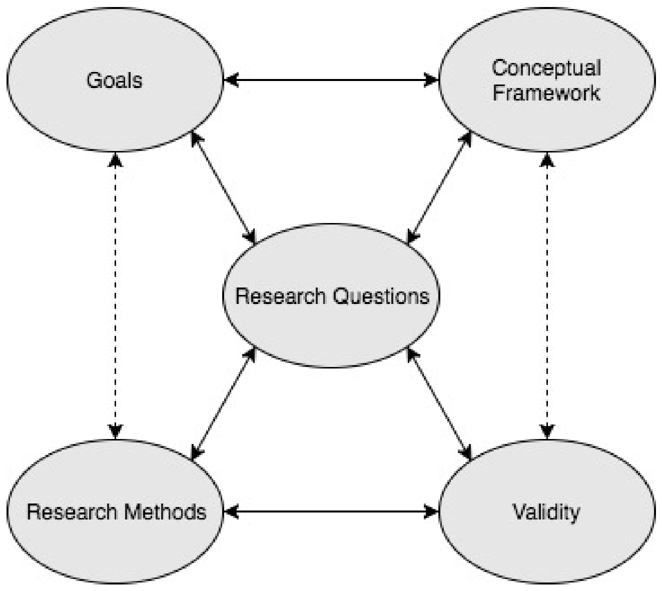

Figure 1: This is where the caption of the figure goes. Remember, figure captions go below the figure.

Table 1: Table caption. Captions for tables go above the table, unlike for figures.

This	is the	header row
1	2	this is a cell in the first row
3	4	this is a cell in the second row

3 Working with bibtex references

The DH Benelux Journal uses bibtex for references and citations. You can cite other work using bibtex keys Maxwell (2013). Your article should end with a section called 'References' that includes a link to your bibtex file that contains the bibliographic information of the work you cite (see below). Upon generating the print version with a LaTeX engine, references to the cited works are included.

4 Adding figures and tables

This section has two sub-sections.

4.1 Adding figures

If you include figures or tables, use reference keys (e.g. Figure 1), to create an unambiguous reference from the text to each figure and table, e.g. to Table 1.

4.2 Adding tables

References

Joseph A Maxwell. *Qualitative research design: An interactive approach, 3rd edition*. Sage publications, 2013.