CoTran: An LLM-based Code Translator using Reinforcement Learning with Feedback from Compiler and Symbolic Execution

Paper #2573

Abstract. In this paper, we present an LLM-based code translation method and an associated tool called CoTran, that translates whole-programs from one high-level programming language to another. Current LLM-based code translation methods lack a training approach to ensure that the translated code reliably compiles or bears substantial functional equivalence to the input code. In our work, we train an LLM via reinforcement learning, by modifying the fine-tuning process to incorporate compiler feedback, and symbolic execution (symexec)-based testing feedback to assess functional equivalence between the input and output programs. The idea is to guide an LLM-in-training, via compiler and symexec-based testing feedback, by letting it know how far it is from producing perfect translations. We conduct extensive experiments comparing CoTran with 14 other code translation tools, including human-written transpilers, LLM-based translation tools, and ChatGPT. Over a benchmark comprising more than $57{,}000$ code pairs in Java and Python, we demonstrate that CoTran outperforms the other tools on relevant metrics such as compilation accuracy (CompAcc) and functional equivalence accuracy (FEqAcc). For example, our tool achieves 48.68% FEqAcc, 76.98% CompAcc for Python-to-Java translation, whereas the nearest competing tool (PLBART-base) only gets 38.26% and 75.77% respectively. Also, built upon CodeT5, CoTran achieves $+12.94\%$, $+14.89\%$ improvement on FEqAcc and $+4.30\%$, $+8.14\%$ on CompAcc for Java-to-Python and Python-to-Java translation respectively.

1 Introduction

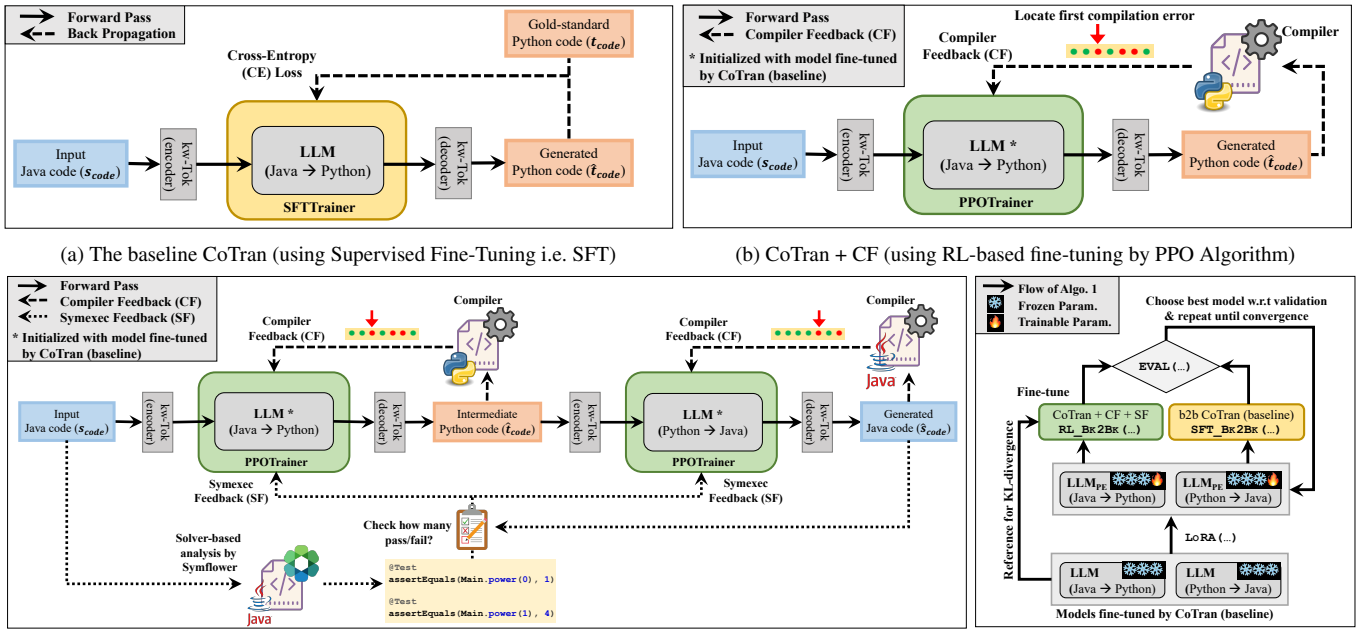
Automatic translation of code from one high-level language to another is an important area of software engineering research with uses in code migration and cross-platform interoperability [14, 25]. Traditional code translation tools, often called *transpilers* have a human-written set of rules and, thus, can be quite expensive to build.

To address the issue of cost related to transpilers, many researchers [9, 23] have recently proposed the use of Large Language Models (LLMs) [42] for code translation. The rise of LLMs is perhaps the most important development in AI in recent years, and even more remarkably they are being used successfully for many software engineering tasks such as code synthesis [38] and code completion [44]. In this context, many researchers have recently proposed LLMs for translating a single function from a source (e.g., Java) to a target language (e.g., Python) [34, 23, 1]. Unfortunately, a function-to-function translation is not sufficient (Refer Section 5.3: Finding 8) for whole-program translation tasks. Further, current LLM-based methods lack a proactive approach to ensure that the translated code reliably compiles or bears substantial functional equivalence to the input code.

To address these issues, while retaining the positive aspects of LLM-based code translation methods, we modify the fine-tuning mechanism of LLMs to incorporate compiler- and symbolic execution (symexec)-based testing feedback. The idea behind our method is to employ *corrective feedback loops* [12] between learning (the LLM) and reasoning (compiler & symexec), for which we have a two-fold strategy. First, if the output code produced by the LLM-in-training does not compile, then a *compiler feedback (CF)* is computed. Secondly, if the output and input codes are not equivalent (w.r.t. a symexec-generated test suite), then a suitable *symexec feedback (SF)* is computed. A reinforcement learning (RL) based optimization framework incorporates these feedback and fine-tunes the LLM. Our results on Java-to-Python (J2P) and Python-to-Java (P2J) translation indicate that these new feedback functions significantly improve the correctness and efficacy, relative to previous techniques. Another interesting feature is that we use two back-to-back LLMs, one for source-to-target ($S \rightarrow T$) and another for target-to-source ($T \rightarrow S$), that are trained together. The advantage of having two such complementary models is that it makes it much easier to automate functional equivalence checking between the input (of $S \rightarrow T$ model) and output (of $T \rightarrow S$ model), both in S.

Contributions.

- We describe a whole-program translation method (and tool CoTran) based on training LLM through interleaved Supervised Fine-Tuning (SFT)- and RL-based optimization, to incorporate CF and SF into the fine-tuning process. Currently, CoTran is configured to translate J2P and P2J. It can be effortlessly adapted to facilitate translation between other language pairs (S, T) given: *(i)* a dataset of equivalent code pairs in S and T, *(ii)* compilers for both languages and *(iii)* automatic test-case generation tool for either S or T – all of which are readily available for popular languages. A crucial insight of our work is that LLMs being trained for software engineering tasks (e.g., code translation/synthesis) can benefit greatly from feedback via compilers, program analysis, and test generation tools during fine-tuning. (Section 3.2)
- Our fine-grained compiler feedback (CF) guides an LLM during fine-tuning, helping it assess the proximity of generated translations to a perfectly compiling one. It is much more effective than a Boolean yes/no feedback e.g. for P2J, CoTran achieves +11.57% higher functional equivalence accuracy and +17.36% higher compilation accuracy compared to PPOCoder [37]. (Section 3.1)
- Our symexec feedback (SF) is used to fine-tune $S \rightarrow T \rightarrow S$ back-to-back LLMs. We generate unit-tests on the input code and check them on the output translated code, to provide feedback on their *inequivalence*. Our method is entirely agnostic to any specific test-case generation tool. As far as we know, no existing tools utilize symexec-based test generation for functional equivalence



(a) The baseline CoTran (using Supervised Fine-Tuning i.e. SFT)

(b) CoTran + CF (using RL-based fine-tuning by PPO Algorithm)

(c) CoTran + CF + SF (using RL-based fine-tuning on b2b LLMs, refer func. **`RL_Bk2Bk`** in Algo. 1)

(d) RL+SFT Interleaved Training Loop

Figure 1: **Training LLMs with Compiler Feedback (CF) and Symbolic Execution Feedback (SF):** (a) The CoTran (baseline) model is an LLM that uses the proposed keyword tokenizer (kw-Tok) and is trained through SFT to optimize cross-entropy loss. (b) CoTran with CF is trained using an RL-based PPOTrainer where the compiler provides feedback in the form of a reward. For P2J LLM, `javac` compiler is used. For J2P LLM, we use the `pylint` [39] static code analyzer as Python is an interpreted language. (c) To train CoTran with both CF and SF, we use back-to-back (b2b) J2P and P2J LLMs which have dedicated RL-based PPOTrainers. Solver-based analysis of the input Java code generates a set of JUnit tests, that are verified on the output Java code to compute SF. The reward for each LLM is a combination of the SF and respective CF. (d) We start with the forward (J2P) and backward (P2J) LLMs that are already fine-tuned by CoTran (baseline). All the parameters of these b2b LLMs are frozen. We continue fine-tuning a parameter-efficient (PE) version of the LLMs by interleaving RL and SFT, whereby only some minor additional parameters are trained through Low-Rank Adaptation (LoRA) [16] (Refer Algorithm 1).

checking and associated feedback function. (Section 3.1)

- We perform an extensive empirical evaluation and ablation study of CoTran against 11 state-of-the-art LLM-based translation tools and 3 human-written transpilers. A range of metrics is assessed incl., functional equivalence[1] accuracy (FEqAcc), compilation accuracy (CompAcc), BLEU [29], CodeBLEU [33] and the proposed average first error position ($\text{errPos}_{1\text{st}}$). CoTran and its variants outperform all other tools of similar size for both J2P and P2J. Compared to the nearest competing tool on FEqAcc, CoTran gets +9.62% (vs. PPOCoder [37]) in J2P and +10.42% (vs. PLBART-base [1]) in P2J. CoTran even outperforms ChatGPT [27] (a much larger model) on all the metrics in P2J and all but one metric (FEqAcc) in J2P. Also, our tool outperforms human-written transpilers e.g., for J2P, CoTran gets +53.43% in FEqAcc, +38.84% in CompAcc, and +38.47% in $\text{errPos}_{1\text{st}}$ (vs. TSS CodeConv [41]). (Section 5)
- We introduce a Keyword Tokenizer (kw-Tok) specialized for code translation which for J2P and P2J, accounts for +3.57%, +6.62% on FEqAcc and, +3.28%, +4.79% on CompAcc. (Section 3)
- We contribute a large, well-curated dataset AVATAR-TC (built on top of AVATAR [2]) having 57,000+ Java-Python code pairs with human-written test-cases (TCs). Compared to AVATAR, faulty codes are manually fixed and TCs (by problem-setters) are collected from coding platforms, enabling the calculation of translator FEqAcc. Based on what we know, this is the first large-scale dataset thoroughly testing code pairs with human-written TCs. (Section 4)

[1] We cannot guarantee that two programs are equivalent as it is generally an undecidable problem, and hard to solve in practice. By functional equivalence of two codes, we mean that they produce equivalent results w.r.t. a test suite.

2 Related Work

Rule-based Transpilers. Rule-based handcrafted transpilers are usually built using traditional compiler techniques and concepts such as parsing and abstract syntax trees. Examples of such transpilers include `java2python` [26] and `py2java` [10]. TSS code converter [41] is a commercial J2P transpiler. Overall, these transpilers vary by the intricacies and difficulty level of constructs (e.g. lambdas, anonymous inner class). Many such tools state a disclaimer with limitations that the translated codes should not be expected to compile and run readily.

Transformer and ML-based Code Translation Tools. The encoder-decoder architecture of Transformers revolutionized Natural Language (NL) translation, using contextualized representations of words [42]. This led to the evolution of advanced language models for code translation task, such as encoder-only CodeBERT [9] and decoder-only CodeGPT [23]. CodeT5 [45] is an encoder-decoder architecture that leverages code semantics from developer-assigned identifiers. Another tool is PLBART [1] which is a unified transformer trained through denoising autoencoding, and performs a range of tasks among NL and Programming Language (PL). Conversely, tree-to-tree [6] is an attention-based LSTM neural network.

LLM-based methods using compiler/unit testing. TransCoder-ST [35] is an unsupervised code translation tool that uses self-training and automated unit tests to verify source-target code equivalence. However, it uses unit tests to create a synthetic dataset of equivalent codes in two languages, instead of incorporating it for model improvement during training. Recently, a Boolean compiler feedback was used to train LLM for code generation in an RL-based scheme [44]. PPOCoder [37] is an RL-based code translation framework that ad-

ditionally uses CodeBLEU-inspired feedback in the LLM training process. While both the tools use the same Boolean compiler feedback that is true if the generated code compiles and false otherwise, our proposed compiler feedback function is fine-grained – it captures how far it is from a perfectly compilable code. Back-to-back (b2b) code translation although proposed earlier [34], is not leveraged for symexec-based functional equivalence checking.

3 The CoTran Method

The Code Translation Learning Problem. Let S denote a source language and T be a target language. The code translation learning problem is to learn a language translator $f_{ST} : S \rightarrow T$, such that the output of the translator is a T-program that is *syntactically correct* (as per the grammar of T) and is *functionally equivalent* to the input S-program (input-output equivalence w.r.t. a test suite).

Brief Overview of our Method. Please refer to Figure 1 for an architectural overview of the CoTran tool. In order to learn the language translator, we build off a sequence-to-sequence (seq2seq) attention-based transformer model consisting of an encoder and decoder stack [42]. For the remainder of the paper, we use the term Large Language Models (LLMs) to refer to such models. The proposed training pipeline consists of two steps. In the first step, we separately train two models $\text{LLM}_f : S \rightarrow T$ and $\text{LLM}_b : T \rightarrow S$ (the forward and backward baseline CoTran models resp.) by supervised fine-tuning (SFT) to optimize *cross-entropy loss*. In the second step, we jointly fine-tune these two models in a parameter-efficient way by training them on the back-to-back translation task i.e. $S \rightarrow T \rightarrow S$. We use an RL+SFT interleaved training loop that incorporates *compiler-* and *symexec-feedback*. Figure 1 illustrates the feedback mechanisms.

Keyword Tokenizer (kw-Tok). The code $\mathbf{s}_{code} \in S$ is first tokenized based on the syntax (grammar) of the corresponding language. Next, the code-specific tokens thus obtained, are converted to a sequence of token-ids using a RoBERTa tokenizer, which is modified for the PL translation task. To avoid splitting the PL keywords into multiple subtokens, we add all the keywords (e.g. `volatile`, `transitive` in Java; `elif`, `instanceof` in Python) and operators (e.g. >=, **=) of S and T to the tokenizer vocabulary. These are collected from the list of terminal symbols of these languages' formal grammar. Grammars-v4 [30] provides a collection of formal grammars for most common languages in use. We also add special tokens to the vocabulary e.g., `NEW_LINE`, `INDENT`, `DEDENT` for Python. We refer to this modified RoBERTa tokenizer as *Keyword Tokenizer (kw-Tok)*. This modification ensures that *kw-Tok* generates a *single token-id* for all the special tokens, keywords, and operators of languages S and T. Thus, given a training example $(\mathbf{s}_{code}, \mathbf{t}_{code}) \in$ dataset D, the tokenizer preprocesses these to a sequence of token ids viz., $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ and $\mathbf{t} = (t_1, t_2, \ldots, t_m)$. The LLM ($S \rightarrow T$) takes the source sequence $\mathbf{s}$ and generates a target sequence $\widehat{\mathbf{t}} = (\widehat{t}_1, \widehat{t}_2, \ldots, \widehat{t}_q)$.

3.1 Definitions: Cross-Entropy Loss, Compiler and Symbolic Execution Feedback

Please refer to Figure 1 for a pictorial representation of the compiler and symexec feedback. Let θ_f and θ_b denote the set of trainable parameters for LLM_f and LLM_b respectively.

Cross-Entropy (CE) Loss. As is typical [45, 42] in translation tasks, LLMs are trained to minimize the cross-entropy (CE) loss through SFT. For the forward LLM that generates $\widehat{\mathbf{t}}$ given a tokenized training instance $(\mathbf{s}, \mathbf{t})$, the CE loss is defined as:

$$\mathcal{L}_{CE}^{\theta_f}\left(\mathbf{t}, \widehat{\mathbf{t}}\right) = -\frac{1}{\ell}\sum_{i=1}^{\ell}\sum_{j=1}^{|V|}\mathbb{1}_{ij}\log P_{ij}^{\theta_f} \tag{1}$$

where, $\ell = \max\left(\left|\widehat{\mathbf{t}}\right|, |\mathbf{t}|\right)$ and V is the tokenizer vocabulary. $\mathbb{1}_{ij}$ is 1 *iff* the i^{th} token in reference translation $\mathbf{t}$ is the j^{th} word of V and, $P_{ij}^{\theta_f}$ is the probability that the i^{th} token in predicted translation $\widehat{\mathbf{t}}$ is the j^{th} word of V. The CE loss is, however, more suitable for machine translation of NLs than PLs. For NLs, the grammar is lenient, and an approximate translation is oftentimes good enough. But in PL translation where the grammar is strict and functional equivalence is paramount, the aim is to generate compilable codes in the target language and, to make sure that $\widehat{\mathbf{t}}$ is functionally equivalent to $\mathbf{s}$.

Compiler Feedback (CF). We use a compiler to assess the syntactic correctness of an LLM-generated translation and in turn, provide feedback to the LLM. This guides the model in determining how the generated translation fares relative to a perfectly compilable code. On this note, the position of the first token that raises a compilation error is an important cue. The closer it is to the end of a code, the closer the code is to a perfect compilation. Accordingly, for the forward LLM that generates $\widehat{\mathbf{t}}$ given a tokenized training instance $(\mathbf{s}, \mathbf{t})$, we formulate the feedback as:

$$\omega_{\text{compiler}}\left(\widehat{\mathbf{t}}\right) = \begin{cases} +2 & \text{, if } \widehat{\mathbf{t}} \text{ compiles} \\ \frac{f(\text{compiler}_T, \widehat{\mathbf{t}})}{\left|\widehat{\mathbf{t}}\right|+1} & \text{, otherwise} \end{cases} \tag{2}$$

where, the function $f(\cdot,\cdot)$ applies the T-language compiler on $\widehat{\mathbf{t}}$ and returns the token position $\left(\in \left[1, \left|\widehat{\mathbf{t}}\right|\right]\right)$ of the first syntax error. So, ω_{compiler} is +2 for a perfect compilation and goes on decreasing from +1 to 0 as the token position of the first syntax error is closer to the beginning of the code. However, the LLM should not game the system to generate $\widehat{\mathbf{t}}$ as a dummy code with no compilation error e.g. a Hello-World program. To penalize such cases we posit that $\widehat{\mathbf{t}}$ should also be close-by in length to $\mathbf{t}$. Accordingly, CF is the product of ω_{compiler} and the value of a $(0, 1]$ Gaussian distribution at $\left|\widehat{\mathbf{t}}\right|$, which is centered at $|\mathbf{t}|$ and has a standard deviation of $\frac{|\mathbf{t}|}{4}$, as follows:

$$\omega_{CF}\left(\mathbf{t}, \widehat{\mathbf{t}}\right) = \omega_{\text{compiler}}\left(\widehat{\mathbf{t}}\right) \times e^{-\frac{1}{2}\left(\frac{\left|\widehat{\mathbf{t}}\right|-|\mathbf{t}|}{|\mathbf{t}|/4}\right)^2} \tag{3}$$

Symbolic Execution Feedback (SF). We also introduce a *symexec feedback* (SF) that lets the LLM-in-training know the extent to which the generated translation is *(in)-equivalent* [2] to the source-language code. As it can be challenging to assess functional inequivalence among two codes of different languages ($\mathbf{s}$ and $\widehat{\mathbf{t}}$), we use two back-to-back (b2b) LLMs: $S \rightarrow T$ and $T \rightarrow S$ such that the same generated test suite can be reused for inequivalence testing. Given a tokenized training instance $(\mathbf{s}, \mathbf{t})$, the forward LLM ($S \rightarrow T$) translates $\mathbf{s}$ to an intermediate target-language sequence $\widehat{\mathbf{t}}$, which the backward LLM ($T \rightarrow S$) translates back to $\widehat{\mathbf{s}}$ in an attempt to reconstruct $\mathbf{s}$. To compute SF, we use solver-based analysis by an industrial-strength symexec tool called Symflower [47]. Through symexec, Symflower generates essential unit-tests even for functions that involve complex data types. It computes the necessary inputs and expected output to cover all linearly independent control-flow paths in a function. For J2P, S is Java and T is Python. We use Symflower to automatically generate JUnit tests ($\mathcal{J}_{\mathbf{s}}$) for the functions in $\mathbf{s} \in S$. These unit-tests are checked on $\widehat{\mathbf{s}}$ and both the LLMs get feedback about how many tests pass. Accordingly, SF is formulated as:

[2] It goes without saying that one cannot guarantee functional equivalence between two programs with a purely testing approach. However, we can establish inequivalence via a sufficient amount of testing.

$$\omega_{SF}(\mathbf{s},\widehat{\mathbf{s}}) = \frac{\epsilon + \sum_{j\in\mathcal{J}_{\mathbf{s}}} \mathbb{1}_{j(\widehat{\mathbf{s}})\equiv\text{Success}}}{\epsilon + |\mathcal{J}_{\mathbf{s}}|} \quad (4)$$

where, ϵ is a small positive value and $\mathbb{1}_{j(\widehat{\mathbf{s}})\equiv\text{Success}}$ is 1 when the j^{th} JUnit test on $\mathbf{s}$ successfully passes on $\widehat{\mathbf{s}}$, else it is 0. Note that, our b2b training loop (Section 3.2) incorporates CF on individual LLMs, in addition to SF. So, a feedback on $(\mathbf{s},\widehat{\mathbf{s}})$-inequivalence with a feedback on compilability of $\widehat{\mathbf{t}}$ and $\widehat{\mathbf{s}}$, correlates to $(\mathbf{s},\widehat{\mathbf{t}})$-inequivalence.

3.2 The CoTran Training Loop

Given a training instance $(\mathbf{s}_{code},\mathbf{t}_{code}) \in$ dataset D, kw-Tok transforms these to a sequence of token ids viz., $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ and $\mathbf{t} = (t_1, t_2, \ldots, t_m)$. Now, we elucidate a two-step procedure for training the LLMs:

Training the translation models $\mathbf{LLM_f}$ and $\mathbf{LLM_b}$. In the first phase, we separately fine-tune LLM_f and LLM_b to minimize the CE loss over respective translation tasks. LLM_f translates $\mathbf{s}$ to generate a target-language (T) sequence $\widehat{\mathbf{t}} = (\widehat{t}_1, \widehat{t}_2, \ldots, \widehat{t}_q)$. Through SFT, it is optimized to minimize the CE loss between $\widehat{\mathbf{t}}$ and $\mathbf{t}$ i.e. $\mathcal{L}_{CE}^{\theta_f}(\mathbf{t},\widehat{\mathbf{t}})$ as defined in Eqn. 1. Similarly, LLM_b translates $\mathbf{t}$ to generate a S-sequence $\widehat{\mathbf{s}} = (\widehat{s}_1, \widehat{s}_2, \ldots, \widehat{s}_r)$ and is trained to minimize $\mathcal{L}_{CE}^{\theta_b}(\mathbf{s},\widehat{\mathbf{s}})$.

Jointly fine-tuning models $\mathbf{LLM_f}$ and $\mathbf{LLM_b}$ by back-to-back (b2b) translation. Next, we further fine-tune LLM_f and LLM_b, but this time together through back-to-back (b2b) translation in order to incorporate CF and SF during training. Given a source-language code, a successful code translation calls for generating a target-language code that *compiles* and is *functionally equivalent* to the input code. Here, optimizing LLMs with CE is not sufficient. So, we define feedback (CF and SF) to let an LLM-in-training know how close it is to a perfect translation. However as CF and SF are non-differentiable functions, they cannot be directly used as loss functions to fine-tune an LLM. So, it is essential to construct an RL setting.

Further, as jointly fine-tuning both LLMs is resource-intensive we follow a parameter-efficient approach. For every linear layer of these LLMs that is either query or value, we create Low-Rank Adaptation [16] layers viz. a projection-up matrix A and projection-down matrix B, whose matrix-multiplication is initially zero. Except those in A and B, all the original parameters of LLM_f and LLM_b are frozen – making the training process space-time efficient.

The scheme for jointly fine-tuning LLM_f and LLM_b is given in Algorithm 1. In short, we interleave RL-based fine-tuning by Proximal Policy Optimization (PPO) [36] and SFT-based optimization of CE loss by Adam optimizer. First, LLM_f ($S \to T$) translates $\mathbf{s}$ to generate an intermediate T-sequence $\widehat{\mathbf{t}}$. With this as input, LLM_b ($T \to S$) tries to reconstruct sequence $\mathbf{s}$ and generates a S-sequence $\widehat{\mathbf{s}}$. In RL-based fine-tuning, the reward for LLM_f and LLM_b is the sum of SF computed among $\mathbf{s}$ and $\widehat{\mathbf{s}}$ and the respective CF among $\mathbf{t},\widehat{\mathbf{t}}$ and $\mathbf{s},\widehat{\mathbf{s}}$. To ensure that the LLMs being fine-tuned do not diverge much from the reference LLMs we started with, a KL-divergence [28] (d_{KL}) term is subtracted from the reward. This ensures that the PPO algorithm does not over-optimize and is appropriately penalized when the trained model starts to diverge too much from their references. Conversely, for the SFT-based optimization, the forward and backward models are trained by back-propagating the respective CE losses between the predicted translation and the corresponding gold-standard translation.

Algorithm 1: RL+SFT Interleaved Training for CoTran

Input : M_f (forward LLM); M_b (backward LLM); Tok (kw-Tok); `tP(·)` (trainable param); D_{trn}, D_{val} (training & validation data)
Output : Learned LLMs M_f (for $S \to T$) and M_b (for $T \to S$)

Function `RL_Bk2Bk` (*LLM$_f$, LLM$_f^{ref}$, LLM$_b$, LLM$_b^{ref}$, PPO$_f$, PPO$_b$, D*) **:**
  for *epoch* $\in [1, E]$ **do**
    foreach $(\mathbf{s},\mathbf{t}) \in D$ **do**
      $\widehat{\mathbf{t}} \leftarrow \text{LLM}_\text{f}(\mathbf{s})$; $\widehat{\mathbf{s}} \leftarrow \text{LLM}_\text{b}(\widehat{\mathbf{t}})$
      $r_\text{f} \leftarrow \omega_{\text{CF}}(\mathbf{t},\widehat{\mathbf{t}}) + \omega_{\text{SF}}(\mathbf{s},\widehat{\mathbf{s}}) - \beta \cdot d_{\text{KL}}(\text{LLM}_\text{f}, \text{LLM}_\text{f}^{\text{ref}})$
      $r_\text{b} \leftarrow \omega_{\text{CF}}(\mathbf{s},\widehat{\mathbf{s}}) + \omega_{\text{SF}}(\mathbf{s},\widehat{\mathbf{s}}) - \beta \cdot d_{\text{KL}}(\text{LLM}_\text{b}, \text{LLM}_\text{b}^{\text{ref}})$
      BackProp: $\text{LLM}_\text{f} \leftarrow \underset{\theta_\text{f}}{\text{PPO}_\text{f}}(\text{LLM}_\text{f}, r_\text{f})$; `// θ_f: tP(LLM_f)`
      BackProp: $\text{LLM}_\text{b} \leftarrow \underset{\theta_\text{b}}{\text{PPO}_\text{b}}(\text{LLM}_\text{b}, r_\text{b})$; `// θ_b: tP(LLM_b)`
  return LLM_f, LLM_b
Function `SFT_Bk2Bk` (*LLM$_f$, LLM$_b$, Adam$_f$, Adam$_b$, D*) **:**
  for *epoch* $\in [1, E]$ **do**
    foreach $(\mathbf{s},\mathbf{t}) \in D$ **do**
      $\widehat{\mathbf{t}} \leftarrow \text{LLM}_\text{f}(\mathbf{s})$; $\widehat{\mathbf{s}} \leftarrow \text{LLM}_\text{b}(\widehat{\mathbf{t}})$
      BackP: $\text{LLM}_\text{f} \leftarrow \underset{\theta_\text{f}}{\text{Adam}_\text{f}}\left(\text{LLM}_\text{f}, \mathcal{L}_{CE}^{\theta_\text{f}}(\mathbf{t},\widehat{\mathbf{t}})\right)$; `// θ_f: tP(LLM_f)`
      BackP: $\text{LLM}_\text{b} \leftarrow \underset{\theta_\text{b}}{\text{Adam}_\text{b}}\left(\text{LLM}_\text{b}, \mathcal{L}_{CE}^{\theta_\text{b}}(\mathbf{s},\widehat{\mathbf{s}})\right)$; `// θ_b: tP(LLM_b)`
  return LLM_f, LLM_b
Function `LoRA` (*LLM*) **:**
  LLM' ← deep copy of LLM ; `// θ: tP(LLM), θ': tP(LLM')`
  foreach *query/value layer* $L_i \in \mathbb{R}^{d_{in}\times d_{out}}$ *of LLM* **do**
    Initialize projection-up layer $A_i \in \mathbb{R}^{d_{\text{in}}\times r}$ randomly from $\mathcal{N}(0,1)$, projection-down layer $B_i \leftarrow \{0\}^{r\times d_{\text{out}}}$
    Replace L_i of LLM' with $L_i + \frac{\alpha}{r}(A_i \times B_i)$
  $\theta' \leftarrow \theta \cup A \cup B$. Freeze params in θ' except A, B
  return LLM'
Function `EVAL` (*LLM$_f$, LLM$_b$*) **:**
  return $\omega_{\text{SF}}(\mathbf{s},\widehat{\mathbf{s}})$ averaged over all $\mathbf{s}$ in the tokenized D_{val}, with LLM_f, LLM_b as back-to-back models

MAIN FUNCTION

$\text{M}_f', \text{M}_b' \leftarrow$ `LoRA` (M_f), `LoRA` (M_b)
Initialize RL Optimizers: $\text{PPO}_f, \text{PPO}_b$ for M_f', M_b'
Initialize SFT Optimizers: $\text{Adam}_f, \text{Adam}_b$ for M_f', M_b'
$D_{\text{trn}}^{Tok} \leftarrow [(Tok(\mathbf{s}_{code}), Tok(\mathbf{t}_{code}))$ for $(\mathbf{s}_{code},\mathbf{t}_{code})$ in $D_{\text{trn}}]$
do
  $\text{M}_f^{\text{SFT}}, \text{M}_b^{\text{SFT}} \leftarrow$ `SFT_Bk2Bk` $(M_f', M_b', Adam_f, Adam_b, D_{trn}^{Tok})$
  $\text{M}_f^{\text{RL}}, \text{M}_b^{\text{RL}} \leftarrow$ `RL_Bk2Bk` $(M_f', M_f, M_b', M_b, PPO_f, PPO_b, D_{trn}^{Tok})$
  $\text{valAcc}_{\text{SFT}}, \text{valAcc}_{\text{RL}} \leftarrow$ `EVAL` (M_f^{SFT}, M_b^{SFT}), `EVAL` (M_f^{RL}, M_b^{RL})
  $\text{M}_f', \text{M}_b' \leftarrow (\text{valAcc}_{\text{RL}} \geq \text{valAcc}_{\text{SFT}})$ **?** $\text{M}_f^{\text{RL}}, \text{M}_b^{\text{RL}}$ **:** $\text{M}_f^{\text{SFT}}, \text{M}_b^{\text{SFT}}$
while *valAcc$_{RL}$, valAcc$_{SFT}$ converged or maxEpochs reached*;

4 The AVATAR-TC Dataset

This paper introduces a new dataset **AVATAR-TC** (built on top of the AVATAR [2]) that has pairs of whole-programs in Java and Python (a statically- and dynamically-typed language, with different syntactic styles), each accompanied by human-written test-cases (TCs). Based on what we know, AVATAR-TC is the first such large-scale dataset where code compilability (syntactical correctness) is ensured, and code pairs have undergone thorough testing w.r.t. human-written TCs.

Note that, these TCs are not used in training any of the CoTran variants (which rely on automated unit-test generation); in this paper, we use them to evaluate translators with FEqAcc (Section 5.2).

Data Source. Similar to AVATAR, we gather a collection of code pairs written in Java and Python by scraping five *competitive coding websites* that host regular contests: Aizu [3], AtCoder [4], Codeforces [7], G-CodeJam [8], LeetCode [20] and two *coding platforms*: GeeksForGeeks [13], Project Euler [31]. These serve as great resources for mining code solutions of a problem statement across multiple

Table 1: **Benchmark Suite:** Statistics of the AVATAR-TC dataset

Sub-dataset	# problem-stmts with test-cases			# pairs of Java-Python codes		
	Train	Valid	Test	Train	Valid	Test
Aizu	762	41	190	14,019	41	190
AtCoder	619	19	97	13,558	19	97
Codeforces	1,625	96	401	23,311	96	401
Google CodeJam	59	1	4	347	1	4
LeetCode	81	7	18	81	7	18
GeeksForGeeks	3,753	268	995	3,753	268	995
Project Euler	110	11	41	110	11	41
Total	7,009	443	1,746	55,179	443	1,746

languages. Also, a participant's code gets checked on multiple test-cases (TCs) curated by the problem-setter. For AVATAR-TC, we web-crawled these data sources and collected such human-written TCs to complement each problem statement.

Data Cleaning. Several code pairs in AVATAR dataset did not compile and/or pass our collected TCs. Consequently in AVATAR-TC, we preprocessed codes afresh from their sources. Utilizing the `javalang` [40] and `tokenize` [32] modules, they were parsed into code-specific tokens. We manually corrected minor faults in code pairs, that did not match the expected output when provided with the TC inputs. Code pairs with major issues were discarded. Our criteria for output matching include case insensitivity, whitespace removal, punctuation disregard (unless significant to the output), and normalization of numeric or floating-point values to a common representation.

Statistics of AVATAR-TC dataset. The train/validation/test partitioning of problem statements is kept the same as AVATAR, except removal of some pairs during data cleaning. This resulted in $57{,}368$ Java-Python pairs at a train : validation : test ratio of $76:5:19$, across $9{,}198$ problems. For the train split, at most 25 pairs corresponds to one problem, while for validation and test there is a unique one-one pair-problem mapping. To ensure *out-of-distribution testing*, no problem overlaps across splits. Refer Table 1 for AVATAR-TC statistics.

5 Experiments

5.1 Experimental Setup and Competing Tools

CoTran (baseline) refers to an LLM fine-tuned without any feedback loop (Refer Figure 1a). It uses the pre-trained CodeT5-base [45] architecture from Huggingface [17]. It is fine-tuned with CE loss and uses the proposed keyword-based tokenizer (kw-Tok). The maximum length for the source and target sequences is set at 512. For additional design choices adopted during implementing CoTran, please refer to Appendix A.2. The baseline CoTran and its variants are compared against (See Table 2): (a) three *human-written transpilers*, (b) three SoTA *LLM-based unsupervised translation tools* (trained on function pairs from ~ 2.5M open-sourced repositories of the GitHub dataset from Google BigQuery Public Datasets), (c) *ChatGPT* [27] and, (d) seven *LLM-based supervised translation tools*. All the tools above are compared on the same set of $1{,}746$ whole-programs from AVATAR-TC (Test). To ensure a fair comparison, each of the supervised tools and CoTran variants are trained on whole-program pairs from AVATAR-TC (Train). For ChatGPT, we use OpenAI API to access the `gpt-3.5-turbo-0301` model. This version of the ChatGPT model was released on March 1, 2023, which predates AVATAR-TC's public availability on Anonymous GitHub. This ensures a fair evaluation by lessening the chances of OpenAI mining AVATAR-TC (Test) pairs, for inclusion in ChatGPT's training data. Following a standardized protocol [46] for ChatGPT, we use "`Translate [`S`] to [`T`]:[`$\mathbf{s}_{code}$`]\n Do not return anything other than the translated code.`" as the prompt. Temperature and top_p are set at 0 (this ensures reproducibility and does not notably alter the translation performance).

5.2 Evaluation Metrics

We evaluate our method using *greedy decoding* that considers only the top translation with the highest log probability. The different metrics for evaluating code translation quality are:

BLEU, CodeBLEU score. BLEU [29] computes the 'closeness' with the reference translation through n-gram overlaps. CodeBLEU [33] additionally checks weighted n-gram match, syntactic AST match (SM) and semantic data-flow match (DM). Both range in $[0, 100]$.

Exact String Match (EM). $\%$-age of generated codes that exactly match the reference translation. Note that, EM can be low even if the generated codes are compilable and functionally equivalent.

Compilation Accuracy (*CompAcc*) and Functional-Equivalence Accuracy (*FEqAcc*). *CompAcc* is defined as the percentage of generated translations that compile correctly. *FEqAcc* is the percentage of generated translations that are IO equivalent to the source-language code w.r.t. a set of human-written test-cases.

Additionally, we propose three new quality measures, namely:

Average First Error Position ($\mathbf{errPos_{1st}}$). $\text{errPos}_{1\text{st}}$ is a fine-grained version of *CompAcc*, relating to the closeness of the translations from a perfect compilation. Averaged over all translations on test set, $\text{errPos}_{1\text{st}}(\widehat{\mathbf{t}})$ calculates the position of the first token responsible for a syntactic error in $\widehat{\mathbf{t}}$, normalized by $|\widehat{\mathbf{t}}|$. It is computed by `pylint` and `javac` for Python and Java resp. Let's consider that a translator achieves $\text{errPos}_{1\text{st}} = e\%$. This implies that, on average, the first compilation error (if any) is located within the last $(100 - e)\%$ portion of the generated translations. As e approaches 100, the human developer only needs to inspect a small section of each translated code to rectify the error, thereby facilitating ease of manual debugging.

Average #Errors per Code (EpC). EpC is the average count of compilation (syntactic) errors per translated code. Zero errors indicate full syntactic correctness. A higher count implies that substantial effort from a human end-user is required for rectification.

Ratio of FEqAcc and CompAcc ($\frac{\mathbf{f}}{\mathbf{c}}$rate). $\frac{\text{f}}{\text{c}}$rate is the %-age rate at which a translator generates functionally-equivalent codes compared to the compilable ones. In a real-world deployment, checking equivalence of translator-generated codes is infeasible. Thus, it is beneficial when a syntactically correct code produced by a translator implies functional equivalence i.e., ideally $\frac{\text{f}}{\text{c}}$rate tends to 100%. $\frac{\text{f}}{\text{c}}$rate is computed as the percentage ratio of FEqAcc and CompAcc.

5.3 Analysis of Empirical Results & Ablation Study

In Table 2, we compare CoTran over J2P and P2J translation against 14 competing tools. Among them, PLBART-base, CodeT5-base, and PPOCoder perform among the top for both J2P and P2J translations. Surprisingly, the transpilers perform poorly (Refer Appendix A.4). ChatGPT exhibits relatively poor performance in P2J i.e., -27.03% in FEqAcc, -52.01% in CompAcc, and -66.07% in $\text{errPos}_{1\text{st}}$, compared to the best CoTran method. A low $\text{errPos}_{1\text{st}}$ indicates that ChatGPT-generated translations are hard to debug. However, in J2P, ChatGPT gets 76.06% FEqAcc which is the best among all listed methods in the table. Note that, CoTran models are built on CodeT5-base with ~220M params. We believe that if we trained a model as large as ChatGPT (rumored to be 100B+ params) using our method, then that would beat ChatGPT overall. Although ChatGPT allows users to fine-tune, the interface they provide is not tailored to receiving symbolic feedback from tools such as compilers, testers, and solvers. Hereafter, **we report CoTran improvements w.r.t CodeT5-base**.

Table 2: **Code Translation Results:** Comparison of CoTran against 14 other tools for Java-Python (J2P) and Python-Java (P2J) translation. (In each column, the highest value is marked in **bold**, second-highest underlined.)

Method / Tool	Model	Java → Python (J2P)						Python → Java (P2J)					
		FEqAcc	*CompAcc*	errPos$_{1st}$	CodeBLEU	BLEU	EM	*FEqAcc*	*CompAcc*	errPos$_{1st}$	CodeBLEU	BLEU	EM
Transpilers	java2python [26]	3.32	41.46	28.62	20.31	17.54	0	-	-	-	-	-	-
	TSS CodeConv [41]	0.46	58.30	54.26	41.87	24.44	0	-	-	-	-	-	-
	py2java [10]	-	-	-	-	-	-	0	0	1.61	41.56	48.59	0
Recent competing tools *(unsupervised trng.)*	TransCoder [34]	0.46	88.09	63.57	35.07	32.07	0	0	0	4.57	35.02	35.06	0
	TransCoder-DOBF [18]	0.46	63.00	47.10	39.98	33.84	0	0	0	3.11	33.33	32.72	0
	TransCoder-ST [35]	0.46	91.58	74.68	40.04	37.30	0	0	0	4.67	29.88	28.15	0
ChatGPT	GPT-3.5-turbo [27]	**76.06**	95.36	90.88	52.11	53.19	0.29	21.65	24.97	30.86	54.08	55.58	0
Recent competing tools *(supervised trng. on AVATAR-TC)*	CodeBERT [9]	12.31	84.77	79.57	46.00	48.10	0.46	0.74	**96.79**	**99.51**	26.10	19.62	0
	GraphCodeBERT [15]	10.88	85.05	79.78	45.53	47.26	0.57	0.46	89.75	98.05	23.72	16.21	0
	CodeGPT [23]	24.86	78.92	89.21	38.38	38.64	1.49	13.40	45.13	94.50	40.51	37.96	0.52
	CodeGPT-adapted [23]	24.17	76.75	89.31	36.84	37.36	1.55	20.50	52.00	97.60	41.46	38.15	1.03
	PLBART-base [1]	38.55	91.47	90.79	54.77	59.34	1.32	38.26	75.77	96.64	55.96	59.24	0.97
	CodeT5-base [45]	40.95	92.84	**93.76**	55.34	60.03	2.41	33.79	68.84	98.02	57.64	60.16	0.86
	PPOCoder [37]	44.27	93.47	91.44	55.16	59.51	1.89	37.11	59.62	96.77	55.04	58.52	0.52
Our tool	**CoTran (baseline)**	44.52	96.12	92.07	55.44	58.71	2.11	40.41	73.63	92.16	**59.11**	61.12	**1.66**
Our tool with CF only	CoTran + CF (RL-based training)	47.02	96.56	91.58	56.10	60.59	2.23	42.78	74.80	96.91	58.55	61.26	1.60
	CoTran + CF (RL+SFT interleaved)	49.83	96.79	92.08	56.07	60.61	2.23	45.93	75.77	96.89	58.28	61.21	1.60
Our tool (b2b) with CF, SF	CoTran + CF + SF (RL-based training)	50.45	96.79	92.15	56.17	60.60	2.23	43.92	75.14	96.93	58.59	**61.28**	1.60
	CoTran + CF + SF (RL+SFT interleaved)	53.89	**97.14**	92.73	**56.24**	**60.69**	**2.29**	**48.68**	76.98	96.93	58.38	61.19	1.60

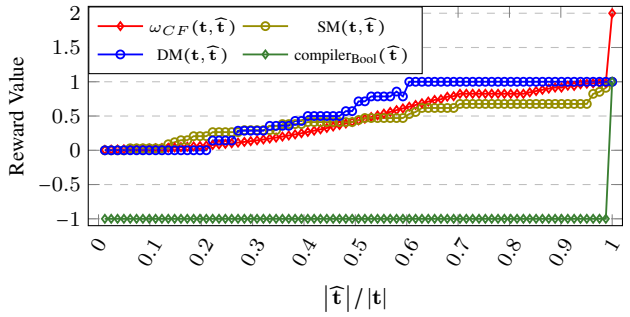


Figure 2: **Reward Analysis:** Plot of different rewards for RL

Hypothesis: *LLMs are good at code translation. Having said that, incorporating compiler and symexec feedback (CF, SF) during fine-tuning significantly improves its capability of producing compilable and functionally equivalent translations.*

Our results validate this hypothesis when evaluated across the diverse AVATAR-TC dataset. In the process, we have made several findings:

Finding 1: *RL turned out to be much more effective than SFT schemes we tried, for incorporating feedback during training.*

We considered two *non-RL schemes* of SFT-based LLM training, combining CE loss and CF, SF. However, they underperform compared to the RL-based methods. Please refer to Appendix A.3 for more details.

Finding 2: *For RL reward, a Boolean feedback from compiler and other existing feedback are not as effective as ours.*

For RL-based optimization of LLMs, it is essential to fabricate a good reward function. CompCoder [44] attempts RL-based code generation using a Boolean feedback compiler$_{\text{Bool}}(\widehat{\mathbf{t}})$ that returns $\{-1,+1\}$. PPOCoder [37] uses the sum of compiler$_{\text{Bool}}(\widehat{\mathbf{t}})$, syntactic match score SM$(\mathbf{t},\widehat{\mathbf{t}})$ and dataflow match score DM$(\mathbf{t},\widehat{\mathbf{t}})$ as the RL reward. We hypothesize that even though these functions are good tools to compare $\widehat{\mathbf{t}}$ with $\mathbf{t}$, they are not the best when it comes to an RL reward.

In Figure 2, we take $\mathbf{t}$ as tokenized version of a simple Java code for reversing an integer using for-loop. $\widehat{\mathbf{t}}$ is a truncated version of $\mathbf{t}$ till a certain number of tokens. We vary the truncation factor $|\widehat{\mathbf{t}}|/|\mathbf{t}|$ from 0 to 1 and plot the respective reward. For a significant portion of the x-axis (especially when $\widehat{\mathbf{t}}$ is close to an empty or perfectly-compilable translation), values of compiler$_{\text{Bool}}(\widehat{\mathbf{t}})$, SM$(\mathbf{t},\widehat{\mathbf{t}})$ and DM$(\mathbf{t},\widehat{\mathbf{t}})$ are all constant. So, CompCoder and PPOCoder give the same RL reward for several closely-related translations $\widehat{\mathbf{t}}$'s. As such, the RL agent cannot interpret whether it is improving or deteriorating. Conversely, our proposed reward $\omega_{CF}(\mathbf{t},\widehat{\mathbf{t}})$ is much more resilient in identifying small changes in $\widehat{\mathbf{t}}$. It drives the RL agent towards smaller goals which ultimately helps to reach the final goal of a perfect compilation. Thus, without even RL+SFT, CoTran + CF (RL only) performs better than PPOCoder, although both are built on CodeT5-base.

Finding 3: *Interleaving RL and SFT improves the LLM's performance, compared to an RL-only fine-tuning approach.*

As RL-based training on the b2b models can deviate them from their respective CE loss objective, interleaving them occasionally with SFT-based training helps improve the overall translation performance. To incorporate CF during training (**CoTran + CF**), we fine-tune the baseline model further to maximize CF as a reward in an RL-based setting. Using only RL-based training (Refer Figure 1b), the improvement in J2P and P2J translation on FEqAcc is +6.07%, +8.99% resp., while on CompAcc it is +3.72%, +5.96% resp. The RL+SFT interleaved training boosts this to +8.88%, +12.14% in FEqAcc and +3.95%, +6.93% in CompAcc. Note that for CoTran + CF, the interleaved training is similar to Algorithm 1 with the difference that ω_{SF} is not considered (there are no b2b LLMs, only the forward LLM). On incorporating both CF and SF (**CoTran + CF + SF**) in a solely RL-based setting with b2b LLMs (Figure 1c), the improvement in J2P and P2J translation performance is +9.50%, +10.13% resp. on FEqAcc and +3.95%, +6.30% on CompAcc. This increases to +12.94%, +14.89% resp. on FEqAcc and +4.30%, +8.14% on CompAcc, upon performing RL+SFT interleaved training (Figure 1d and Algorithm 1). Also, it achieves 92.73% and 96.93% on *errPos*$_{1st}$ for J2P and P2J resp, thus signifying easy-to-debug translations.

Finding 4: *Adding keywords to tokenizer vocabulary by kw-Tok improves the code translation performance.*

For J2P and P2J translation resp., *kw-Tok* itself accounts for +3.57%, +6.62% increase on FEqAcc and, +3.28%, +4.79% increase on CompAcc (CoTran baseline v/s CodeT5-base).

Finding 5: *Incorporating compiler and symexec feedback during fine-tuning reduces the count of syntactic errors per translated code.*

In Figure 3, we compare the number of compilation (syntactic) errors per code (EpC) produced in Python-to-Java (P2J) translation on AVATAR-TC (Test). ChatGPT exhibits significantly poorer performance with an EpC of 10.98 ± 9.67 (i.e., mean ± stdev). CodeT5-base reduces this to 0.95 ± 2.58. Our CoTran + CF + SF (RL+SFT) further lowers the EpC value to 0.71 ± 2.15. This indicates that the proposed feedback results in producing more easy-to-debug translations.

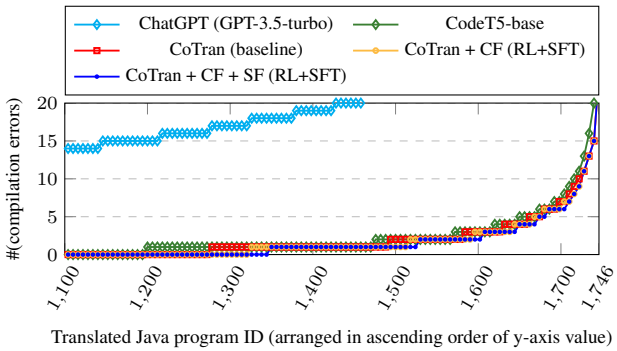


Figure 3: **Count of compilation errors per code (EpC)**: #Errors per translated code in P2J translation, sorted in the ascending order

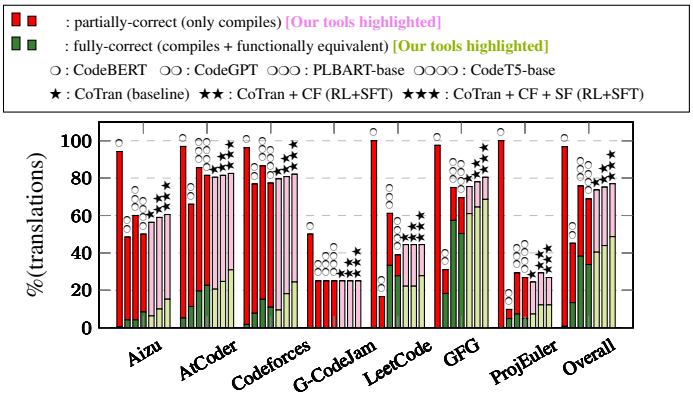


Figure 4: **Sub-dataset wise performance w.r.t. $\frac{f}{c}$rate:** Comparison of P2J translation, across the 7 sub-datasets of AVATAR-TC (Test)

Finding 6: *Integrating compiler and symexec feedback during fine-tuning markedly improves $\frac{f}{c}$rate metric across diverse codebases.*
In Figure 4, we compare the P2J performance of four SoTA tools and three CoTran-based methods over the different sub-datasets of AVATAR-TC (Test). CodeBERT demonstrates the highest rate of compilable translations (with over 90% CompAcc across most sub-datasets). However, the resultant translations lack meaningfulness, with only around 1% of them being functionally equivalent. In contrast, CodeGPT, PLBART-base, and CodeT5-base yield functionally equivalent codes at rates of 29.69%, 50.49%, and 49.08% respectively, relative to the number of compilable translations they generate across all sub-datasets. This $\frac{f}{c}$rate increases to 63.24% for CoTran + CF + SF (RL+SFT). Our method consistently *ranks among the top two* in $\frac{f}{c}$rate for each sub-dataset. Note that, G-CodeJam is a small sub-dataset, and all tools perform similarly poorly when translating P2J on it.

Finding 7: *It is easier for a code translator to achieve good performance if the source language is statically-typed.*
Java is a statically-typed language, while Python is dynamically-typed. When translating J2P, Java's explicit variable-type declarations are ignored as they are redundant in Python. Conversely, in Python, variable-types are not explicitly declared, requiring the translator to infer them during P2J translation. Even for humans experienced in both languages, deducing types for variables in Python can be challenging and demand repeated engrossed mental evaluations. Consequently, learning P2J is much more challenging than J2P. Thus, the source language being statically-typed makes learning code translation easier.

Finding 8: *Function-to-function translation is not sufficient for whole-program translation tasks.*
The TransCoder-based unsupervised tools are trained on function-level translations. As per Roziere et al. [34], this keeps training batches shorter and unit-test-based model evaluation simpler. But consequently, these tools cannot efficiently generate compilable, functionally-equivalent translations for whole-programs. For instance, in J2P, all three TransCoder-based tools fail drastically in FEqAcc (0.46%). Similarly, for P2J, no (0%) translated code is compilable.

Correlation between CF and SF. Table 2 suggests an implicit correlation between compiler and symexec feedback (CF, SF). Fine-tuning LLM over CF improves generation of compilable codes. This also produces more functionally-equivalent codes even without direct SF fine-tuning. For instance, in J2P, incorporating CF through RL+SFT only marginally increases CompAcc (w.r.t. baseline) from 96.12 to 96.79%, but significantly boosts FEqAcc from 44.52 to 49.83%. Similarly, CF+SF through RL+SFT not only enhances FEqAcc but also CompAcc, such as in P2J where CompAcc increases from 73.63 to 76.98%, and FEqAcc increases from 40.41 to 48.68%.

Benefits of automatic test-case generation (TCgen) during training. During fine-tuning $S \rightarrow T \rightarrow S$ back-to-back LLMs (Section 3.2), we automatically generate unit-tests on $\mathbf{s}$ and test them on $\widehat{\mathbf{s}}$. Note that, equivalence checking of programs is an undecidable problem. Our idea is to assess the *inequivalence* of $\mathbf{s}$ and $\widehat{\mathbf{s}}$ (detectable via sufficient testing) w.r.t. a test suite produced by TCgen tool and thereby, to calculate symexec feedback (SF) for the LLM-in-training. To our knowledge, no existing LLM-based code translation tool employs *automatic symexec-based TCgen* for functional equivalence checking and feedback. All existing tools like PPOCoder [37], TransMap [43], RLTF [22], CodeRL [19] use *human-written* test cases for fixing LLM-generated codes. In contrast, CoTran does not require anything extra than what is required for a CE loss-based supervised fine-tuning of LLM i.e. a dataset of S, T code pairs. Also, compared to human-written test-cases, symexec-based testing has a higher probability of covering all linearly independent control-flow paths.

Generalizability regarding choice of TCgen tool. We want to emphasize that our approach is entirely agnostic to any specific TCgen tool, including Symflower. The selection of Symflower is driven by the fact that we tested CoTran with Java as S in the $S \rightarrow T \rightarrow S$ back-to-back training loop, and Symflower stands out as a commercial industrial-strength symexec engine for Java. Its efficiency and reliability make it a preferred choice. MLB, JBMC, and GDart (the top 3 verification tools for Java in SV-COMP 2024 [5]) can also be considered as potential alternatives to Symflower. In fact, similar efficient TCgen tools (e.g. EvoSuite [11] for Java, KLOVER [21] for C++, Pynguin [24] for Python) are available for most popular languages. If not, LLVM-based TCgen tools (e.g., KLEE) offer an alternative, as many languages can be readily translated into LLVM IR.

Reproducibility. All our code, AVATAR-TC dataset, and the appendix are in the supplementary, which includes a README file outlining execution steps. URL: https://anonymous.4open.science/r/CoTran.

6 Conclusion and Future Work

In summary, we present an LLM-based code translation method (CoTran) that incorporates feedback from compiler and symexec-based solver during fine-tuning. The paper highlights that LLMs being trained for software engineering tasks e.g., code translation/synthesis, can greatly improve through feedback from compilers, program analysis, and test generation. Another key insight is the efficacy of fine-tuning LLMs with fine-grained feedback, pinpointing proximity to an ideal solution, than simple directives like 'try again'. Results show that CF and SF, especially with RL+SFT interleaved training, significantly improve code translation, yielding more compilable and functionally equivalent translations when compared to 14 SoTA tools. We plan to extend CoTran to translate legacy codes to modern languages.

References

[1] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified Pre-training for Program Understanding and Generation. In *Proc. 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. ACL. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211.

[2] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang. AVATAR: A Parallel Corpus for Java-Python Program Translation. *arXiv preprint arXiv:2108.11590*, 2021.

[3] Aizu-Online-Judge. Programming Challenge, 2023. https://judge.u-aizu.ac.jp/onlinejudge/.

[4] AtCoder, 2023. https://atcoder.jp/.

[5] D. Beyer. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–329. Springer, 2024.

[6] X. Chen, C. Liu, and D. Song. Tree-to-Tree Neural Networks for Program Translation. *Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[7] Codeforces, 2023. https://codeforces.com/.

[8] CodeJam. Google's Coding Competitions, 2023. https://codingcompetitions.withgoogle.com/codejam.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. ACL. doi: 10.18653/v1/2020.findings-emnlp.139.

[10] N. Fomin. `py2java`: Python to Java Language Translator, 2019. https://pypi.org/project/py2java/.

[11] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419, 2011.

[12] V. Ganesh, S. A. Seshia, and S. Jha. Machine Learning and Logic: A New Frontier in Artificial Intelligence. *Formal Methods in System Design*, 60 (3):426–451, 2022.

[13] GeeksforGeeks. A CS Portal for Geeks, 2023. https://www.geeksforgeeks.org/.

[14] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján, and H. Mössenböck. Cross-language Interoperability in a Multi-language Runtime. *ACM Transactions on Programming Lang. and Systems (TOPLAS)*, 40(2):1–43, 2018.

[15] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations (ICLR)*, 2021.

[16] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*, 2021.

[17] HuggingFace. The AI Community Building the Future, 2023. https://huggingface.co/.

[18] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample. DOBF: A Deobfuscation Pre-training Objective for Programming Languages. *Advances in Neural Information Processing Systems (NeurIPS)*, 34: 14967–14979, 2021.

[19] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[20] LeetCode, 2023. https://leetcode.com/.

[21] G. Li, I. Ghosh, and S. P. Rajan. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 609–615. Springer, 2011.

[22] J. Liu, Y. Zhu, K. Xiao, Q. Fu, X. Han, W. Yang, and D. Ye. RLTF: Reinforcement Learning from Unit Test Feedback. *arXiv preprint arXiv:2307.04349*, 2023.

[23] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664*, 2021.

[24] S. Lukasczyk and G. Fraser. Pynguin: Automated Unit Test Generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[25] B. G. Mateus, M. Martinez, and C. Kolski. Learning Migration Models for Supporting Incremental Language Migrations of Software Applications. *Information and Software Tech.*, 153, 2023.

[26] T. Melhase, B. Kearns, L. Li, I. Curt, and S. Saladi. `java2python`: Simple but Effective Tool to Translate Java Source Code into Python, 2016. https://github.com/natural/java2python.

[27] OpenAI. ChatGPT [Large Language Model], 2023. https://chat.openai.com.

[28] D. Palenicek. A Survey on Constraining Policy Updates using the KL Divergence. *Reinforcement Learning Algorithms: Analysis and Applications*, pages 49–57, 2021.

[29] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proc. 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[30] T. Parr, T. Everett, et al. Grammars-v4: Grammars written for ANTLR (ANother Tool for Language Recognition) v4, 2023. https://github.com/antlr/grammars-v4.

[31] ProjectEuler, 2023. https://projecteuler.net/.

[32] Python. `tokenize`: Tokenizer for Python, 2023. https://docs.python.org/3/library/tokenize.html.

[33] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. CodeBLEU: A Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[34] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample. Unsupervised Translation of Programming Languages. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:20601–20611, 2020.

[35] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample. TransCoder-ST: Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=cmt-6KtR4c4.

[36] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[37] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy. Execution-based Code Generation using Deep Reinforcement Learning. *arXiv preprint arXiv:2301.13816*, 2023.

[38] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. IntelliCode Compose: Code Generation using Transformer. In *Proc. 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[39] S. Thenault. pylint: Python code Static Checker, 2023. https://pypi.org/project/pylint/.

[40] C. Thunes. `javalang`: Pure Python Java parser and tools, 2020. https://github.com/c2nes/javalang.

[41] TSS. The Most Accurate and Reliable Source Code Converters, 2023. (Tangible Software Solutions): https://www.tangiblesoftwaresolutions.com/.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[43] B. Wang, R. Li, M. Li, and P. Saxena. TransMap: Pinpointing Mistakes in Neural Code Translation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 999–1011, 2023.

[44] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu. Compilable Neural Code Generation with Compiler Feedback. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 9–19, 2022.

[45] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proc. 2021 Conference on Empirical Methods in NLP*, pages 8696–8708. ACL, 2021. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

[46] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang. CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation. *arXiv preprint arXiv:2310.04951*, 2023.

[47] M. Zimmermann and E. Haslinger. Symflower: Smart Unit Test Generator for Java, 2023. https://symflower.com/en/.