Beyond Lexical Consistency: Preserving Semantic Consistency for Program Translation

Yali Du, Yi-Fan Ma, Zheng Xie, and Ming Li*
National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
{duyl, mayf, xiez, lim}@lamda.nju.edu.cn

Abstract—**Program translation aims to convert the input programs from one programming language to another. Automatic program translation is a prized target of software engineering research, which leverages the reusability of projects and improves the efficiency of development. Recently, thanks to the rapid development of deep learning model architectures and the availability of large-scale parallel corpus of programs, the performance of program translation has been greatly improved. However, the existing program translation models are still far from satisfactory, in terms of the quality of translated programs. In this paper, we argue that a major limitation of the current approaches is lack of consideration of semantic consistency. Beyond lexical consistency, semantic consistency is also critical for the task. To make the program translation model more semantically aware, we propose a general framework named Preserving Semantic Consistency for Program Translation (PSCPT), which considers semantic consistency with regularization in the training objective of program translation and can be easily applied to all encoder-decoder methods with various neural networks (e.g., LSTM, Transformer) as the backbone. We conduct extensive experiments in 7 general programming languages. Experimental results show that with CodeBERT as the backbone, our approach outperforms not only the state-of-the-art open-source models but also the commercial closed large language models (e.g., text-davinci-002, text-davinci-003) on the program translation task. Our replication package (including code, data, etc.) is publicly available at https://github.com/duyali2000/PSCPT.**

Index Terms—**Program Translation, Semantic Consistency, Regularization, Large Language Model**

I. Introduction

Nowadays, programs have become the main tool for building computer applications, the information technology industry, and the digital world [1], [2]. To this end, various programming languages have been invented to develop programs with different demands. Unfortunately, when combining programs written in different programming languages, the developer usually suffers from the intensive labor of manually translating programs from one to another programming language. For example, many industries spend hundreds of millions of dollars to convert code written in older programming languages (e.g., FORTRAN and COBOL) to newer ones (e.g., Java, C++) [3]. To alleviate the burden of program migration and facilitate the development of software systems, program translation, which aims to automatically translate the program from one programming language to another, has drawn significant attention in the software mining community [4]–[8].

*Ming Li is the corresponding author.

Recently, the booming development of machine learning coupled with the availability of an extensive parallel corpus of programs has led to a remarkable enhancement in the performance of program translation. Traditional approaches rely on the statistical machine translation [9], [10], attempting to adapt phrase-based statistical machine translation models with grammatical rules for code migration. Recent studies indicate that programs solving the same problem may have high diversity regarding variable names, method design, and logical flow, and such diversity especially becomes a bottleneck in program translation. To address this issue, recent program translation models aim to mine fixed correspondences between code patterns in different programming languages by minimizing the lexical difference between the generated program and the target program. For example, Zhu *et al.* [11] propose MuST to leverage the similarity between different programming languages like C++ and C and the snippet-level translation to enhance the more complex and challenging program-level translation. Guo *et al.* [12] propose GraphCode-BERT to utilize the data flow graph to extra capture the structural information of programs. Chen *et al.* [1] design a tree-to-tree neural network with the parse trees to align the source and target programs with grammar.

However, given the flexibility of programming language, even a tiny variance in lexical may cause a huge gap in semantics. As illustrated in Figure 1, the translated program is decoded from the source program and contrasted with the parallel target program. In the example, despite encountering a bit of confusion during program translation, the resulting translated program exhibits a significant degree of lexical similarity to the target program, attaining a BLEU score nearing 0.9. However, there is obviously a huge gap between the semantics of the two programs. While the target program would generate an in-order traversal of the binary tree, the translated program would yield a pre-order traversal of the same binary tree. It is terrifying, as a small translated difference is hard to identify by developers, but the translated error may cause a high cost in development. Therefore, in addition to considering lexical consistency, we should further preserve the semantic consistency in program translation.

One question arises here: how to take advantage of semantic consistency to leverage program translation? The semantics of the program can be calculated in any number of ways, which is usually set as the latent feature encoded by neural networks in other domains [13]–[17]. However, the neural

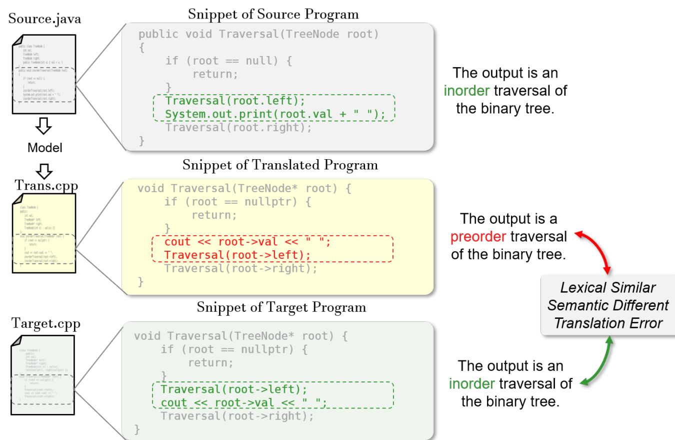


Fig. 1: A tiny lexical variance may cause a huge gap in semantics. The source program would output an in-order traversal of the binary tree, but the translated program would output a preorder traversal of the binary tree.

networks in the earlier training stages are unreliable since the semantic embedding space is still under optimization. To achieve this goal of constructing the latent space containing the semantics of a program in the target programming language, an auto-encoding regularization is applied to reconstruct the target program. Moreover, as the training is conducted in a supervised manner, a semantic-preserving regularization is designed between the latent vectors of the translated and target programs to bias the semantic consistency.

In our work, we argue that semantic consistency should be considered in the program translation. To this end, we propose a new framework named **P**reserving **S**emantic **C**onsistency for **P**rogram **T**ranslation (**PSCPT**). The regularization terms of auto-encoding and semantic-preserving beyond the general objective are designed for program translation. Moreover, the regularized program translation framework can be easily applied to all encoder-decoder methods with various neural networks (*e.g,* LSTM, Transformer) as the backbone. The extensive experiment is conducted on a widely used dataset, which includes 7 general programming languages (i.e., C, C#, C++, Java, Javascript, PHP, and Python). Compared with the state-of-the-art approaches, PSCPT achieves more than 9.40% improvement on average in terms of BLEU score [18] in program translation. Furthermore, to evaluate the quality of the translated program comprehensively in practical application, an experiment is conducted on a new dataset collected from real programming contests, and the results demonstrate that the PSCPT outperforms both the open-source models and the commercial closed large language models (e.g., text-davinci-002, text-davinci-003) in terms of all sorts of metrics at lexical, syntactic, and semantic levels.

In summary, we make the following major contributions:

- We argue that beyond lexical consistency, semantic consistency is crucial to be considered in program translation.
- A novel training framework is proposed for program translation, which measures both lexical and semantic consistency. It is the first attempt to explore semantic consistency regularization in program translation.
- We evaluate the performance of PSCPT on 7 general programming languages and up to 42 translation pairs. Compared with the state-of-the-art, PSCPT achieves more than 9.40% improvement on average in terms of BLEU score in program translation, which indicates preserving semantic consistency improves the program translation models dramatically.

The rest of the paper is organized as follows. In Section 2, the related works are discussed. Then the detail of the proposed regularized framework is introduced in Section 3. Experiments are provided in Section 4. Then the whole paper is concluded in Section 5.

II. Related Work

A. Automatic Program Translation

The previous works applied phrase-based statistical machine translation techniques to program translation [6]–[10], [19], which leveraged grammatical structures of programming languages for code migration. Nguyen *et al.* [10] presented an approach to programming language translation based on statistical language models, which integrated parsing queries to the programming language grammar into a phrase-based translation approach. Nguyen *et al.* [6] used Word2Vec representation of different programming languages and learned a transformation matrix for mapping. SMT [20] was a statistical phrase-based and tree-to-string machine translation technique, that can automatically learn statement-wise pseudo-code generators and require less human effort.

Recently, various deep learning techniques were employed to program translations [1], [3], [21]–[23]. Zhu *et al.* proposed a multilingual program translation model that used code snippets translation as a pre-training method to improve the accuracy of program translation [11]. Moreover, inspired by the success of pre-training models in natural language processing, a number of pre-training models are proposed in the software mining community, and the program translation is always used as one of the downstream tasks to evaluate the pre-training models [24]–[28]. For instance, GraphCodeBERT [12] imported structural information to enhance the code representation by adding the data flow graph as an auxiliary of input tokens, which improved the performance of code representation upon CodeBERT [29]. The most widely used loss function of program translation is the cross-entropy [30] calculated between the predicted tokens and referenced tokens, which preserves the lexical consistency in program translation. However, beyond lexical consistency, semantic consistency is also crucial to be considered in program translation.

B. Evaluation Metrics

There are various metrics that attempt to evaluate the performance of program translation approaches from lexical-level, syntactic-level, and semantic-level similarities. BLEU [18] is the most popular lexical similar metric in translation research, originated in 2002 from the machine translation research

community. The introduction of BLEU has facilitated the automated training and optimization of machine translation systems, expediting the research process. BLEU works by comparing n-grams in the prediction and reference. In the most typical implementation, n ranges from 1 to 4 and is used to compute a BLEU_n score:

$$\text{BLEU}_n = \frac{\sum_{t_n} min\{C_p(t_n), C_r(t_n)\}}{P(n)}, \tag{1}$$

where t_n is the n-gram, $C_p(t_n)$, $C_r(t_n)$ are the counts of the n-gram in the translated program and the target program, respectively, and P_n is the total number of n-grams. The single aggregate BLEU is reported in most papers, which is basically the product of the BLEU_n scores [13]. However, there are some limitations for BLEU to evaluate the code of the programming languages. Compared with natural language that has evolved naturally in humans through use and repetition, code is artificially designed to produce various kinds of output [31]. Weighted N-Gram Match Score [31] is introduced to assign different weights for different n-grams, where the keywords may have higher weights. Accuracy is another used occasionally lexical similar metric in program translation [12], which is highly dependent on the token hit or missed in the fixed position:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=0}^{n} \mathbb{I}(g_i = r_i), \tag{2}$$

where n is the number of the tokens of the reference program, g_i is the i-th token of the translated program, and r_i is the i-th token of the referenced program.

In addition to the lexical similarity, the syntactic similarity is usually measured by matching the abstract syntactic tree (AST) of the programs named AST Match Score [31]. In one abstract syntactic tree, each node denotes a construct occurring in the source program. The leaves of the abstract syntactic tree represent the names of the function and all the variables. All the sub-trees of the abstract syntactic trees of the translated program and the target program are extracted respectively[1] to calculate the syntactic similarity.

The semantic similarity is usually measured by matching the data flow graph (DFG) of the programs named DataFlow Match Score [31]. A data flow graph represents the relationship graph of the variables in the program, in which nodes represent variables and edges represent where the value of each variable comes from [12]. In order to calculate the semantic dataflow match score, it is necessary to acquire the dataflow graphs of both the translated program and the target program. The dataflow items are then normalized through a process of variable renaming. CodeBLEU [31] is an evaluation metric that evolves from the BLEU [18] by considering the importance of Weighted N-Gram Match, AST Match Score, and DataFlow Match Score at the same time.

$$\begin{aligned} \text{CodeBLEU} &= 0.25 \cdot \text{BLEU} + 0.25 \cdot \text{BLEU}_w \\ &+ 0.25 \cdot \text{Match}_{AST} + 0.25 \cdot \text{Match}_{DF}, \end{aligned} \tag{3}$$

[1]https://github.com/tree-sitter/tree-sitter

where BLEU is standard BLEU [18], BLEU_w is the Weighted N-Gram Match Score, Match_{AST} is the AST Match Score, Match_{DF} is the DataFlow Match Score.

III. The Proposed Method

The goal of program translation is to generate a translated program in the target programming language from the source program in the source programming language, where the program translation preserved both lexical and semantic consistency. Let $\mathcal{D} = \{(x_1, y_1), (x_2, y_2) \ldots (x_n, y_n)\}$ denotes the pairwise set of the program translation dataset, where $\{x_1, x_2, \ldots, x_n\} \in \mathcal{X}$ denotes the set of source programs, $\{y_1, y_2, \ldots, y_n\} \in \mathcal{Y}$ denotes the set of target programs.

The learning objective of program translation is to learn a generation function f: $\mathcal{X} \rightarrow \mathcal{Y}$ by minimizing the following regularized objective function:

$$\min_f \sum_i \mathcal{L}(f(x_i), y_i) + \alpha \cdot \Psi_{\text{AE}} + \beta \cdot \Psi_{\text{SP}}, \tag{4}$$

where $\mathcal{L}(\cdot, \cdot)$ is the loss function to preserve lexical consistency and Ψ_{AE}, Ψ_{SP} are the auto-encoding and semantic-preserving regularization terms. The trade-off between $\mathcal{L}(\cdot, \cdot)$, Ψ_{AE} and Ψ_{SP} is balanced by α and β.

The overall regularized training framework is presented in Section III-A. The program translation model is discussed in Section III-B, as well as the regularization to preserve semantic consistency is discussed in Section III-C.

A. The Overall Framework

The regularized training approach learns the model from the objective which consists of the supervised loss function as well as two regularization terms. The overall framework of the proposed approach is shown in Figure 2.

The training process of our model includes two phases. First, only auto-encoding regularization is performed to train the capacity of the model to reconstruct the program of the target language. The reconstruction process only involves programs in the target language, encoded by the target encoder and decoded by the decoder. As the reconstruction only used the monolingual translation pairs, it is more effective to train the model than using the bilingual translation pairs. With the auto-encoding regularization of the monolingual program in the target language, the target encoder is trained to be robust to map the target program into the semantic-enriched latent space used to guide the following semantic consistency preserving phase. And the decoder is trained to possess the capability of generating programs in the target language.

Second, the holistic program translation model is trained to preserve both lexical and semantic consistency. In this phase, the end-to-end program translation involves encoding the source program using the source encoder and decoding it through the decoder, where the translated program is contrasted with the target program in the lexical space of the target language. The translated target program is subsequently encoded once again by the target encoder and compared with the latent vector of the target program within the semantic

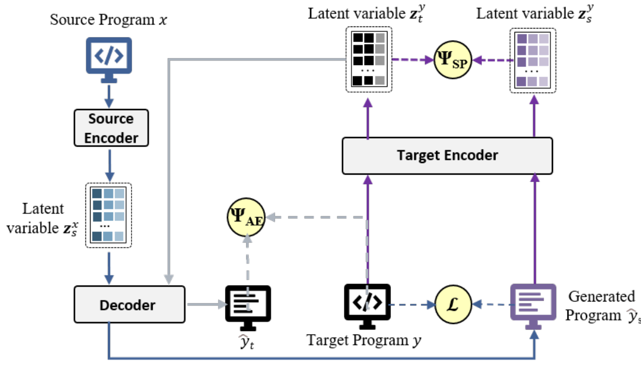


Fig. 2: An example framework of PSCPT. The training objective includes program translation loss preserving lexical consistency, auto-encoding regularization, and semantic-preserving regularization.

space. Therefore, the token sequences of the target program offer lexical supervision within the lexical space, while the latent vectors of the target program provide semantic supervision within the latent semantic space.

B. Program Translation

The source programs are pre-processed by tokenization into token sequences. Then the sequences are concatenated and encoded by the source encoder. Let $\mathbf{x}$, $\mathbf{y}$, and $\hat{\mathbf{y}}_s$ denote the source program, the target program, and the translated program, respectively. The program translation from the source program to the target program can be defined as the probability $p(\mathbf{y}|\mathbf{x})$ over the translation $\mathbf{y}$ by source program $\mathbf{x}$.

The transformer architecture is employed for the model, wherein the layers utilize a masked multi-head self-attention operation followed by a feed-forward network. Many works have shown that the transformer architecture can achieve promising performance on various language processing tasks [29], [32], [33]. The encoder is designed to map the input sequence of tokens into contextual representations, and the decoder aims to generate the output sequence based on it. We define $\mathrm{Encoder_s}$ as the source encoder, $\mathrm{Encoder_t}$ as the target encoder, and $\mathrm{Decoder}$ as the decoder to generate the translated program.

A latent vector $\mathbf{z}_s^x$ is introduced to capture the semantics of the source program, which is inferred by source encoder $\mathrm{Encoder_s}$ given source program $\mathbf{x}$.

$$\mathbf{z}_s^x = \mathrm{Encoder_s}(\mathbf{x}). \tag{5}$$

The general program translation model aims to define a probability $p(\hat{\mathbf{y}}_s|\mathbf{z}_s^x, \mathbf{x})$ over $\hat{\mathbf{y}}_s$ by the latent vector of source program $\mathbf{z}_s^x$. The translated program $\hat{\mathbf{y}}_s$ generated by the decoder given $\mathbf{z}_s^x$ is defined as:

$$\hat{\mathbf{y}}_s = \mathrm{Decoder}(\mathbf{z}_s^x, \mathbf{y}). \tag{6}$$

The loss function of program translation is based on the cross-entropy loss [30] to maximize the similarity of the tokens of the translated program and the target program.

$$\mathcal{L} = -\sum_{i=1}^{m} \log p_{\mathrm{Decoder}}(\mathbf{y}^i|\mathbf{z}_s^x, \mathbf{y}^{<i})), \tag{7}$$

where m is the length of the translated program.

In the process of testing, the decoded embedding of translated program $\hat{\mathbf{y}}_s$ can be fed into the output layers mapped to tokens by sharing the input embedding layers of the encoder after a dense layer and $tanh$ as the activation function:

$$\mu^s = tanh(W_\mu^s \hat{\mathbf{y}}_s + b_\mu^s), \tag{8}$$

$$\hat{\mathbf{y}}_{token}^s = W^s \mu^s + b^s. \tag{9}$$

For one thing, $tanh$ increases the nonlinearity of the model to improve the representation capability. For another, $tanh$'s lower bound prevents the value of μ from being so small that the model degenerates into a discriminative model.

C. Preserving Semantic Consistency

With the help of regularization terms, we expect to encourage the program translation to preserve semantic consistency, which constrains two aspects. First, auto-encoding regularization is applied to construct the latent space containing the semantics of a program in the target programming language. Second, semantic-preserving regularization is utilized to preserve semantic consistency by contrasting the semantics between the translated program and the target program in the semantic latent space of the target language.

Auto-Encoding Regularization The auto-encoding regularization is proposed to train the encoder to construct a semantic-enriched latent space and train the decoder to utilize the latent vector to generate the sequence of the target programming language. Let $\hat{\mathbf{y}}_t$ denote the translated program by auto-encoding. The target programs are pre-processed by tokenization into token sequences. Then the sequences are concatenated and encoded by the target encoder.

A latent vector $\mathbf{z}_t^y$ is introduced to represent the semantics of the target program in the semantic-enriched latent space, which is inferred by target encoder $\mathrm{Encoder_t}$ from the $\mathbf{y}$.

$$\mathbf{z}_t^y = \mathrm{Encoder_t}(\mathbf{y}). \tag{10}$$

The auto-encoding regularization aims to define a probability $p(\hat{\mathbf{y}}_t|\mathbf{z}_t^y, \mathbf{y})$ over $\hat{\mathbf{y}}_t$ by the latent vector of the target program $\mathbf{z}_t^y$. The decoded target program $\hat{\mathbf{y}}_t$ is defined as:

$$\hat{\mathbf{y}}_t = \mathrm{Decoder}(\mathbf{z}_t^y, \mathbf{y}). \tag{11}$$

The auto-encoding regularization based on the cross-entropy loss [30] to maximize the similarity of the reconstructed target program and the target program can be written as :

$$\Psi_{AE} = -\sum_{i=1}^{m} \log p_{\mathrm{Decoder}}(\mathbf{y}^i|\mathbf{z}_t^y, \mathbf{y}^{<i}), \tag{12}$$

where m is the length of the translated program.

Semantic-Preserving Regularization The loss function to preserve the lexical consistency of program translation has made a remarkable achievement [3], [11], [12]. However, there may be little obfuscation in lexical that could cause a significant semantic shift, and simply aligning the tokens of the translated program and target program at the lexical level can not tackle the problem. To avoid the significant

semantic shift, we claim that there exists the latent vector distribution for each program that can be used to appropriately guide the consistency training at the semantic level. As a semantic-enriched latent space of the target language has been constructed by the phase of auto-encoding, we implement the semantic preserving with the latent vectors of the translated program and the target program.

The latent vector $\mathbf{z}_s^y$ is introduced to represent the semantics of the translated target program, which is inferred by target encoder $\mathrm{Encoder_t}$ from the translated target program $\hat{\mathbf{y}}_s$.

$$\mathbf{z}_s^y = \mathrm{Encoder_t}(\hat{\mathbf{y}}_s). \tag{13}$$

The similarity between the translated program $\mathbf{z}_s^y$ and the true target program $\mathbf{z}_t^y$ in the latent space denotes the ability of the model to preserve the semantics of the program. Now we implement the semantic preserving by maximizing the similarity or minimizing the distance over the latent vectors $\mathbf{z}_s^y$ and $\mathbf{z}_t^y$. For the target program, $\mathrm{Encoder_t}$ can learn the semantic distribution of the program in the target programming language. Therefore, $\mathbf{z}_t^y$ can be regarded as a supervisor of $\mathbf{z}_s^y$ in the semantic-enriched latent space of the target language. We force $\mathbf{z}_s^y \cong \mathbf{z}_t^y$ by imposing L_2-distance loss [34] over the distance between $\mathbf{z}_s^y$ and $\mathbf{z}_t^y$:

$$\Psi_{SP} = ||\mathbf{z}_s^y - \mathbf{z}_t^y||_2^2. \tag{14}$$

By constraining the semantic-preserving regularization term, we can squeeze the latent vector $\mathbf{z}_s^y$ into the ground truth of $\mathbf{z}_t^y$ from all directions in the semantic-enriched latent space.

For more details, the Pseudocode of the training process of PSCPT is described in Algorithm 1.

Algorithm 1: Pseudocode of PSCPT.

```
Input: x, y are input of the source program and
       target program. α, β, and auto_num are hyper
       parameters, where auto_num is number of
       epochs for auto-encoding. opt is the optimizer.
1  for epoch in range(epochs) do
2      for (x, y) in loader do
3          z_t^y ← Encoder_t(y)
4          ŷ_t ← Decoder(z_t^y, y)
5          if epoch ≤ auto_num then
6              G ← ∇_θ(Ψ_AE(ŷ_t, y))
7          else
8              z_s^x ← Encoder_s(x)
9              ŷ_s = Decoder(z_s^x, y)
10             z_s^y ← Encoder_t(ŷ_s)
11             G ← ∇_θ(L(ŷ_s, y) + α(Ψ_AE(ŷ_t, y)) +
                 β(Ψ_SP(z_s^y, z_t^y))
12         end
13         Update θ
14         θ ← opt(θ, G)
15     end
16 end
```

IV. Experiment

This section describes our quantitative experiment on a widely used dataset and a new dataset collected from practical applications, in which we study the effectiveness of PSCPT in program translation described in Section III.

A. Dataset Description

We conduct experiments on the *CoST* dataset [11], which is a large and comprehensive dataset to evaluate the performance of program translation approaches. The dataset consists of both snippet-level and program-level parallel data from 7 programming languages (i.e., C, C#, C++, Java, Javascript, PHP, and Python) and up to 42 programming language pairs, which was collected from the GeeksForGeeks[2] website. The detailed statistics of the dataset are shown in Table I, while the train, validation, and test sets are split the same as *CoST* [11].

In order to assess the effectiveness of program translation in real-world applications, a new dataset is compiled from LeetCode[3]. The dataset contains 115 paired solutions commonly used in competitions in Java programming language and C++ programming language. The dataset is randomly separated with a ratio of 1:1 to the train set and test set.

[2]https://www.geeksforgeeks.org/

[3]https://leetcode.cn/

TABLE I: The statistic of the *CoST* dataset. The upper triangle (in normal font) shows the number of parallel snippets, and the lower triangle (in bold font) shows the number of parallel programs. **Py** is short for Python, and **JS** is short for Javascript.

Lang	C	C#	C++	Java	JS	PHP	Py
C	-	2123	2188	2135	1232	700	1779
C#	**273**	-	13326	13905	7601	3192	11404
C++	**267**	**1442**	-	13929	7596	3165	11930
Java	**281**	**1495**	**1497**	-	7729	3194	11713
JS	**196**	**994**	**996**	**1009**	-	2917	7165
PHP	**135**	**552**	**548**	**552**	**512**	-	545
Py	**263**	**1383**	**1419**	**1417**	**962**	**545**	-

B. Experimental Settings

We choose the following state-of-the-art program translation methods as baselines:

- **Naïve Copy**: A direct copy from the source program to the output of the translation, which denotes how similar the source language and the target language are.
- **DOBF** [35]: A pre-training objective for programming languages, which leverages the structural aspect of programming languages and recovering the original version of obfuscated source code.
- **CodeBERT** [29]: A bimodal pre-training model, trained with a hybrid objective function that incorporates the pre-training task of standard masked language modeling and replaced token detection.
- **GraphCodeBERT** [12]: A pre-trained model for the programming language that considers the inherent structure of code using data flow in the pre-training stage to encode the relationships between variables.
- **MusT-PT** [11]: An effective program translation technique

TABLE II: Performance evaluation in terms of BLEU on the CoST dataset, where GCBERT is short for GraphCodeBERT.

		Snippet Level							Program Level						
Lang	Method	C	C#	C++	Java	JS	PHP	Py	C	C#	C++	Java	JS	PHP	Py
C	Naive Copy	-	68.88	85.58	69.17	54.85	37.48	37.84	-	68.74	85.02	69.34	53.90	38.1	36.09
	DOBF	-	39.38	40.17	41.57	33.54	34.51	17.93	-	27.64	20.90	27.15	20.77	24.15	25.87
	CodeBERT	-	51.92	60.84	51.70	40.57	34.49	31.49	-	34.41	33.64	33.75	29.90	29.32	27.34
	GCBERT	-	28.99	35.78	30.48	21.59	20.46	14.30	-	41.51	47.48	41.82	27.44	35.64	35.51
	MusT-PT	-	80.68	**88.58**	79.24	80.35	**82.94**	66.49	-	78.39	84.92	76.84	66.13	70.62	55.71
	PSCPT	-	**86.70**	82.51	**80.02**	**80.76**	74.44	**76.72**	-	**80.10**	**86.53**	**78.97**	**80.91**	**81.99**	**69.51**
C#	Naive Copy	68.91	-	67.29	78.13	58.90	35.01	36.49	68.74	-	67.51	78.69	57.61	35.55	34.62
	DOBF	38.33	-	38.94	47.84	28.70	49.32	25.14	26.38	-	27.50	31.63	23.62	34.90	22.94
	CodeBERT	52.65	-	79.28	83.90	76.99	68.62	64.72	34.93	-	65.74	80.11	53.72	45.67	47.14
	GCBERT	28.23	-	84.34	80.46	63.70	50.70	53.39	30.69	-	81.96	90.74	79.37	67.86	71.81
	MusT-PT	81.12	-	85.34	85.80	82.74	81.64	71.11	78.78	-	84.72	87.76	70.00	70.66	62.03
	PSCPT	**87.19**	-	**85.98**	**87.74**	**83.99**	**78.31**	**72.80**	**83.84**	-	**87.23**	**93.99**	**82.27**	**85.95**	**74.66**
C++	Naive Copy	85.66	67.33	-	67.32	55.44	37.68	36.92	85.02	67.51	-	67.38	54.07	38.47	34.89
	DOBF	43.32	42.25	-	42.03	40.01	49.29	25.77	31.84	37.43	-	48.70	34.05	15.67	23.73
	CodeBERT	63.24	77.21	-	78.39	75.10	70.75	68.92	45.57	64.15	-	56.47	48.65	40.59	56.73
	GCBERT	32.30	81.89	-	69.18	61.26	52.00	62.92	27.21	82.79	-	81.89	76.05	71.13	77.35
	MusT-PT	**87.55**	82.98	-	80.27	81.01	**83.29**	71.20	84.20	81.15	-	79.15	68.85	71.18	64.10
	PSCPT	84.42	**83.72**	-	**82.39**	**81.79**	81.82	**75.79**	**86.38**	**87.56**	-	**87.71**	**80.86**	**84.49**	**81.79**
Java	Naive Copy	69.14	78.03	67.25	-	57.33	33.82	35.50	69.40	78.77	67.48	-	55.99	33.66	33.60
	DOBF	39.21	44.26	38.80	-	40.23	48.87	24.83	32.23	65.02	22.01	-	55.78	35.75	24.90
	CodeBERT	54.98	86.02	79.14	-	78.54	70.21	66.41	46.85	80.88	69.88	-	55.15	47.66	48.56
	GCBERT	28.80	82.07	70.70	-	64.75	51.35	53.73	39.86	91.36	83.97	-	80.12	67.39	73.35
	MusT-PT	81.16	90.13	85.23	-	81.87	80.39	70.06	78.71	89.93	84.28	-	69.53	69.83	61.12
	PSCPT	**81.32**	**91.60**	**85.32**	-	**83.42**	**80.93**	**72.01**	**80.29**	**93.22**	**88.71**	-	**84.07**	**86.13**	**79.41**
JS	Naive Copy	54.58	58.61	55.29	56.61	-	30.44	41.58	53.00	56.70	53.29	54.44	-	31.53	39.77
	DOBF	33.16	41.50	39.91	44.16	-	46.93	24.05	22.13	38.06	20.69	37.78	-	26.03	21.21
	CodeBERT	40.93	75.38	73.83	74.58	-	63.85	60.99	32.88	58.43	50.42	60.13	-	46.14	44.34
	GCBERT	21.69	61.98	60.37	63.08	-	47.72	43.25	28.04	76.27	75.22	75.94	-	62.17	65.83
	MusT-PT	78.54	78.91	78.95	78.03	-	78.69	66.47	70.20	73.32	73.01	73.39	-	76.44	63.88
	PSCPT	**79.14**	**80.38**	**80.58**	**80.84**	-	**79.12**	**66.84**	**72.69**	**83.51**	**77.11**	**80.57**	-	**79.45**	**77.49**
PHP	Naive Copy	37.46	34.99	37.64	33.85	30.66	-	23.67	38.10	35.55	38.47	33.61	32.01	-	23.04
	DOBF	25.78	40.88	38.30	42.98	38.11	-	25.64	17.62	31.95	30.18	25.88	25.89	-	20.80
	CodeBERT	30.06	65.67	67.68	64.02	62.06	-	57.01	30.82	47.14	43.04	45.83	43.45	-	39.42
	GCBERT	14.41	42.45	43.86	44.04	38.05	-	35.85	25.38	61.13	61.49	59.39	53.15	-	59.73
	MusT-PT	76.67	77.96	**79.41**	76.42	**77.64**	-	69.34	67.88	70.34	70.04	67.30	73.54	-	63.97
	PSCPT	**76.87**	**78.25**	76.87	**79.91**	71.54	-	**69.72**	**79.00**	**84.56**	**83.76**	**83.58**	**81.71**	-	**78.01**
Py	Naive Copy	37.77	36.42	36.90	35.24	41.53	23.59	-	35.74	34.31	34.62	33.00	39.79	22.85	-
	DOBF	28.77	34.07	36.23	33.48	30.71	45.68	-	15.47	29.22	24.99	35.64	27.31	28.21	-
	CodeBERT	35.82	61.50	71.06	65.99	62.34	63.73	-	53.17	49.16	57.63	52.93	52.30	45.69	-
	GCBERT	21.75	48.04	55.60	47.63	39.97	44.68	-	28.24	62.08	69.28	63.33	52.79	59.76	-
	MusT-PT	70.64	72.35	75.37	70.89	70.46	75.49	-	58.70	63.23	66.16	64.57	66.47	70.90	-
	PSCPT	**77.05**	**71.62**	**76.77**	**71.52**	**71.34**	**76.55**	-	**69.77**	**81.12**	**75.76**	**80.26**	**72.74**	**77.94**	-

TABLE III: Scalability evaluation of translation backbone in terms of BLEU, where PSC is short for Preserving Semantic Consistency.

Model	Java-Py	Py-Java	Java-C++	C++-Java	Java-C#	C#-Java	Py-C++	C++-Py	Py-C#	C#-Py	C++-C#	C#-C++
LSTM	13.22	17.90	22.35	19.14	12.58	16.17	20.16	13.82	16.93	13.53	18.43	20.81
LSTM+PSC	13.47	19.43	22.56	21.25	16.43	19.35	21.39	14.05	17.81	14.12	24.87	22.56
TransCoder	24.98	21.98	30.09	30.42	44.85	29.40	23.03	23.52	40.40	18.81	41.91	25.30
TransCoder+PSC	39.50	35.64	49.13	57.10	65.86	63.49	41.67	38.68	52.99	38.34	59.45	54.07
Transformer	31.22	38.15	44.38	43.93	47.34	45.60	37.42	33.90	36.91	32.64	45.32	42.65
Transformer+PSC	47.82	53.72	57.39	57.24	57.61	57.69	55.61	48.69	53.35	46.56	57.02	57.53
CodeBERT	48.56	52.93	69.88	56.47	80.88	80.11	57.63	56.73	49.16	47.14	64.15	65.74
PSCPT	**79.41**	**80.26**	**88.71**	**87.71**	**93.22**	**93.99**	**75.76**	**81.79**	**81.12**	**74.66**	**87.56**	**87.23**

that utilizes multilingual languages and exhibits strong generalizability enhances the performance of translation, particularly for low-resource languages.

To compare with these baselines, we follow the best hyper-parameters suggested in their studies. For hyper-parameters in our method, the numbers of transformer layers of the encoder and decoder are set as 12 and 6, respectively. The model dimension and attention heads in transformer layers are set as 768 and 12. To accelerate the training process, the parameters of CodeBERT [29] are utilized to initialize the encoder. The batch size, the number of training epochs, and the trade-off hyper-parameters α, β, and $auto_num$ are determined based on the performance of the validation set, which is set as 64, 200, 2, 3, and 50 in the experiment. Then, the training set and the validation set are mixed up to train the model. Following MusT-PT [11], the snippet-level training set is utilized to enhance the program-level translation. This process is repeated for 3 times and the average performance on the test set is reported. Moreover, the comments in the source code are kept in the dataset the same as Transcoder [3], which can

TABLE IV: The effectiveness of the auto-encoding and semantic-preserving regularization terms for program translation, where SP is short for Semantic-Preserving, and AE is short for Auto-Encoding. The ablation study is dependent on the configuration of three hyper-parameters, α, β, and $auto_num$. When the auto-encoding is ablated, the hyper-parameters α and $auto_num$ are both set to zero. When the semantic-preserving is ablated, the hyper-parameters β are set to zero.

SP	AE	Java-Py	Py-Java	Java-C++	C++-Java	Java-C#	C#-Java	Py-C++	C++-Py	Py-C#	C#-Py	C++-C#	C#-C++
✗	✗	48.65	52.93	69.88	56.47	80.88	80.11	57.63	56.73	49.16	47.14	64.15	65.74
✓	✗	55.83	55.94	80.39	77.52	89.59	88.81	60.84	64.31	53.64	53.32	71.99	78.40
✗	✓	73.24	76.55	85.06	84.98	91.86	91.06	72.80	71.95	72.72	67.18	83.19	82.37
✓	✓	**79.41**	**80.26**	**88.71**	**87.71**	**93.22**	**93.99**	**75.76**	**81.79**	**81.12**	**74.66**	**87.56**	**87.23**

increase the number of anchor points across languages. The AdamW [36] optimizer is used to update model parameters with the initial learning rate 1e-5. The linear weight decay is used for scheduling the learning rate. All the experiments are conducted with the NVIDIA Tesla A100 with 128GB RAM on the Ubuntu operating system.

C. Experiment Results Compared with Other Methods

In the experiment, we use the BLEU [18] score as the evaluation metric to evaluate the n-gram overlap between the translated code and the ground-truth target code, which is the most widely used metric in program translation [11], [12], [35], [37]. A higher BLEU score indicates better evaluation performance, which varies from 0 to 100 as a percentage. Table II shows the experimental results. It can be observed that PSCPT performs significant performance gains over the state-of-the-art approaches on most of the translation pairs, especially at the program level.

At the program level, the BLEU scores of PSCPT are 32.28 higher than the naive copy on average of 42 pairs. Compared with the state-of-the-art, PSCPT achieves more than 9.40% improvement on average in terms of BLEU score in program translation. Especially, compared with MuST [11], PSCPT reduces the error rate by 24.46%, 31.30%, 32.00%, 32.72%, 23.99%, 37.40%, and 28.25% on average in terms of BLEU score in program translation from the source program C, C#, C++, Java, Javascript, PHP, and Python, respectively.

At the snippet level, the BLEU scores of PSCPT are 29.07 higher than the naive copy on average of 42 pairs. Compared with the start-of-the-art method MusT [11], PSCPT has demonstrated superior performance on approximately 90% pairs, which has shown the effectiveness of preserving semantic consistency in translation.

In conclusion, the results have conclusively demonstrated that preserving semantic consistency plays a crucial role in enhancing the performance of code translation. Notably, the improvement in performance is more significant at the program level than the snippet level, highlighting that it is more important for the long program to focus on the semantic information of programs.

D. Scalability Evaluation of Translation Backbone

We further evaluate the scalability of the framework in different backbones for program translation, including LSTM [38], Transformer [39], Transcoder [3], [40], [41],

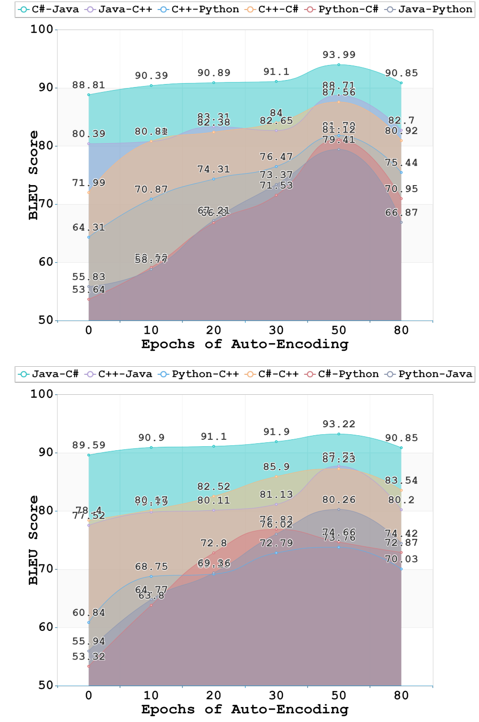


Fig. 3: Performance of PSCPT with different auto-encoding epochs in terms of BLEU. The source and target programming languages in the translation are opposed in the two figures.

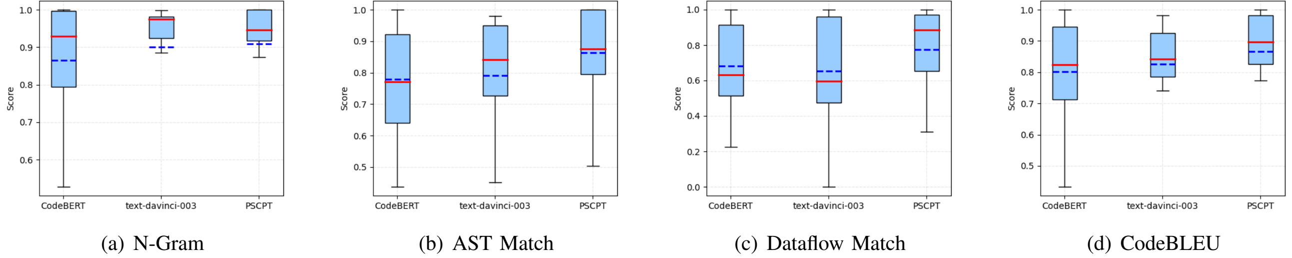
and BERT [29]. As shown in Table III, the transformer-based model is more effective than the LSTM-based model in program translation, which captures internal dependencies within the input sequence while allowing the model to establish long-range relationships between different positions. In addition, supervised models tend to outperform unsupervised models. Supervised models directly learn and model the mapping between source and target programs by leveraging annotated data. In contrast, unsupervised models must discover this mapping without explicit guidance, potentially increasing the difficulty of the problem. No matter based on any backbone, the performance of program translation has significant improvement by preserving semantic consistency on various translation pairs, which illustrates the scalability of our approach in program translation.

E. Ablation Study of Components of PSCPT

We evaluate the effectiveness of the auto-encoding and semantic-preserving regularization terms by trade-off hyper-

TABLE V: Performance evaluation of CodeBERT, PSCPT, and text-davinci-003 on the dataset collected from practical application in terms of N-Gram Match, Weighted N-Gram Match, AST Match, Dataflow Match, CodeBLEU, and Accuracy metrics.

Lang	Model	N-Gram Match (BLEU)	Weighted N-Gram Match	AST Match	Dataflow Match	CodeBLEU	Accuracy
Java-C++	CodeBERT	85.40	86.33	85.70	71.28	82.18	43.48
	text-davinci-002	67.00	67.01	36.75	33.68	51.11	17.85
	text-davinci-003	80.53	81.16	59.47	46.09	66.81	20.02
	PSCPT	**89.69**	**90.16**	**86.56**	**71.71**	**84.53**	**56.52**
C++-Java	CodeBERT	85.05	86.04	73.66	66.64	77.85	39.13
	text-davinci-002	74.06	74.11	46.70	42.01	59.22	19.46
	text-davinci-003	86.77	87.27	73.01	56.81	75.97	19.17
	PSCPT	**88.77**	**89.34**	**84.92**	**75.51**	**84.63**	**47.83**



(a) N-Gram (b) AST Match (c) Dataflow Match (d) CodeBLEU

Fig. 4: Box plots of the performance of CodeBERT, PSCPT, and text-davinci-003 in terms of BLEU, AST Match, Dataflow Match, and CodeBLEU scores in the direction from Java to C++. The value of the Metrics range from 0 to 1, and higher is better. Dashed blue lines indicate the median of the scores, and solid red lines denote the mean of the scores.

parameters α, β, and $auto_num$. The ablation study depends on the configuration of three hyper-parameters, α, β, and $auto_num$. When the auto-encoding is ablated, the hyper-parameters α and $auto_num$ are both set to zero. When the semantic-preserving is ablated, the hyper-parameters β are set to zero. Comparison between the first and second lines in Table IV illustrates the effectiveness of semantic-preserving regularization. And the comparison between the first and the third line in Table IV illustrates the efficacy of auto-encoding regularization. Moreover, a comparison between the second and fourth lines in Table IV indicates that a semantic-enriched latent space constructed by auto-encoding regularization is the premise of semantic-preserving regularization between translated and target programs.

We further evaluate the sensitivity of the auto-encoding with a gradual increase in the $auto_num$ for auto-encoding. As shown in Figure 3, along with epochs of auto-encoding increase, the performance of PSCPT will first rise and then decline in terms of BLEU score. The rising curve proves again that bad auto-encoding will decrease the learning of the semantics of programs. The decline curve shows that the model is overfitting to the noise in the training data.

F. Evaluation of PSCPT in Practical Application

In the experiment, the lexical similarity-based metric (BLEU, Accuracy), syntactic similarity-based metric (AST Match Score), and semantic similarity-based metric (Dataflow Match Score, CodeBLEU) are calculated to evaluate the performance between the translated and the ground-truth target code. Compared with the CodeBERT and PSCPT in Table V of the experimental results, we observe that the major improvement comes from preserving semantic consistency on all sorts of metrics. Especially, the metrics are improved by 12.02% and 2.82% on BLEU and CodeBLEU in the Java-C++ direction and 27.67% and 8.87% in BLEU and CodeBLEU in the C++-Java direction.

The commercial closed large language models are also evaluated on the dataset, where the performance of text-davinci-002 is weaker than text-davinci-003. We invoke the text-davinci-002 and text-davinci-003 API[4] through its official Python bindings. As the large language models can not be fine-tuned, the prompts are given to elicit desired responses for the program translation. In addition, we can observe that the performance of large language models is comparable with PSCPT on BLEU metric, but has obviously lower results on AST Match and Dataflow Match metrics. Although large language models have shown impressive abilities in the generation, there is also no guarantee that the translation will be strictly correct, especially when the program is long. They are predicted and generated by learning statistical regularities in large amounts of data. Therefore, if there are errors, biases, or inaccuracies in the training corpus, the model may replicate these problems during generation. Moreover, the positive linear correlations between accuracy and other metrics are not significant since accuracy is very sensitive to the order of the tokens and must require the translated program and target program exactly the same in the order of tokens.

Figure 4 exhibits the results of box plots of CodeBERT, PSCPT, and text-davinci-003 for BLEU, AST Match, Dataflow Match, and CodeBLEU scores in the direction from Java to C++ for every program. The box plots present the distribution of the results in lexical, syntactic, and semantic metrics. Over-

[4]https://openai.com/api/

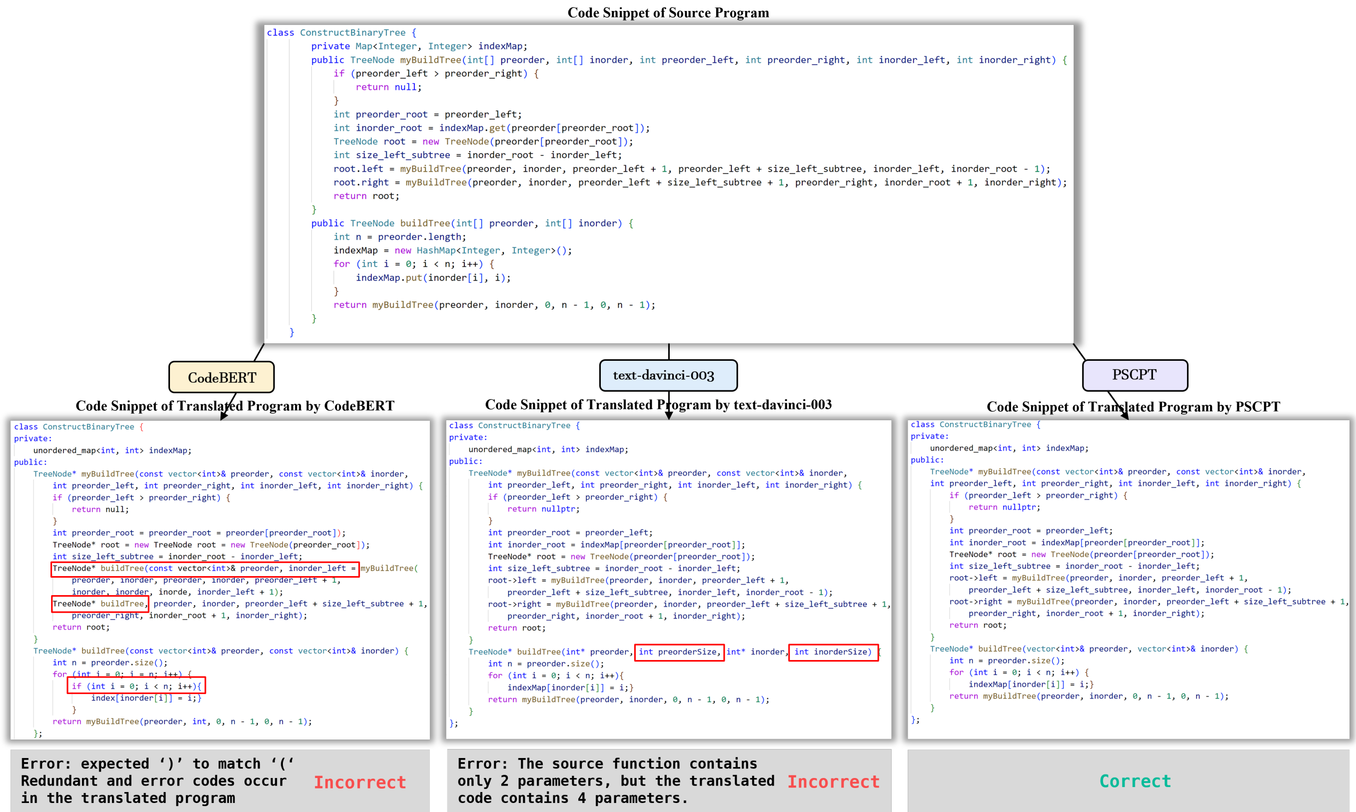


Fig. 5: The case of program translation from Java to C++. The program is to construct and return the binary tree, given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree. The top is the source program in Java. The bottom left, bottom middle, and bottom right are the translated programs in C++ translated by CodeBERT, text-davinci-003, and PSCPT, respectively.

all, PSCPT has a higher upper bound than CodeBERT and text-davinci-003 on all metrics. Compared with text-davinci-003, we can see that PSCPT receives higher scores in AST Match and Dataflow Match. It indicates that programs translated by PSCPT tend to be more similar to target programs at the syntactic and semantic levels than text-davinci-003.

G. *Qualitative Analysis*

For qualitative analysis of our approach, we present the case study containing the source program, and target programs translated by CodeBERT, PSCPT, text-davinci-003, respectively, as shown in Figure 5. The top is the source program in Java. The bottom left, bottom middle, and bottom right are the translated programs in C++ translated by CodeBERT, text-davinci-003, and PSCPT, respectively. In this case, the aim of the Java program is to construct and return the binary tree, given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree. The program translated by PSCPT tends to be more accurate and more readable than the CodeBERT. Moreover, the programs translated by the large language model sometimes contain redundant or incorrect codes. The input of the "buildTree" function in the source program only contains two parameters, but four input parameters are generated from the large language model.

V. Conclusion

In this paper, we argue that semantics are crucial for program translation, which can be used by preserving semantic consistency to improve program translation. We propose an effective regularized framework for program translation named PSCPT, which uses auto-encoding regularization to construct the semantic-enriched latent space and exploit the semantic-preserving regularization to guide the alignment of the semantics in program translation. We conduct extensive experiments in 7 programming languages. Experimental results show that PSCPT outperforms not only the state-of-the-art open-source models but also the commercial closed large language models (e.g., text-davinci-002, text-davinci-003). Extensive experiments prove that preserving semantic consistency in program translation is essential and effective.

In future work, we plan to promote preserving semantic consistency in program translation with more different approaches. In addition, more studies of preserving semantic consistency will be exploited in other software domains.

References

[1] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Advances in Neural Information Processing Systems 31, Montréal, Canada*, pp. 2552–2562, 2018.

[2] Y. Ma, Y. Du, and M. Li, "Capturing the long-distance dependency in the control flow graph via structural-guided attention for bug localization," in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pp. 2242–2250, ijcai.org, 2023.

[3] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems, NeurIPS, virtual*, 2020.

[4] R. C. Waters, "Program translation via abstraction and reimplementation," *IEEE Trans. Software Eng.*, vol. 14, no. 8, pp. 1207–1228, 1988.

[5] W. C. Chu, "A re-engineering approach to program translation," in *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada*, pp. 42–50, IEEE Computer Society, 1993.

[6] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping API elements for code migration with vector representations," in *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016.

[7] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, pp. 81:1–81:37, 2018.

[8] S. Karaivanov, V. Raychev, and M. T. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA*, pp. 173–184, ACM, 2014.

[9] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation*, pp. 651–654, ACM, 2013.

[10] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 585–596, IEEE Computer Society, 2015.

[11] M. Zhu, K. Suresh, and C. K. Reddy, "Multilingual code snippets training for program translation," in *Thirty-Sixth AAAI Conference on Artificial Intelligence*, AAAI Press, 2022.

[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, Virtual Event, Austria*, 2021.

[13] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," in *ICPC*, 2022.

[14] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, "Deep graph matching and searching for semantic code retrieval," *ACM Trans. Knowl. Discov. Data*, 2021.

[15] C. Du, F. Zhuang, Q. He, and Z. Shi, "Multi-task semi-supervised semantic feature learning for classification," in *2012 IEEE 12th International Conference on Data Mining*, pp. 191–200, IEEE, 2012.

[16] X. Huo, Y. Yang, M. Li, and D.-C. Zhan, "Learning semantic features for software defect prediction by code comments embedding," in *2018 IEEE international conference on data mining (ICDM)*, IEEE, 2018.

[17] X. Luo, J. Wu, C. Zhou, X. Zhang, and Y. Wang, "Deep semantic network representation," in *2020 IEEE International Conference on Data Mining (ICDM)*, pp. 1154–1159, IEEE, 2020.

[18] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia, PA, USA*, pp. 311–318, ACL, 2002.

[19] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 574–584, IEEE Computer Society, 2015.

[20] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, Lincoln, NE, USA*, IEEE Computer Society, 2015.

[21] G. Lample, A. Conneau, L. Denoyer, and M. Ranzato, "Unsupervised machine translation using monolingual corpora only," in *6th International Conference on Learning Representations, Vancouver, BC, Canada*, OpenReview.net, 2018.

[22] Y. Wen, Q. Guo, Q. Fu, X. Li, J. Xu, Y. Tang, Y. Zhao, X. Hu, Z. Du, L. Li, C. Wang, X. Zhou, and Y. Chen, "Babeltower: Learning to auto-parallelized program translation," in *International Conference on Machine Learning, Baltimore, Maryland, USA*, PMLR, 2022.

[23] J. D. Weisz, M. J. Muller, S. Houde, J. T. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection not required? human-ai partnerships in code translation," in *IUI '21: 26th International Conference on Intelligent User Interfaces, College Station, TX, USA, April 13-17, 2021*, pp. 402–412, ACM, 2021.

[24] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Virtual Event / Punta Cana, Dominican Republic*, pp. 8696–8708, Association for Computational Linguistics, 2021.

[25] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Vancouver, Canada*, pp. 440–450, Association for Computational Linguistics, 2017.

[26] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," in *7th International Conference on Learning Representations*, OpenReview.net, 2019.

[27] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "Treebert: A tree-based pre-trained model for programming language," in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, Virtual Event*, Proceedings of Machine Learning Research, pp. 54–63, AUAI Press, 2021.

[28] Y. Du and Z. Yu, "Pre-training code representation with semantic flow graph for effective bug localization," *CoRR*, vol. abs/2308.12773, 2023.

[29] Z. Feng, D. Guo, D.-Y. Tang, N. Duan, X.-C. Feng, M. Gong, L.-J. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP, Online Event*, pp. 1536–1547, Association for Computational Linguistics, 2020.

[30] J. Shore and R. Johnson, "Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy," *IEEE Transactions on information theory*, vol. 26, no. 1, pp. 26–37, 1980.

[31] 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.

[32] L. Chai and M. Li, "Pyramid attention for source code summarization," in *Advances in Neural Information Processing Systems*, 2022.

[33] Y. Du, Y. Wei, W. Ji, F. Liu, X. Luo, and L. Nie, "Multi-queue momentum contrast for microvideo-product retrieval," in *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining, WSDM 2023*, pp. 1003–1011, ACM, 2023.

[34] W. James and C. Stein, "Estimation with quadratic loss," *Breakthroughs in statistics: Foundations and basic theory*, pp. 443–460, 1992.

[35] M. Lachaux, B. Rozière, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," in *Advances in Neural Information Processing Systems 34, virtual*, 2021.

[36] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *7th International Conference on Learning Representations,*, OpenReview.net, 2019.

[37] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, virtual*, 2021.

[38] P. An, Z. Wang, and C. Zhang, "Ensemble unsupervised autoencoders and gaussian mixture model for cyberattack detection," *Information Processing & Management*, vol. 59, no. 2, p. 102844, 2022.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, (Red Hook, NY, USA), p. 6000–6010, 2017.

[40] B. Rozière, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," in *ICLR*, 2022.

[41] M. Szafraniec, B. Rozière, H. Leather, F. Charton, P. Labatut, and G. Synnaeve, "Code translation with compiler representations," *CoRR*, vol. abs/2207.03578, 2022.