Automated Snippet-Alignment Data Augmentation for Code Translation

Anonymous ACL submission

Abstract

Code translation aims to translate a piece of code from its source language to the target language. It is widely used in different software development scenarios such as software migration, multilingual development, and system refactoring. With the rapid advancement of Large Language Models (LLMs), researchers have begun applying them to code translation. However, the scarcity of parallel corpora hinders models from learning semantic and syntactic alignment knowledge across programming languages. To address this issue, we propose a data augmentation method that leverages LLMs to automatically generate snippet-alignment data, which can provide more fine-grained syntactic alignment knowledge than program-alignment data. In addition, we also explore two effective training approaches to consistently enhance model performance by leveraging snippet-alignment data. Experiments on the widely used programming languages Python, Java, and C++ demonstrate that our augmented snippet-alignment data and training approaches can lead to further performance improvements compared to fine-tuning only on program-alignment data.

1 Introduction

Code translation aims to translate source code from one programming language to another (Chen et al., 2018). The advancement of automated code translation techniques has enhanced productivity in various software development scenarios, such as: (1) migrating legacy software systems to modern programming languages for better maintainability (e.g., from COBOL to Java (Lachaux et al., 2020)), (2) refactoring code bases to utilize distinctive characteristics of the target language (e.g., from C to its memory-safe alternative, Rust (Hong, 2023; Eniser et al., 2025)), and (3) enabling efficient multilingual development, consequently expanding the applicability of softwares (Macedo et al., 2025).

```python
import math
def sumOfTwoCubes(n):
    lo = 1
    hi = round(math.pow(n, 1 / 3))
    while (lo <= hi):
        curr = (lo*lo*lo + hi*hi*hi)
        if (curr == n):
            return True
...
```

```cpp
#include <bits/stdc++.h>
using namespace std;
bool sumOfTwoCubes(int n) {
    long long int lo = 1;
    long long int hi = (long long int)cbrt(n);
    while (lo <= hi) {
        long long int curr = (lo*lo*lo + hi*hi*hi);
    if (curr == n) return true;
...
```

(a) Program-Alignment

```python
if (curr == n):
        return True
```

```cpp
if (curr == n) return true;
```

(b) Snippet-Alignment

Figure 1: An example of program-alignment data and snippet-alignment data. Though program-alignment data can help models learn semantic alignment knowledge, it is typically extensive in length. Whereas snippet-alignment data itself is shorter and can naturally help models learn syntactic alignment knowledge.

Recent advancements in large language models (LLMs) have demonstrated their strong potential in code translation (Pan et al., 2024; Tao et al., 2024). However, the scarcity of existing parallel corpora in code translation, along with their limited supported languages, has made it challenging for LLMs to sufficiently learn the alignment across programming languages (Zhu et al., 2022a; Ahmad et al., 2023b; Yan et al., 2023; Khan et al., 2024). This has motivated researchers to explore data augmentation techniques for code translation.

From a data granularity perspective, existing par-

allel corpora can be divided into two categories: program-alignment data and snippet-alignment data. Though program-alignment data can help models capture semantic alignment knowledge, the long length of the data makes it challenging for models to learn syntactic alignment knowledge effectively (Pan et al., 2024; Xin-Ye et al., 2025). In contrast, the shorter length and fine-grained characteristics of snippet-alignment data can naturally help models learn syntactic alignment patterns efficiently (Zhu et al., 2022a).

Existing works primarily focus on augmenting program-alignment data (Xie et al., 2023; Chen et al., 2024). For snippet-alignment data, Zhu et al. (2022a) constructed a dataset called XLCoST, containing parallel code snippets collected from GeeksForGeeks website. The construction process was based on the standard template provided by GeeksForGeeks, which limited its generalisability because the template was only used on this specific website.

To overcome these challenges, we propose a novel automated snippet-alignment data augmentation pipeline. In short, this pipeline takes program-alignment data as input and leverages LLMs to generate snippet-alignment data. The design principles of this pipeline draw inspiration from the construction methodology of XLCoST (Zhu et al., 2022a). To ensure the computational resource requirements do not increase substantially as the amount of language increases, we explicitly decompose the generation part of the pipeline into two discrete stages rather than employing a direct end-to-end generation approach. Concretely, the generation part of the pipeline operates in two stages: (1) leveraging LLMs to generate and insert comments into source programs, followed by (2) rewriting target programs that preserve both the content and order of the source-program comments, using the original target program as reference.

We also explore two simple methods to utilize snippet-alignment data: Mix-k and 2-Stage. These two approaches represent distinct data utilization paradigms. The Mix-k approach replaces program-alignment data with snippet-alignment data in a ratio of k to construct a unified training set, whereas the 2-Stage approach employs sequential training, first on one data granularity before transitioning to the other.

Then we conducted a series of experiments on DeepSeek-Coder-Instruct 1.3B/6.7B (Guo et al., 2024), which are widely used code LLMs supporting various programming languages. Experiments on TransCoder-test (Lachaux et al., 2020) demonstrate that our proposed methods and augmented data can consistently enhance model performance compared to fine-tuning only on program-alignment data, measured by pass@1.

Our contributions are summarized as follows:

- We propose an automated snippet-alignment data augmentation pipeline, which can automatically generate snippet-alignment data from program-alignment data.
- We explore two simple methods Mix-k and 2-Stage to leverage snippet-alignment data during the training procedure.
- Experimental results demonstrate that our proposed methods and augmented data can consistently enhance model performance.

2 Related Work

2.1 Data Augmentation for Code Translation

Due to the scarcity of parallel corpora in code translation, researchers have begun to explore data augmentation techniques. Xie et al. (2023) borrowed a term called *comparable corpora* from natural language translation, which referred to texts on similar topics in various languages (Gete and Etchegoyhen, 2022). It proposed three methods to generate comparable corpora, including collecting from existing programming contests, generating from natural language documentations, and retrieving from existing data. It also proposed a method of generating multiple references for the source code by generating several candidates, followed by filtering through test cases, which could provide more diverse training signals for the model. Zhu et al. (2024) first fine-tuned a small model on snippet-alignment data (Zhu et al., 2022a), then leveraged this model to generate parallel corpora from monolingual data, which significantly reduced reliance on parallel corpora. Chen et al. (2024) designed a rule-based method to transform the parallel data into a more diverse dataset. It also proposed a method to retrieve the similar source code and target code from a large code database to construct new parallel data.

However, the existing works mentioned above mainly focus on augmenting program-alignment data, and lack the exploration of snippet-alignment data augmentation. In contrast, our work mainly

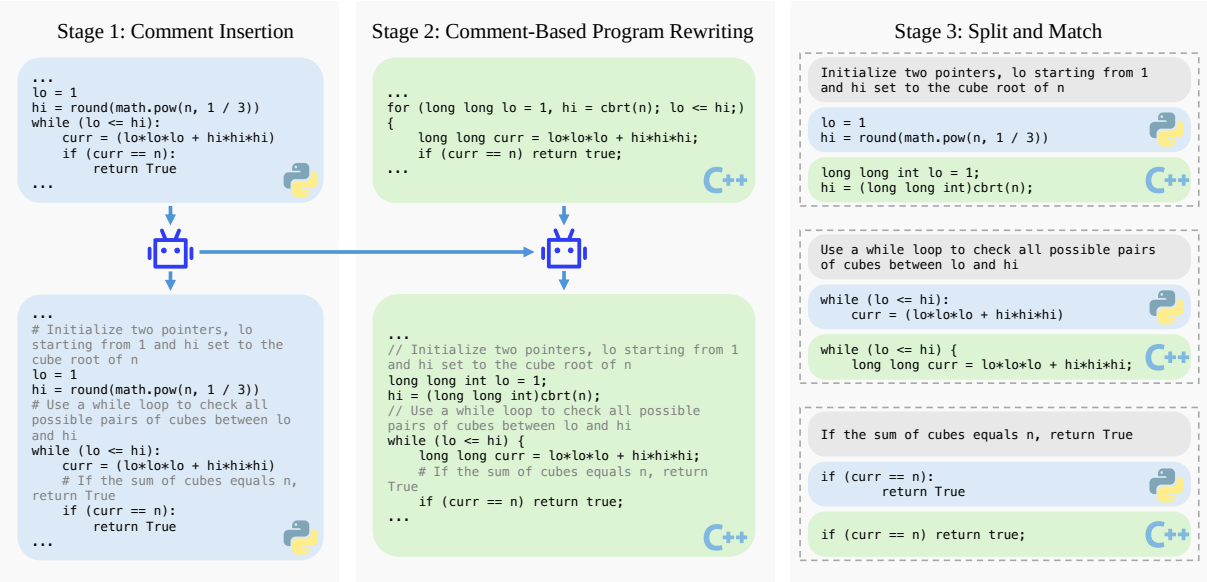

Figure 2: Overview of our data augmentation pipeline. It consists of two LLM-involved stages and a post-processing stage. In **Stage 1** (the left part), the LLM takes the source program as input and outputs it with comments inserted. In **Stage 2** (the middle part), the LLM takes the comment-inserted source program and the original target program as input, and rewrites the target program to preserve the same content and order of comments in the source program. In **Stage 3** (the right part), we can split and match the code snippets according to the comments.

focuses on augmenting snippet-alignment data from the existing program-alignment data, **which bridges this gap in existing works**.

2.2 Snippet-Alignment Datasets in Code Translation

Compared with program-alignment data, snippet-alignment data proves much more challenging to acquire. One of the primary obstacles lies in assessing semantic equivalence for isolated code snippets without contextual information, where program-alignment data can verify its semantic equivalence through test cases.

There are only few existing works that construct snippet-alignment datasets. Zhu et al. (2022b) first constructed a dataset called CoST, a multilingual code snippet translation dataset containing 7 commonly used programming languages. All data in CoST was collected from GeeksForGeeks, which provided a standard template. This ensured that different solutions to the same problem maintained the same set of comments in the same order. Through this template, the authors extracted the code snippets and comments for solutions in different languages, which were then aligned based on the comments. When the template did not function as expected, the authors manually verified the code and either modified it or discarded it. Subsequently, Zhu et al. (2022a) constructed XLCoST, which significantly increased the amount of data available in CoST and extended its coverage to multiple programming tasks, including code translation, code summarization, and code synthesis.

Both datasets employ a template-based segmentation method to derive snippet-alignment data from program-alignment data, though manual verification remains necessary in certain cases. Crucially, this template-dependent approach lacks generalizability, as most program-alignment datasets do not have corresponding pre-defined templates, which motivates us to explore an automated data augmentation method that can be applied to any program-alignment dataset.

3 Automated Snippet-Alignment Data Augmentation

3.1 Design Principles

Before detailing the implementation details of the pipeline, we will first present our design principles. The primary design decision for this pipeline is whether to generate data from scratch or from existing program-alignment data. For one thing, existing parallel data inherently guarantees semantic alignment between programs (Zhu et al., 2022a; Ahmad et al., 2023b; Yan et al., 2023; Khan et al., 2024), therefore, we can mainly focus on establishing the semantic alignment between code snip-

pets. For another, generating data solely by LLM from scratch may lead to low diversity (Wang et al., 2023; Luo et al., 2024), which could be even worse in snippet-alignment data due to its inherently shorter length. Thus, we ultimately elect to generate snippet-alignment data from existing program-alignment data.

However, directly utilizing program-alignment data to generate snippet-alignment data still confronts two major challenges. Firstly, it should be noted that different solutions corresponding to the same problem may use distinct algorithms or data structures, with variations in implementation details. This may prevent code snippets from being matched directly without one of the parallel programs being modified. In other words, rewriting the target program may be necessary before matching to ensure that the internal snippets of the target program align with those of the source program. Secondly, as the number of languages N increases, generating snippet-alignment data for all possible language pairs through a direct end-to-end generation requires $C(N,2) = \frac{N(N-1)}{2}$ iterations, which would lead to a significant increase in computational resources. For a certain problem, if we can align the segmentation patterns of all the other languages with that of a specific language, the number of required iterations will be reduced to N.

Therefore, we decompose the generation part of the pipeline into two stages explicitly. The first stage caches the snippet segmentation result to reduce computational resources. In the second stage, the pipeline gives LLMs the flexibility to adaptively rewrite target programs as needed, thereby resolving internal snippet-alignment issues.

3.2 Pipeline Implementation

Figure 2 illustrates an overview of how to generate snippet-alignment data from the original program-alignment data. Concretely, the proposed pipeline consists of two LLM-involved generation stages and a post-processing stage: (1) automatically generating and inserting code comments for source programs, followed by (2) rewriting target programs that preserve both the content and order of the source-program comments using the original target program as a reference, and (3) splitting the parallel comment-inserted programs and matching corresponding code snippets according to the comments.

The first two stages of the entire pipeline are driven by LLMs, which also serve as its core components. Following these two stages, we can easily obtain snippet-alignment data through the final post-processing stage by employing simple heuristic rules and string processing functions.

All the prompts designed for the first two stages will be provided in the Appendix B.

3.2.1 Comment Insertion

Drawing inspiration from the construction methodology of XLCoST (Zhu et al., 2022a), we chose to use comments as snippet separators because, in contrast to traditional software engineering tools such as Abstract Syntax Tree (AST) and Control Flow Graph (CFG) (Zhong et al., 2024; Du et al., 2025), comments are not language-specific and can take full advantage of the code comprehension capabilities of LLMs (Cui et al., 2024).

Given a source program S, we first prompt an LLM M to generate and insert comments into S. The insertion positions are entirely determined by M, guided solely by the principle of inserting as many comments as possible in order to ensure that each code snippet is not excessively lengthy.

After this stage, M will output a new source program $S' = (s_0, c_1, s_1, c_2, s_2 ... c_k, s_k)$ containing k comments and at most $k+1$ snippets, where k is determined by M, s_i denotes the i-th snippet of S' and c_i denotes the i-th comment of S'. Note that s_0 can have a length of 0, since in some cases the first comment can be inserted at the very beginning. The lengths of the other snippets are non-zero.

3.2.2 Comment-Based Program Rewriting

In the second stage, given a set of target programming languages $L = [l_1, l_2 ... l_m]$ containing m kinds of languages and the original target program set $\tau = [T_1, T_2 ... T_m]$, where l_i denotes the i-th target language and T_i denotes the original target program written in l_i, M takes S' and one original target program T_i as input, and outputs the rewritten program T_i'.

Specifically, M will rewrite T_i based on the content and order of the comments in S', using T_i as a reference. Notably, T_i' may remain identical to T_i when comments are disregarded, provided that M believes that each snippet is already aligned and no rewriting process is required.

After this stage, for each T_i, M will output a new target program $T_i' = \left(t_0^i, c_1, t_1^i, c_2, t_2^i ... c_k, t_k^i\right)$ that theoretically containing exactly the same amount of comments and snippets as S', where t_i^w denotes the i-th snippet of T_w and c_i denotes the i-th comment

Dataset	Size (pairwise)	Granularity
XLCoST-Program	26972	Program
XLCoST-Augmented	135654	Snippet
TransCoder-test	1788	Program

Table 1: Statistical information on training set and test set. The two upper rows represent the training set, while the bottom row corresponds to the test set.

of T_w. Then we will obtain a new target program set $\tau' = [T'_1, T'_2 ... T'_m]$.

3.2.3 Split and Match

After the two stages above, we obtain S' and τ' with identical comment content and comment order theoretically. Suppose S' as another target program T'_{m+1}, then we can formally construct a new target program set $\tau'' = \left[T'_1, T'_2 ... T'_m, T'_{m+1}\right]$, where $T'_{m+1} = S'$.

Then for each pair of target programs T'_i and T'_j, $i, j \in [1, m+1]$, we then extract each t^i_p and t^j_p, $p \in [0, k]$, from T'_i and T'_j according to c_i to construct the final snippet-alignment data. If we find any discrepancy in the number or content of comments between T'_i and T'_j, we will simply ignore this pair and continue the matching procedure.

4 Experiment

4.1 Setting

Datasets. In Table 1, we summarize the statistical information of the datasets used in our experiments. We employ XLCoST (Zhu et al., 2022a) as the training set and the foundational dataset for our data augmentation, which covers 7 commonly used programming languages and contains both program-alignment and snippet-alignment data.

Considering the computational resources, we have chosen Python, Java, and C++ as our basic language setting, as they are widely used in different software development scenarios (Lachaux et al., 2020, 2021; Ahmad et al., 2023a; Xin-Ye et al., 2025), and also cover both dynamic programming languages and static programming languages. For the program-alignment data, we select XLCoST-Program, i.e., the program-alignment data in XLCoST, as the source of program-alignment data used for subsequent training.

For the snippet-alignment data, we employ the pipeline described in Section 3.2 to construct a snippet-alignment dataset XLCoST-Augmented by augmenting the program-alignment data in the XLCoST dataset. Specifically, we utilize DeepSeek-V3-0324 (DeepSeek-AI et al., 2025) through its official API interface throughout the entire pipeline for Comments Insertion and Program Rewriting. After the augmentation and post-processing procedures, we ultimately obtained 135,654 pairs of snippet-alignment data.

Following the previous works (Yang et al., 2024; Du et al., 2025; He et al., 2025), we have conducted a comprehensive set of experiments on TransCoder-test (Lachaux et al., 2020), a widely used public code translation dataset containing 1,788 pairs of program-alignment data in Python, Java, and C++. Each problem includes ten corresponding test cases to verify the semantic equivalence between the translated program and the source program, which enables more accurate measurement of the performance of LLMs in code translation compared with those datasets without test cases (Lu et al., 2021; Zhu et al., 2022a).

Evaluation Metric. Since we have access to the test cases of TransCoder-test, we can directly use the pass@k metric to evaluate the semantic equivalence (Chen et al., 2021). To calculate pass@k, we first generate k code samples for each problem, and a problem is considered passed if at least one of the generated samples passes all the test cases.

Due to the issue of high variance brought by sampling, Chen et al. (2021) refined it into a more stable metric as described in Equation 1.

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

For reproducibility purposes, we set $k = 1$ and employ a greedy decoding strategy, which also reduces resource consumption during inference.

Training Details. For our backbone model, we use DeepSeek-Coder-Instruct 1.3B/6.7B, which are powerful open-source code LLMs supporting various programming languages (Guo et al., 2024). Following previous work (He et al., 2025), we employ instruction tuning on the backbone using a simple code translation instruction template to better align its behaviour with code translation.

For the hyperparameters during training, we set the batch size to 128 for the training set comprising program-alignment data and to 512 for the training set comprising only snippet-alignment data. We also set the max sequence length to 1536 for

	Method	Epoch	J2P	C2P	P2C	J2C	P2J	C2J	AVG
DeepSeek-Coder Instruct 1.3B	Baseline	1	80.84	81.10	76.15	86.92	82.96	87.08	82.51
		2	**83.12**	**82.75**	77.62	87.24	83.12	88.36	83.70
	Mix	1	80.68	81.65	77.98	88.20	83.60	86.92	83.17
	2-Stage	1+1	82.47	79.63	**83.49**	**91.71**	**88.80**	**91.55**	**86.28**
DeepSeek-Coder Instruct 6.7B	Baseline	1	85.23	84.40	80.92	88.84	83.28	90.43	85.52
		2	**86.69**	**86.06**	78.90	89.31	78.25	89.00	84.70
	Mix	1	85.23	84.59	80.37	89.00	85.55	89.63	85.73
	2-Stage	1+1	84.42	80.55	**88.81**	**93.78**	**91.07**	**94.26**	**88.82**

Table 2: Performance comparison between baseline and our proposed methods. The header X2Y denotes translating from language X to Y, where P, J, and C stand for Python, Java, and C++, respectively. AVG stands for the average performance across all language pairs. The Mix method here denotes Mix-0.7, and the 2-Stage method here denotes 2-Stage-PS. We conduct all experiments twice and report the higher of the two results.

program-alignment data and to 1024 for snippet-alignment data. For all the training procedure, we set the learning rate to 2e-5 and the warmup ratio to 0.1, using cosine lr_scheduler_type. We save the model once the entire training process is complete. For all the training, we use 2 NVIDIA A100-SXM4-80GB.

For the baseline, we directly fine-tune the model on program-alignment data. For our methods, there are two different ways to fine-tune the model leveraging our augmented snippet-alignment data. Mix-k denotes that some of the program-alignment data is replaced with the corresponding snippet-alignment data, where k is a hyperparameter that controls the replacement ratio. 2-Stage-XY means that we first train the model on data of one specific granularity X, and then train the model on data of another granularity Y. Here, we use P and S to denote program-alignment data and snippet-alignment data, respectively.

4.2 Main Results

We present the performance comparison of our proposed methods and the baseline in Table 2. We obtain a strong baseline performance simply by fine-tuning the model solely on program-alignment data, and this is typically the paradigm adopted when we want to adapt a code LLM for code translation. Moreover, when we introduce snippet-alignment data, we observe consistent performance gains. Under a one-epoch training setup, a partial replacement of program-alignment data can yield measurable improvements: +0.66% for the 1.3B model and +0.21% for the 6.7B model.

Due to the significant disparities in length and difficulty between the two data granularities, simply mixing snippet data may only yield limited performance gains, which has motivated our 2-Stage approach that explicitly splits the training process into two steps.

Specifically, we first train the model on program-alignment data to learn semantic alignment knowledge, followed by training on snippet-alignment data to learn syntax alignment knowledge. Through this, we observe a significant average performance improvement of +2.58% for the 1.3B model and +3.30% for the 6.7B model, proving the effectiveness of both our augmented data and our 2-Stage approach.

This result is particularly intriguing because the models are ultimately trained on snippet-alignment data yet evaluated on program-alignment data, where a granularity mismatch exists. Despite this discrepancy, this approach still yields consistent performance improvements on average.

Conventionally, optimal evaluation performance is expected when the data granularity of the training set—at least in the final fine-tuning stage—matches that of the test set. More intriguingly, the average performance gains primarily stem from X2J and X2C translations, albeit at the cost of X2P performance. This phenomenon complements the behavior of baseline: when increasing training epochs from 1 to 2, X2P improves consistently—even as the average performance of 6.7B model declines in the second epoch. This may suggest that continued training on program-alignment data could further improve the performance of X2P, whereas snippet-alignment data could preferentially boost the performance of X2J and X2C.

4.3 Analysis

Granularity Order of 2-Stage Training. To further investigate how the training order of different

	Epoch	Order	J2P	C2P	P2C	J2C	P2J	C2J	AVG
DeepSeek-Coder Instruct 1.3B	1	P	**80.84**	**81.10**	76.15	86.92	82.96	87.08	82.51
		S	76.79	76.15	**82.75**	**90.59**	**86.53**	**89.79**	**83.77**
	2	PP	**83.12**	**82.75**	77.62	87.24	83.12	88.36	83.70
		SP	82.79	**82.75**	77.25	86.92	83.44	88.20	83.56
		SS	79.06	78.72	82.20	89.95	86.53	90.43	84.48
		PS	82.47	79.63	**83.49**	**91.71**	**88.80**	**91.55**	**86.28**
DeepSeek-Coder Instruct 6.7B	1	P	**85.23**	**84.40**	80.92	88.84	83.28	90.43	85.52
		S	83.12	79.82	**87.52**	**92.66**	**91.07**	**94.26**	**88.08**
	2	PP	**86.69**	**86.06**	78.90	89.31	78.25	89.00	84.70
		SP	85.71	84.59	81.47	89.00	84.74	89.95	85.91
		SS	83.60	81.47	88.26	92.98	**91.23**	93.62	88.53
		PS	84.42	80.55	**88.81**	**93.78**	91.07	**94.26**	**88.82**

Table 3: Performance comparison of all order combinations of snippet-alignment and program-alignment data.

data granularities affects model performance, we conduct a series of experiments to evaluate the performance among different order combinations of data granularities.

Table 3 shows the performance comparison of all the order combinations. When fixing the number of training epochs to 1, we observe that training solely on snippet-alignment data yields an average performance improvement of +1.26% for the 1.3B model and +2.56% for the 6.7B model. When we extend the training epochs to 2, we consistently observe a similar phenomenon in the results.

This aligns with our observations in Section 4.2, where snippet-alignment data is shown to enhance model performance specifically for X2J and X2C translations.

However, training solely on snippet-alignment data fails to yield optimal results, necessitating the incorporation of program-alignment data. As a result, we can observe that 2-Stage-PS improves the X2P performance to some extent compared with 2-Stage-SS trained solely on snippet-alignment data, and also increases the performance on X2C and X2J slightly, further illustrating that the training process should make full use of data of different granularities rather than relying on data of a single granularity.

Snippet Training as a Powerful Foundational Training Stage. From the results in Table 3, we can observe that training on snippet-alignment data before program-alignment data (2-Stage-SP) can yield an average performance improvement compared with training solely on program-alignment data. This leads us to investigate that how model performance would vary if we add just one epoch of snippet-alignment training before the program-alignment training stage while holding the number of program-alignment training epochs constant?

The results in Table 4 indicate that adding only one epoch of snippet-alignment training prior to the program-alignment training stage can consistently improve model performances for almost all language combinations.

Besides, while the model exhibits overfitting as training epochs increase, especially for the 6.7B model, introducing snippet-alignment data may mitigate performance degradation—reducing the loss margin from -0.98% to -0.46% for the 1.3B model. For the 6.7B model, when P-Epoch is 2, the performance of the model can grow further rather than decline.

Moreover, even with the same total number of training epochs, SPP still significantly outperforms PPP by +1.76% for the 1.3B model and +4.56% for the 6.7B model. SP outperforms PP by +1.21% for the 6.7B model; for the 1.3B model, its performance is comparable to that of PP.

Overall, the results above demonstrate both the effectiveness of our generated snippet-alignment data and the promising potential of regarding snippet-alignment training as a powerful foundational training stage for achieving consistent model performance gains.

Data Quality of LLM-Augmented Snippets. To further investigate the quality of LLM-Augmented snippet-alignment data, we present detailed statistical information of XLCoST-Augmented in Table 5. After the augmentation process, we obtain XLCoST-Augmented-Initial containing 139,556 pairs of snippet-alignment data. We then perform a post-processing stage on these data, including extracting the generated code from the output of

	P-Epoch	Order	J2P	C2P	P2C	J2C	P2J	C2J	AVG
DeepSeek-Coder Instruct 1.3B	1	P	80.84	81.10	76.15	**86.92**	82.96	87.08	82.51
		SP	**82.79**	**82.75**	**77.25**	**86.92**	**83.44**	**88.20**	**83.56**
	2	PP	83.12	82.75	77.62	**87.24**	83.12	88.36	83.70
		SPP	**83.60**	**84.22**	**79.82**	86.76	**83.93**	**88.52**	**84.48**
	3	PPP	82.96	81.65	76.88	**87.40**	80.52	86.92	82.72
		SPPP	**83.44**	**84.59**	**78.35**	86.92	**81.82**	**89.00**	**84.02**
DeepSeek-Coder Instruct 6.7B	1	P	85.23	84.40	80.92	88.84	83.28	**90.43**	85.52
		SP	**85.71**	**84.59**	**81.47**	**89.00**	**84.74**	89.95	**85.91**
	2	PP	**86.69**	86.06	78.90	**89.31**	78.25	89.00	84.70
		SPP	86.53	**86.97**	**82.02**	88.84	**84.58**	**89.79**	**86.46**
	3	PPP	85.55	83.49	74.68	88.52	72.24	86.92	81.90

Table 4: Performance comparison of adding only one epoch of snippet-alignment training prior to the program-alignment training stage.

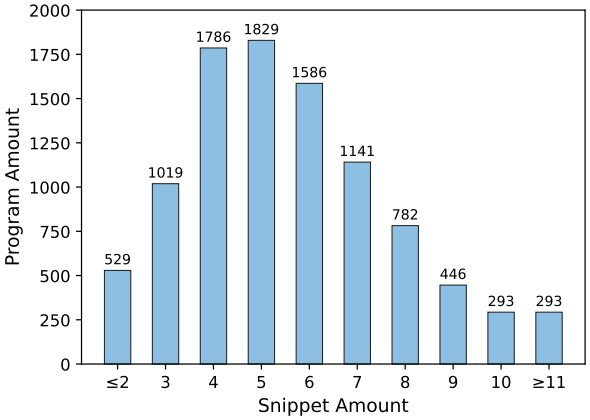

Figure 3: The distribution of the amount of code snippets in source programs after the Comment Insertion stage described in Section 3.2.1.

Dataset	Size	Percentage
XLCoST-Augmented-Initial	139556	100%
- Parsing Error	24	~0%
- Comments Not Match Error	425	0.31%
- Filtering Error	3453	2.47%
XLCoST-Augmented	135654	97.20%

Table 5: Statistical information of data filtered by the entire post-processing stage.

LLM, comparing the comment content and order of parallel data, and filtering out snippets that contains useless information (e.g., those consisting solely of *import* or *#include* statements).

Finally, we have discarded 3902 pairs of snippet-alignment data, and the overall data usability rate is 97.2%, demonstrating high quality of LLM-Augmented data and high effectiveness of our augmentation pipeline.

Moreover, Figure 3 illustrates the distribution of the number of code snippets of source programs following the Comment Insertion stage outlined in Section 3.2.1. The total amount of original source program is 9,704.

After our statistics, the average number of lines of code in the original source programs is 21.72 lines, and approximately 84% source programs are inserted with 3 to 8 comments after the Comment Insertion stage, with the number of average code snippets being 5.63. These statistics demonstrate that the two LLM-involved augmentation stages function as expected and show great potential for generating large amounts of snippet-alignment data.

5 Conclusion

In this paper, we propose an automated snippet-alignment data augmentation method, which can generate snippet-alignment data from program-alignment data. It first utilizes LLMs to insert comments into the source program and then rewrites the target program to ensure internal alignment of the parallel data. After these two steps, snippet-alignment data can be obtained by extracting and matching code snippets based on the content and order of the shared comments. Moreover, we explore two different methods to utilize snippet-alignment data during training. Experimental results demonstrate that our augmented data and the proposed methods can consistently improve model performance. A key observation is that training solely on data of single data granularity can only lead to a suboptimal performance. In contrast, fully leveraging both program-alignment data and snippet-alignment data can achieve optimal performance on average.

Limitations

Firstly, due to the limitation of computational resources, we did not validate the effectiveness of our proposed data augmentation pipeline on more datasets. The XLCoST dataset we use already contains snippet-alignment data, which makes it easier for LLMs to perform relevant augmentation steps such as Comment Insertion and Program Rewriting. We have yet to explore how LLMs ultimately perform when the data is more heterogeneous, i.e., having huge differences in implementation details or length. Secondly, we have only conducted our experiments on DeepSeek-Coder-Instruct of 1.3B and 6.7B, and have not yet verified the validity of our method or data on other models. Thirdly, although our proposed method 2-Stage-PS is optimal in terms of average performance, it is not optimal in every language setting, and in particular, it sacrifices the performance of X2P, where we have not yet delved into the reasons for the performance degradation.

Ethics Statement

All of the models used in this paper are publicly available, and we have used them in accordance with their respective licences and terms of use.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023a. Summarize and generate to back-translate: Unsupervised translation of programming languages. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1528–1542, Dubrovnik, Croatia. Association for Computational Linguistics.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023b. AVATAR: A parallel corpus for Java-python program translation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 2268–2281, Toronto, Canada. Association for Computational Linguistics.

Binger Chen, Jacek Golebiowski, and Ziawasch Abedjan. 2024. Data augmentation for supervised code translation learning. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 444–456, New York, NY, USA. Association for Computing Machinery.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Jielun Cui, Yutong Zhao, Chong Yu, Jiaqi Huang, Yuanyuan Wu, and Yu Zhao. 2024. Code comprehension: Review and large language models exploration. In *2024 IEEE 4th International Conference on Software Engineering and Artificial Intelligence (SEAI)*, pages 183–187.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 181 others. 2025. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Yali Du, Hui Sun, and Ming Li. 2025. Post-incorporating code structural knowledge into llms via in-context learning for code translation. *Preprint*, arXiv:2503.22776.

Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2025. Towards translating real-world code with llms: A study of translating to rust. *Preprint*, arXiv:2405.11514.

Harritxu Gete and Thierry Etchegoyhen. 2022. Making the most of comparable corpora in neural machine translation: a case study. *Lang. Resour. Evaluation*, 56(3):943–971.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Minghua He, Fangkai Yang, Pu Zhao, Wenjie Yin, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025. Execoder: Empowering large language models with executability representation for code translation. *Preprint*, arXiv:2501.18460.

Jaemin Hong. 2023. Improving automatic c-to-rust translation with static analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 273–277.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting*

of the Association for Computational Linguistics (Volume 1: Long Papers), pages 6766–6805, Bangkok, Thailand. Association for Computational Linguistics.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Preprint*, arXiv:2006.03511.

Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. In *Advances in Neural Information Processing Systems*, volume 34, pages 14967–14979. Curran Associates, Inc.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. 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, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Xu Wang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024. Semi-instruct: Bridging natural-instruct and self-instruct for code large language models. *Preprint*, arXiv:2403.00338.

Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R. Cogo, and Bram Adams. 2025. Intertrans: Leveraging transitive intermediate translations to enhance LLM-based code translation. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the potential of large language models in code translation: How far are we? In *31st Asia-Pacific Software Engineering Conference, APSEC 2024, Chongqing, China, December 3-6, 2024*, pages 353–362. IEEE.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada. Association for Computational Linguistics.

Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. 2023. Data augmentation for code translation with comparable corpora and multiple references. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13725–13739, Singapore. Association for Computational Linguistics.

Li Xin-Ye, Du Ya-Li, and Li Ming. 2025. Enhancing llms in long code translation through instrumentation and program state alignment. *Preprint*, arXiv:2504.02017.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5067–5089, Singapore. Association for Computational Linguistics.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1(FSE).

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022a. Xlcost: A benchmark dataset for cross-lingual code intelligence. *Preprint*, arXiv:2206.08474.

Ming Zhu, Ismini Lourentzou, and Danfeng Yao. 2024. Alignment-enhancing parallel code generation for semi-supervised code translation.

Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022b. Multilingual code snippets training for program translation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(10):11783–11790.

A Case Study.

We will show some examples of programs that the baseline fail to translate, but our model, trained by the 2-Stage method, can translate successfully. In the figures, we have marked the wrong codes in red font and the correctly translated codes in green font. We can find that, in Figure 4, the baseline model misinterprets the meaning of string subtraction in Java, while the 2-Stage model, trained with the snippet-alignment data, faithfully translates string subtraction into Unicode subtraction. In Figure 5, the model misinterprets the meaning of the assignment statement and simplifies it incorrectly, while our model faithfully completes the translation of the assignment statement These cases of failure caused by a simple syntax error highlights the importance of snippet training.

B Pipeline Prompt

We will demonstrate the prompt template that we have designed for the entire pipeline described in Section 3.2.

```
<original_Java_program>
...
static int f_gold(String str) {
    int res = str.charAt(0) - '0';
    for (int i = 1; i < str.length(); i++) {
        if (str.charAt(i) == '0'  str.charAt(i) == '1'  res < 2) {
            res += (str.charAt(i) - '0');
        } else {
            res *= (str.charAt(i) - '0');
        }
    }
    return res;
}
...

<translated_Python_program_by_Baseline>
...
def f_filled ( str ) :
    res = int ( str [ 0 ] )
    for i in range ( 1 , len ( str ) ) :
        if ( str [ i ] == '0' or str [ i ] == '1' or res < 2 ) :
            res += int ( str [ i ] )
        else :
            res *= int ( str [ i ] )
    return res
...

<translated_Python_program_by_2-Stage>
...
def f_filled ( str ) :
    res = ord ( str [ 0 ] ) - ord ( '0' )
    for i in range ( 1 , len ( str ) ) :
        if ( str [ i ] == '0' or str [ i ] == '1' or res < 2 ) :
            res += ( ord ( str [ i ] ) - ord ( '0' ) )
        else :
            res *= ( ord ( str [ i ] ) - ord ( '0' ) )
    return res
...
```

Figure 4: A J2P problem where baseline fails to translate and our 2-Stage model translates it successfully.

```
<original_Java_program>
...
static int f_gold(int arr[], int n) {
  Map<Integer, Integer> um = new HashMap<>();
  int curr_sum = 0;
  for (int i = 0; i < n; i++) {
    curr_sum += (arr[i] == 0) ? -1 : arr[i];
    um.put(curr_sum, um.get(curr_sum) == null ? 1 : um.get(curr_sum) + 1);
  }
...
}
<translated_Python_program_by_Baseline>
...
def f_filled ( arr , n ) :
    um = { }
    curr_sum = 0
    for i in range ( n ) :
        curr_sum += ( arr [ i ] == 0 ) - 1 or arr [ i ]
        if curr_sum in um :
            um [ curr_sum ] += 1
        else :
            um [ curr_sum ] = 1
...

<translated_Python_program_by_2-Stage>
...
def f_filled ( arr , n ) :
    um = { }
    curr_sum = 0
    for i in range ( n ) :
        curr_sum += - 1 if arr [ i ] == 0 else arr [ i ]
        if curr_sum in um :
            um [ curr_sum ] += 1
        else :
            um [ curr_sum ] = 1
...
```

Figure 5: Another J2P problem where baseline fails to translate and our 2-Stage model translates it successfully.

```
You are an expert in code summarization. You will be provided with a piece of code, and your
    task is to add a few comments within the code based on its overall functionality.
These comments should follow the restrictions below:
1. The position of these comments must be between lines of code. DO NOT add the comment at the
    end of a particular line of code.
2. Each comment should correspond to several lines of code. A comment corresponding to only a
    single line of code is forbidden.
3. These comments should fully cover all lines of code, and the code segments corresponding to
    different comments must not overlap, i.e. each line of code must have only one
    corresponding comment.
4. Control the number of comments to be added, if the code is long you can add more comments.
    Ensure that a programming beginner can clearly understand every segment of the code by
    reading the comments.

The Input Code is marked with <Code> and </Code>, and your Output Code should also be marked
    with <Code> and </Code>.

Here is an example:
Input Code:
<example_source_program>

First of all, you can add some comment tags <ct> to the positions where you think comments are
    needed, which can help you break down this task.
Remember that the position of these comment tags must be between lines of code.
And remember, these comments should fully cover all lines of code, and the code segments
    corresponding to different comments must not overlap, i.e. each line of code must have only
    one corresponding comment.
You should also control the number of comments to be added, if the code is long you can add
    more comments.
Since the Input Code may not be formatted, you should **format** the Input Code before
    inserting the comments, so that it can be more readable.

Here is the edited code:
<edited_example_source_program>

After this step, you can replace the comment tags with real comments.
Each comment should explain the functionality of the corresponding code segment. The comment
    could have been written more thoroughly if the corresponding code segment is long.

Output Code:
<example_source_program_with_comment_inserted>

Here is the <src_lan> code to add comments. Please follow the restrictions and procedures
    above, and you only need to return the Output Code.
Input Code:
<Code>
<tag>
</Code>
```

Figure 6: The prompt designed for Comment Insertion stage described in Section 3.2.1. Due to length constraints, we have omitted the example provided for the model in the template

<example_source_program> can be replaced by the following example:
<Code>
```
def maxPresum ( a , b ) :
    X = max ( a [ 0 ] , 0 )
    for i in range ( 1 , len ( a ) ) :
        a [ i ] += a [ i - 1 ]
        X = max ( X , a [ i ] )
    Y = max ( b [ 0 ] , 0 )
    for i in range ( 1 , len ( b ) ) :
        b [ i ] += b [ i - 1 ]
        Y = max ( Y , b [ i ] )
    return X + Y
A = [ 2 , - 1 , 4 , - 5 ]
B = [ 4 , - 3 , 12 , 4 , - 3 ]
print ( maxPresum ( A , B ) )
```
</Code>

<edited_example_source_program> can be replaced by the following example:
<Code>
```
<ct>
def maxPresum(a, b):
    X = max(a[0], 0)
    Y = max(b[0], 0)
    <ct>
    for i in range(1, len(a)):
        a[i] += a[i - 1]
        X = max(X, a[i])
    <ct>
    for i in range(1, len(b)):
        b[i] += b[i - 1]
        Y = max(Y, b[i])
    return X + Y
<ct>
A = [2, -1, 4, -5]
B = [4, -3, 12, 4, -3]
print(maxPresum(A, B))
```
</Code>

<example_source_program_with_comment_inserted> can be replaced by the following example:
<Code>
```
# To compute the sum of the maximum prefix sum of two arrays, we first initialize X and Y as
     the maximum of the first element of the array A [] and B [] or 0.
def maxPresum(a, b):
    X = max(a[0], 0)
    Y = max(b[0], 0)
    # Iterate through the array A [] to compute the maximum prefix sum of array A [].
    for i in range(1, len(a)):
        a[i] += a[i - 1]
        X = max(X, a[i])
    # Iterate through the array B [] to compute the maximum prefix sum of array B []. Then
     return the sum of the maximum prefix sum of two arrays.
    for i in range(1, len(b)):
        b[i] += b[i - 1]
        Y = max(Y, b[i])
    return X + Y
# Create two arrays to test the above function
A = [2, -1, 4, -5]
B = [4, -3, 12, 4, -3]
print(maxPresum(A, B))
```
</Code>

Figure 7: The concrete examples omitted in Figure 6.

````
You are an expert in code translation. You will be provided with two pieces of code that
    implement the same functionality but are written in two different programming languages.
The first piece of code contains several comments that can be used to break the code into
    smaller segments, and each comment describes the functionality of the corresponding segment
    of code, while the second piece of code doesn't have any comments.
Your task is to regenerate the second piece of code with comments based on the first piece of
    code and its comments, using the second piece of code as a reference. Specifically, you
    need to translate the first piece of code segment by segment from top to bottom.
Since the second piece of reference code is functionally identical to the first one, you should
    prioritize copying the relevant parts of the second piece of code during the translation
    process.
Since I will later match the corresponding segments of the two pieces of code based on the
    comments, the comments in the second piece of code must match those in the first piece of
    code exactly in quantity, content, and order.

Here is an example of how to do this. The codes below are written in Python and C++, and the
    Python code is provided with comments:
<example_source_program_with_comment>

<example_orginal_target_program>

Since the second piece of code may not be formatted, you should format the second piece of code
    before the translation process, so that it can be more readable and be easily divided into
    segments.
Then we can regenerate the second piece of code segment by segment based on the comments of the
    first piece of code, using the second piece of code as a reference:
<example_rewritten_target_program>

Since the reference code could not be aligned well with the first piece of code, we modified
    the second piece of code to ensure strict alignment with the first, based on the number,
    content and order of the comments from the first piece of code.
In this example, some lines of the reference code can be reused, but you are still allowed to
    generate entirely new code, as long as you ensure that the second piece of code matches the
    first segment by segment.

Here are the codes of this task:
```<lan_tag_1>
<tag1>
```

```<lan_tag_2>
<tag2>
```
````

Figure 8: The prompt designed for Comment-Based Program Rewriting stage described in Section 3.2.2. Due to length constraints, we have omitted the example provided for the model in the template

<example_source_program_with_comment> can be replaced by the following example:
```python
# To compute the sum of the maximum prefix sum of two arrays, we first initialize X and Y as
    the maximum of the first element of the array A [] and B [] or 0.
def maxPresum(a, b):
    X = max(a[0], 0)
    Y = max(b[0], 0)
    # Iterate through the array A [] to compute the maximum prefix sum of array A [].
    for i in range(1, len(a)):
        a[i] += a[i - 1]
        X = max(X, a[i])
    # Iterate through the array B [] to compute the maximum prefix sum of array B []. Then
    return the sum of the maximum prefix sum of two arrays.
    for i in range(1, len(b)):
        b[i] += b[i - 1]
        Y = max(Y, b[i])
    return X + Y
# Create two arrays to test the above function
A = [2, -1, 4, -5]
B = [4, -3, 12, 4, -3]
print(maxPresum(A, B))
```

<example_orginal_target_program> can be replaced by the following example:
```cpp
#include <bits/stdc++.h>
using namespace std;

int maxPresum(vector<int> a, vector<int> b) {
    int X = max(a[0], 0);
    int i = 1;
    while (i < a.size()) {
        a[i] += a[i - 1];
        X = max(X, a[i]);
        i++;
    }

    int Y = max(b[0], 0);
    i = 1;
    while (i < b.size()) {
        b[i] += b[i - 1];
        Y = max(Y, b[i]);
        i++;
    }

    return X + Y;
}

int main() {
    vector<int> A = {2, -1, 4, -5};
    vector<int> B = {4, -3, 12, 4, -3};
    cout << maxPresum(A, B) << endl;
}
```

Figure 9: The first two concrete examples omitted in Figure 8.

<example_rewritten_target_program> can be replaced by the following example:
```cpp
// To compute the sum of the maximum prefix sum of two arrays, we first initialize X and Y as
     the maximum of the first element of the array A [] and B [] or 0.
#include <bits/stdc++.h>
using namespace std;
int maxPresum(vector<int> a, vector<int> b) {
    int X = max(a[0], 0);
    int Y = max(b[0], 0);
    // Iterate through the array A [] to compute the maximum prefix sum of array A [].
    for (int i = 1; i < a.size(); i++) {
        a[i] += a[i - 1];
        X = max(X, a[i]);
    }
    // Iterate through the array B [] to compute the maximum prefix sum of array B []. Then
     return the sum of the maximum prefix sum of two arrays.
    for (int i = 1; i < b.size(); i++) {
        b[i] += b[i - 1];
        Y = max(Y, b[i]);
    }
    return X + Y;
}
// Create two arrays to test the above function
int main() {
    vector<int> A = {2, -1, 4, -5};
    vector<int> B = {4, -3, 12, 4, -3};
    cout << maxPresum(A, B) << endl;
}
```

Figure 10: The last concrete example omitted in Figure 8.