UniCoder: Scaling Code Large Language Model via Universal Code

Tao Sun[1,*], **Linzheng Chai**[1,*], **Jian Yang**[1,*,†], **Yuwei Yin**[2], **Hongcheng Guo**[1], **Jiaheng Liu**[1], **Bing Wang**[1], **Liqun Yang**[1], **Zhoujun Li**[1]
[1]State Key Laboratory of Complex & Critical Software Environment, Beihang University;
[2]Department of Computer Science, University of British Columbia
{buaast, challenging, jiaya, hongchengguo}@buaa.edu.cn;
{liujiaheng, bingwang, lqyang, lizj}@buaa.edu.cn; yuweiyin@cs.ubc.ca

Abstract

Intermediate reasoning or acting steps have successfully improved large language models (LLMs) for handling various downstream natural language processing (NLP) tasks. When applying LLMs for code generation, recent works mainly focus on directing the models to articulate intermediate natural-language reasoning steps, as in chain-of-thought (CoT) prompting, and then output code with the natural language or other structured intermediate steps. However, such output is not suitable for code translation or generation tasks since the standard CoT has different logical structures and forms of expression with the code. In this work, we introduce the universal code (**UniCode**) as the intermediate representation. It is a description of algorithm steps using a mix of conventions of programming languages, such as assignment operator, conditional operator, and loop. Hence, we collect an instruction dataset UniCoder-Instruct to train our model UniCoder on multi-task learning objectives. UniCoder-Instruct comprises natural-language questions, code solutions, and the corresponding universal code. The alignment between the intermediate universal code representation and the final code solution significantly improves the quality of the generated code. The experimental results demonstrate that UniCoder with the universal code significantly outperforms the previous prompting methods by a large margin, showcasing the effectiveness of the structural clues in pseudo-code.[1]

1 Introduction

The field of code translation and generation has advanced significantly (Szafraniec et al., 2023; Yan et al., 2023) with the advent of code-specific large language models (LLMs). Code LLMs, such as StarCoder (Li et al., 2023b) and Code-Llama (Rozière et al., 2023), are capable of generating executable code by analyzing natural language prompts. Chain-of-thought (CoT) prompting (Wei et al., 2022b) has emerged as the leading technique in enhancing LLMs, where the intermediate steps provide a structured pathway from the problem statement to the solution, effectively mirroring the human problem-solving process.

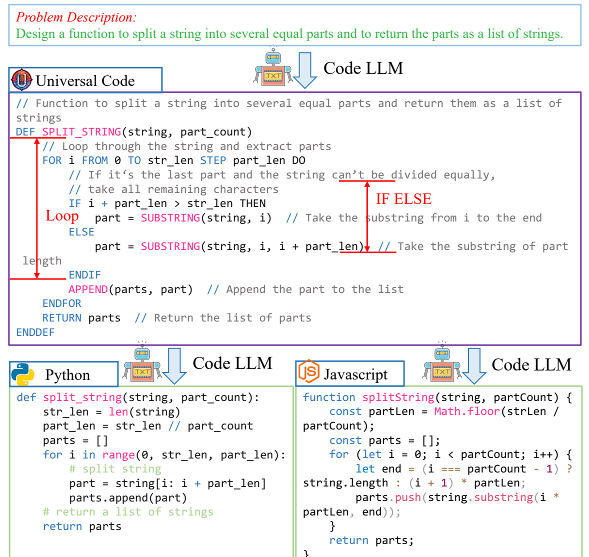

Figure 1: An example of UniCoder. The Code LLM solves the code generation question by "translating" the pseudocode description (Universal Code) into executable code of the target programming language.

Considering the low accuracy of CoT in coder generation, structure CoT (SCoT) (Li et al., 2023a) is proposed to minimize the gap between the intermediate steps and the generated code. More intuitively, using a universal code as the intermediate representation to handle multiple programming languages (PL) is promising. Here, universal code is a blueprint for implementing an algorithm, which helps to make the design of algorithms logically clear and readily comprehensible. Moreover, it is universal across different programming languages (PL-agnostic) since it typically does not follow spe-

[*]Equal contribution.
[†]Corresponding Author.
[1]https://github.com/ASC8384/UniCoder

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1812–1824
August 11-16, 2024 ©2024 Association for Computational Linguistics

cific syntax and omits execution details. Yet, *how the universal code is used for code translation and generation in multilingual scenarios remains underexplored.*

In this work, we scale up the code LLMs to support multiple programming languages via the universal code (**UniCode**), which is used as an efficient and language-independent intermediate representation of the key algorithm principles. Specifically, we first define **UniCode** by specifying grammar rules and providing paradigms, followed by prompting GPT-4 (OpenAI, 2023) to create an instruction dataset UNICODER-INSTRUCT comprising natural-language questions, code solutions, and the corresponding universal code, as shown in Figure 1. Then, the UNICODER model is built by performing instruction tuning (Wei et al., 2022a) on multi-task learning objectives, including zero-shot question-answer generation (question→code), question-universal-code generation (question→**UniCode**→code), universal-code-solution translation (**UniCode**→code), and Universal-code-of-Thought (UoT) objectives. In UoT, the model is required to generate the universal code before the executable code.

UNICODER is evaluated on the Python benchmark (Humaneval (Chen et al., 2021) and MBPP (Austin et al., 2021)) and the extended multilingual benchmark MultiPL-E. The results demonstrate that UNICODER consistently achieves state-of-the-art performance across all languages, notably surpassing the previous baselines. Furthermore, the ablation study verifies the efficacy of the proposed method, and extra discussions provide insights into the effect of our method. The contributions are summarized as follows:

- We introduce the universal code **UniCode**, which is agnostic to programming languages, allowing LLMs to grasp the essence of algorithms step by step. In addition, the instruction dataset UNICODER-INSTRUCT is collected and provided for follow-up research.
- We propose UNICODER, a code generation method that uses multi-task learning objectives to fine-tune the code LLMs with the help of **UniCode**. The objectives include question-answer generation (QA), question-universal-code generation (QP), universal-code-answer translation (PA), and Universal-code-of-Thought (UoT).
- As extensive experiments show, our method

````
Definition of Universal Code:
1. Comments: Use `//` for single-line comments and `/* ... */` for multi-line comments.
2. Variables: Choose clear, type-free names for variables.
3. Input/Output: Keep input/output straightforward.
4. Conditionals: Employ `IF`, `ELSIF`, and `ELSE` with proper indentation.
5. Loops: Use `FOR`, `WHILE`, or `DO...WHILE` loops, specifying conditions and indenting code.
6. Functions/Procedures: Name them descriptively and consider parameters.
7. Formatting: Maintain consistent 2-4 space indentation for clarity.

Example:
```pseudocode
//This is the QuickSort algorithm which sorts an array by recursively partitioning it around a pivot.
QUICKSORT(Arr[], LOW, HIGH) {
   if (LOW < HIGH) {
       PIVOT = PARTITION(Arr, LOW, HIGH);
       QUICKSORT(ARR, LOW, PIVOT - 1);
       QUICKSORT(ARR, PIVOT + 1, HIGH);
   }
}
```
````

Figure 2: Definition of the universal code.

UNICODER consistently outperforms the previous baselines on different benchmarks, including HumanEval, MBPP, and MultiPL-E. To further verify the effectiveness of the universal code, we propose UNICODER-BENCH to test the capabilities of code LLMs.

2 UNICODER-INSTRUCT

Definition of Universal Code. Universal code is designed for expressing algorithms in a form that is easily understood by humans, blending programming language syntax with natural language descriptions and mathematical notation to outline the steps of an algorithm without the complexity of full coding details. It omits machine-specific implementations to focus on the core logic, making it a popular choice for documentation in educational materials and the preliminary design phases of software development. By abstracting away from the intricacies of actual code, pseudocode facilitates clear communication of algorithmic concepts across various programming environments. The definition of the universal code, as shown in Figure 2, is based on the following principles:

- **Comments**: Provide explanations and context for code segments, making it easier for others to understand the intent and functionality.

```
{Definition of Universal Code}
### Question
{Question}
### Response
{Answer}
### Your Task
Please combine the above Question and
Response to comply with the pseudocode
standard to write the corresponding
pseudocode of solution. Adopt a meticulous
methodology, breaking down the generation
process into manageable steps. Just output
the generated pseudocode for the solution
and do not include the Question and
Response in the output.

The output format is as follows, Use
```pseudocode to put the generated
pseudocode in markdown quotes:

```pseudocode
{{Offers a pseudocode version of the
solution.}}
```
```

Figure 3: Prompt of generating **UniCode**.

- **Variables**: Enhance code readability and maintainability by using meaningful names that convey the purpose of the variables without relying on data type specifications.
- **Input/Output**: Simplify the interaction with data entering and leaving the system, ensuring these operations are clear and easy to trace.
- **Conditionals**: Clarify decision-making processes within the code by using structured and indented conditional statements that define clear execution paths.
- **Loops**: Facilitate the repetition of code blocks in a controlled manner, with clearly defined start and end conditions, making the iterative processes understandable.
- **Functions/Procedures**: Increase modularity and reusability by naming functions and procedures descriptively, and by using parameters effectively to encapsulate functionality.
- **Formatting**: Improve the overall visual organization of the code by applying consistent indentation, which helps in delineating hierarchical structures and logical groupings within the code.

**Construction From Instruction Dataset.** For a programming language $L$, given the existing code instruction pair $(q_\alpha, a_\alpha) \in D_s^L$, where $q_\alpha$ and $a_\alpha$ are question and answer from $D_s^L$, we create the universal code instruction dataset $D_{u_\alpha}^L$ by prompting LLMs to generate the universal code $p_\alpha$ and then add $(q_\alpha, a_\alpha, p_\alpha)$ into $D_{u_\alpha}^L$. Figure 2 shows the definition of the code universal and Figure 3 is the prompt for LLMs to generate **UniCode**. {Definition of Universal Code}, {Question}, and {Answer} denote the slots for definition of the universal code $p_\alpha$, the question of the instruction data $q_\alpha$, and the answer of the instruction $a_\alpha$, respectively. Given $K$ different programming languages $L_{all} = \{L_k\}_{k=1}^K$, the multilingual programming instruction dataset with the universal code $D_{u_\alpha} = \{D_{u_\alpha}^{L_k}\}_{k=1}^K$ are created for supervised fine-tuning (SFT) (Ouyang et al., 2022). In this work, we adopt the open-source instruction dataset.

**Construction From Code Snippets.** For the unsupervised data (code snippets) widely existing on many websites (e.g., GitHub), we also construct the instruction dataset with the universal code from raw code snippets. Specifically, we ask the LLM to generate the question $q_\beta$ and the corresponding code answer $a_\beta$ pair based on the original code snippet $c$ using the prompt "Please generate the self-contained question and answer based on the given code snippet". Then, we generate **UniCode** $p_\beta$ and construct $(q_\beta, a_\beta, p_\beta)$ triplets the same way as in Paragraph 2. In addition, an LLM scorer is applied to filter out the low-quality $(q_\beta, a_\beta, p_\beta)$ triplets. Therefore, given raw code snippets of different programming languages $L_k \in \{L_k\}_{k=1}^K$, we can construct instruction dataset with the universal code $D_{u_\beta} = \{D_{u_\beta}^{L_k}\}_{k=1}^K$ directly from such unsupervised data. Finally, we combine these two instruction datasets to obtain $D_u = D_{u_\alpha} \cup D_{u_\beta}$, where $D_u^{L_k} = D_{u_\alpha^{L_k}} \cup D_{u_\beta^{L_k}}$ for each program langauge $L_k \in L_{all}$.

**Evaluation Task for Universal Code.** To test the capability of the LLMs in generating **UniCode** from questions and translating **UniCode** into answers, we design a code reconstruction task for evaluation. Given the code snippet $c$, we require the LLM to generate **UniCode** $p$ and then translate it into the code $c'$. The evaluation metric is not the similarity between $c$ and $c'$ but whether the restored code $c'$ can pass the test cases. We expand the HumanEval and MBPP datasets to create our benchmark UNICODER-BENCH comprising 164 HumanEval samples and 500 MBPP test samples.
```

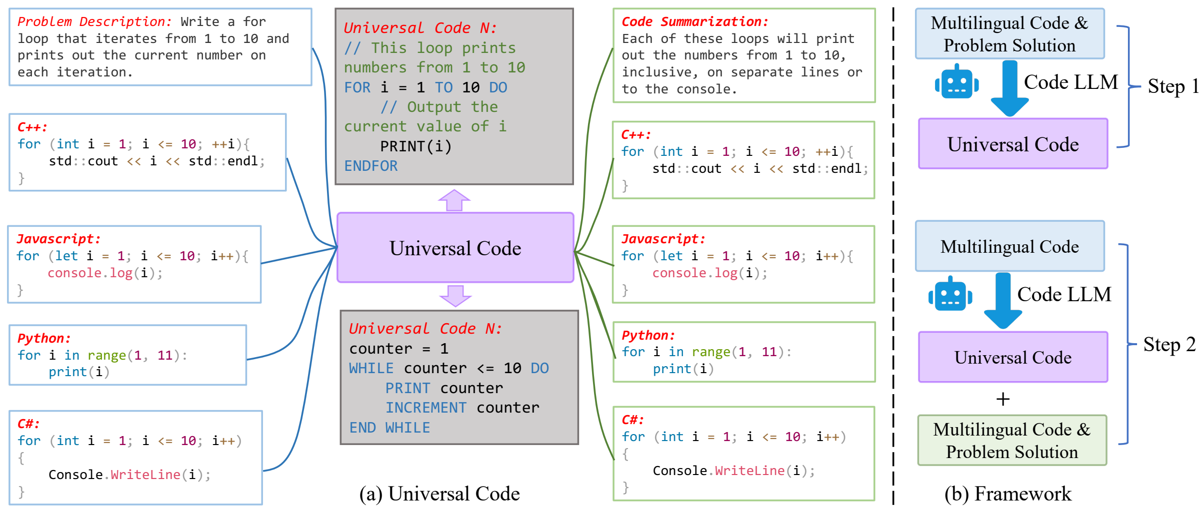

Figure 4: **Overview of UNICODER.** (a) The function of universal code **UniCode**; (b) The framework of our method UNICODER. The universal code as the intermediate representation, our proposed framework can support code generation, code translation, and code summarization. In (a), the LLM encodes the code snippets of multilingual programming languages or the problem description questions into **UniCode**. Then **UniCode** is translated into the target output, i.e., the executable code of multilingual programming languages with a descriptive code summarization. In (b), we first ask the LLM to generate **UniCode** with few-shot prompts. In the second stage, the instruction dataset, containing questions, answers, and **UniCode**, is fed into the code LLM for fine-tuning.

3 UNICODER

3.1 Model Overview

In Figure 4, we first define the concept of the universal code with the essential components and then prompt the LLM to generate **UniCode** p based on the existing instruction data (questions q and answers a) and the raw code snippets c. **UniCode** is regarded as the intermediate representation for different tasks, including code generation, code translation, and code summarization. Our proposed model UNICODER is trained on the instruction dataset D_u with the multilingual objectives to fully unleash the potential of **UniCode**.

3.2 Code LLM with Universal Code

Given the instructions dataset with K multilingual programming languages $D_u = \{D_u^{L_k}\}_{k=1}^{K}$, the pre-trained code LLM $\mathcal{M}$ trained on D_u can support Universal-code-of-Thought (UoT). It can be described as:

$$P(p, a|q) = P(p|q; \mathcal{M})P(a|q, p; \mathcal{M}) \quad (1)$$

where q (question) and a (answer) are the instruction pair from D_u. Given the question q, the code LLM $\mathcal{M}$ first generates **UniCode** p and then outputs the final answer a, where p provides key algorithm ideas with natural language comments.

3.3 Multi-task Supervised Fine-tuning

To fully unleash the potential of the **UniCode**, we design multiple objectives to enhance the understanding and generation capability of code LLM.

Multi-task Fine-tuning.

$$\mathcal{L}_{all} = \mathcal{L}_{qa} + \mathcal{L}_{qp} + \mathcal{L}_{pa} + \mathcal{L}_{uot} \quad (2)$$

where $\mathcal{L}_{qa}$ is the question-answer generation objective, $\mathcal{L}_{qp}$ is the question-universal-code generation objective, $\mathcal{L}_{pa}$ is the universal-code-answer translation objective, and $\mathcal{L}_{uot}$ is the Universal-code-of-Thought (UoT) objective.

Here, we introduce all four training objectives. For all the following objectives, the multilingual corpora $D_u = \{D_u^{L_k}\}_{k=1}^{K}$ are given. $\mathcal{M}$ is the code LLM and K is the number of programming languages.

Question-Answer Objective. The training objective $\mathcal{L}_{qa}$ of the standard instruction fine-tuning can be described as:

$$\mathcal{L}_{qa} = -\sum_{k=1}^{K} \mathbb{E}_{q,a \sim D_u^{L_k}} [\log P(a|q; \mathcal{M})] \quad (3)$$

where q and a are the question and answer pair.

Question-Universal-Code Objective. The training objective $\mathcal{L}_{qp}$ of the auxiliary universal code

generation task can be described as:

$$\mathcal{L}_{qp} = -\sum_{k=1}^{K} \mathbb{E}_{q,p\sim D_{L_k}} [\log P(p|q; \mathcal{M})] \quad (4)$$

where q and p are the question and **`UniCode`**.

Universal-Code-Answer Objective. The training objective $\mathcal{L}_{pa}$ of generating the executable code answer from **`UniCode`** can be described as:

$$\mathcal{L}_{pa} = -\sum_{k=1}^{K} \mathbb{E}_{p,a\sim D_{L_k}} [\log P(a|p; \mathcal{M})] \quad (5)$$

where p and a are **`UniCode`** and the answer.

Universal-Code-of-Thought Objective. The training objective $\mathcal{L}_{uot}$ of generating **`UniCode`** and then the executable code answer can be described as:

$$\mathcal{L}_{uot} = -\sum_{k=1}^{K} \mathbb{E}_{q,p,a\sim D_{L_k}} [\log P(p, a|q; \mathcal{M})] \quad (6)$$

where q, a, and p are the question, answer, and **`UniCode`**, respectively.

4 Experimental Setup

4.1 Instruction Dataset

GPT-4 (`gpt-4-1106-preview`) (OpenAI, 2023) is used as the foundation model to generate the UNICODER-INSTRUCT. We randomly extract code snippets within 1024 tokens from the StarCoder dataset (Li et al., 2023b) and let GPT-4 summarize the code snippets as the universal code. Based on each code snippet and the corresponding universal code, a self-contained coding problem with a correct solution is created.

4.2 Baselines

Proprietary Models. Based on a neural architecture known as generative pre-trained Transformers (GPT) (Vaswani et al., 2017; Radford et al., 2018), GPT-3.5 and GPT-4 are LLMs trained on massive datasets of text, code, math equations, and more. They are also trained to follow instructions (Ouyang et al., 2022), which allows them to generate human-like responses. We use GPT-3.5 Turbo and GPT-4 as the proprietary models because they perform excellently in various code understanding and generation tasks.

Open-Source Models. To narrow the gap between open-source and closed-source models, a series of open-source models and instruction datasets are proposed to improve code LLMs and bootstrap their instruction-following ability. Starcoder (Li et al., 2023b), Code Llama (Rozière et al., 2023), and DeepSeek-Coder (Guo et al., 2024a) with different model sizes are introduced into the based model. OctoCoder (Muennighoff et al., 2023), WiazrdCoder (Luo et al., 2023), MagiCoder (Wei et al., 2023), and WaveCoder (Yu et al., 2023) are further fine-tuned on these based code LLMs.

Decontainmation. We apply data decontamination before training our UNICODER models to decontaminate the code snippets from the starcoder data (Li et al., 2023b), by removing exact matches from HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), DS-1000 (Lai et al., 2023), and GSM8K (Cobbe et al., 2021).

4.3 Evaluation Benchmark

HumanEval. The HumanEval test set (Chen et al., 2021) is a crafted collection of 164 Python programming problems to test the abilities of code generation models. For each problem, there are roughly 9.6 test cases to check whether the generated code works as intended. Humaneval has become one of the most popular benchmarks to measure how well these code-writing AI models perform, making it a key tool in the field of AI and machine learning for coding.

MBPP. The MBPP dataset (Austin et al., 2021), comprising approximately 1,000 Python programming challenges sourced from a crowd of contributors, is tailored for beginners in programming, focusing on core principles and the usage of the standard library. The MBPP test set comprised of 500 problems is selected to evaluate the few-shot inference of the code LLMs.

MultiPL-E. The MuliPL-E test set (Cassano et al., 2022) translates the original HumanEval test set to other 18 programming languages, i.e., Javascript, Java, Typescript, C++, and Rust. We use the MultiPL-E to evaluate the multilingual capabilities of the code LLMs.

4.4 Evaluation Metrics

Pass@k. We adopt the Pass@k metric (Chen et al., 2021) to improve the reliability of our evaluation. We then count the total number of success-

fully passing test cases, denoted as k, to compute the Pass@k, thereby enhancing the accuracy and consistency of the performance assessment.

$$\text{Pass@k} = \mathbb{E}\left[1 - \frac{\binom{n}{k-c}}{\binom{n}{k}}\right] \quad (7)$$

where n is the total number of generated samples for each problem, and c is the number of correct generated code snippets passing all the test cases ($n > k \geq c$).

4.5 Impletmentation Details

We expand the open-source Evol-Instruct dataset `evol-code-alpaca-v1` (Xu et al., 2023) with nearly 110K samples into the instruction dataset with the universal code. For the code snippets collected from starcoderdata [2], we choose 5K code snippets of each language (Python, Javascript, C++, Java, Rust, and Go) to construct the synthetic instruction dataset with universal code. Finally, we obtain the instruction dataset UNICODER-INSTRUCT contains nearly 140K training samples. Code-Llama and DeepSeek-Coder-Base are used as the foundational code LLMs for supervised fine-tuning (SFT). We fine-tune these foundation LLMs on nearly 150K samples generated from `evol-codealpaca-v1` and the starcoder pre-training data. UNICODER is fine-tuned on `Standford_Alpaca`[3] with 8 NVIDIA A100-80GB GPUs. The learning rate first increases into 8×10^{-5} with 50 warmup steps and then adopts a cosine decay scheduler. We adopt the Adam optimizer (Kingma and Ba, 2015) with a global batch size of 128 samples, truncating sentences to 1536 tokens.

5 Results and Discussion

5.1 Main Results

Python Code Generation. Table 1 shows that UNICODER significantly beats previous strong open-source baselines using UoT, closing the gap with GPT-3.5 and GPT-4. Magicoder (Wei et al., 2023) and Wavecoder (Yu et al., 2023) both prove the effectiveness of instruction datasets from code snippets. Further, UNICODER outperforms the WizardCoder with 15B parameters and Evol-Instruct techniques with the help of the **`UniCode`**.

[2] https://huggingface.co/datasets/bigcode/starcoderdata

[3] https://github.com/tatsu-lab/stanford_alpaca

Multilingual Code Understanding. Table 2 shows that UNICODER significantly outperforms strong baselines Magicoder and WaveCoder, which both leverage the code snippets to construct the instruction dataset. Given the multilingual correct code snippet, the code LLM is tasked to generate an explanation of the code and then regenerate the code only based on its own explanation. For the different backbones (Code Llama and Deepseek-Coder), our method beats most previous methods, especially in other languages, which demonstrates that UNICODER-INSTRUCT can bring the capability of multilingual understanding and generation.

5.2 Discussion

Ablation Study. To verify the efficacy of each component, we conduct the ablation study step by step on HumanEval and MBPP. In Table 3, we observe that removing the multi-tasks objective (only keeping the UoT objective: Equation 6) will have a -1.6 performance drop in HumanEval and a -1.3 drop in MBPP. Removing **`UniCode`** will further degrade the performance. The results support the effectiveness of each component of UNICODER.

Effect on Universal Code. To discuss the effect of the different formats of the universal code, we use different definitions of universal code for UNICODER. Specifically, we randomly sample 5K samples to generate the instruction dataset with different formats of **`UniCode`**.

- **`UniCode` 1**: It describes the naming conventions, variable declaration, operators, conditional statements, loops, and function structure that pseudocode should have.
- **`UniCode` 2**: It separates the first set of standards and provides code examples for each, instead of applying them all together in the examples.
- **`UniCode` 3**: It describes the code structure, variable rules, control structures, functions, comments, and assignment rules that pseudocode should have.
- **`UniCode` 4**: It is similar to the first standard but specifies type-free names for variables.
- **`UniCode` 5**: It provides an abstract, high-level architectural description, without setting standards for the code itself.
- **`UniCode` 6**: It uses latex algorithm and algorithmic packages for description.

Models	Base Model	Params	Instruction Data	Model Weight	HumanEval	MBPP
Proprietary Models						
GPT-3.5	-	-	-	-	72.6	81.6
GPT-4	-	-	-	-	85.4	83.0
Open-source Models						
StarCoder (Li et al., 2023b)	-	15B	✗	✓	33.6	43.3
WizardCoder (Luo et al., 2023)	StarCoder	15B	✓	✓	57.3	51.8
OctoCoder (Muennighoff et al., 2023)	StarCoder	15B	✓	✓	46.2	43.5
WaveCoder-SC (Muennighoff et al., 2023)	StarCoder	15B	✓	✓	50.5	51.0
Code-Llama (Rozière et al., 2023)	-	7B	✗	✓	33.5	41.4
Code-Llama-Instruct (Rozière et al., 2023)	Code Llama	7B	✓	✓	34.8	44.4
WaveCoder-CL (Yu et al., 2023)	Code Llama	7B	✓	✓	48.1	47.2
Magicoder-CL (Wei et al., 2023)	Code Llama	7B	✓	✓	60.4	64.2
UNICODER (our method)	Code Llama	7B	✓	✓	65.4	65.2
DeepseekCoder (Guo et al., 2024a)	-	6.7B	✗	✓	49.4	60.6
WaveCoder-DS (Yu et al., 2023)	Deepseek-Coder	6.7B	✓	✓	64.0	62.8
UNICODER (our method)	Deepseek-Coder	6.7B	✓	✓	**70.6**	**64.3**

Table 1: Evaluation results of Pass@1 on the HumanEval and MBPP benchmark. We use self-reported scores whenever available. All methods use greedy decoding and We use the reported scores of the previous work.

Model	Params	Programming Language						
		Java	Javascript	C++	PHP	Swift	Rust	Avg.
Proprietary models								
GPT-3.5	-	69.2	67.1	63.4	60.9	-	-	-
GPT-4	-	81.6	78.0	76.4	77.2	-	-	-
Open-source models								
CodeLlama (Rozière et al., 2023)	34B	40.2	41.7	41.4	40.4	35.3	38.7	39.6
CodeLlama-Python (Rozière et al., 2023)	34B	39.5	44.7	39.1	39.8	34.3	39.7	39.5
CodeLlama-Instruct (Rozière et al., 2023)	34B	41.5	45.9	41.5	37.0	37.6	39.3	40.5
WizardCoder-CL (Luo et al., 2023)	34B	44.9	55.3	47.2	47.2	44.3	46.2	47.5
StarCoderBase (Li et al., 2023b)	15B	28.5	31.7	30.6	26.8	16.7	24.5	26.5
StarCoder (Li et al., 2023b)	15B	30.2	30.8	31.6	26.1	22.7	21.8	27.2
WizardCoder-SC (Luo et al., 2023)	15B	35.8	41.9	39.0	39.3	33.7	27.1	36.1
CodeLlama (Rozière et al., 2023)	7B	29.3	31.7	27.0	25.1	25.6	25.5	27.4
CodeLlama-Python (Rozière et al., 2023)	7B	42.4	51.9	42.3	46.5	29.4	33.6	29.7
UNICODER (Our method)	7B	**46.4**	**50.2**	**39.2**	**40.4**	**41.2**	**32.4**	**41.6**

Table 2: Evaluation results of Pass@1 (%) performance on the MultiPL-E benchmark. The baseline results are partly from the previous work (Wei et al., 2023).

ID	Methods	HumanEval	MBPP
①	UNICODER	70.6	64.3
②	① - Multi-tasks Objective	67.4	60.2
③	② - Universal Code	66.8	59.8

Table 3: Ablation study of our proposed method on HumanEval and MBPP. UNICODER is fine-tuned on the UNICODER-INSTRUCT with the multi-task objectives.

ID	Methods	HumanEval	MBPP
①	**UniCode** 1	53.2	51.5
②	**UniCode** 2	52.8	51.2
③	**UniCode** 3	53.5	50.5
④	**UniCode** 4	53.8	49.5
⑤	**UniCode** 5	49.5	50.2
⑥	**UniCode** 6	48.2	48.4
⑦	**UniCode** 1∼4	**55.5**	**52.2**

Table 4: Evaluation results of our method with different formats of the universal code.

In Table 4, we can observe that the evaluation results of **UniCode** 1∼**UniCode** 4 have better performance. Compared to the universal code format **UniCode** 5 and **UniCode** 6, **UniCode** 1∼**UniCode** 4 has a clear definition and common structure, which brings more support for code generation. Notably, the experiment ⑦ performs the best by combing the training data of ①∼④. The experimental results show that the concrete definition of **UniCode** and the combination of it can effectively improve the model performance.

5.3 Code-UniCode-Code

To compare the capabilities of different code LLMs, we create a test set by prompting the code

Method	Params	Python	Other Languages
Code-Llama-Instruct	7B	33.3	26.2
Code-Llama-Alpaca	7B	44.2	29.1
UNICODER	7B	45.2	31.3

Table 5: Pass@1 scores of our method UNICODER and two Code-Llama baselines for Code-**UniCode**-Code.

LLM to generate **UniCode** and translate it into the executable code. We check the correctness of each translated code with the test cases, denoted as Pass@1 of the universal code. Code-Llama-7B is fine-tuned on the Code Alpaca dataset and our dataset UNICODER-INSTRUCT separately. The results of fine-tuned Code-Llama models on UNICODER-BENCH are shown in Table 5. Our method UNICODER is more accurate in passing the test cases than the Code-Llama baselines, demonstrating its excellent code understanding and generation abilities.

6 Related Work

Code Understanding and Generation. Code understanding and generation as the key tasks to substantially facilitate the project development process, including code generation (Chen et al., 2021; Austin et al., 2021; Zhang et al., 2023), code translation (Szafraniec et al., 2023), automated testing (Deng et al., 2023), bug fixing (Muennighoff et al., 2023), code refinement (Liu et al., 2023c), code question answering (Liu and Wan, 2021), and code summarization (Ahmad et al., 2020). Researchers Chai et al. (2023) have undertaken extensive endeavors to bridge natural language and programming languages. With less ambiguous prompt styles, Mishra et al. (2023) using pseudocode improves the performance of NLP tasks. Oda et al. (2015) uses traditional machine learning to achieve code to pseudocode conversion. Jiang et al. (2022) also shows that designers and programmers can speed up the prototyping process, and ground communication between collaborators via prompt-based prototyping. To verify that the generated code is correct, there are some code synthesis evaluation frameworks, including EvalPlus (Liu et al., 2023b), HumanEval (Chen et al., 2021), HumanEval-X (Zheng et al., 2023), and MBPP (Austin et al., 2021).

Large Language Models for Code. Since CodeBERT (Feng et al., 2020) first connected code tasks with pre-trained models, large language models for code have developed rapidly, demonstrating extraordinary performance on almost all code tasks, rather than a single task. Prominent large models include Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), SantaCoder (Allal et al., 2023), Starcoder (Li et al., 2023b), WizardCoder (Luo et al., 2023), InCoder (Fried et al., 2022), CodeT5 (Wang et al., 2021), CodeGeeX (Zheng et al., 2023), Code Llama (Rozière et al., 2023), and CodeQWen (Bai et al., 2023). To improve the performance of code generation, researchers used optimized prompts (Liu et al., 2023a; Reynolds and McDonell, 2021; Zan et al., 2023; Beurer-Kellner et al., 2023), bring test cases (Chen et al., 2023) and collaborative roles (Dong et al., 2023). There are also some related studies on using large language models for other code tasks, such as dynamic programming (Dagan et al., 2023), compiler optimization (Cummins et al., 2023), multilingual prompts (Di et al., 2023), and program of thoughts (Chen et al., 2022) (PoT).

Chain-of-Thought Prompting. To unleash the potential of LLMs in addressing complex reasoning tasks, chain-of-thought (CoT) prompting (Wei et al., 2022b; Kojima et al., 2022) extends in-context learning with step-by-step reasoning processes, which handles complex reasoning tasks in the field of the code and mathematics by encouraging them to engage in step-by-step reasoning processes. Following this line of research, X-of-Thought (XoT) reasoning (CoT and its structural variants further) (Chai et al., 2024; Yao et al., 2023; Li et al., 2023a; Lei et al., 2023; Guo et al., 2023; Ji et al., 2024; Guo et al., 2024b) further expands the capabilities and applications of LLMs in complex reasoning and planning scenarios.

Intermediate Repersentation In the field of natural language processing, there exist many works using intermediate representation (Gan et al., 2021; Yang et al., 2022, 2024, 2019, 2020b,a; Liang et al., 2024), such as text generation and translation. The universal code is used as the intermediate representation, which typically omits details that are essential for the machine implementation of the algorithm. We perform the coarse-to-fine pattern for the code generation and translation, where the universal code first summarizes the algorithm process and then the programming language gives the accurate solution. The Unicode provides explicit help for code generation such as Chain-of-thought in LLM.

7 Conclusion

In this work, we put forth a state-of-the-art framework UNICODER for both code translation and code generation. Using the universal code **UniCode** as the intermediate representation, we effectively bridge different programming languages and facilitate code tasks. In addition, we collect a dataset UNICODER-INSTRUCT with 140K instruction instances from existing instruction datasets and the raw code snippets. After being fine-tuned on UNICODER-INSTRUCT with multi-task learning objectives, our model generates **UniCode** and translates it into the final answer (executable code). The evaluation results on code translation and generation tasks demonstrate that our method significantly improves the generalization ability, showing the efficacy and superiority of UNICODER.

Limitations

We acknowledge the following limitations of this study: (1) The evaluation focuses on benchmark datasets (Humaneval, MBPP, and MultiPL-E), and the model's effectiveness in real-world programming scenarios or industry applications is not fully explored. (2) Our method is developed and evaluated primarily on programming language benchmarks. Its effectiveness in other domains or for non-programming-related tasks is not assessed, which limits the generalizability of our findings.

Acknowledege

This work was supported in part by the National Natural Science Foundation of China (Grant Nos. U1636211, U2333205, 61672081, 62302025, 62276017), a fund project: State Grid Co., Ltd. Technology R&D Project (ProjectName: Research on Key Technologies of Data Scenario-based Security Governance and Emergency Blocking in Power Monitoring System, Proiect No.: 5108-202303439A-3-2-ZN), the 2022 CCF-NSFOCUS Kun-Peng Scientific Research Fund and the Opening Project of Shanghai Trusted Industrial Control Platform and the State Key Laboratory of Complex & Critical Software Environment (Grant No. SKLSDE-2021ZX-18).

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 4998–5007. Association for Computational Linguistics.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*, abs/2309.16609.

Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2023. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI):1946–1969.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Linzheng Chai, Jian Yang, Tao Sun, Hongcheng Guo, Jiaheng Liu, Bing Wang, Xinnian Liang, Jiaqi Bai, Tongliang Li, Qiyao Peng, and Zhoujun Li. 2024. xcot: Cross-lingual instruction tuning for cross-lingual chain-of-thought reasoning. *arXiv preprint arXiv:2401.07037*, abs/2401.07037.

Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. 2023. Ernie-code: Beyond english-centric cross-lingual pretraining for programming languages. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 10628–10650. Association for Computational Linguistics.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, abs/2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, abs/2211.12588.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Kim M. Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, abs/2309.07062.

Gautier Dagan, Frank Keller, and Alex Lascarides. 2023. Dynamic planning with a LLM. *arXiv preprint arXiv:2308.06391*, abs/2308.06391.

Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014*, abs/2304.02014.

Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. 2023. Codefuse-13b: A pretrained multilingual code large language model. *arXiv preprint arXiv:2310.06266*, abs/2310.06266.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, abs/2304.07590.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, abs/2204.05999.

Shiwei Gan, Yafeng Yin, Zhiwei Jiang, Lei Xie, and Sanglu Lu. 2021. Skeleton-aware neural sign language translation. In *MM '21: ACM Multimedia Conference, Virtual Event, China, October 20 - 24, 2021*, pages 4353–4361. ACM.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024a. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Hongcheng Guo, Jian Yang, Jiaheng Liu, Liqun Yang, Linzheng Chai, Jiaqi Bai, Junran Peng, Xiaorong Hu, Chao Chen, Dongfeng Zhang, Xu Shi, Tieqiao Zheng, Liangfan Zheng, Bo Zhang, Ke Xu, and Zhoujun Li. 2023. OWL: A large language model for IT operations. *CoRR*, abs/2309.09298.

Hongcheng Guo, Wei Zhang, Anjie Le, Jian Yang, Jiaheng Liu, Zhoujun Li, Tieqiao Zheng, Shi Xu, Runqiang Zang, Liangfan Zheng, et al. 2024b. Lemur: Log parsing with entropy sampling and chain-of-thought merging. *arXiv preprint arXiv:2402.18205*.

Hangyuan Ji, Jian Yang, Linzheng Chai, Chaoren Wei, Liqun Yang, Yunlong Duan, Yunli Wang, Tianzhen Sun, Hongcheng Guo, Tongliang Li, et al. 2024. Sevenllm: Benchmarking, eliciting, and enhancing abilities of large language models in cyber threat intelligence. *arXiv preprint arXiv:2405.03446*.

Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J. Cai. 2022. Promptmaker: Prompt-based prototyping with large language models. In *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022, Extended Abstracts*, pages 35:1–35:8. ACM.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations,*

ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Bin Lei, Pei-Hung Lin, Chunhua Liao, and Caiwen Ding. 2023. Boosting logical reasoning in large language models through a new framework: The graph of thought. *arXiv preprint arXiv:2308.08614*, abs/2308.08614.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023b. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, abs/2305.06161.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *arXiv preprint arXiv:2203.07814*, abs/2203.07814.

Yaobo Liang, Quanzhi Zhu, Junhe Zhao, and Nan Duan. 2024. Machine-created universal language for cross-lingual transfer. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 18617–18625. AAAI Press.

Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023a. Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*, abs/2305.08360.

Chenxiao Liu and Xiaojun Wan. 2021. CodeQA: A question answering dataset for source code comprehension. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2618–2632. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, abs/2305.01210.

Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach Dinh Le, and David Lo. 2023c. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. *arXiv preprint arXiv:2307.12596*, abs/2307.12596.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Mayank Mishra, Prince Kumar, Riyaz Bhat, Rudra Murthy V, Danish Contractor, and Srikanth Tamilselvam. 2023. Prompting with pseudo-code instructions. *arXiv preprint arXiv:2305.11790*, abs/2305.11790.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. OctoPack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, abs/2308.07124.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate

pseudo-code from source code using statistical machine translation (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 574–584. IEEE Computer Society.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. *OpenAI blog*.

Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama Japan, May 8-13, 2021, Extended Abstracts*, pages 314:1–314:7. ACM.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. 2023. Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022a. Finetuned language models are zero-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022b. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, abs/2312.02120.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

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, Singapore, December 6-10, 2023*, pages 5067–5089. Association for Computational Linguistics.

Jian Yang, Hongcheng Guo, Yuwei Yin, Jiaqi Bai, Bing Wang, Jiaheng Liu, Xinnian Liang, Linzheng Chai, Liqun Yang, and Zhoujun Li. 2024. m3p: Towards multimodal multilingual translation with multimodal prompt. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 10858–10871. ELRA and ICCL.

Jian Yang, Shuming Ma, Dongdong Zhang, Zhoujun Li, and Ming Zhou. 2020a. Improving neural machine translation with soft template prediction. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 5979–5989. Association for Computational Linguistics.

Jian Yang, Shuming Ma, Dongdong Zhang, Shuangzhi Wu, Zhoujun Li, and Ming Zhou. 2020b. Alternating language modeling for cross-lingual pre-training. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 9386–9393. AAAI Press.

Jian Yang, Yuwei Yin, Shuming Ma, Dongdong Zhang, Shuangzhi Wu, Hongcheng Guo, Zhoujun Li, and Furu Wei. 2022. UM4: unified multilingual multiple teacher-student model for zero-resource neural machine translation. In *Proceedings of the Thirty-First*

International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022, pages 4454–4460. ijcai.org.

Ze Yang, Wei Wu, Jian Yang, Can Xu, and Zhoujun Li. 2019. Low-resource response generation with template prior. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 1886–1897. Association for Computational Linguistics.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, abs/2305.10601.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187*, abs/2312.14187.

Daoguang Zan, Ailun Yu, Bo Shen, Jiaxin Zhang, Taihong Chen, Bing Geng, Bei Chen, Jichuan Ji, Yafen Yao, Yongji Wang, and Qianxiang Wang. 2023. Can programming languages boost each other via instruction tuning? *arXiv preprint arXiv:2308.16824*, abs/2308.16824.

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, abs/2303.12570.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, abs/2303.17568.