Case2Code: Scalable Synthetic Data for Code Generation

Yunfan Shao[1,2], **Linyang Li**[2†], **Yichuan Ma**[1,2], **Peiji Li**[1,2], **Demin Song**[2],
Qinyuan Cheng[1,2], **Shimin Li**[1], **Xiaonan Li**[1], **Pengyu Wang**[1], **Qipeng Guo**[2],
Hang Yan[2,3], **Xipeng Qiu**[1†], **Xuanjing Huang**[1], **Dahua Lin**[2,3]

[1]School of Computer Science, Fudan University
[2]Shanghai AI Laboratory
[3]The Chinese University of Hong Kong
{yfshao19, xpqiu}@fudan.edu.cn
{lilinyang, yanhang}@pjlab.org.cn

Abstract

Large Language Models (LLMs) have shown outstanding breakthroughs in code generation. Recent work improves code LLMs by training on synthetic data generated by some powerful LLMs, which can be challenging to scale due to the dependence on a teacher model and high generation costs. In this paper, we focus on synthesizing code data at scale and propose a **Case2Code** task by exploiting the expressiveness and correctness of programs. **Case2Code** is an inductive inference task that aims to infer underlying code implementations by observing input-output examples or program behaviors, By incorporating LLMs to generate program inputs, and executing the program with these inputs to obtain the program outputs, we can synthesize diverse and high-quality **Case2Code** data at scale for training and evaluating code LLMs. Experimental results show that case-to-code induction is challenging for current representative LLMs if they are untrained. Models trained with **Case2Code** improve performance not only on distribution case-to-code induction but also various coding-generation tasks, demonstrating the great potential of large-scale synthetic data and inductive learning.[1]

1 Introduction

The success of large language models (LLMs), exemplified by GPT-4 (OpenAI, 2023) has revolutionized the AI community. One of the most impressive abilities of LLMs is the code generation ability, exemplified by writing code blocks or programs to accomplish complex instructions and satisfy user specifications (Hong et al., 2024; Zhuo et al., 2024; Sun et al., 2024).

To further improve the performance of code generation with open-source LLMs, recent work adopts synthetic data via self-instruct (Wang et al.,

---

†Corresponding Authors.
[1]Code and datasets will be available at https://github.com/choosewhatulike/case2code.

Figure 1: Examples of existing code generation and case2code tasks. Compared with existing code generation instructions, inductive learning tasks like case2code are rare in the training data, which makes it challenging for LLMs to perform.

2023b; Taori et al., 2023) to distill the strong capability of code generation from teacher models. Specifically, practitioners often collect and devise code instructions and then generate high-quality responses using a powerful teacher LLM to synthesize the training data. Then, the data can be used to fine-tune a weak LLM to bootstrap their code generation ability. For example, Code Alpaca (Chaudhary, 2023) incorporates ChatGPT to generate 20K code instructions under 21 seed tasks for fine-tuning. WizardCoder (Luo et al., 2023) adopts Evol-Instruct to the realm of code, using powerful closed-source LLMs to synthetic high-quality code instruction-following samples to enhance open-sourced LLMs code generation performance.

Despite the success of these code synthetic data, they rely on a powerful teacher for data generation (Yu et al., 2023b), which can be challenging to scale up as they are bound by the capability of

the teacher and suffer from high costs. Moreover, models can perform poorly on tasks like inductive learning as these instructions are rare in the training corpus (Figure 1). Therefore, in this paper, to synthesize high-quality code data at scale and complement existing code training data, we introduce **Case2Code**, a diverse and challenging synthetic task for LLMs.

Inspired by inductive inference tasks (Balog et al., 2016; Devlin et al., 2017; Ellis et al., 2021), we focus on large-scale data synthesis with programs in the real world. In Case2Code, samples are synthesized from real-world productive functions, which are closer to the actual distribution of general LLM applications and production. Specifically, the Case2Code challenge requires the LLM to infer the underlying program based on several input-to-output cases generated by the real-world program. In Case2Code learning, LLMs are supposed to write solutions formulated by codes based on the example outputs, which is one common scenario in the real-world working process, using examples to convey knowledge.

To obtain large-scale and diverse Case2Code data, we first gather a diverse collection of executable code texts that cover a wide range of real-world applications. Then, we generate the input-output transformation cases with the assistance of LLMs and code interpreters. By incorporating LLMs to write input examples for each program and execute the program with these inputs to gather the corresponding outputs, we can synthesize large-scale Case2Code samples with diverse data transformations and complicated control logic. The data synthetic framework does not require powerful LLMs with advanced code generation capabilities, resulting in the possibility of a weak-to-strong learning paradigm.

Based on the synthetic data, we can form a unique and challenging task to evaluate and further train the LLMs and study the case-to-code induction ability of LLMs. In the Case2Code challenge, we first test how current LLMs perform in making the code induction task. We then train LLMs with Case2Code data to further study whether such data can improve the in-distribution code induction ability and generalize to other commonly used code generation tasks. Experimental results show that Case2Code is a challenging task for LLMs, even for powerful LLMs like LLaMA3-70B, GPT-3.5, and GPT-4. With constructed Case2Code data, we can boost LLMs to learn to make such inductive in-

ference tasks, while such ability can be transferred to help improve general code generation tasks such as HumanEval and MBPP.

To summarize, in this paper, we:

(1) We introduce a scalable data synthetic framework that aims to generate high-quality and diverse inductive code generation samples for evaluating and training LLMs, **Case2Code**.

(2) We evaluate the inductive learning ability of current representative LLMs, demonstrating the necessity of synthesizing inductive data like **Case2Code** for LLMs.

(3) We explore methods of training LLMs on large-scale **Case2Code** data, showing not only great improvements on the Case2Code challenge but also a consistent generalization of the trained models in general code generation tasks.

2 Related Work

Our work discusses the reasoning ability of LLMs, touching on the following grounds:

2.1 Inductive Inference

Inductive inference is rarely discussed in LLM reasoning, most research focuses on specific scenarios with limited inductive reasoning. One pioneer work is prerequisite toy tasks (Weston et al., 2015) where the task goal is to solve simple induction. Later, Yang et al. (2022) introduces various world-wide knowledge such as botany, history, and geography into the facts given and asks neural models to predict whether a given rule is correct. In the realm of code, several works focus on Programming by Example (PBE), which aims to induce a valid program given the expected inputs and the corresponding outputs. These works train and evaluate inductive program synthesis models for constrained scenarios with limited search spaces, such as operations on list, string, and manually-defined objects (Balog et al., 2016; Devlin et al., 2017; Ellis et al., 2021; Shi et al., 2023; Wu et al., 2024; Li and Ellis, 2024). Different from previous studies, our proposed Case2Code task leverages diverse code in the real world as a powerful platform for LLMs to learn inductive inference under various challenging scenarios.

2.2 Synthetic Data

Recent works focus on building high-quality instruction-following or question-answering training data through strong LLMs such as GPT-4 to
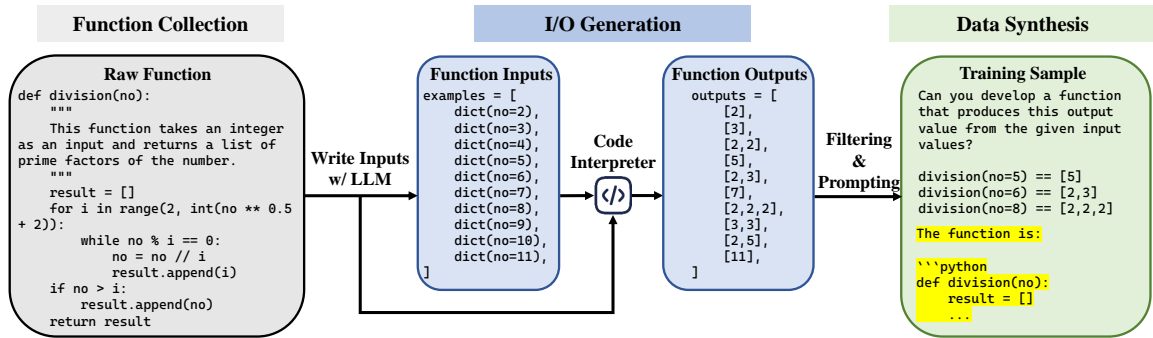
Figure 2: Our synthetic framework incorporates an LLM and a code interpreter to construct Case2Code training samples at scale automatically.

enhance smaller LLMs (Yu et al., 2023a; Mitra et al., 2024; Luo et al., 2023). While a particular line of work focuses on studying different strategies to diverse the instructions and control the quality of LLM generation, including self-consistency (Wang et al., 2022), rejection sampling (Huang et al., 2023; Yuan et al., 2023; Wang et al., 2023a), program-of-thought (PoT) (Sun et al., 2024), tree-structure CoT (ToT) searching (Yao et al., 2023), Monte Carlo Tree Searching (Silver et al., 2016; Chen et al., 2024), etc. These methods still require a strong LLM as the teacher with high costs of model inference, limiting the scalability.

3 Method

In this section, we illustrate the framework for synthesizing Case2Code data in detail, which focuses on producing large-scale and high-quality inductive reasoning data in the code domain. Unlike other synthetic data frameworks that distill high-quality training data from a strong teacher LLM to provide supervision signals to improve student LLMs, our Case2Code synthetic framework introduces a writer LLM to assist the synthesis of data samples. Thus the overall data quality does not directly rely on the performance of the LLM generator. And we can efficiently obtain reliable Case2Code training data at scale.

3.1 Problem Formulation

The inductive reasoning task aims to find a general hypothesis based on a small set of observations to explain a phenomenon. In this paper, we define Case2Code, an inductive reasoning task in the code domain. Case2Code is a program synthesis task that targets the reconstruction of unknown programs based on observations of the program behaviors.

Formally, for a functional program $\mathcal{P}$, we have a set of n input-output examples $\mathcal{S}_{\mathcal{P}} = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, where $y_i = \mathcal{P}(x_i), i = 1, 2, ..., n$. The goal of Case2Code is to implement a program $\mathcal{P}'$ that captures the functionality of the program $\mathcal{P}$ based on the observed set of input-output example cases $\mathcal{S}_{\mathcal{P}}$. And for any new input case $x_{\text{new}} \notin \mathcal{S}_{\mathcal{P}}$, the implemented program $\mathcal{P}'$ should satisfy that $\mathcal{P}(x_{\text{new}}) = \mathcal{P}'(x_{\text{new}})$.

3.2 Framework Overview

In our synthetic data generation framework, we focus on generating large-scale and diverse Case2Code data automatically. As shown in Figure 2, we first collect diverse programs from large-scale datasets with rule-based filters. Then we incorporate LLMs to write diverse example inputs and utilize the code interpreter to calculate their corresponding outputs for each program. Finally, we filter out low-quality programs based on their outputs and convert the obtained triple (program, inputs, outputs) into Case2Code data for inductive reasoning in the code domain.

Note that the correctness of our synthetic data does not depend on the capabilities of the used LLMs. Therefore, we can synthetic high-quality Case2Code data at scale using small LLMs with low costs.

3.3 Collecting Programs

To obtain massive data samples for inductive reasoning learning, we first need to acquire massive and diverse programs that take input arguments, do some complicated processes, and return output values. Instead of prompting LLMs to generate functions that meet these requirements, we collect human-written high-quality programs in the wild to enhance diversity.

11058

Specifically, we sample valid Python functions from The Stack (Kocetkov et al., 2022) to construct our reasoning dataset. We incorporate the out-of-box Abstract Syntax Tree (AST) parsing tool [2] to parse each file in The Stack to obtain Python functions. We only keep self-contained high-quality functions that satisfy all of these filtering rules: (1) pass the syntax check; (2) have one or more input arguments and return values; and (3) do not rely on third-party packages or external I/O operations. After collecting these functions, we can easily execute and verify these functions to obtain diverse Case2Code data with a simple and fast code interpreter at scale, which avoids extra file or network operations that require a sophisticated sandbox.

3.4 Generating Inputs

Once we collect large-scale functions, the next step is to obtain the corresponding input-output pairs for each function to construct the Case2Code data. It is infeasible to write test cases for each function manually. So, we utilize LLMs to generate suitable input examples for these functions. We prompt LLMs to write some example input arguments for each function based on the corresponding function implementation. Detailed prompt is listed in Table 7 in the appendix.

To generate suitable input arguments, the LLM needs first to analyze the implementation of the functions, then infer the possible types and value ranges of the input arguments, and finally come up with correct input arguments. However, we argue that a powerful LLM is not the key factor for our synthetic data. As we find that while strong LLMs can write high-quality inputs to generate Case2Code training data that boosts the reasoning performance of weak LLMs, the weak LLM can also write inputs for creating Case2Code data to self-improve their reasoning ability (see Sec 4.4). Therefore, the generation process can be scaled efficiently at a low cost by using small LLMs.

3.5 Obtain Outputs

After collecting self-contained functions and the corresponding inputs, it is intuitive to incorporate a code interpreter to run these functions on their inputs for output curation. Since the LLM-generated input examples can contain errors, we introduced a filtering procedure to reject invalid inputs or functions based on their returned outputs. Specifically,

if the outputs of a function do not change as the inputs change (e.g. always return the same output or exceptions), the function is considered invalid and will be filtered out.

Moreover, we also filter out functions that generate very long output values to ensure the length of the generated Case2Code data is within the context window size of current LLMs. Note that we do not filter out inputs that lead to exceptions or runtime errors, as we believe that failure call attempts can also provide valuable information for inductive reasoning to reconstruct the function.

3.6 Post-processing

The final step is to convert the obtained functions and their corresponding input-output pairs into Case2Code style data. Formally, for a given function $\mathcal{P}$ and its n test cases $\mathcal{S}_{\mathcal{P}} = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, we randomly sample m examples ($m <= n$) as the observed set $\mathcal{S}'_{\mathcal{P}}$. We generate the prompted data that facilitate the LLM to conduct inductive reasoning on the observed examples $\mathcal{S}'_{\mathcal{P}}$ to reconstruct the given function $\mathcal{P}$. Converted training examples are shown in Table 8 in the appendix.

We find that the diversity of the prompts can substantially affect the generalization of the model reasoning performance (as shown in Sec 4.4). Therefore, we manually construct about 10 prompts with different styles to enhance the data diversity.

4 Experiment

In this section, we illustrate the experimental setups and discuss the experimental results to demonstrate the challenge of solving Case2Code problems and show the effectiveness of large-scale Case2Code synthetic data.

4.1 Experimental Setup

Data Construction We randomly sampled about 2.3 million functions from The Stack pre-training dataset, in which we already performed data deduplication with the evaluation benchmarks (e.g. HumanEval, MBPP, etc). We conduct the data synthetic pipeline incorporating InternLM2-7b (Cai et al., 2024) to generate input examples for each function. The temperature is set to 0.2 and the top_p is set to 0.95. The generation takes about 500 GPU hours using A800 GPUs. Then we use 64 CPUs to execute and filter functions, which takes about 1 hour. The execution is under a

---

	Size	HumanEval	HumanEval+	MBPP	MBPP+	Case2Code
GPT-4	-	90.2	86.6	85.7	73.3	43.6
GPT-3.5	-	76.8	70.7	82.5	69.7	34.2
LLaMA2-Chat	7B	14.0	11.6	26.8	20.3	0.2
	13B	23.1	19.5	37.0	27.6	8.2
	34B	22.6	-	33.0	-	-
	70B	36.6	28.7	46.3	35.1	7.8
CodeLLaMA-Instruct	7B	37.8	35.4	59.5	46.8	14.2
	13B	42.7	38.4	63.5	52.6	19.0
	34B	51.8	43.9	69.3	56.3	22.6
LLaMA3-Instruct	8B	61.6	56.7	70.1	59.3	23.2
	70B	77.4	72.0	82.3	69	34.0

Table 1: Accuracy of various representative LLMs on the code generation datasets and the Case2Code test set.

constrained Python environment to ensure safety. We eventually obtained 1.3M high-quality functions with input-output pairs for Case2Code reasoning. We hold out 500 samples for evaluation and the rest for training. For the hold-out evaluation samples, we further prompted GPT-4 (`gpt-4-turbo-2024-04-09`) to generate additional input examples and collect the corresponding outputs for a more strict inductive reasoning evaluation.

Training Setup   To demonstrate the generalization and effectiveness of our synthetic training data, we conduct three variants of Case2Code training: direct fine-tuning, mixed pre-training, and mixed fine-tuning. All Case2Code variants are trained for 5k steps with a batch size of 64, a maximum context window size of 4096, and apply linear warmup and cosine decay of the learning rate from the peak value of 2e-5 to 5e-6. All model training is completed on two servers of eight A800 GPUs. We conduct training on open-sourced models, i.e. InternLM2-7B (Cai et al., 2024) and LLaMA3-8B (AI@Meta, 2024) to verify the effectiveness of synthetic training data on different model series.

Evaluation Setup   We evaluate the coding ability of trained LLMs with HumanEval, MBPP. To conduct strict evaluation, we use EvalPlus, an extension to the original HumanEval and MBPP with massive additional test cases. For models that are not instructed tuned, we apply zero-shot prompting and four-shot prompting for HumanEval and MBPP evaluation, respectively. And for instructed-aligned LLMs, we use zero-shot prompting on all

these benchmarks. To evaluate inductive reasoning on code, we test various LLMs on solving Case2Code tasks, with zero-shot prompting. When evaluating the instructed models that are not tuned on Case2Code task, we find the performance is unstable and sensitive to the prompts. We manually optimized the prompts for Case2Code evaluation to elicit the actual inductive reasoning ability of these models. We use greedy decoding during the inference for all experiments.

Models   We compare the trained models with several families of representative LLMs: GPT series(OpenAI, 2023), CodeLLaMA (Rozière et al., 2023), LLaMA2 (Touvron et al., 2023) and LLaMA3 (AI@Meta, 2024). For GPT series, we evaluate GPT-3.5 (`gpt-3.5-turbo-0125`) and GPT-4 (`gpt-4-turbo-2024-04-09`). For other model series, we evaluate their available open-sourced versions.

4.2 Zero-shot Case2Code is Challenging for Current LLMs

As shown in Table 1, we report the zero-shot Case2Code performance of different representative LLMs and their programming performance. We can find that the zero-shot Case2Code performance of representative models is strongly related to their corresponding program synthesis performance. Models with higher program synthesis scores tend to achieve higher Case2Code performance. And larger models often outperform small models. This indicates that Case2Code can become a good benchmark to reflect the code reasoning performance of different LLMs. However, the

	Train w/ Ours	HumanEval	HumanEval+	MBPP	MBPP+	Case2Code
InternLM2-7B-Base	✗	31.1	21.3	51.4	40.3	27.2†
w/ Direct Fine-tuning	✓	**44.5**	34.8	56.0	40.4	**44.4**
w/ Mixed Pre-training	✓	43.9	**40.9**	**58.4**	**42.6**	41.4
InternLM2-7B	✗	39.0	33.4	56.8	**54.1**	25.6†
w/ Direct Fine-tuning	✓	43.3	40.9	54.5	40.6	**44.5**
w/ Mixed Pre-training	✓	47.6	37.2	58.4	45.6	42.4
w/ Insturction-tuning	✗	49.4	43.9	58.0	50.4	6.2
w/ Mixed Instruction-tuning	✓	**64.6**	**56.7**	**63.4**	52.4	44.0
LLaMA3-8B	✗	35.4	20.1	59.1	45.1	29.2†
w/ Direct Fine-tuning	✓	43.2	39.0	50.6	35.1	44.8
w/ Mixed Pre-training	✓	47.6	40.9	55.6	41.1	42.6
w/ Insturction-tuning	✗	49.8	45.7	57.6	47.9	8.6
w/ Mixed Instruction-tuning	✓	**64.8**	**57.9**	**71.2**	**53.1**	**45.0**

Table 2: Results of models trained with our synthetic dataset and the corresponding generalization performance. Case2Code performance are evaluated with zero-shot prompting, except results with †, which are evaluated with four-shot prompting.

zero-shot Case2Code scores of LLMs have a large gap compared with their coding accuracy, which demonstrates that existing LLMs are better at some types of reasoning (e.g. writing programs based on instructions) than others (e.g. inductive programs by their behaviors). This can be explained as the LLMs are trained with massive program generation data but fewer samples similar to Case2Code that need inductive reasoning. Similar to the Reverse Curse (Berglund et al., 2023), models trained with deductive reasoning data struggle to transfer to inductive reasoning tasks.

4.3 Generalization of Training on Case2Code

One essential issue of synthetic data is its generalization ability. Therefore, we train different LLMs with our synthetic Case2Code dataset under various settings to explore how it affects the learning of code reasoning of LLMs.

4.3.1 Direct Fine-tuning

First, we find that LLMs that are directly trained on the Case2Code reasoning samples can effectively learn coding based on cases. As shown in Table 2, by direct fine-tuning, Internlm2-7B and LLaMA3-8B can significantly outperform the few-shot prompting baselines by up to 18.9%, achieve up to 44.5% and 42.0% accuracy on Case2Code evaluation set, respectively, which even outperforms the more powerful LLMs like LLaMA3-70B, GPT-3.5, and comparable with GPT-4 (results in Table 1). Moreover, models

trained with Case2Code reasoning also improve their program synthesis performance on benchmarks like HumanEval and MBPP. This indicates that the Case2Code reasoning is general and challenging. Training on Case2Code samples not only boosts the inductive reasoning performance in distribution but enhances the code understanding and code generation abilities of LLMs. As the Case2Code samples can be synthetic at scale, we believe that synthesizing large-scale and high-quality inductive reasoning data is a promising path to consistently improve LLMs without exhausting data.

4.3.2 Mixed Training

Then, we explore how to better incorporate our synthetic Case2Code data into different stages of LLM training to enhance the reasoning ability of LLMs in general. Specifically, we train LLMs with two variants of data mixing, either during pre-training or in the supervised fine-tuning (SFT) stage. The first mixing strategy introduces natural language pre-training texts from the Pile (Gao et al., 2021) and the code pre-training samples from The Stack (Kocetkov et al., 2022). The mixing ratio is 1:1:2 for samples from the Pile, The Stack, and the Case2Code dataset, respectively. On the other hand, we incorporate a supervised fine-tuning (SFT) dataset from WizardCoder (Luo et al., 2023) to demonstrate that the performance gain of Case2Code training does not come from the understanding of instructions but the learning of in-

ductive reasoning of code execution. We combine the SFT dataset with Case2Code samples in a 1:3 ratio, as the size of our synthetic dataset is much larger.

Mixed Pre-training As shown in Table 2, when incorporated into the pre-training stage, the Case2Code training data helps the model to connect the execution states with the function implementation, which further facilitates the program synthesis performance of these LLMs. Compared with directly fine-tuned on Case2Code dataset, training these samples with pre-training texts enables the generalization of inductive reasoning of code states learned by the Case2Code task.

Mixed Instruction-tuning When trained with instruction-following datasets, the Case2Code data also improves the performance of the programming with instruction tasks, as reported in Table 2. We evaluate the SFT models with the zero-shot instructed version of programming synthesis tasks, HumanEval, and MBPP. We find that incorporating Case2Code data boosts the performance of various LLMs on code generation tasks. Compared to the corresponding SFT baselines, InternLM2-7B improves on HumanEval from 49.4% to 64.6%, with more than 10% improvements. LLaMA3-8B achieves 64.6%, 57.9%, and 71.2% on HumanEval, HumanEval+, and MBPP, respectively, with significant improvements compared to the SFT version. These results demonstrate the effectiveness of learning on Case2Code and the necessity of incorporating inductive reasoning data into LLM training.

4.4 Ablation Study

In this section, we conduct ablation studies to demonstrate the effectiveness of the Case2Code synthetic pipeline across different families and scales of LLMs.

Prompt Diversity of Training Data Since the synthetic Case2Code training data is converted by triples of (*programs*, *inputs*, *outputs*), during the construction, the prompt templates are utilized to embed the input-output pairs to form natural language texts for LLM to learn. As the LLM can only rely on these converted prompts to learn the Case2Code, it is important to understand the effectiveness of how different prompt templates affect the training of LLMs. Intuitively, the diversity of prompt templates plays an important role in the learning of LLMs. Therefore, we compare
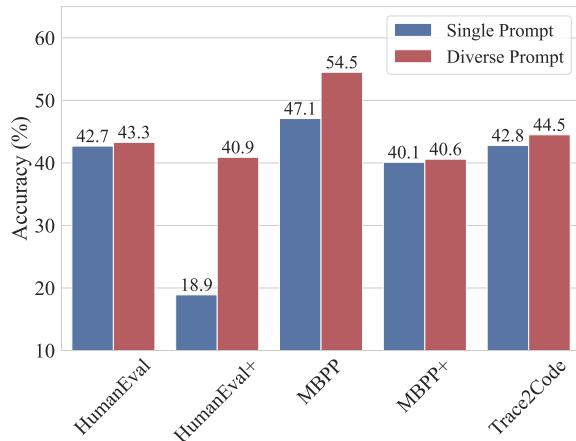

Figure 3: Downstream results when directly fine-tuning InternLM2-7B with different Case2Code prompt templates. Diverse prompts not only help the model to learn Case2Code reasoning but also significantly advance the generalization of the code inductive reasoning.

synthetic data prompted using a single template style with data utilizing diverse styles of templates. The result is reported in Figure 3, in which diverse prompts may have little effect on the in-domain Case2Code performance, however, the diversity significantly affects the accuracy of LLMs on out-of-domain program synthesis tasks. It is indicated that diversity can be critical during LLM learning, which also has been discussed in other domains like in general natural language processing tasks (Wei et al., 2022) and alignment (Ouyang et al., 2022; Wang et al., 2023b).

	HumanEval	HumanEval+	Case2Code
LLaMA3-8B	35.4	20.1	29.2
w/ Case2Code (full)	**47.6**	**40.9**	**42.6**
w/ Case2Code (code only)	38.41	28.66	28.2

Table 3: Models trained with Case2Code (Full) achieve higher accuracy on multiple datasets than models trained on Case2Code (Code only), indicating the effectiveness of combining the I/O messages for Case2Code learning.

Importance of I/O pairs We extract code data from the case2code training data and remove other parts like prompts form the training set, Case2Code (code only). Then, we train Llama3-8B under the same setups of the mixed pre-training in Section 4.3.2, replacing the Case2Code (full) training set with the Case2Code (code only) set. Compared with mixed pre-trained models with Case2Code (code only) dataset, the Case2Code (full) trained models achieve much higher accuracy both on in-
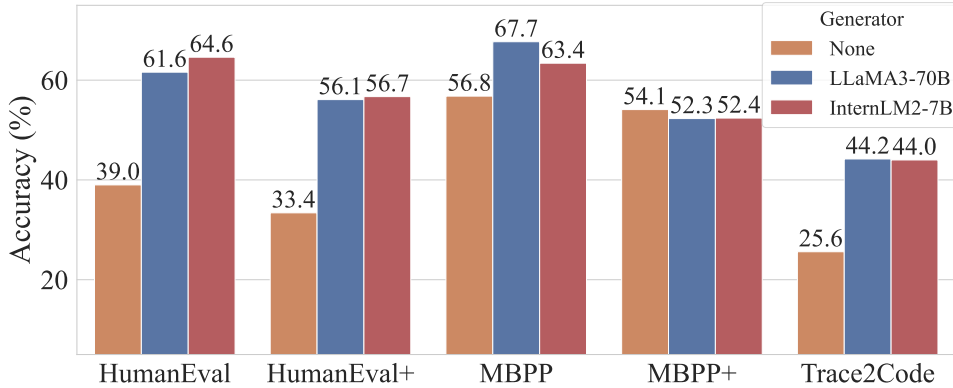
Figure 4: Downstream results when fine-tuning InternLM2-7B with synthetic data using different input example generators. Generator "None" refers to the baseline InternLM2-7B not trained on any Case2Code data. The computational overhead of using LLaMA3-70B is $4.5\times$ that of InternLM2-7B.

	TP	TGS	Costs	# Samples
InternLM2-7B	1	1600 tokens/s	$1\times$	1.3M
LLaMA3-70B	4	720 tokens/s	$4.5\times$	700K

Table 4: Efficiency of using different LLM Writers for Input Generation. "TP" refers to the size of the tensor parallel for inference. "TGS" refers to the inference throughput (tokens/s) of each LLM instance. "Costs" refers to the relative compute costs of different LLM generators. Due to the large TP and low throughput, the large LMs can be more costly than the small LMs when inferencing on the same number of GPUs. In our data synthetic process, using LLaMA3-70B costs about $9\times$ compute resources compared to small models like InternLM2-7B. Due to the high costs of LLaMA3-70B, we only sub-sample the raw data to run the data synthesis. The total costs are still $4.5\times$ compared to InternLM2-7B.

distribution Case2Code task and out-of-distribution tasks like HumanEval and HumanEval+, demonstrating the effectiveness of combining the I/O messages with the corresponding code and the incorporation of synthetic case2code samples.

Choice of Input Generator During the synthesis of Case2Code data, a critical step is prompting the LLM to write several input examples for each program. These inputs are then executed with the corresponding programs one by one to obtain the program outputs, thus we can utilize these important contexts to construct Case2Code training data. To explore whether the reasoning ability of the LLM writer affects the synthetic data quality, we replace the LLM generator from Interlm2-7B to LLaMA3-70B, and rerun the data synthesis pipeline to obtain a new version of Case2Code training data.

Due to the high costs of LLaMA3-70B, we only generate half the size of our original synthetic data. Detailed generation costs are reported in Table 4. We train Interlm2-7B with this version of Case2Code dataset under the instruction-tuning setup to evaluate the data quality. As shown in Figure 4, compared with the InternLM2-7B generator, large LMs like LLaMA3-70B can write high-quality input samples that help trained LLMs to achieve comparable code reasoning capability with fewer training data. It indicates that the input generation step can affect the overall synthetic data quality, suggesting data collectors choose a strong LLM to be the input writer if compute resources are sufficient. However, we note that LLaMA3-70B contains too many parameters that are $4.5\times$ more costly than InternLM2-7B. By generating inputs with InternLM2-7B, our Case2Code data synthesis framework maintains generation efficiency and data quality. It also demonstrates the possibility of self-improving for LLMs on their code reasoning capabilities.

Improvements on Different Model Scale We want to explore whether the Case2Code data synthesized using a small model can still improve a large model, and how the model scale affects the learning process. Therefore, we use Case2Code data generated with InternLM2-7B to train models in the InternLM2 series to investigate these questions. The training is taken under the setting of data mixing with SFT dataset (Luo et al., 2023) and the results are shown in Table 5. Our synthetic data consistently enhances the code reasoning performance of various sizes of LLMs, even though one of the student models is almost three

	HumanEval	HumanEval+	MBPP	MBPP+	Case2Code
InternLM2-1.8B	32.3	29.9	43.6	24.3	27.8
InternLM2-7B	64.6	56.7	63.4	52.4	42.2
InternLM2-20B	**73.1**	**65.2**	**77.4**	**55.4**	**46.0**

Table 5: Code results with different scales of models, after supervised fine-tuning on the instruction-following dataset mixed with Case2Code synthetic data.

times larger than the model used for data synthesis. These results demonstrate the possibilities of weak-to-strong supervision in code-related tasks at scale.

5 Conclusion

We first construct a new benchmark Case2Code to evaluate the inductive reasoning capability of LLMs in the code domain. Then, we propose a data synthetic framework to construct Case2Code training samples at scale. By just using small LLMs and a code interpreter, we can collect high-quality Case2Code data from pre-training code texts automatically and efficiently. By training on various LLMs in multiple settings, we demonstrate the Case2Code can improve not only the inductive reasoning ability of LLM but also the general coding capabilities. We believe synthetic Case2Code is a promising way to continue improving the LLMs when human-generated data is exhausted.

Limitations

In this work, we study Case2Code, a synthetic task for learning inductive reasoning capabilities. Our work is still limited in several aspects:

- Potential harmful programs: we gather and filter programs from the pre-training code corpus, which excludes code that may contain dangerous operations like system calls, file manipulation, and network traffic that require careful safety checks and vulnerability mitigation. In the future one can incorporate a safe and reliable execution environment that supports these operations for Case2Code synthesis.

- Programming languages: we focus on synthesizing Case2Code data using Python programs, as it is a commonly used programming language and can be easily and reliably manipulated and executed. Future work can extend the data synthesis framework to more

programming languages and applications.

- Long context: some inputs or outputs of the given programs can be extremely long, which can be challenging to fit into the context window of current LLMs. Future work can explore efficient methods of representing and learning long-context case-to-code induction.

- Data modality: we represent cases in our Case2Code data as texts for LLM training, however, real-world programs often interact with multi-modal inputs and outputs like audio, image, and video. How to effectively collect and learn multi-modal inductive reasoning remains a big challenge.

Acknowledgments

References

AI@Meta. 2024. Llama 3 model card.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. 2023. The reversal curse: Llms trained on "a is b" fail to learn "b is a". *CoRR*, abs/2309.12288.

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, Xiaoyi Dong, Haodong Duan, Qi Fan, Zhaoye Fei, Yang Gao, Jiaye Ge, Chenya Gu, Yuzhe Gu, Tao Gui, Aijia Guo, Qipeng Guo, Conghui He, Yingfan Hu, Ting Huang, Tao Jiang, Penglong Jiao, Zhenjiang Jin, Zhikai Lei, Jiaxing Li, Jingwen Li, Linyang Li, Shuaibin Li, Wei Li, Yining Li, Hongwei Liu, Jiangning Liu, Jiawei Hong, Kaiwen Liu, Kuikun Liu, Xiaoran Liu, Chengqi Lv, Haijun Lv, Kai Lv, Li Ma, Runyuan Ma, Zerun Ma, Wenchang Ning, Linke Ouyang, Jiantao Qiu, Yuan Qu, Fukai

Shang, Yunfan Shao, Demin Song, Zifan Song, Zhi-hao Sui, Peng Sun, Yu Sun, Huanze Tang, Bin Wang, Guoteng Wang, Jiaqi Wang, Jiayu Wang, Rui Wang, Yudong Wang, Ziyi Wang, Xingjian Wei, Qizhen Weng, Fan Wu, Yingtong Xiong, and et al. 2024. Internlm2 technical report. *CoRR*, abs/2403.17297.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. 2024. Alphamath almost zero: process supervision without process. *ArXiv*, abs/2405.03553.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Push-meet Kohli. 2017. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pages 835–850.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2021. The pile: An 800gb dataset of diverse text for language modeling. *CoRR*, abs/2101.00027.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for A multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2023. Large language models can self-improve. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 1051–1068. Association for Computational Linguistics.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 TB of permissively licensed source code. *CoRR*, abs/2211.15533.

Wen-Ding Li and Kevin Ellis. 2024. Is programming by example solved by llms? *CoRR*, abs/2406.08316.

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. *CoRR*, abs/2306.08568.

Arindam Mitra, Hamed Khanpour, Corby Rosset, and Ahmed Awadallah. 2024. Orca-math: Unlocking the potential of slms in grade school math. *CoRR*, abs/2402.14830.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/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*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Kensen Shi, Joey Hong, Manzil Zaheer, Pengcheng Yin, and Charles Sutton. 2023. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *CoRR*, abs/2307.13883.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489.

Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, Qipeng Guo, Xipeng Qiu, Pengcheng Yin, Xiaoli Li, Fei Yuan, Lingpeng Kong, Xiang Li, and Zhiyong Wu. 2024. A survey of neural code intelligence: Paradigms, advances and beyond. *CoRR*, abs/2403.14734.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. 2023a. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *CoRR*, abs/2312.08935.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. 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), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. 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 Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart Van Merriënboer, Armand Joulin, and Tomas Mikolov. 2015. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*.

Jiarong Wu, Lili Wei, Yanyan Jiang, Shing-Chi Cheung, Luyao Ren, and Chang Xu. 2024. Programming by example made easy. *ACM Trans. Softw. Eng. Methodol.*, 33(1):4:1–4:36.

Zonglin Yang, Li Dong, Xinya Du, Hao Cheng, Erik Cambria, Xiaodong Liu, Jianfeng Gao, and Furu Wei. 2022. Language models as inductive reasoners. *arXiv preprint arXiv:2212.10923*.

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*, abs/2305.10601.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023a. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*.

Yue Yu, Yuchen Zhuang, Jieyu Zhang, Yu Meng, Alexander J. Ratner, Ranjay Krishna, Jiaming Shen, and Chao Zhang. 2023b. Large language model as attributed training data generator: A tale of diversity and bias. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Chuanqi Tan, and Chang Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *CoRR*, abs/2308.01825.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

A Prompts Used in Case2Code

We demonstrate the prompts used during the Case2Code synthesis, training, and evaluation as follows:

- The prompt template for evaluating zero-shot Case2Code performance of various LLMs is listed in Table 6.

- We show the prompt for using LLMs as input generators for synthesizing Case2Code data in Table 7.

- We randomly sample some Case2Code data to demonstrate in Table 8.

Prompt Template for Zero-shot Case2Code Evaluation.

```
{prompt}

Please write the correct names of arguments. As the function you implement will be
called by: {func_name}(**input_dict). Keep the original type. No need to convert
the output to string.
```

Table 6: Prompt template for zero-shot Case2Code evaluation. We inject {prompt} and {func_name} for each test sample for evaluation.

Prompt for LLM Input Generator

```
Given the function, first analyze the types of the function arguments, then write
10 different example inputs for the function, each example should be a dict with
function arguments' names and their values.
Output format:
```python
examples = [
    dict(argname=argvalue),
    ....
]
```

Function:
```python
def test_func(a: int, b: str) -> str:
    return str(a) + b
```
Examples:
```python
examples = [
    dict(a=1, b='a'),
    dict(a=2, b='b'),
    dict(a=3, b='c'),
    dict(a=4, b='d'),
    dict(a=5, b='e'),
    dict(a=6, b='f'),
    dict(a=7, b='g'),
    dict(a=8, b='h'),
    dict(a=9, b='i'),
    dict(a=10, b='j'),
]
```

Function:
```python
{code}
```
Examples:
```

Table 7: Prompt for LLM input generator, we replace {code} with programs collected in for Case2Code.

Case2Code Examples

```
Input: dict(s="abcba", center=2), Output: "(5, 0, 4)"
Input: dict(s="abcdefg", center=3), Output: "(1, 3, 3)"
Input: dict(s="aba", center=1), Output: "(3, 0, 2)"
Input: dict(s="racecar", center=3), Output: "(7, 0, 6)"
Input: dict(s="madam", center=2), Output: "(5, 0, 4)"
Input: dict(s="abcabcabc", center=4), Output: "(1, 4, 4)"
Input: dict(s="xyzyx", center=2), Output: "(5, 0, 4)"
Input: dict(s="hello", center=2), Output: "(1, 2, 2)"
Input: dict(s="ab", center=0), Output: "(1, 0, 0)"
Input: dict(s="a", center=0), Output: "(1, 0, 0)"

Write a function that generates the output from the input.
Function: greatest_palindrome_size_odd

The function is:

```python
def greatest_palindrome_size_odd(s, center):
    right = center + 1
    left = center - 1
    size = 1
    optimum_right = optimum_left = center
    while left >= 0 and right < len(s):
        if s[left] == s[right]:
            size += 2
            optimum_left = left
            optimum_right = right
            right += 1
            left -= 1
        else:
            break
    return size, optimum_left, optimum_right
```
```

```
Arguments and results:
Input: dict(seq="ATCG", complementarity={'A': 'U', 'T': 'A', 'C': 'G', 'G': 'C'}),
Output: "'CGAU'"
Input: "ATCG", {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}, Output: "'CGAT'"
Input: seq:"ACGT", complementarity:{'A': 'U', 'T': 'A', 'C': 'G', 'G': 'C'}, Output:
"'ACGU'"
Input: "ACGT", {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}, Output: "'ACGT'"

Please write a function to process the input arguments and produce the specified
outputs.

Start with the function:
reverse_complement

The function is:

```python
def reverse_complement(seq, complementarity):
    bases = list(seq)
    bases = [complementarity[base] for base in bases]
    reversed_complement = ''.join(bases)
    return reversed_complement[::-1]
```
```

Table 8: Case2Code data examples.