What can Large Language Models Capture about Code Functional Equivalence?

Nickil Maveli **Antonio Vergari** **Shay B. Cohen**
School of Informatics, University of Edinburgh
10 Crichton Street, Edinburgh, EH8 9AB
{nickil.maveli,avergari,scohen}@ed.ac.uk

Abstract

Code-LLMs, large language models pre-trained on code corpora, have shown great progress in learning rich representations of the structure and syntax of code, successfully using it to generate or classify code fragments. At the same time, understanding if they are able to do so because they capture code semantics, and how well, is still an open question. In this paper, we tackle this problem by introducing SeqCoBench, a benchmark for systematically assessing how Code-LLMs can capture code functional equivalence. SeqCoBench contains over 20 code transformations that either preserve or alter the semantics of Python programs. We conduct extensive evaluations in different settings, including zero-shot and parameter-efficient finetuning methods on state-of-the-art (Code)-LLMs to see if they can discern semantically equivalent or different pairs of programs in SeqCoBench. We find that the performance gap between these LLMs and classical match-based retrieval scores is *minimal*, with both approaches showing a concerning lack of depth in understanding code semantics.[1]

1 Introduction

Comprehending the semantics of code is crucial to generate new code accurately as well as to understand and verify existing code. Capturing code semantics would entail the ability to predict *code functional equivalence*, i.e., the property of two functions to produce the same outputs when given the same inputs, yielding the same observable behaviour, even if their implementations differ syntactically. In other words, functionally equivalent functions are interchangeable from the perspective of a program's functionality.

Identifying such functional equivalences is important for software development and formal verification, as it enhances software quality by detecting redundant code, encouraging reusability, preventing bug spread (Mondal et al., 2018), and boosting developer productivity. For example, when code is refactored (Shirafuji et al., 2023) or optimized (Shypula et al., 2024), it is desirable to automatically confirm that the new and old implementations behave the same. In static analysis, it is useful for reducing the risk of unexpected behaviour in a piece of code (Ding et al., 2023), reducing the effort required to verify a system's correctness and to find errors or inconsistencies in the code.

While determining the equivalence between two code segments is an undecidable problem in general (Poonen, 2014), in practice, this can be partially achieved by focusing on a narrower input and code domain and by running unit tests on it. These execution-based code evaluation strategies have become increasingly widespread for evaluating *code generation tasks*, such as program synthesis, code translation and code summarization (Huang et al., 2022; Wang et al., 2023c) with Code-LLMs, LLMs pre-trained on large code corpora (Rozière et al., 2024; Li et al., 2023).

However, execution-based evaluation comes with drawbacks. Firstly, it cannot scale to complex codebases that resemble real-world software domains. Currently, the test cases apply mostly to closed-domain problems having limited coverage due to the presence of either built-in functions (Li et al., 2022) or handpicked libraries from a specific field (Lai et al., 2023). Secondly, it is infeasible to cover all possible inputs, edge cases, and execution traces, and while passing tests is a good proxy for functional correctness, it does not necessarily imply the model truly understands the semantics behind the code. This leaves us with the open question: *how much are Code-LLMs that are remarkable at code generation able to identify aspects of semantics such as code functional equivalence?*

In this work, we try to answer this question

[1] Our code and dataset is available at https://github.com/Nickil21/SeqCoBench.

Shared supervision.

Findings of the Association for Computational Linguistics:
NAACL 2025, pages 6880–6918

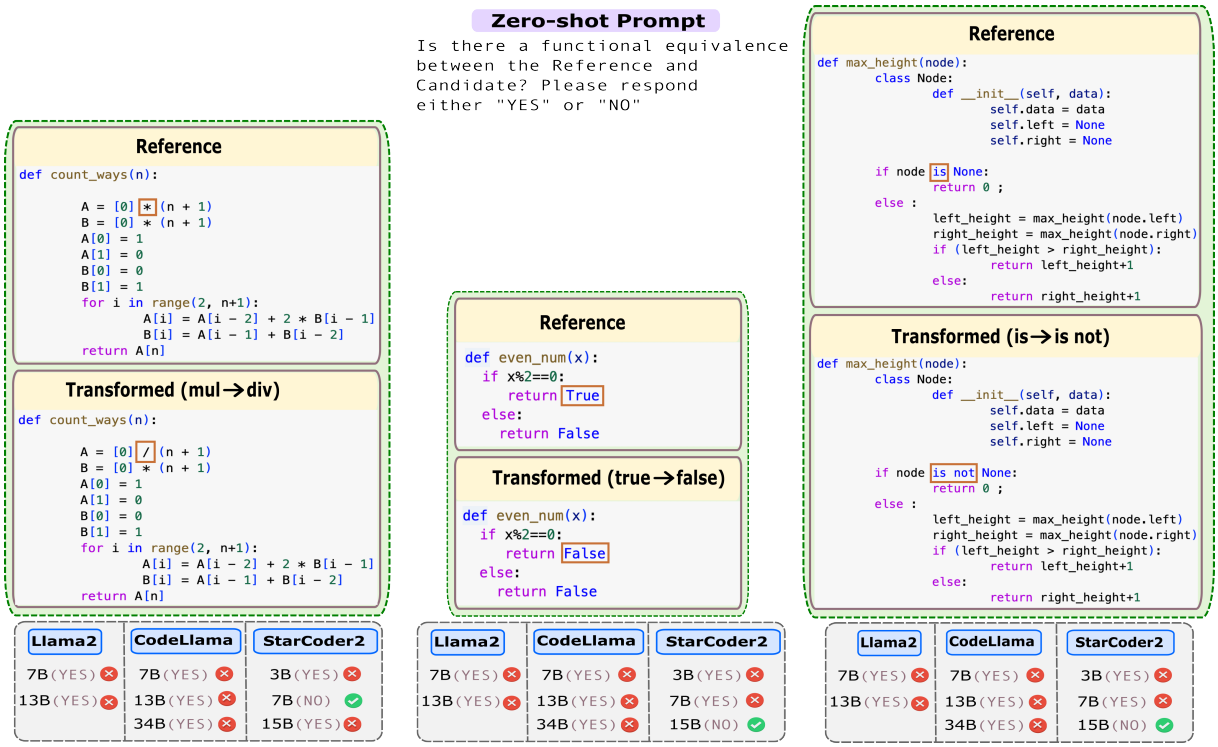

Figure 1: **State-of-the-art (Code)-LLMs struggle to understand subtle changes in one program syntax that, however, dramatically alter its semantics** as shown by three semantic-altering transformations from SeqCoBench involving arithmetic, boolean and identity operator misuses (Section 4.2).

by introducing the ***Semantic Equivalence Code Benchmark*** (SeqCoBench). SeqCoBench is a challenging benchmark comprising Python programs generated by applying semantic transformations to a reference program, with the objective of preserving (resp. altering) its semantics while altering (resp. preserving) most of its syntax. When called to evaluate if the two pieces of code in a pair from SeqCoBench are functionally equivalent or not, state-of-the-art (Code)-LLMs can get confused, as illustrated in Fig. 1. Our findings indicate that Code-LLMs have a weak sense of code semantics that breaks when we introduce subtle variations in our SeqCoBench dataset, which helps to systematically measure this effect.

To summarize our contributions: **(a)** We design SeqCoBench to comprise various semantic transformations (Section 4) according to different types of code clones (Section 3) and to understand better which fragment of the syntax-semantics spectrum LLMs capture. **(b)** We use SeqCoBench to extensively evaluate not only state-of-the-art (Code)-LLMs but also classical match-based metrics to capture code functional equivalence (Section 5). This includes experiments on zero-shot learning and parameter-efficient fine-tuning (PEFT) settings, noting that Code-LLMs struggle on SeqCoBench and perform on par with classical match-based similarity metrics. **(c)** We investigate which transformations are most challenging to reason with on our benchmark.

2 Related Work

Code generation benchmarks. The common benchmarks, HumanEval (Chen et al., 2021) and Mostly Basic Python Problems (MBPP; Austin et al. 2021), help evaluate Python code synthesis based on functional correctness on relatively simple functions. HumanEval comprises 164 human-curated Python programming challenges proposed by OpenAI. Each task contains a docstring, function signature, function body, and a set of unit tests. Whereas MBPP consists of 974 crowd-sourced and hand-crafted Python functions. Each task contains a natural language description, code solution, and three test cases.

To increase test coverage, EvalPlus (Liu et al., 2023) augments HumanEval test cases to automatically generate and diversify additional test inputs. To tackle dataset contamination, LiveCodeBench (Jain et al., 2024) gathers new coding problems not seen during model training, and

EvoEval (Xia et al., 2024) which uses LLMs to transform existing benchmarks into novel coding tasks. Our approach focuses on the equivalence of the code itself rather than evaluating the correctness of a generated solution against a predefined problem.

Some benchmarks test LLMs' capability to automate real-world software development processes. SWE-Bench (Jimenez et al., 2024) automatically resolves GitHub issues by generating code patches that pass the existing test cases. RepoEval (Zhang et al., 2023a) evaluates repository-level code completion tasks at various levels of granularity. In contrast, we evaluate functional equivalence without considering the broader context of a codebase or real-world software engineering tasks. While we focus on determining whether two code snippets are functionally equivalent, CRUXEval (Gu et al., 2024) specifically evaluates a model's reasoning skills in predicting inputs/outputs for code understanding and execution abilities.

Code evaluation metrics. We can broadly categorise code evaluation metrics (CEM) into two main types: reference-based metrics, which compare the generated code to a known reference, and reference-free metrics, which assess the quality of the generated code without relying on a reference by executing them on test cases. In this study, we focus on reference-based metrics.

Reference-based metrics typically include *match-based* metrics, which rely on lexical exact token matching, and *LLM-based* metrics, which employ models pre-trained on code. Match-based metrics include n-gram matching metrics like BLEU (Papineni et al., 2002), ROUGE (Lin, 2004), etc., and also incorporate syntactic and semantic properties like CodeBLEU (Ren et al., 2020), CrystalBLEU (Eghbali and Pradel, 2023), etc. LLM-based metrics include embedding-based metrics like BERTScore (Zhang et al., 2020), CodeBERTScore (Zhou et al., 2023), CodeScore (Dong et al., 2023), etc. We provide the details of the different evaluation metrics in Appendix A. There is a need to build robust CEM as existing metrics show a weak correlation with functional correctness on HumanEval as shown in Appendix H. We cover notable work not already mentioned in Appendix J.

3 Functional Equivalence of Programs

Determining the semantic equivalence of two code snippets is a challenging task that *lies on a spectrum*, as different codes can compute the same function, but in different ways. Consider a space of programs $\mathcal{P}$, each explicitly accepting an input $x \in X$ and outputting an output $y \in Y$. The semantics we attach to such programs is based on their input-output mapping, i.e., the function $f : X \to Y$ they implement. However, while two different programs $p, p' \in \mathcal{P}$ can be *functionally equivalent* if they implement the same input-output mapping, they can concretely implement such a mapping in a very different way.

For example, two pieces of code can have different syntactic variations – from minor changes as inserting whitespaces to renaming variables – as well as implement two different algorithms that however encode the same function. This can be done simply by using different library APIs or dependencies, different abstraction levels, or algorithms with different time and space complexity, e.g., sorting a list of integers can be equally done with mergesort or insertion sort and both will pass the same unit tests that check for input-output consistency.

We operationalise the question of whether LLMs can capture different functional equivalence relationships in this spectrum through the notion of *code clones* (Saini et al., 2018). Detecting code clones is a proxy task for checking functional equivalence that is highly relevant in software engineering, e.g., to retrieve similar code snippets for code search (Sun et al., 2023) or detect duplicates within a codebase (Yang et al., 2023). Classifying code clones into types can help us systematize possible functional equivalence classes. Following previous work (Roy and Cordy, 2007; Bellon et al., 2007), we categorise code clones into four types based on their complexity and degree of similarity.

type-1 clones comprise two almost syntactically identical pieces of code that differ only for minor variations in the layout, e.g., by the presence of whitespace and comments.

type-2 clones resemble two syntactically identical code fragments except for variations in variable and function names, identifiers, literal values, types, etc. They also comprise **type-1** differences.

type-3 clones include two syntactically identical code fragments except for additions, deletions, modifications of several statements, and the differences already specified for **type-2** clones.

type-4 clones, also known as semantic clones. They exhibit identical functional behaviour despite having different syntax, control flow, data flow, or

programming languages.

This typology of clones helps us devise diverse groups of transformations that can preserve or alter the semantics of a program while also modifying its syntax. Such transformations enable us to create a challenging benchmark to systematically evaluate whether LLMs can capture functional equivalence and inspect at which level they can disentangle syntax from semantics, as discussed next.

4 The SeqCoBench Dataset

Traditionally, code benchmarks for evaluating LLMs have focused on assessing their ability to generate single-function programs based on natural language descriptions. This is done for example in popular benchmarks such as HumanEval (Chen et al., 2021) and Mostly Basic Python Problems (MBPP; Austin et al. 2021), which comprise human-curated Python code snippets with a docstring, function signature, function body, and a set of unit tests to check if generated code satisfies specifications.

Instead, as we want to evaluate the ability of LLMs to capture functional equivalence between already existing pieces of code, we construct our SeqCoBench by creating pairs of code snippets that are labelled to be functionally equivalent or not. More formally, given a program p_i, we generate the tuple $(p_i, t(p_i), \ell_i)$ where t is a *semantic-preserving* (SP; Section 4.1) or *semantic-altering* (SA; Section 4.2) transformation that generates a new code snippet whose semantics changes accordingly and $\ell_i \in \{0, 1\}$ is a label indicating if the pair has been generating through a SP (1) or SA (0) transformation.

We build SeqCoBench by applying a set of SP or SA transformations to programs appearing in MBPP, they provide unit tests that help us check if our transformations correctly operate on the program semantics. We prepare the train/valid/test splits following a 60/16/24 ratio and ensure that there is no overlapping of the original code across different data splits.

4.1 Semantic-Preserving Transformations

Given a program p, we aim to generate a new code snippet p' that is ***functionally equivalent*** to p (i.e., they encode the same input-output mapping f) ***while maximizing the token-level differences between the original code*** where appropriate. To this end, we consider the four SP transformations. We group them by the corresponding clone type (Section 3) while trying to cover all types. Note that we omit type-1 clones as they typically include comment- and docstring-level perturbations written in natural language and are not technically part of code semantics. They are used for documentation purposes only and do not affect the execution or behaviour of the code itself.

While adversarial perturbations that alter the function's intended meaning can cause the LLM to ignore the function body completely and instead give more emphasis to the perturbed entity's instructions, reformulating the perturbations using text augmentation strategies such as back-translation (Wang et al., 2023a), synonym substitutions, etc., can make it less challenging for LLMs. We create these transformations using the NatGen package (Chakraborty et al., 2022). We leave out transformations on the original code where the necessary requirements to perform the transformation are not met (e.g., lack of for/while loop or boolean operators in the code snippets).

Rename Variables (RV) type-2 We primarily use three different adaptations. *Naive:* It renames the most common variable name to VAR_i. *CB:* It identifies the variable name that appears most frequently in the partial code snippet and then substitutes all occurrences of that variable name throughout the prompt with a new name suggested by CodeBERT. *RN:* It identifies the variable name that occurs most frequently within the given partial code snippet and then generates a *random* string composed of an equal mix of alphabetic and numeric characters. Finally, it substitutes all instances of the most commonly used variable name with this newly generated random string.

Dead Code Insertion (DCI) type-3 It creates unreachable code blocks at a random location. These could be unused variables or redundant assignments. We place these statements in a block around either a looping (e.g., for, while) or a branching structure (e.g., if-else, switch), if any.

Dead Code Insertion Example

```
x = 5
y = x + 2
print(y)
```

```
x = 5
z = 10 # Dead code
y = x + 2
if False:
    print("This will never execute") # Dead code
print(y)
```

Operand Swap (OS) type-3 It swaps the first occurrence of boolean operators and, if needed, changes the operator to preserve semantic equiv-

alence. For example, if the original code had the condition `a > b`, the transformation could change it to `b < a`, swapping the operands "a" and "b" while also changing the operator from ">" to "<" to preserve the same logical meaning.

Loop Transformation (LT) type-4 It converts the first occurrence of for-loop into its equivalent while-loop and vice-versa.

In the for→while case, we initialize the counter outside the loop and use a while condition that checks the loop counter against the termination condition. Then, we increment/decrement the counter inside the body, which remains unchanged. For the reversed case, we initialize the loop counter in the for-loop statement and include the termination condition taken from the while-loop. The loop counter increment/decrement is merged in the for loop, and the body remains unchanged.

Loop Transformation Example

```
total = 0
for i in range(n):
    total += i
```

```
total = 0, i = 0
while i < n:
    total += i
    i += 1
```

Loop transformation is challenging as loops often contain critical logic and control flow determining the code's functionality. Modifying or transforming loops risks changing the program's intended behaviour. In contrast, simpler code modifications like dead code insertion, variable renaming, or operand swaps have more localized effects and require less global reasoning about data dependencies or code semantics. LLMs can perform these transformations more reliably by learning from the training data patterns. We show the structure of transformations for a representative program taken from the dataset in Appendix I.

4.2 Semantic-Altering Transformations

In this case, our goal is to generate a program $t(p)$ that is functionally not equivalent to the original code p while ***maximizing token-level similarity to the original code***. In other words, generate pairs of programs that are *not* clones but might fool a superficial comparison. Accordingly, we consider six families of SA transformations. As before, we leave out transformations on the original code where the necessary requirements to perform the transformation are not met (e.g., unavailability of identity or boolean operators).

Arithmetic Operators Misuse (AOM). We search for the first occurrence of an arithmetic operator and modify it to its semantic opposite counterpart. For example, we replace `a + b` with `a - b`, `a * b` with `a / b`, and the other way around. Similarly, we replace augmented assignment operators `a += b` to `a -= b` and `a *= b` to `a /= b`.

Dissimilar Code Selection (DCS). We randomly select five distinct code snippets from the base dataset, excluding the original code p, and create five additional code pairs using p as a reference.

Identity Operators Misuse (IOM). We look for the first occurrence of an identity operator and adjust it to its corresponding semantic opposite. For instance, we replace `a is c` with `a is not c`, and vice-versa.

Boolean Operators Misuse (BOM). It searches for the first occurrence of a boolean literal and replaces it with its logical negation. For example, we replace the keyword `True` with `False`, and vice-versa.

Logical Operators Misuse (LOM). It scans for the first occurrence of a logical operator (e.g., `and`, `or`) and swaps it with another operator. For instance, we interchange `a > 1 and a < 5` with `a > 1 or a < 5`, and vice-versa.

Comparison Operators Misuse (COM). It searches for the first occurrence of a comparison operator and replaces it with its logical opposite operator type. For e.g., we replace `a > b` with `a < b`, `a >= b` with `a <= b`, `a == b` with `a != b`, and vice-versa.

Split	Size	Unique Functions	Transformations	
			SP	SA
Train	7214	565	3085	4129
Valid	2943	229	1291	1652
Test	4415	341	1860	2555

Table 1: An overview of SeqCoBench statistics. Transformation-wise breakdown of counts is shown in the Appendix E.

Table 1 summarizes the dataset statistics after applying the transformations to the code fragments in MBPP and splitting into train/valid/test. Furthermore, we clarify how the motivation to construct SeqCoBench differs from code clone detection and code obfuscation techniques in the Appendix K.

While many of the transformations, especially the SP ones, overlap with BiFi (Yasunaga and Liang, 2021) and are well-studied in software engineering literature, our motivation is completely different and tackles an altogether different scenario. We focus on a controlled environment where we

can understand strictly which simple transformations can confuse LLMs by discriminating syntax from semantics. SP transformations provide a controlled way to introduce changes while maintaining functional equivalence. This allows for a more precise analysis of what the LLM considers significant or insignificant regarding code structure and syntax. From a practical perspective, without grasping the code semantics completely, LLM-generated code may contain subtle logical errors or edge cases that are difficult to detect through surface-level code evaluation, such as identifying subtle bugs introduced during code refactoring that shouldn't alter functionality. This increases the importance of careful code review by experienced developers.

5 Experiments

Through the use of our dataset, we aim to answer the following research questions:
RQ1: How good are state-of-the-art LLMs at zero-shot classification on SeqCoBench?
RQ2: Which are the most/least challenging semantic transformations for Code-LLMs?
RQ3: Can the performance improve with fine-tuning on some transformations?

We demonstrate that Code-LLMs are far from "solving" our dataset, leaving it for future work to further use our dataset as a benchmark for analysing the level of code understanding in such LLMs.

5.1 Models

To evaluate the benchmark, we choose general and state-of-the-art (Code)-LLMs that have performed well on code-related benchmarks (such as HumanEval) and are open-sourced for commercial use, as our test models. We exclude closed-source LLMs because we want to see the impact of fine-tuning and we are worried about possible data leaks, as HumanEval, MBPP, and relative benchmarks might have already been used to train the largest closed-source LLMs. See the model details in Appendix D.

5.2 RQ1: Match-based vs. LLM-based Metrics Performance

To evaluate the model performance, we measure the area under the precision-recall curve to calculate the average precision (AP) score:

$$\mathrm{AP} = \sum_n (R_n - R_{n-1})\, P_n, \qquad (1)$$

Type	Metric	Size	AP
Match-based	Rouge1	–	50.91
	Rouge2	–	50.67
	RougeL	–	48.48
	Meteor	–	52.05
	ChrF	–	55.25
	BLEU	–	48.46
	CrystalBLEU	–	48.35
	CodeBLEU	–	50.65
LLM-based	Comet	–	52.41
	CodeScore	126M	46.48
	BERTScore	110M	54.87
	CodeBERTScore	125M	47.45
	Generic		
	Llama2	7B	41.33
		13B	43.32
	Code-Specific		
	CodeLlama	7B	44.30
		13B	**70.85**
		34B	46.59
	StarCoder2	3B	34.11
		7B	33.91
		15B	50.75

Table 2: LLM-based metrics struggle to differentiate between semantically equivalent and non-equivalent code snippets, sometimes performing worse than surface-level match-based metrics. This indicates a lack of understanding of code semantics and reasoning based on underlying logic.

where P_n and R_n correspond to the precision and recall at the n^{th} threshold. AP accounts for ranking by rewarding models that rank correct predictions higher and are calculated per class, allowing performance evaluation on individual classes. We note that AP is the area under the precision-recall curve (AUC-PR) curve and is a more accurate metric for slightly imbalanced datasets than the usual AUC-ROC. This imbalance mainly happens when the positive class (SP) is lesser in magnitude than the negative class (SA). This is because AP focuses more on the performance of the positive class and is more sensitive to improvements in the positive class predictions compared to AUC-ROC. At the same time, AUC-ROC can give a false sense of high performance on imbalanced data.

We aim to investigate whether LLMs perform significantly better than old-school, syntax-based metrics and incorporate all CEMs proposed in the literature. While BLEU is meant to compare at the ngram-matching level, we also consider their improved variations, which include code-related modifications such as CodeBLEU and CrystalBLEU,

Transformation			*Embedding-based*				*Zero-shot-prompt-based*							
Type	**Category**	**Name**	**Comet**	**CS**	**BS**	**CBS**	**Llama2**		**CodeLlama**			**StarCoder2**		
							7B	13B	7B	13B	34B	3B	7B	15B
SA	AOM	$+ \rightarrow -$	4.46	**87.52**	0.76	1.39	20.85	14.77	14.19	15.69	16.25	24.64	14.99	16.92
		$- \rightarrow +$	4.29	**88.52**	0.57	3.30	23.29	13.95	14.04	20.44	19.44	24.93	17.35	17.66
		$\div \rightarrow \times$	4.19	**81.21**	0.71	2.40	13.11	12.95	13.15	17.46	16.28	11.96	12.62	17.63
		$\times \rightarrow \div$	4.83	**94.76**	1.01	1.45	17.38	17.40	15.49	15.18	16.64	22.32	14.37	18.20
	BOM	`False` → `True`	4.32	2.29	0.20	1.56	14.47	13.95	10.49	14.15	14.68	**21.39**	13.04	14.47
		`True` → `False`	4.14	1.91	0.23	1.75	14.94	12.55	9.22	13.12	13.44	**22.17**	10.04	9.73
	COM	`==` → `!=`	4.75	**85.09**	0.58	1.16	23.88	14.42	15.01	13.71	15.85	23.65	12.61	13.39
		`!=` → `==`	3.80	**88.77**	0.20	0.43	9.53	9.30	14.66	19.70	10.11	14.66	10.87	7.19
		$> \rightarrow <$	4.55	**91.16**	0.61	3.85	26.02	14.97	13.67	15.42	18.75	20.26	11.87	16.93
		$< \rightarrow >$	4.50	**30.78**	0.40	1.00	11.78	16.51	13.52	14.55	12.93	13.11	14.04	17.27
	DCS	Dissimilar Code Inject	30.42	**38.50**	13.39	14.07	23.09	31.49	22.44	17.09	21.55	32.43	15.02	16.48
	IOM	`is` → `is not`	2.54	3.12	0.14	0.62	12.68	9.79	10.35	12.96	13.75	**16.40**	12.36	13.57
		`is not` → `is`	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**
	LOM	`and` → `or`	4.30	**27.00**	0.25	0.76	11.54	14.61	13.54	15.66	11.99	22.52	13.45	16.37
		`or` → `and`	3.05	4.24	0.30	0.78	7.71	12.80	6.16	18.04	14.32	**24.55**	13.47	16.80
SP	DCI	Dead Code Insert	11.64	**72.95**	5.79	7.24	16.02	17.05	11.99	21.98	16.72	16.39	9.28	13.99
	LT	`for` ↔ `while` Loop	10.37	**77.64**	7.46	8.81	12.56	16.17	9.20	16.95	14.56	16.10	14.73	14.25
	OS	Operand Swap	8.20	**77.25**	4.92	5.01	21.61	18.29	11.15	20.44	14.83	14.15	12.60	13.56
	RV	Rename Variable Cb	24.46	**74.29**	9.74	6.60	22.32	18.22	11.57	21.25	17.04	14.80	13.50	16.68
		Rename Variable Naive	18.17	**72.38**	7.88	8.57	17.33	16.51	9.34	18.87	16.27	13.36	14.45	14.34
		Rename Variable Rn	34.78	**70.16**	12.57	11.34	19.31	20.28	11.64	19.99	17.39	13.45	12.44	16.97

Table 3: LLM-based metrics struggle to classify SA transformations due to their susceptibility to subtle input variations. Our findings show that LLM variants specifically trained for coding tasks outperform their more general-purpose counterparts. Here, *CS:* CodeScore, *BS:* BERTScore, *CBS:* CodeBERTScore.

which offer an upper limit for the performance of match-based metrics. In addition, we need to know whether relying on LLM-based metrics has a significant advantage (at the cost of latency, memory consumption, etc.) compared to match-based metrics. Table 2 shows the results on the SeqCoBench test set. We observe that CodeLlama (CL) outperforms Llama2 across different model sizes. This can be attributed to the fact that CL models were initialized with Llama2 weights but then further trained on a massive 500B token dataset heavily focused on code and code-related content. This specialized training data allows CL to develop a deeper understanding of programming languages, libraries, and coding conventions than the more general Llama2.

We notice that BERTScore outperforms its code-enhanced metric, CodeBERTScore (CBS). We hypothesize this could be due to multiple reasons. Firstly, while encoding the surrounding context (e.g. natural language descriptions) is beneficial for code generation, in the case of understanding tasks without generation, this additional context encoding in CBS may not provide any advantage and could even introduce noise. Secondly, it leverages pre-trained language models for code like CodeBERT, which heavily relies on the names of variables and functions to understand code semantics. When these names are obfuscated or changed, it struggles to comprehend the underlying logic and meaning of the code (Wang et al., 2024). CL-13B is the best-performing overall, outperforming the larger 34B model. While the 34B model has more parameters and performs better on benchmarks, we speculate specific coding tasks or prompts may better suit the 13B model's capabilities. The smaller model size could lead to better generalization or less overfitting on some particular tasks. Training details are shown in Appendix C. The prompt format is shown in Appendix F.

We report additional experiments that show how few-shot improves performance for some LLMs and how zero-shot CoT prompting affects the model performance in Appendix G. We can see that zero-shot CoT prompting has only marginal improvements compared to the standard zero-shot prompting. We note that zero-shot prompting can be more adaptable across different programming languages without needing to adjust the prompting strategy, as it relies more on the model's general un-

derstanding of code functionality. LLMs trained on vast amounts of code can often make accurate judgments about code equivalence without needing to "think through" the problem explicitly, effectively leveraging their pre-trained knowledge.

Answer to RQ1: *As there are only marginal improvements, we believe LLM-based metrics are not superior compared to match-based metrics despite their strong contextual understanding abilities. Bigger LLMs tend to outperform their smaller counterparts.*

5.3 RQ2: Impact of Semantic Transformations

To assess the performance of various CEMs on each transformation, we consider the true positive label for SP transformations as 1 and 0 for SA transformations. As it becomes a single-class classification per transformation, the precision will always be 1 due to zero false positives. In this case, the recall score measures the fraction of all actual positive instances correctly identified. So, we measure the area under the recall curve (AURC) corresponding to all the chosen thresholds, which varies between 0 and 1 and use the obtained result for our analysis as shown:

$$\begin{aligned} \texttt{R}, \texttt{T} &= \texttt{recall\_curve}(y, \hat{y}, posLabel) \\ \texttt{AURC} &= \texttt{auc(T,R)} \end{aligned} \tag{2}$$

where `recall_curve` refers to the plot of recall scores against different thresholds. y is the true label, $\hat{y}$ is the score probabilities and $posLabel$ is the label of the positive class. `auc` calculates the area under the recall curve using the trapezoidal rule. We generally observe that the metrics have difficulty understanding SA transformations compared to the SP transformations. We take the row with the smallest to the largest sum to determine the level of difficulty of the transformations.

Among SA transformations, "Dissimilar Code Inject" is the least challenging as unrelated code fragments are less likely to share common variables, functions, or data structures, making it simpler to isolate and compare the code snippets independently. On the other hand, the "`is` $\rightarrow$ `is not`" appears to be the most challenging, as it struggles with identity operator misuse. Among SP transformations, "Operand Swap" is the most challenging, while "Rename Variable" seems the least challenging. Table 3 shows the transformation-wise breakdown of LLM-based metrics. CodeScore (CS), an automatic metric, outperforms zero-shot prompted Code-LLMs, presumably due to using both NL context and the reference. We show the transformation-wise breakdown of match-based metrics in the Appendix (Table 10). We often observe a high similarity score among different metrics for an SA variant compared to its SP one, as shown in the Appendix (Figure 4).

Answer to RQ2: *The transformed code containing the least challenging transformations is associated with maximum syntactic differences, whereas the most challenging transformations often occur in similar contexts and are relatively close in the embedding space but have completely opposite behaviour.*

We note that for AP, the models have to distinguish between SP v/s SA labels, which requires the models to be more selective in their positive predictions, ensuring that it is more often correct when predicting a positive class (i.e., SP). To clarify, CL-13B predicts mostly higher probabilities (> 0.8) for SA transformations than CS; hence, it gets a weaker AURC score. Thus, we can infer that CL-13B does a better job classifying the two snippets as either SP or SA, but once we know the transformation is of a specific type, CS performs better.

5.4 RQ3: Impact of Finetuning

While finetuning on a diverse set of transformations can improve performance on seen examples, it does not necessarily guarantee effective generalization to novel, unseen transformations. We propose a leave-one-out evaluation strategy for assessing the performance of PEFT methods on unseen semantic transformations. The approach involves finetuning the model on N-1 transformation for a given category, then evaluating on the held-out N^{th} transformation. We repeat the leave-one-out approach N times to assess performance on each held-out transformation. Table 4 shows the PEFT results on the SeqCoBench test set using the AURC score. We note that Llama2-7B outperforms CL-7B on most transformations.

Answer to RQ3: *PEFT improves performance on SP transformations while facing challenges with SA transformations. Among different PEFT methods, LoRA is the most effective one for SP, while PrefixTuning is the most successful for SA.*

Transformation		Method	Llama2	CodeLlama
Type	Category		7B	7B
SA	AOM	LoRA	2.69	6.12
		AdaLoRA	**40.57**	9.29
		Prefix-Tuned	16.6	5.06
	BOM	LoRA	0.37	4.04
		AdaLoRA	23.32	10.91
		Prefix-Tuned	**31.16**	20.68
	COM	LoRA	3.17	1.67
		AdaLoRA	14.72	7.44
		Prefix-Tuned	**18.73**	6.01
	DCS	LoRA	**98.69**	77.23
		AdaLoRA	88.28	20.77
		Prefix-Tuned	36.66	35.17
	IOM	LoRA	0.12	3.62
		AdaLoRA	19.57	6.72
		Prefix-Tuned	**32.33**	11.57
	LOM	LoRA	3.84	3.85
		AdaLoRA	9.14	15.60
		Prefix-Tuned	**23.99**	5.72
SP	DCI	LoRA	**81.03**	63.44
		AdaLoRA	56.75	29.50
		Prefix-Tuned	16.46	37.66
	LT	LoRA	**96.01**	78.33
		AdaLoRA	64.76	76.70
		Prefix-Tuned	10.21	26.09
	OS	LoRA	**91.51**	82.76
		AdaLoRA	59.64	75.27
		Prefix-Tuned	0.24	2.08
	RV	LoRA	62.33	**80.63**
		AdaLoRA	26.68	70.14
		Prefix-Tuned	8.92	20.86

Table 4: When finetuning with SP transformations, the PEFT methods learn to be invariant to these transformations. In contrast, SA transformations require updating the core weights to learn the new semantics, and the PEFT methods are not expressive enough to capture such fundamental changes.

6 Conclusion

We propose SeqCoBench, a new challenging benchmark to evaluate how well Code-LLMs capture functional equivalence between code snippets from the code semantics standpoint. We compare the performance of LLM- and match-based metrics on the SeqCoBench and find the performance gap to be minimal. We identify semantic transformations for the Code-LLMs from least to most challenging on a spectrum. We conduct extensive evaluations in different settings, including zero-shot (w/ prompting) and using PEFT methods. In the future, we would like to study code semantics in both static and dynamic settings at different granularities by incorporating approximate, operational, and abstract semantics (Ding et al., 2024). Incorporating symbolic reasoning modules or hybrid approaches that combine neural networks with formal logic can be a promising direction.

Limitations

We investigate open-source LLMs to evaluate for code functional equivalence, so we do not consider ICE-Score (Zhuo, 2024) that requires using closed-source LLMs like GPT-3.5 or GPT-4 in this analysis. Closed-source models are opaque, as their inner workings, data sources, and training methodologies are not disclosed, making it hard to draw meaningful comparisons with open-source models.

In addition, they often come with significant usage costs, and finetuning is not fully supported in experimental access. Currently, the code functions in SeqCoBench are exclusively in Python. However, we aim to broaden the scope to encompass a broader range of programming languages and domains. By doing so, we strive to enhance the diversity and applicability of the dataset, making it more comprehensive and versatile for various software engineering tasks and scenarios.

Moreover, we do not check for compilation of semantic code transformations as we perform a static code evaluation to analyse the code without needing it to be executed or compiled.

Further, we do not account for variations due to prompt and temperature as we do not optimise the prompting format, although we ensure it is kept consistent across different LLMs.

Finally, we refrain from chaining multiple transformations of the same type (either preserving or altering) to make the analysis straightforward.

Acknowledgements

We thank the anonymous reviewers, Ke Wang, and Antonio Miceli Barone for their feedback. This work was supported by the UKRI Centre for Doctoral Training (CDT) in Natural Language Processing through the UKRI grant (EP/S022481/1). We appreciate using computing resources through the CSD3 cluster at the University of Cambridge and the Baskerville cluster at the University of Birmingham. AV was supported by the "UNREAL: Unified Reasoning Layer for Trustworthy ML" project (EP/Y023838/1) selected by the ERC and funded by UKRI EPSRC.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint*. ArXiv:2108.07732 [cs].

Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by "Naturalizing" source code. *arXiv preprint*. ArXiv:2206.07585 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, 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, Josh 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 [cs].

Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 243–253, New York, NY, USA. Association for Computing Machinery.

Michael Denkowski and Alon Lavie. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 376–380, Baltimore, Maryland, USA. Association for Computational Linguistics.

Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. A static evaluation of code completion by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 347–360, Toronto, Canada. Association for Computational Linguistics.

Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. SemCoder: Training Code Language Models with Comprehensive Semantics. *arXiv preprint*. ArXiv:2406.01006 [cs].

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2023. CodeScore: Evaluating Code Generation by Learning Code Execution. *arXiv preprint*. ArXiv:2301.09043 [cs].

Aryaz Eghbali and Michael Pradel. 2023. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1–12, New York, NY, USA. Association for Computing Machinery.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. *arXiv preprint*. ArXiv:2401.03065 [cs].

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 1.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, and Nan Duan. 2022. Execution-based evaluation for data science code generation models. In *Proceedings of the Fourth Workshop on Data Science with Human-in-the-Loop (Language Advances)*, pages 28–36, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint*. ArXiv:2403.07974 [cs].

Akshita Jha and Chandan K. Reddy. 2023. CodeAttack: Code-Based Adversarial Attacks for Pre-trained Programming Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(12):14892–14900.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Jens Krinke and Chaiyong Ragkhitwetsagul. 2022. Bigclonebench considered harmful for machine learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 1–7.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *Proceedings of the 40th International Conference on Machine Learning*, pages 18319–18345. PMLR.

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, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S. Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, 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. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research*.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.

Yujia Li, David 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. *Science*, 378(6624):1092–1097.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Antonio Valerio Miceli Barone, Fazl Barez, Ioannis Konstas, and Shay B. Cohen. 2023. The larger they are, the harder they fail: Language models do not recognize identifier swaps in python. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 272–292, Toronto, Canada. Association for Computational Linguistics.

Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2018. Bug-proneness and late propagation tendency of code clones: A Comparative study on different clone types. *Journal of Systems and Software*, 144:41–59.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Bjorn Poonen. 2014. Undecidable problems: a sampler. *arXiv preprint*. ArXiv:1204.0299 [math].

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal. Association for Computational Linguistics.

Md Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552.

Ricardo Rei, Craig Stewart, Ana C Farinha, and Alon Lavie. 2020. COMET: A neural framework for MT evaluation. In *Proceedings of the 2020 Conference*

on Empirical Methods in Natural Language Processing (EMNLP), pages 2685–2702, Online. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint*. ArXiv:2009.10297 [cs].

Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of computing TR*, 541(115):64–68.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, 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. 2024. Code Llama: Open Foundation Models for Code. *arXiv preprint*. ArXiv:2308.12950 [cs].

Neha Saini, Sukhdip Singh, et al. 2018. Code clones: Detection and management. *Procedia computer science*, 132:718–727.

A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 151–160, Los Alamitos, CA, USA. IEEE Computer Society.

Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*.

Weisong Sun, Yuchen Chen, Guanhong Tao, Chunrong Fang, Xiangyu Zhang, Quanjun Zhang, and Bin Luo. 2023. Backdooring neural code search. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9692–9708, Toronto, Canada. Association for Computational Linguistics.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.

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, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint*. ArXiv:2307.09288 [cs].

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023a. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada. Association for Computational Linguistics.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023b. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada. Association for Computational Linguistics.

Zhilong Wang, Lan Zhang, Chen Cao, Nanqing Luo, and Peng Liu. 2024. A Case Study of Large Language Models (ChatGPT and CodeBERT) for Security-Oriented Code Analysis. *arXiv preprint*. ArXiv:2307.12488 [cs].

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023c. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290, Singapore. Association for Computational Linguistics.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top Leaderboard Ranking = Top Coding Proficiency, Always? EvoEval: Evolving Coding Benchmarks via LLM. *arXiv preprint*. ArXiv:2403.19114 [cs].

Yanming Yang, Ying Zou, Xing Hu, David Lo, Chao Ni, John Grundy, and Xin Xia. 2023. C³: Code Clone-Based Identification of Duplicated Components. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on*

the Foundations of Software Engineering, ESEC/FSE 2023, pages 1832–1843, New York, NY, USA. Association for Computing Machinery.

Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1482–1493, New York, NY, USA. Association for Computing Machinery.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning (ICML)*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2023b. AdaLoRA: Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning. *arXiv preprint*. ArXiv:2303.10512 [cs].

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. *arXiv preprint*. ArXiv:1904.09675 [cs].

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pages 5673–5684, New York, NY, USA. Association for Computing Machinery.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. *arXiv preprint*. ArXiv:2302.05527 [cs].

Terry Yue Zhuo. 2024. ICE-score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2232–2242, St. Julian's, Malta. Association for Computational Linguistics.

A Details of Evaluation Metrics

A.1 Based on Lexical Overlap

These metrics operate on the surface form of the code and account only for an exact lexical token match.

ROUGE (Lin, 2004) measures the recall between n-grams in generated and reference code.

BLEU (Papineni et al., 2002) is the geometric mean of the n-gram precision multiplied by a brevity penalty between generated and reference code.

CodeBLEU (Ren et al., 2020) is a composite metric which is a modification of BLEU and uses the abstract syntax tree and data-flow graph in addition to the surface-level matching.

CrystalBLEU (Eghbali and Pradel, 2023) is again a modification of BLEU which considers the underlying differences between source code and natural language (such as trivially shared n-grams).

Meteor (Denkowski and Lavie, 2014) is a MT metric which is based on the harmonic mean of unigram precision and recall, with the recall being more weighted.

ChrF (Popović, 2015) is again a MT metric which calculates the precision and recall for character n-gram matches and averages it over 1- to 6-character-n-grams.

A.2 Based on Pre-trained LLMs

These metrics rely on LLMs to extract the token embeddings of the hidden layer to calculate the similarity.

COMET (Rei et al., 2020) uses a pre-trained multilingual model to encode generated and reference code separately. These embeddings are concatenated to obtain a quality score.

BERTScore (Zhang et al., 2020) leverages BERT embeddings to compute the pairwise cosine similarity between generated and reference code.

CodeBERTScore (Zhou et al., 2023) uses CodeBERT to encode the context (the natural language description or comment) in addition to generated and reference code. However, it does not use the encoded context to compute cosine similarities.

A.3 Based on Execution

These metrics compare the execution result of generated code by running tests to check for functional correctness.

Pass@k (Chen et al., 2021) generates k solutions for each problem, which is deemed solved if any of the k samples pass the tests.

CodeScore (Dong et al., 2023) provides a framework, UniCE, to finetune LLMs to learn code execution (such as estimating the PassRatio of test cases) of generated code with unified input.

AvgPassRatio (Hendrycks et al., 2021) computes the average pass rate of test cases.

B Details of Correlation Metrics

Kendall-Tau (τ) is a non-parametric statistical measure that quantifies the strength and direction of the rank correlation between two ordinal variables. It is calculated as:

$$\tau = \frac{|\text{concordant}| - |\text{discordant}|}{|\text{concordant}| + |\text{discordant}|}$$

where, concordant pairs are in the same relative order in both rankings, whereas, discordant pairs are in the opposite relative order.

Pearson (r_p) is a statistical measure that quantifies the strength and direction of the linear relationship between two continuous variables. It is calculated as:

$$r_p = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}}$$

where, X and Y corresponds to the values of the reference and candidate variables, respectively.

Spearman (r_s) is a non-parametric measure of the strength and direction of the monotonic relationship between two ranked variables.

$$r_s = \frac{\sum_{i=1}^{n}(R(X_i) - \overline{R(X)})(R(Y_i) - \overline{R(Y)})}{\sqrt{\sum_{i=1}^{n}(R(X_i) - \overline{R(X)})^2 \sum_{i=1}^{n}(R(Y_i) - \overline{R(Y)})^2}}$$

where, $R(X_i)$ and $R(Y_i)$ are the ranks of the i^{th} observations in reference and candidate variables, respectively.

C Training Details

The whole pipeline takes roughly 24 hours to create the full transformations on a single A100 80GB GPU. The zero-shot inference experiments take roughly 12 GPU hours, and fine-tuning takes about 48 GPU hours.

C.1 Model Generate

The configuration specified during the generation step is `max_new_tokens=3` to limit the generated output to 3 new tokens, `top_p=0.9` and `temperature=0.2` to control the randomness of the generated text, `num_return_sequences=3` to generate 3 different output sequences, `top_k=5` to consider only the 5 most likely tokens at each step, `do_sample=True` to use sampling for text generation.

The provided configuration sets up a model to use 4-bit quantization with NF4 type, bfloat16 compute dtype, and optionally enables double quantization for better accuracy.

In our analysis, we exclude instruction-tuned models of CodeLlama and StarCoder due to the potential for learning similar instructions during the finetuning step, such as the code clone detection task.

C.2 Confidence Scores from LLMs

To generate token probabilities, we first generate output sequences from the model using various sampling techniques like top-p, temperature, and top-k. Then, we compute the softmax probabilities of the logits at a specific index to obtain the probabilities of all tokens in the vocabulary. We identify the tokens with probabilities above a certain threshold, decode them, and store their probabilities in a dictionary. Then, we extract the probabilities of the "YES" and "NO" tokens from this dictionary and calculate a score as the ratio of the "YES" probability to the sum of the "YES" and "NO" probabilities. Generating token probabilities from the pre-trained model and analyzing the probabilities of specific tokens provides a way to quantify the model's confidence for a particular output (in this case, "YES" or "NO").

C.3 Finetuning Experiments

We use the popular PEFT methods:

Low-rank Adaptation (LoRA; Hu et al. 2022) introduces two learnable weight matrices, A and B, attached to a frozen pre-trained weight matrix, W, and considers that these updates have a low rank during adaptation.

Adaptive Low-rank Adaptation (AdaLoRA; Zhang et al. 2023b) adaptively allocates more parameters to the more important layers and fewer parameters to less important layers, unlike LoRA, which distributes trainable parameters evenly across all layers.

Prefix Tuning (Li and Liang, 2021) prepends pseudo prefix tokens to the input of a language model.

In the case of LoRA, we use rank as 8, alpha as 32, and dropout as 0.1. For AdaLoRa, we use an initial rank of 12 that will be reduced to 8. The beta1 and beta2 parameters for the Adam optimizer are both set to 0.85. The learning rate will be adjusted between the initial time step of 200 and the final time step of 1000, with a step size 10. We use the alpha value of 32 and a dropout rate of 0.1. For Prefix Tuning, we use 20 virtual tokens.

The training arguments include a per-device batch size and gradient accumulation steps, totalling 2 training epochs. The model undergoes 100 warmup steps and a maximum of 400 steps overall, using a learning rate 3e-4 and enabling fp16 precision. We set logging to occur every 10 steps using the AdamW optimizer. Evaluation and saving occur every 200 steps, with outputs directed to a specified directory and a limit of 3 saved checkpoints. The model does not load the best model at the end of training. We group the sequences by length to speed up training and report results to weights and biases with a run name that includes a timestamp.

D Model Details

Llama2 (Touvron et al., 2023) is a family of LLMs developed by Meta AI, ranging from 7B to 70B parameters. It is an open-source successor to the original Llama model, offering improved performance through a larger training corpus, longer context length, and the use of grouped-query attention.

StarCoder2 (Lozhkov et al., 2024) is a generative model with 3B, 7B, and 15B parameters trained on over 600 programming languages from the Stack v2, along with natural language sources like Wikipedia, ArXiv, and GitHub issues.

Code Llama (Rozière et al., 2024) is initialized using pre-trained weights of Llama2 and trained on code-specific datasets. It then undergoes long-context finetuning and can handle repository-level inputs of 100K tokens.

E Transformation Counts

The average line of code for the transformed version is 9.2247. The problems in the benchmark are designed to be solvable by entry-level programmers. Among these questions, 58% are mathemati-

Transformation Type	Transformation Name	Test	Train	Valid
SA	$+ \rightarrow -$	147	267	114
	$- \rightarrow +$	133	211	73
	$\div \rightarrow \times$	47	82	30
	$\times \rightarrow \div$	93	133	53
	`False` $\rightarrow$ `True`	41	49	23
	`True` $\rightarrow$ `False`	39	50	21
	== $\rightarrow$!=	117	173	62
	!= $\rightarrow$ ==	29	34	26
	$> \rightarrow <$	79	112	38
	$< \rightarrow >$	57	96	39
	Dissimilar Code Inject	1705	2825	1140
	`is` $\rightarrow$ `is not`	10	11	2
	`is not` $\rightarrow$ `is`	1	0	1
	`and` $\rightarrow$ `or`	44	47	17
	`or` $\rightarrow$ `and`	13	39	13
SP	Dead Code Insert	320	539	224
	`for` $\leftrightarrow$ `while` Loop	313	515	217
	Operand Swap	311	512	217
	Rename Variable Cb	290	489	199
	Rename Variable Naive	313	515	217
	Rename Variable Rn	313	515	217

Table 5: Breakdown of counts of different transformations across train, validation, and test sets of SeqCoBench.

cal (e.g., calculating the volume of a sphere), 43% involve list processing, 19% require string manipulation, 9% focus on integer sequences, and 2% revolve around using other data structures. Table 5 lists the counts of different transformations.

F Prompt Template

Figure 2 shows the prompt template used in the zero-shot prompting experiments.

G Additional Prompting Results

Table 6 shows the performance of a few hand-picked closed-source LLMs to assess progress in order to have a more holistic evaluation. We include it to ascertain the range of performance of these sophisticated LLMs on our task.

Table 7 and 8 demonstrate few-shot and zero-shot chain-of-thought (CoT) performance on the SeqCoBench test set. It suggests the model cannot fully infer the task requirements or context directly from the zero-shot prompt. Few-shot examples help bridge this gap by reducing ambiguity and assisting the model in aligning its output to the expected format or logic of the task by recognising patterns in the input-output pairs and applying these patterns to new, unseen data. Since the representative examples resemble the different plausible styles of transformations, the CodeLLMs can learn these patterns but might struggle to understand other variations of transformations (e.g., De Morgan's laws).

```
"""You are a helpful and honest code assistant expert in Python. Is there a functional equivalence between the Code1 and Code2? Please respond either "YES" or "NO".

### Code1:
{code_1}

### Code2:
{code_2}

### Response:
"""
```

Figure 2: Prompt for Code-LLMs on SeqCoBench.

Model	Size	AP
gpt-4o-mini	~8B	83.73
deepseek-coder-instruct-v1.5	7B	84.10
qwen2.5-coder-instruct	32B	85.21

Table 6: Results for zero-shot prompting on the SoTA closed-source LLMs. While they perform better than open-source LLMs (e.g., StarCoder, CodeLlama, etc.), they still struggle to differentiate between functionally equivalent v/s functionally non-equivalent codes. Our claims surrounding RQ1 still hold, as these are trivial tasks for the sophisticated closed-source LLMs.

H Execution-based Functional Correctness

Functional correctness is assessed by running the generated code against a set of test cases and checking if the output matches the expected results. We use the HumanEval (Python only), and HumanEval-X (Zheng et al., 2023) benchmarks to measure the correlation with functional correctness. We filter based on the popularity of the programming language and choose to evaluate on Java, C++, Python, and JavaScript languages. We compute the Pearson, Spearman, and Kendall-Tau correlation coefficients

Model	Size	AP
Generic		
Llama2	7B	55.52
	13B	58.53
Code-Specific		
CodeLlama	7B	80.20
	13B	85.92
	34B	92.54
StarCoder2	3B	71.84
	7B	56.14
	15B	97.81

Table 7: Two-shot with one SP and SA demonstration example inserted into the original prompt. By including examples, few-shot prompts offer more context about analysing and comparing code snippets. This helps the model focus on relevant aspects like logic flow, variable usage, and output rather than superficial differences in syntax or formatting.

as Pearson captures linear relationships, while the other two capture ordinal relationships, which may be non-linear. Section B provides an overview of different correlation metrics. Table 9 shows correlation coefficients of different metrics with the functional correctness on HumanEval for multiple languages. We notice C++ and Javascript have lower correlation scores than Python and Java. Python and Java are primarily object-oriented languages, while C++ supports both object-oriented and functional programming styles. JavaScript, although object-oriented, has a strong functional programming influence. Generating code that effectively utilizes functional programming constructs can be more challenging for models trained primarily on object-oriented codebases.

I SeqCoBench Transformation Examples

Model	Size	AP
Generic		
Llama2	7B	37.51
	13B	40.55
Code-Specific		
CodeLlama	7B	36.13
	13B	68.29
	34B	62.58
StarCoder2	3B	39.28
	7B	34.85
	15B	73.93

Table 8: Zero-shot chain-of-thought results by adding "Let's think step by step" to the original prompt. By prompting the model to think step-by-step, Zero-shot CoT prompting leverages the model's inherent reasoning abilities. We can observe that Zero-shot CoT prompting has comparable performance with the standard zero-shot prompting.

Type	Metric	Java			C++			Python			JavaScript		
		τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p
Match-Based	Rouge1	0.4982	0.3992	0.3903	0.2466	0.3390	0.3286	0.3857	0.3389	0.3284	0.2594	0.2664	0.2360
	Rouge2	0.4569	0.3417	0.3273	0.2273	0.2963	0.2961	0.3457	0.2747	0.2686	0.1888	0.2390	0.1786
	RougeL	0.4778	0.3953	0.3880	0.2454	0.3265	0.3235	0.3607	0.3320	0.3248	0.2012	0.2541	0.2107
	Meteor	0.5576	0.4337	0.4222	0.2714	0.3392	0.3217	0.4428	0.4189	0.4124	0.3019	0.3660	0.3425
	ChrF	0.5576	0.4223	0.4040	0.3257	0.3647	0.3576	0.4306	0.3808	0.3727	0.3088	0.3551	0.3298
	BLEU	0.5148	0.4137	0.4005	0.2565	0.2986	0.2914	0.4049	0.3560	0.3499	0.2970	0.2970	0.2703
	CrystalBLEU	0.5145	0.4145	0.4016	0.2519	0.2953	0.2877	0.4027	0.3562	0.3502	0.2949	0.2945	0.2682
	CodeBLEU	0.4990	0.3458	0.3417	0.2200	0.1600	0.1630	0.3806	0.3219	0.3195	0.3080	0.2643	0.2271
LLM-Based	CodeScore	0.3061	0.2686	0.2865	0.1253	0.1074	0.0962	0.2951	0.2936	0.3012	0.1793	0.1605	0.1498
	BERTScore	0.4977	0.3707	0.3782	0.2391	0.2969	0.2954	0.3550	0.3106	0.2971	0.2357	0.2756	0.2465
	CodeBERTScore	0.5798	0.4080	0.4130	0.3389	0.3473	0.3478	0.4488	0.3817	0.3953	0.3521	0.3227	0.3739

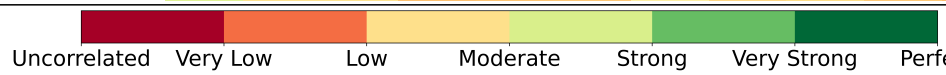

Table 9: Current LLM-based metrics, like CodeBERTScore and CodeScore, show minimal correlation with the execution-based functional correctness on HumanEval across multiple languages. No metric exceeds an average correlation coefficient of $r = 0.31$, highlighting a significant opportunity for developing better metrics.

Original

```
def binary_search(item_list,item):
    first = 0
    last = len(item_list)-1
    found = False
    while( first<=last and not found):
        mid = (first + last)//2
        if item_list[mid] == item :
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

Dead code insertion

```
def binary_search(item_list, item):
    first = 0
    for _i_3 in range(0):
        first = 0
    last = len(item_list) - 1
    found = False
    while first <= last and not found:
        mid = (first + last) // 2
        if item_list[mid] == item:
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

For loop →While loop

```
def binary_search(item_list, item):
    first = 0
    last = len(item_list) - 1
    found = False
    while first <= last and not found:
        mid = (first + last) // 2
        if item_list[mid] == item:
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

Operand swap

```
def binary_search(item_list, item):
    first = 0
    last = len(item_list) - 1
    found = False
    while last >= first and not found:
        mid = (first + last) // 2
        if item_list[mid] == item:
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

Rename variables (CB)

```
def binary_search(item_list, item):
    first = 0
    last = len(item_list) - 1
    found = False
    while first <= last and not found:
        first2 = (first + last) // 2
        if item_list[ first2 ] == item:
            found = True
        else:
            if item < item_list[ first2 ]:
                last = first2 - 1
            else:
                first = first2 + 1
    return found
```

Rename variables (Naive)

```
def binary_search(item_list, item):
    first = 0
    last = len(item_list) - 1
    found = False
    while first <= last and not found:
        VAR_0 = (first + last) // 2
        if item_list[ VAR_0 ] == item:
            found = True
        else:
            if item < item_list[ VAR_0 ]:
                last = VAR_0 - 1
            else:
                first = VAR_0 + 1
    return found
```

Rename variables (RN)

```
def binary_search(item_list, item):
    first = 0
    last = len(item_list) - 1
    found = False
    while first <= last and not found:
        ztc = (first + last) // 2
        if item_list[ ztc ] == item:
            found = True
        else:
            if item < item_list[ztc]:
                last = ztc - 1
            else:
                first = ztc + 1
    return found
```

Figure 3: Examples of the output of semantic-preserving transformations.

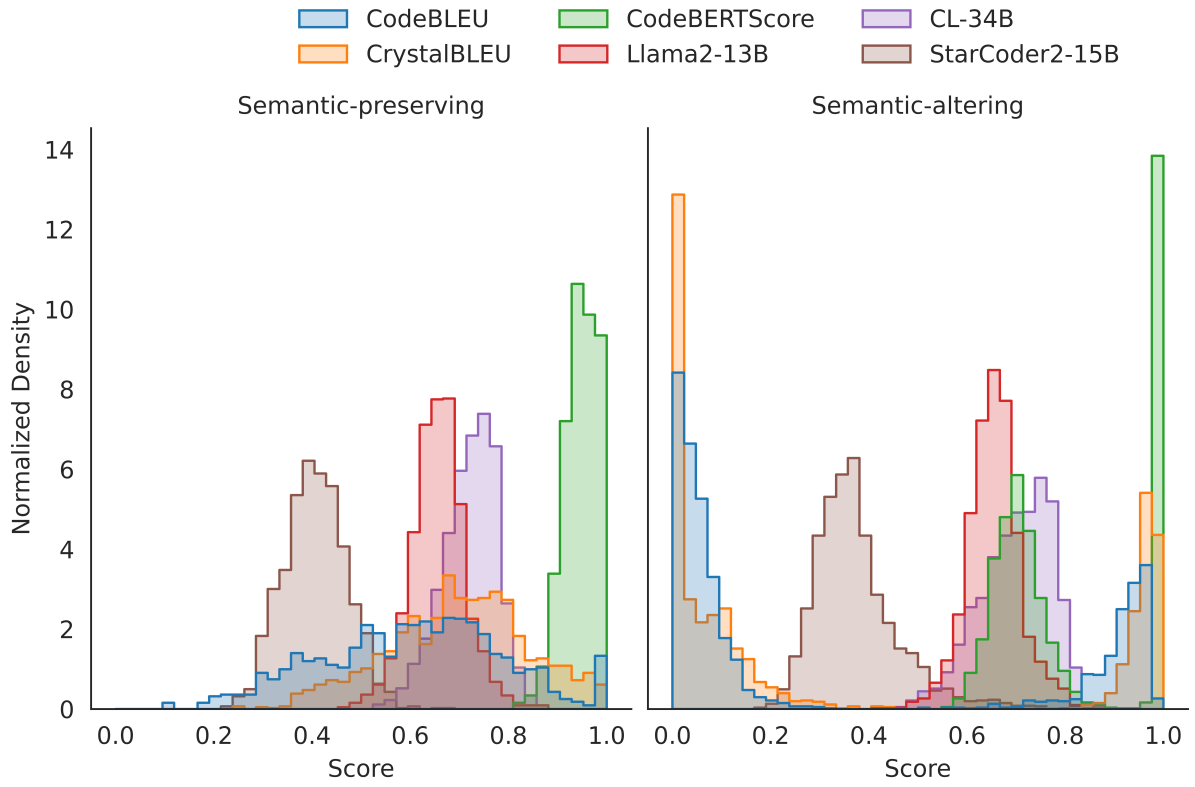

Figure 4: We observe peaks in scores above 0.9 for SA transformations, leading to incorrect semantic similarity calculations. When we consider all the metrics, we observe that the probability distribution of scores for semantic-altering labels is often higher than for SP labels.

Transformation			*N-gram-matching*						*N-gram-matching w/ code-related features*	
Type	**Category**	**Name**	**Rouge1**	**Rouge2**	**RougeL**	**Meteor**	**ChrF**	**BLEU**	**CrystalBLEU**	**CodeBLEU**
SA	AOM	+ → −	2.70	5.70	2.70	2.63	6.35	7.46	7.03	**26.44**
		− → +	2.33	4.96	2.33	3.57	5.43	13.08	11.56	**18.81**
		÷ → ×	2.41	5.23	2.41	2.49	3.88	6.79	7.72	**23.25**
		× → ÷	5.13	8.53	5.13	3.94	6.04	14.51	12.82	**46.31**
	BOM	`False` → `True`	1.86	3.90	1.86	2.73	3.96	7.84	10.56	**16.84**
		`True` → `False`	1.84	3.86	1.84	2.71	4.37	7.77	10.49	**24.57**
	COM	== → !=	2.69	5.68	2.69	2.81	4.00	8.02	10.46	**25.46**
		!= → ==	1.07	2.22	1.07	1.08	2.65	2.91	3.01	**9.23**
		> → <	3.03	5.11	3.03	3.58	7.54	12.95	11.21	**25.90**
		< → >	3.02	5.01	3.02	2.34	7.19	6.99	6.00	**21.38**
	DCS	Dissimilar Code Inject	**29.34**	14.84	25.02	27.92	13.11	6.33	6.32	5.45
	IOM	`is` → `is not`	0.86	3.16	0.86	2.16	2.96	4.87	5.62	**17.48**
		`is not` → `is`	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**	**50.00**
	LOM	`and` → `or`	1.29	2.67	1.29	3.86	3.50	11.20	12.45	**28.65**
		`or` → `and`	1.40	2.88	1.40	2.71	2.31	7.57	8.14	**10.78**
SP	DCI	Dead Code Insert	22.26	27.94	20.59	21.67	31.55	33.34	33.26	**49.88**
	LT	`for` ↔ `while` Loop	19.13	35.11	28.82	26.28	26.00	43.63	45.20	**46.70**
	OS	Operand Swap	10.75	19.42	13.29	16.57	30.10	31.19	30.89	**54.26**
	RV	Rename Variable Cb	16.62	30.72	16.10	23.85	31.57	42.46	41.95	**44.16**
		Rename Variable Naive	16.57	30.62	16.01	23.67	30.70	42.16	41.15	**43.56**
		Rename Variable Rn	16.60	30.67	16.04	23.72	33.28	42.26	41.65	**43.75**

Table 10: Transformation-wise breakdown for Match-based metrics in the zero-shot setting using Area under the Recall curve (AURC) metric.

J Other Work

Robustness against adversarial attacks. There is a strong connection between adversarial attacks and semantic transformations for code. Both involve minor changes to a code input that preserve the original meaning or functionality but cause a machine learning model to make incorrect predictions. We derive inspiration from previous works in software engineering to leverage semantic-preserving code transformations. For instance, Rabin et al. (2021) highlights the importance of considering semantics-preserving transformations when building reliable neural program analysers, and Compton et al. (2020) rely on data augmentation methods based on variable renaming to design efficient models.

A few works focused on addressing the robustness problem for the code on attacks and defences. Miceli Barone et al. (2023) demonstrates how LLMs understand code semantics by approximating α-equivalence of Python code and suggest improved defences against identifier substitution attacks. Yang et al. (2022) argue that adversarial examples should preserve natural semantics in addition to operational semantics. To identify vulnerabilities of Code-LLMs to adversarial attacks, Jha and Reddy (2023) introduces CodeAttack that generates adversarial samples for a code fragment, and Wang et al. (2023b) designs a robustness evaluation benchmark and evaluates Code-LLMs on their ability to generate equivalent code across perturbed prompts while we aim to comprehend and analyze existing code.

Novelty of our approach compared to existing works

We clarify the motivation and real-world applicability of the proposed task setting.

Motivation. By using transformations that preserve semantics, we can better isolate the LLMs ability to understand the code's underlying meaning rather than just its surface-level structure. So, essentially, the task becomes more about detecting equivalence despite superficial differences rather than recognizing that two entirely different implementations achieve the same result. In addition, it allows for a more nuanced sensitivity analysis of the model's behaviour, helping to identify which types of code changes are most likely to affect the model's judgment of equivalence. Our focus is on small lexical changes that preserve semantics or the other way around, not on general control flow restructuring, as even with these small lexical changes, LLMs do not do well. We can most certainly expect them to be able to tackle full restructuring at the moment. We are deepening and further showing issues with LLMs in a similar style to Miceli Barone et al. (2023) but much more extensively and comprehensively.

K Code Clone Detection and Code Obfuscation

While there is a degree of overlap *in spirit* between the transformations appearing in code clone detection datasets and our benchmark, our primary goal is different. As stated above, capturing functional equivalence is a harder task and lets us evaluate how LLMs disentangle syntax from semantics. We note that our transformations are designed to be the simplest as possible (e.g., flipping an operator) to detect how much LLMs get confused by transformations that are trivial for us humans. This is a design choice that lets us carry on a finer grain analysis that is not possible with current benchmarks such as BigCloneBench (Svajlenko et al., 2014) and, as such, should not be rated as *not substantial*.

Current code clone detection benchmarks, e.g., BigCloneBench, propose a different task than ours: they are designed to identify code fragments that share the same high-level functionality, but that can be very different from semantically equivalent problems. For example, two functions that can have different side-effects, or even having different input and output types are still considered "clones". Our semantic equivalence definition is more stringent and better captures what LLMs understand about code execution. Moreover, Krinke and Ragkhitwetsagul (2022) highlight how many clone pairs are falsely tagged as actual clones in BigCloneBench.

We remark that our objective is not to create a dataset to improve LLMs capabilities to detect clones in the real world. Instead, we want a controlled environment where we can exactly understand which simple transformations are able to confuse LLMs by discriminating syntax from semantics. We also argue that simple transformations can be common in real-world scenarios, e.g., out of simple typos and wrong copy-pasting actions that are common in programming. While having real-world large code repositories would be nice, it is a non-trivial task to collect the samples (for e.g., from GitHub) that can be validated using unit tests

and at the same time allow us to have a controlled experimental setting. Hence, we rely on existing benchmarks which have samples tested by human programmers for functional correctness to build our benchmark.

We further note that our approach differs from code obfuscation approaches as the main difference is that the former intentionally reduces readability (up to the point that the code can be hard for humans to follow). Our semantic transformations, on the other hand, want to preserve human readability and intelligibility. Our transformations are simple by design and *hard* enough to show certain failure modes of LLMs that are commonly used and deployed. As we state in Section 4 we design them to make the simplest change in syntax or semantics that allows us to measure that code LLMs struggle to disentangle these two aspects. This is a feature, not a bug. Having more complex transformations (e.g., iterative vs recursive implementations, loop unrolling, etc.) would simply imply that LLMs failing on simple transformations are also failing on more complex ones (but without the simplest ones, we would not know what is the smallest perturbation that confuses the LLMs).

L Examples

When evaluating LLMs for their ability to classify code snippets as functionally equivalent or not, it is essential to consider a variety of examples that highlight both the strengths and limitations of these models. Below, we present a selection of examples that demonstrate scenarios where LLMs may find it easy or hard to determine code equivalence for the different transformations in SeqCoBench.

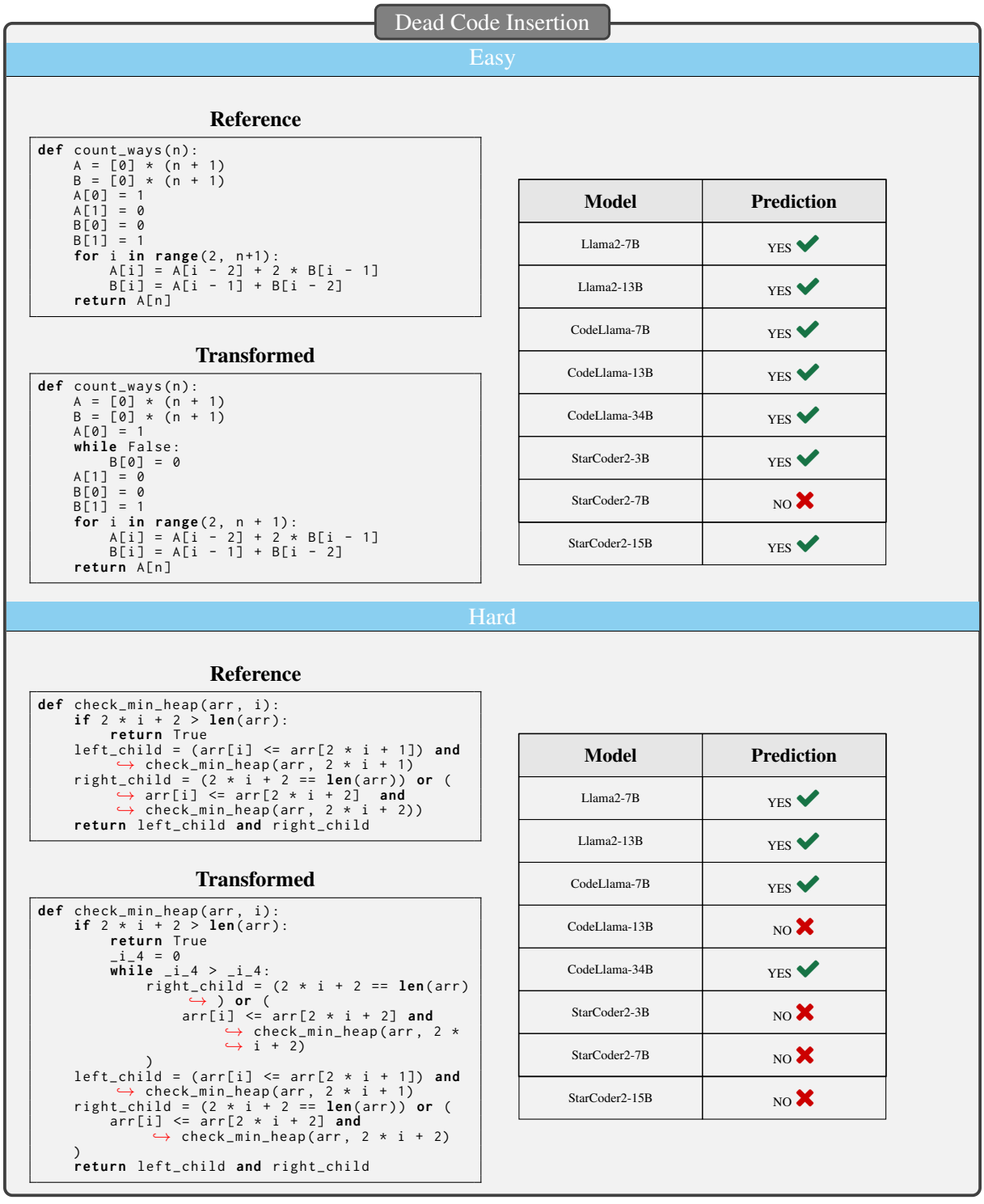

Figure 5: In the easy example, the function uses a loop to populate these arrays according to specific recurrence relations, ultimately returning the value of A[n]. In the transformed version of the code, an unnecessary while False statement is included, which does not affect the functionality but disrupts code clarity. Model predictions indicate that various LLMs successfully recognize the function's correctness, while one variant of StarCoder2 does not. In the hard example, the transformed version attempts to introduce a loop but retains the recursive checks for left and right children, albeit with some redundancy in the definitions of `left_child` and `right_child`. Additionally, the model predictions indicate varying success rates of different LLMs in recognizing the min-heap structure.

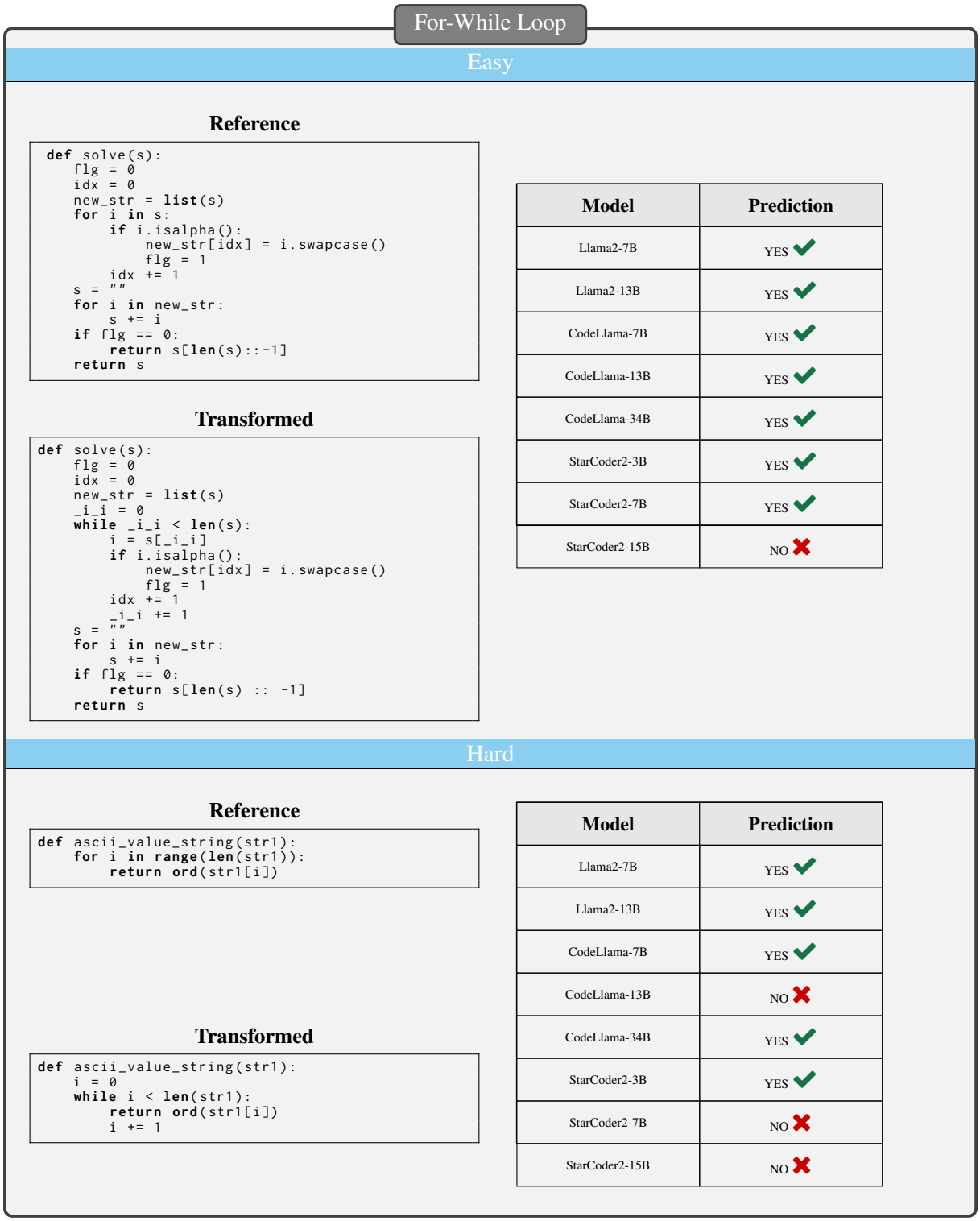

Figure 6: In the easy example, it defines a function that processes a string by swapping the case of its alphabetic characters while preserving the order of non-alphabetic characters. The transformed version of the function uses a `while` loop instead of a `for` loop to achieve the same functionality. Most LLMs confirm its correctness. In the hard example, the original implementation uses a `for` loop to iterate through the string, while the transformed version employs a `while` loop. Model predictions indicate that various LLMs, including Llama2-7B, Llama2-13B, CodeLlama-7B, and CodeLlama-34B, successfully recognize the function's equivalence, whereas CodeLlama-13B, StarCoder2-7B, and StarCoder2-15B do not.

Operand Swap

Easy

Reference

```
import math

def poly(xs: list, x: float):
    return sum([coeff * math.pow(x, i) for i,
        ↪ coeff in enumerate(xs)])

def find_zero(xs: list):
    begin, end = -1., 1.
    while poly(xs, begin) * poly(xs, end) > 0:
        begin *= 2.0
        end *= 2.0
    while end - begin > 1e-10:
        center = (begin + end) / 2.0
        if poly(xs, center) * poly(xs, begin) >
            ↪ 0:
            begin = center
        else:
            end = center
    return begin
```

Transformed

```
import math

def poly(xs: list, x: float):
    return sum([coeff * math.pow(x, i) for i,
        ↪ coeff in enumerate(xs)])

def find_zero(xs: list):
    begin, end = -1.0, 1.0
    while 0 < poly(xs, begin) * poly(xs, end):
        begin *= 2.0
        end *= 2.0
    while end - begin > 1e-10:
        center = (begin + end) / 2.0
        if poly(xs, center) * poly(xs, begin) >
            ↪ 0:
            begin = center
        else:
            end = center
    return begin
```

Model	Prediction
Llama2-7B	YES ✔
Llama2-13B	YES ✔
CodeLlama-7B	YES ✔
CodeLlama-13B	YES ✔
CodeLlama-34B	YES ✔
StarCoder2-3B	YES ✔
StarCoder2-7B	YES ✔
StarCoder2-15B	NO ✘

Hard

Reference

```
def triangle_area(a, b, c):
    if a + b <= c or a + c <= b or b + c <= a:
        return -1
    s = (a + b + c)/2
    area = (s * (s - a) * (s - b) * (s - c)) **
        ↪ 0.5
    area = round(area, 2)
    return area
```

Transformed

```
def triangle_area(a, b, c):
    if a + b <= c or a + c <= b or a >= b + c:
        return -1
    s = (a + b + c) / 2

    area = (s * (s - a) * (s - b) * (s - c)) **
        ↪ 0.5
    area = round(area, 2)
    return area
```

Model	Prediction
Llama2-7B	YES ✔
Llama2-13B	YES ✔
CodeLlama-7B	YES ✔
CodeLlama-13B	YES ✔
CodeLlama-34B	YES ✔
StarCoder2-3B	NO ✘
StarCoder2-7B	NO ✘
StarCoder2-15B	NO ✘

Figure 7: In the easy example, the transformed version of the code slightly modifies the conditions in the function for clarity. Most models, except for StarCoder2-15B, successfully recognize the code's functionality. In the hard example, an error in the conditional check is noted, where a >= b + c should be corrected to b + c <= a, in the transformed version. Model predictions indicate that several models successfully identify the functional equivalence aspect, while others (StarCoder2 variants) do not.

Rename-Variable (CB) Example

Easy

Reference

```
import math

def poly(xs: list, x: float):
    return sum([coeff * math.pow(x, i) for i,
        ↪ coeff in enumerate(xs)])

def find_zero(xs: list):
    begin, end = -1., 1.
    while poly(xs, begin) * poly(xs, end) > 0:
        begin *= 2.0
        end *= 2.0
    while end - begin > 1e-10:
        center = (begin + end) / 2.0
        if poly(xs, center) * poly(xs, begin) >
            ↪ 0:
            begin = center
        else:
            end = center
    return begin
```

Transformed

```
import math

def poly(xs: list, x: float):
    return sum([coeff * math.pow(x, i) for i,
        ↪ coeff in enumerate(xs)])

def find_zero(xs: list):
    center2, end = -1.0, 1.0
    while poly(xs, center2) * poly(xs, end) > 0:
        center2 *= 2.0
        end *= 2.0
    while end - center2 > 1e-10:
        center = (center2 + end) / 2.0
        if poly(xs, center) * poly(xs, center2)
            ↪ > 0:
            center2 = center
        else:
            end = center
    return center2
```

Model	Prediction
Llama2-7B	YES ✔
Llama2-13B	YES ✔
CodeLlama-7B	YES ✔
CodeLlama-13B	YES ✔
CodeLlama-34B	YES ✔
StarCoder2-3B	YES ✔
StarCoder2-7B	YES ✔
StarCoder2-15B	NO ✘

Hard

Reference

```
def triangle_area(a, b, c):
    if a + b <= c or a + c <= b or b + c <= a:
        return -1
    s = (a + b + c)/2
    area = (s * (s - a) * (s - b) * (s - c)) **
        ↪ 0.5
    area = round(area, 2)
    return area
```

Transformed

```
def triangle_area(a, b2, c):
    if a + b2 <= c or a + c <= b2 or b2 + c <= a
        ↪ :
        return -1
    s = (a + b2 + c) / 2
    area = (s * (s - a) * (s - b2) * (s - c)) **
        ↪  0.5
    area = round(area, 2)
    return area
```

Model	Prediction
Llama2-7B	YES ✔
Llama2-13B	YES ✔
CodeLlama-7B	YES ✔
CodeLlama-13B	YES ✔
CodeLlama-34B	NO ✔
StarCoder2-3B	NO ✘
StarCoder2-7B	NO ✘
StarCoder2-15B	NO ✘

Figure 8: In the easy example, the transformed code modifies variable names for clarity without changing the underlying logic. Model predictions indicate that several LLMs (Llama2 and CodeLlama) successfully handle the task, while one (StarCoder2-15B) does not. In the hard example, the transformed version of the function uses a different variable name for one of the sides (b2 instead of b) but maintains the same logic. Model predictions indicate that various LLMs (Llama2 and CodeLlama) successfully solve the task, while others (StarCoder2) do not.

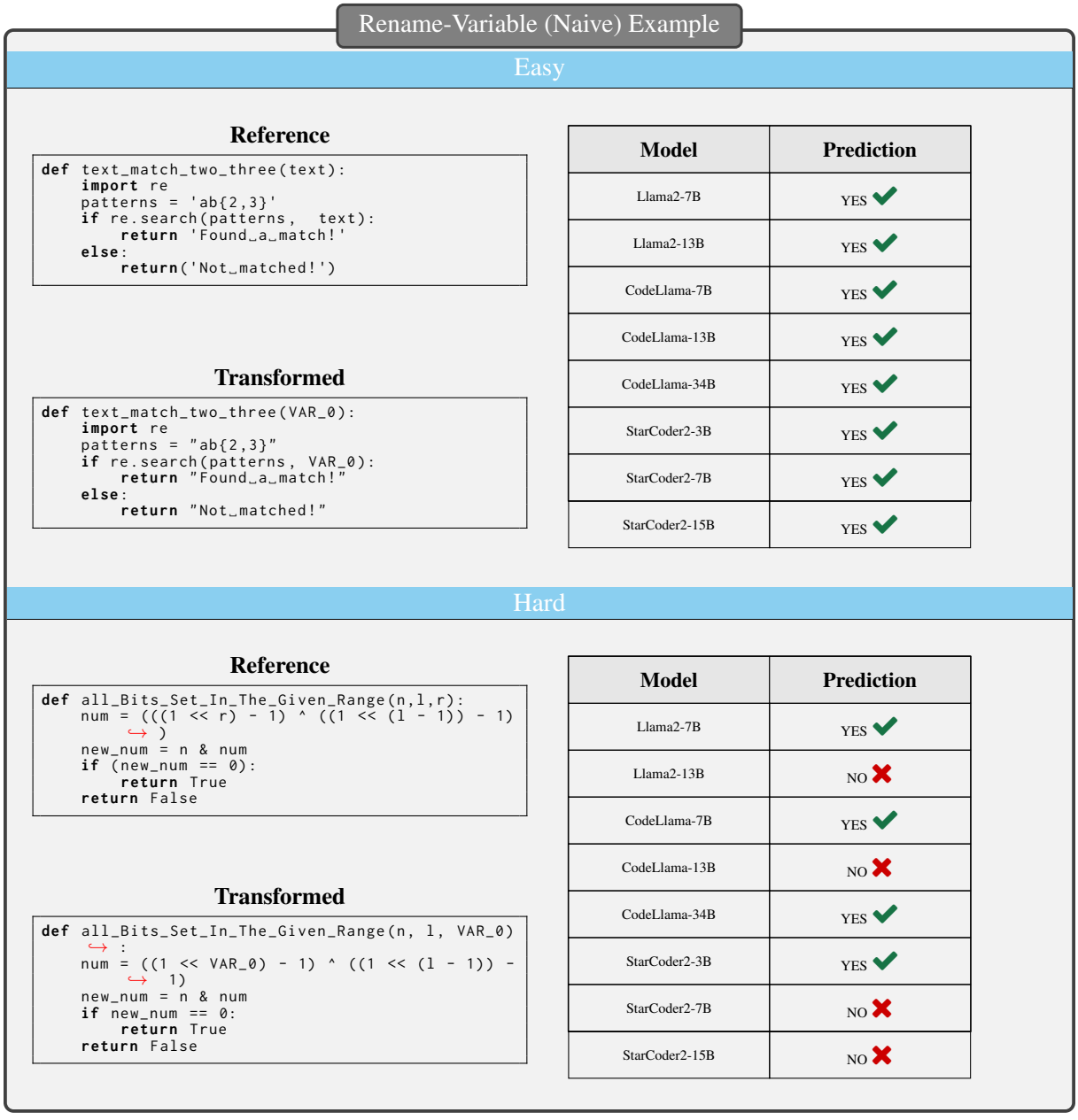

Figure 9: In the easy example, the transformed version of the function replaces the parameter name with `VAR_0`, but the functionality remains unchanged. All the LLMs successfully predict the function's behavior, indicating that they recognize the code's intent and structure. In the hard example, the function is transformed to use a variable `VAR_0` instead of `r`, and various model predictions indicate whether they agree with the function's expected output, with some models confirming the correctness and others not.

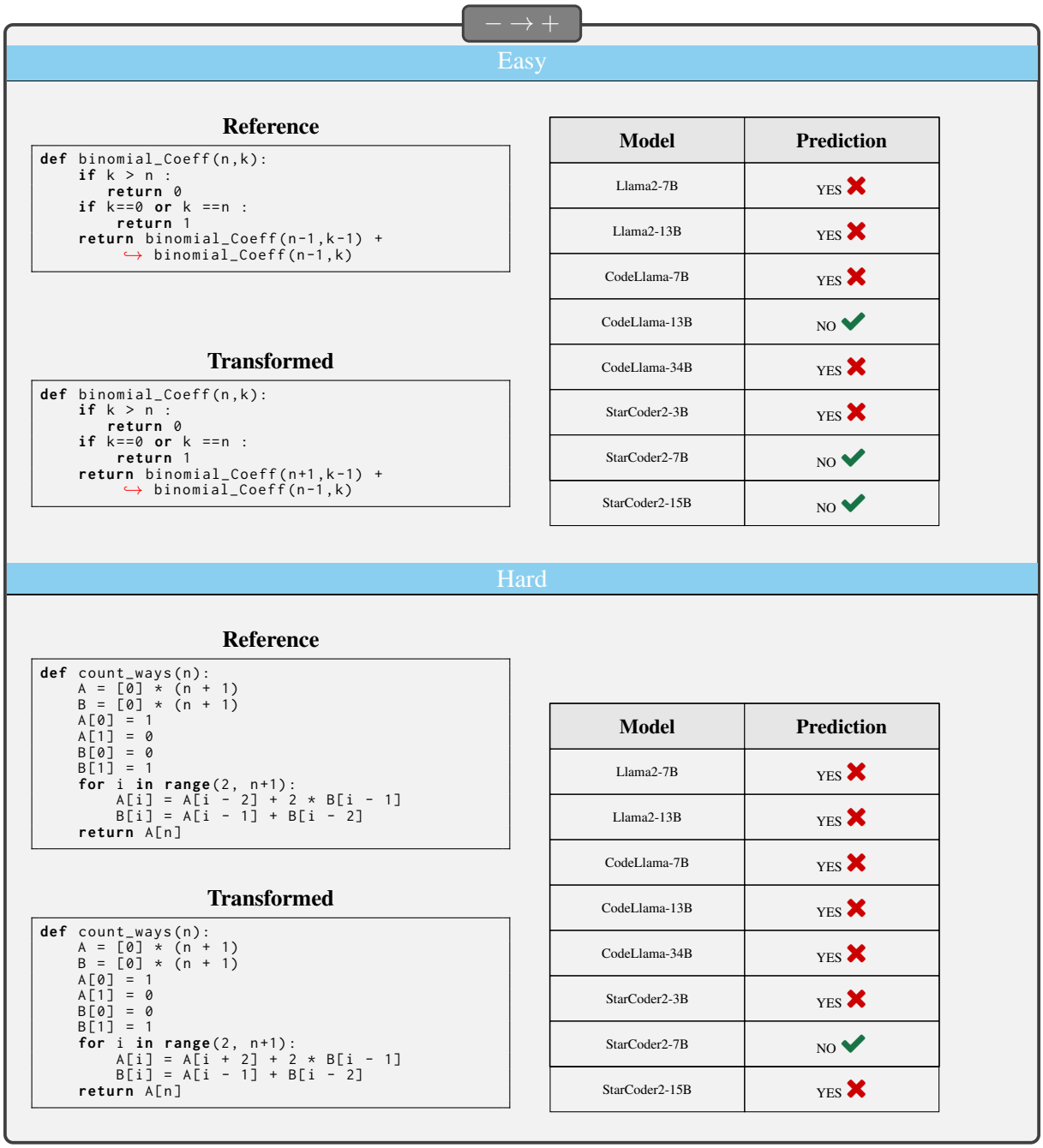

Figure 10: In the easy example, the reference version correctly implements the recursive logic, returning 0 if k is greater than n, 1 if k is 0 or equal to n, and otherwise summing two recursive calls. The transformed version modifies the first recursive call to use n + 1 instead of n - 1, which alters the logic but retains the original base cases. In terms of model predictions, various LLMs show differing capabilities in handling the transformed function, with some successfully predicting the output while others do not. In the hard example, the function uses a loop to fill these sequences based on specific recurrence relations. However, a transformation in the code erroneously modifies the index in the calculation for A[i], changing it from A[i - 2] to A[i + 2], which likely leads to incorrect results. Various LLMs produced erroneous classification, while one variant of StarCoder2 predicted correctly.

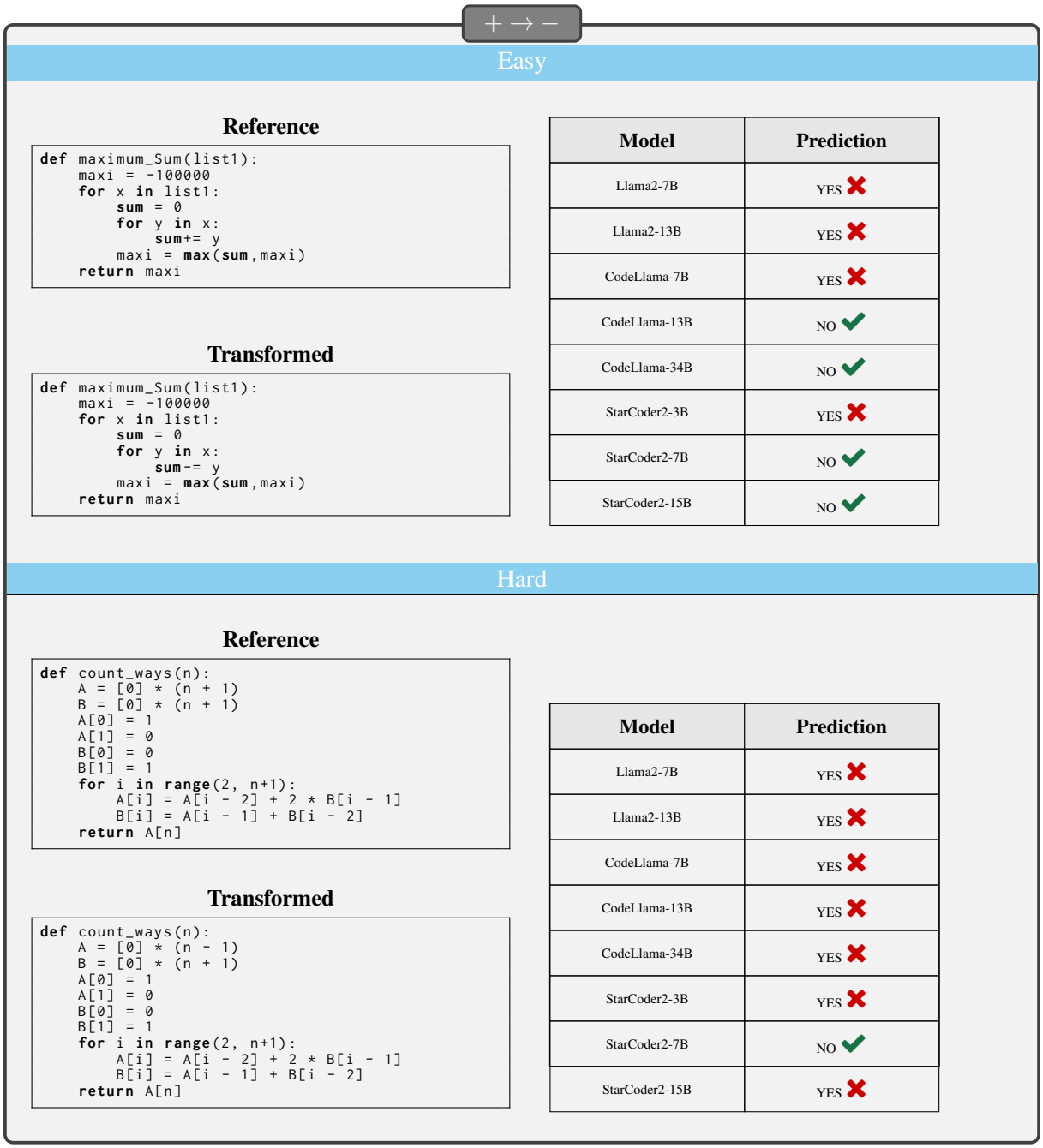

Figure 11: In the easy example, the function sums the inner lists' elements and updates a maximum value in the original version. The transformed version, however, subtracts the elements instead of adding them, which fundamentally alters the function's purpose. The model predictions indicate that various LLMs have some difficulty in classifying them as functionally nonequivalent. In the hard example, the function initializes the base cases for A and B, then iteratively fills these lists based on previously computed values to derive the total ways to reach the n^{th} step. The first version of the function allocates an array A of size n + 1, while the transformed version incorrectly allocates A of size n - 1, which could lead to an index error. Model predictions indicate that various LLMs fail to discern the function's validity, with some exceptions.

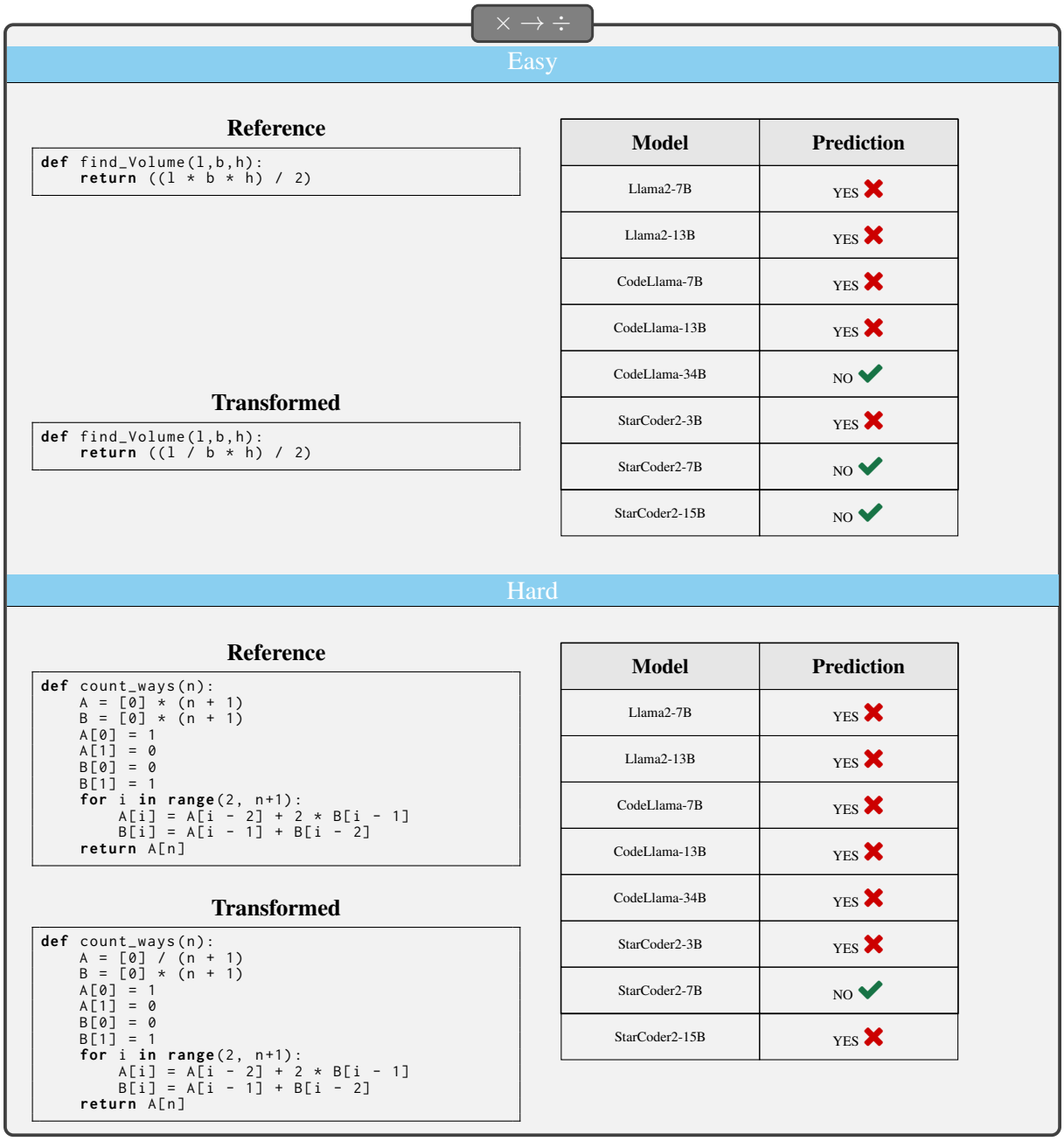

Figure 12: In the easy example, the reference code correctly calculates the volume of a rectangular prism using the formula ($l \times b \times h$), while the transformed code incorrectly divides l by b and then divides the result by 2, which does not match the formula. In the hard example, the function sets base cases for A and B, then iteratively fills the lists using previously computed values. The transformed code contains a minor error in the initialization of list A, where the division operator is incorrectly used instead of the multiplication operator. The model predictions indicate that various LLMs, unsuccessfully classify the function's structure and logic, with some models showing varying levels of accuracy.

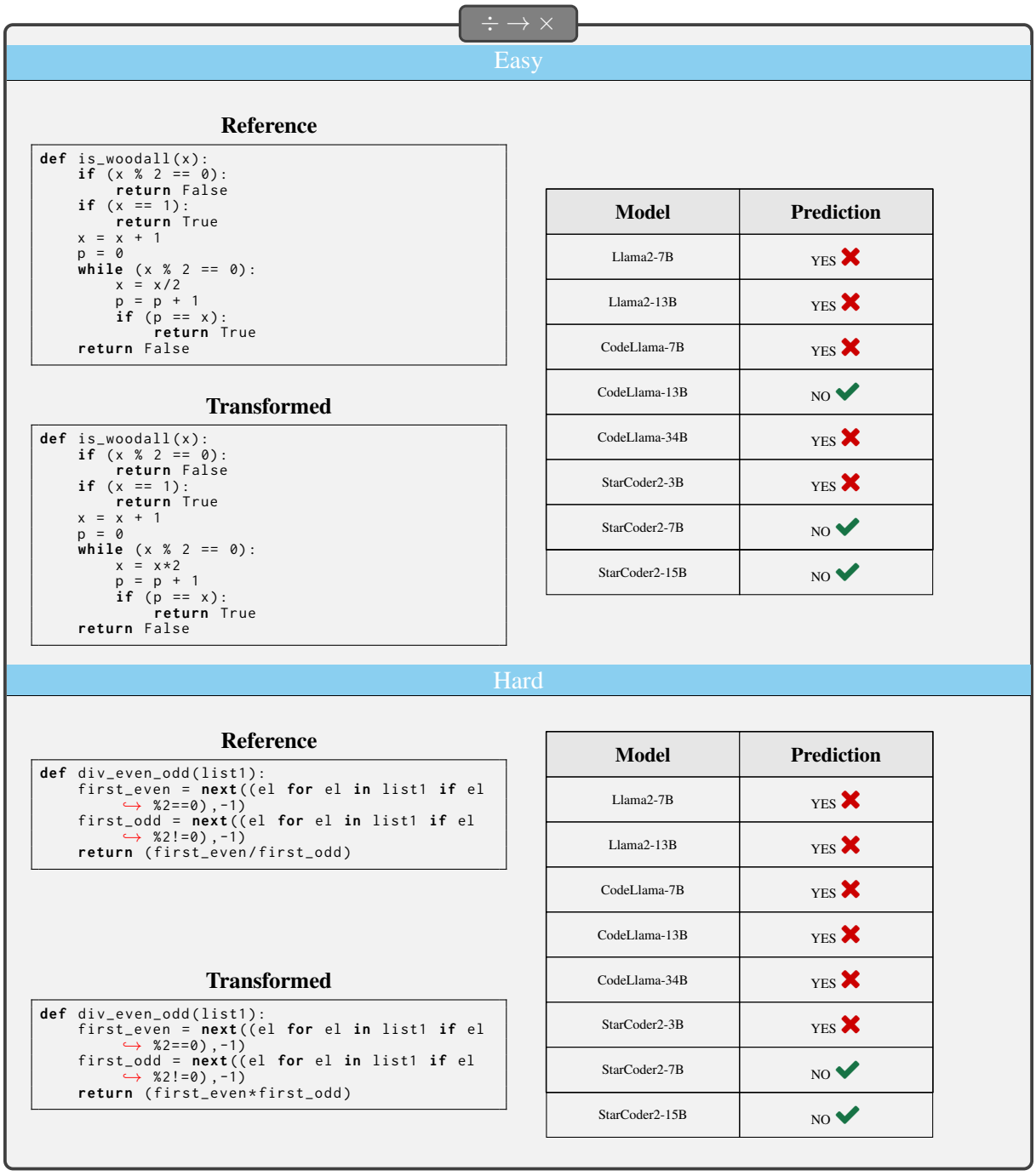

Figure 13: In the easy example, the function checks if this count equals the modified value of x to determine if it is a Woodall number. The transformed version incorrectly multiplies x by 2 in the loop instead of dividing, which may lead to different results. Model predictions from various LLMs indicate varying success rates in identifying the correctness of the function, with some models confirming the original function as valid while others question the transformed version. In the hard example, it returns the result of dividing the first even number by the first odd number in the original version, while the transformed version changes the operation to multiplication instead. The model predictions indicate that various LLMs unsuccessfully classify the original and transformed functions, with some discrepancies noted for certain versions of StarCoder2.

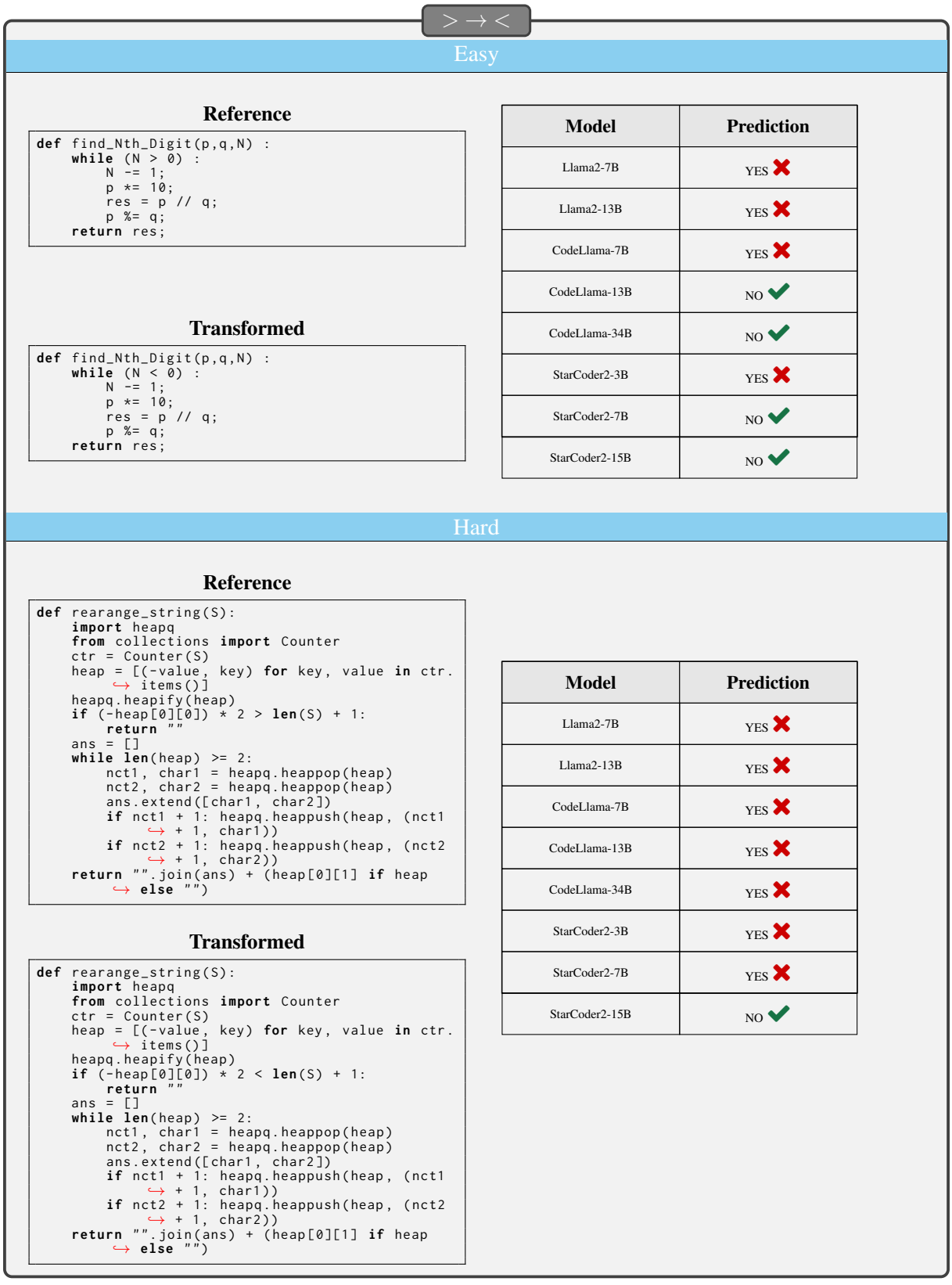

Figure 14: In the easy example, the transformed version incorrectly checks for N being less than 0, which would lead to an infinite loop since N would never become negative. Some models successfully predict the output while others do not. In the hard example, the transformation in the code changes the condition for returning an empty string from > to <, affecting the validity check for the arrangement. The model predictions indicate that various LLMs unsuccessfully recognize the function's intent and structure, with the StarCoder2-15B model predicting correctly.

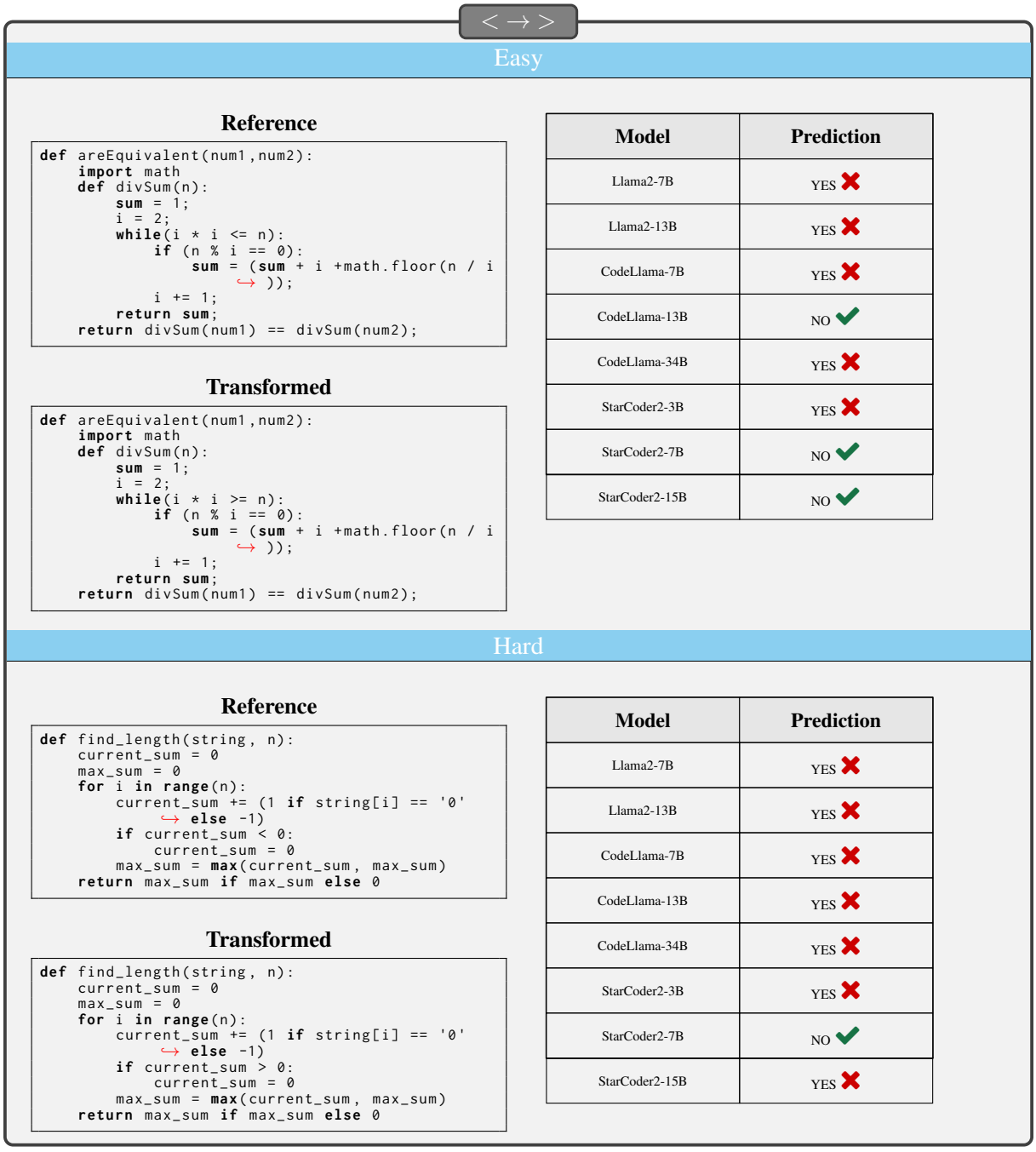

Figure 15: In the easy example, the provided code defines a function which checks if two numbers are equivalent based on the sum of their divisors. In the original code, the loop condition checks if $i \times i$ is less than or equal to n, while in the transformed version, this condition is incorrectly changed to $i \times i$ being greater than or equal to n. The model predictions indicate varying results from different LLMs regarding the equivalence of the two versions, with some models affirming the equivalence and others disagreeing. In the hard example, the provided code defines a function that calculates the maximum sum of a modified array derived from a binary string, where '0' is treated as +1 and '1' as -1. The transformed version of the function incorrectly resets current_sum when it is positive instead of negative, which alters its intended functionality. Various LLMs unsuccessfully predict the behavior of both versions of the function, with some exceptions noted for StarCoder2-7B model.

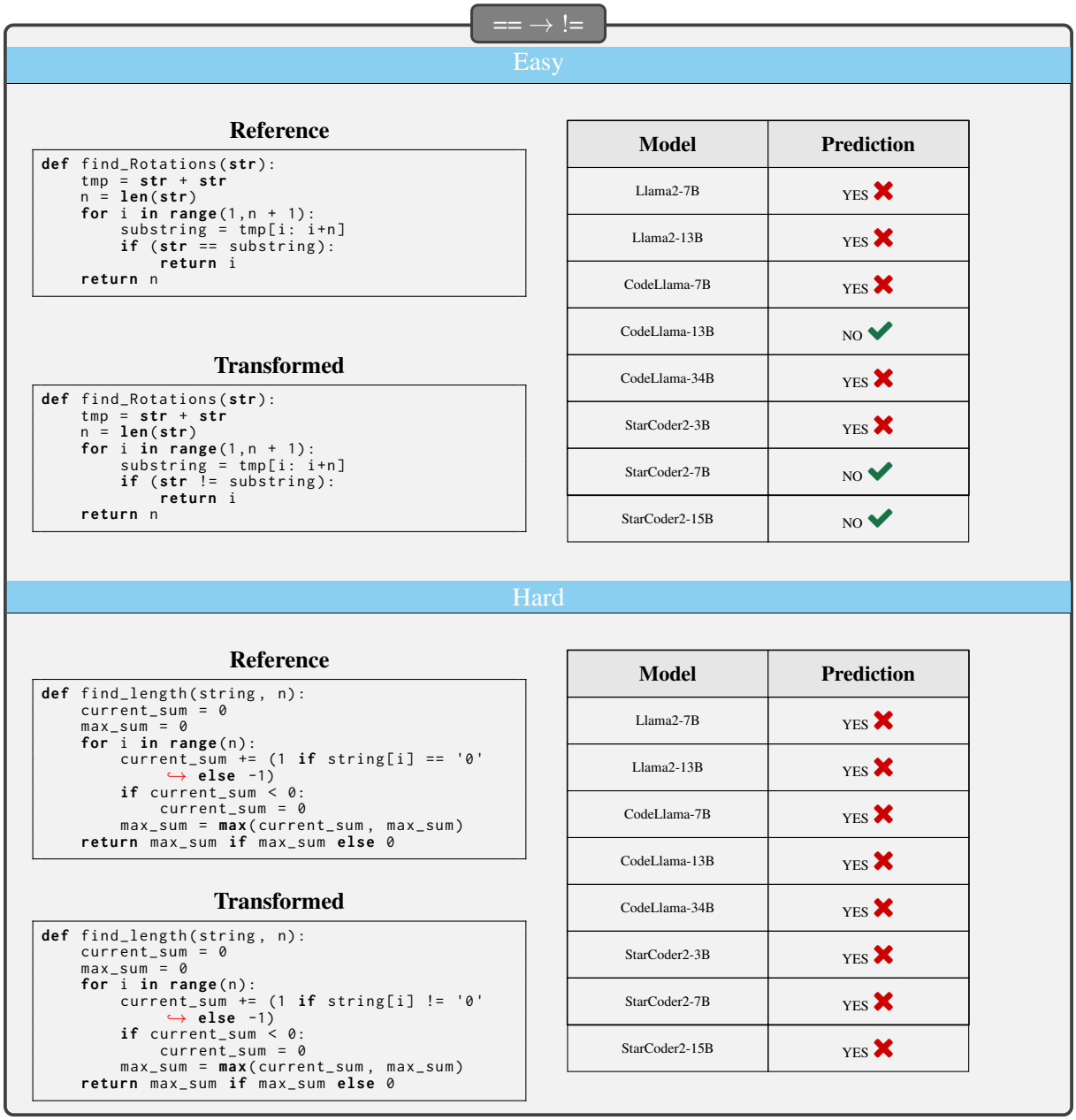

Figure 16: In the easy example, the provided code defines a function identifies the starting index of a substring in a concatenated string, where the substring is a rotation of the original string. The original version checks for equality between the original string and each substring derived from the concatenated string, returning the index when a match is found. The transformed version, however, changes the condition to check for inequality, returning the index when the substring does not match the original string. The model predictions indicate that various LLMs have differing capabilities in recognizing the functionality of the original and transformed code, with some models successfully identifying the original logic while others do not. In the hard example, the provided code snippets define a function that calculates the maximum length of a contiguous subarray in a string, treating '0's and '1's differently. In the original version, the function increments a counter for '0's and decrements for '1's, while the transformed version does the opposite, incrementing for '1's and decrementing for '0's. The model predictions indicate that all LLMs fail to recognize and validate the functionality of both versions of the code.

!= → ==

Easy

Reference

```
def removals(arr, n, k):
    def find_ind(key, i, n, k, arr):
        ind = -1
        start = i + 1
        end = n - 1;
        while (start < end):
            mid = int(start + (end - start) / 2)
            if (arr[mid] - key <= k):
                ind = mid
                start = mid + 1
            else:
                end = mid
        return ind
    ans = n - 1
    arr.sort()
    for i in range(0, n):
            j = find_ind(arr[i], i, n, k, arr)
            if (j != -1):
                ans = min(ans, n - (j - i + 1))
    return ans
```

Transformed

```
def removals(arr, n, k):
    def find_ind(key, i, n, k, arr):
        ind = -1
        start = i + 1
        end = n - 1;
        while (start < end):
            mid = int(start + (end - start) / 2)
            if (arr[mid] - key <= k):
                ind = mid
                start = mid + 1
            else:
                end = mid
        return ind
    ans = n - 1
    arr.sort()
    for i in range(0, n):
        j = find_ind(arr[i], i, n, k, arr)
        if (j == -1):
            ans = min(ans, n - (j - i + 1))
    return ans
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	NO ✓
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	NO ✓
StarCoder2-15B	NO ✓

Hard

Reference

```
def div_even_odd(list1):
    first_even = next((el for el in list1 if el
        ↪ %2==0),-1)
    first_odd = next((el for el in list1 if el
        ↪ %2!=0),-1)
    return (first_even/first_odd)
```

Transformed

```
def div_even_odd(list1):
    first_even = next((el for el in list1 if el
        ↪ %2==0),-1)
    first_odd = next((el for el in list1 if el
        ↪ %2==0),-1)
    return (first_even/first_odd)
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	YES ✗
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	NO ✓
StarCoder2-15B	NO ✓

Figure 17: In the easy example, the transformed code contains a logical error in the condition that checks if j is -1, which lead to incorrect results. The model predictions indicate that some models successfully recognize the logic while others do not. In the hard example, the logic is flawed in the transformed version as it incorrectly searches for the first odd number using the same condition as the even number, leading to an unexpected outcome. Most LLMs recognize the original function's equivalence wrongly, while some models flag the transformed version as incorrect.

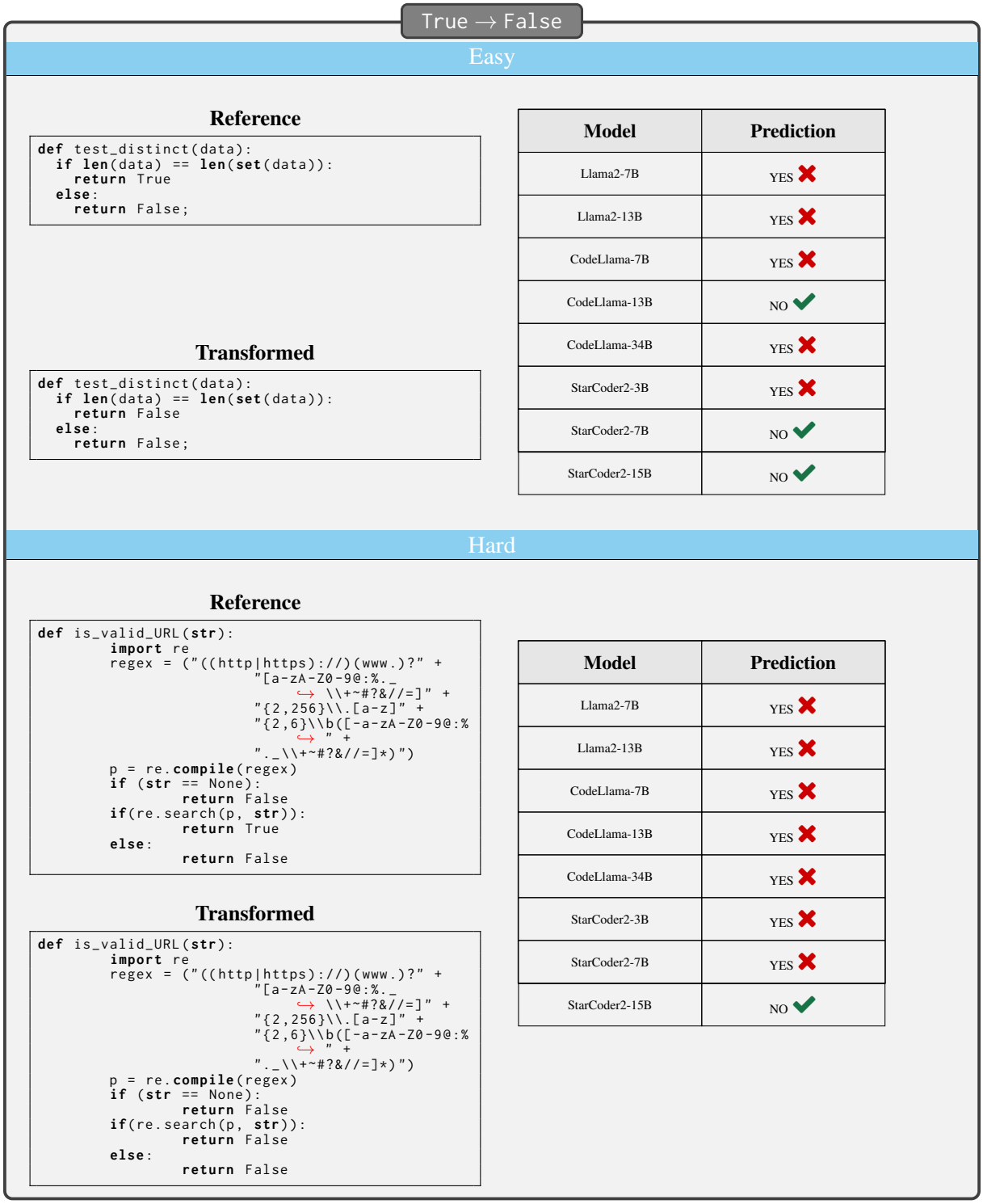

Figure 18: In the easy example, the original code correctly returns True if the length of data matches the length of the set created from data, indicating all elements are unique; otherwise, it returns False. However, the transformed code incorrectly returns False regardless of the input, failing to accurately determine the distinctness of the elements. Model predictions from various LLMs show mixed results, with some models correctly identifying the original function's intent while others misinterpret the transformed version's logic. In the hard example, the provided code defines a function that uses regular expressions to validate whether a given string is a valid URL. The function returns False if the input string is None, and it is supposed to return True if the string matches the regex pattern; however, the transformed code incorrectly returns False for all cases. The model predictions indicate that various LLMs, including Llama2 and CodeLlama, consistently incorrectly validate the function as equivalent except for the StarCoder2-15B model, which predicts correctly.

False → True

Easy

Reference

```
def check_equilateral(x,y,z):
    if x == y == z:
        return True
    else:
        return False
```

Transformed

```
def check_equilateral(x,y,z):
    if x == y == z:
        return True
    else:
        return True
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	NO ✓
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	NO ✓
StarCoder2-15B	NO ✓

Hard

Reference

```
def binary_search(item_list,item):
    first = 0
    last = len(item_list)-1
    found = False
    while( first<=last and not found):
        mid = (first + last)//2
        if item_list[mid] == item :
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

Transformed

```
def binary_search(item_list,item):
    first = 0
    last = len(item_list)-1
    found = True
    while( first<=last and not found):
        mid = (first + last)//2
        if item_list[mid] == item :
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	YES ✗
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	YES ✗
StarCoder2-15B	YES ✗

Figure 19: In the easy example, the code checks if a triangle is equilateral by comparing the lengths of its three sides. If all sides are equal, it returns True, indicating an equilateral triangle. However, in the transformed code, the else block always returns True regardless of the condition, making the function always return True, even for non-equilateral triangles. The LLM predictions show inconsistent results, with some models correctly identifying the issue in the transformed code while others do not. In the hard example, the provided code snippet implements a binary search algorithm to determine if a specified item exists within a sorted list. The transformed version of the code mistakenly sets the found variable to True at the start, which would cause the loop to exit immediately, effectively rendering the search ineffective. All the LLMs incorrectly identify the transformed code as functionally equivalent to the original code.

and → or

Easy

Reference

```
def check_String(str):
    flag_l = False
    flag_n = False
    for i in str:
        if i.isalpha():
            flag_l = True
        if i.isdigit():
            flag_n = True
    return flag_l and flag_n
```

Transformed

```
def check_String(str):
    flag_l = False
    flag_n = False
    for i in str:
        if i.isalpha():
            flag_l = True
        if i.isdigit():
            flag_n = True
    return flag_l or flag_n
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	NO ✓
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	NO ✓
StarCoder2-15B	NO ✓

Hard

Reference

```
def is_tree_balanced(root):
    class Node:
        def __init__(self, data):
            self.data = data
            self.left = None
            self.right = None
    def get_height(root):
        if root is None:
            return 0
        return max(get_height(root.left),
            ↪ get_height(root.right)) + 1
    if root is None:
        return True
    lh = get_height(root.left)
    rh = get_height(root.right)
    if (abs(lh - rh) <= 1) and is_tree_balanced(
    root.left) is True and is_tree_balanced(
        ↪ root.right) is True:
        return True
    return False
```

Transformed

```
def is_tree_balanced(root):
    class Node:
        def __init__(self, data):
            self.data = data
            self.left = None
            self.right = None
    def get_height(root):
        if root is None:
            return 0
        return max(get_height(root.left),
            ↪ get_height(root.right)) + 1
    if root is None:
        return True
    lh = get_height(root.left)
    rh = get_height(root.right)
    if (abs(lh - rh) <= 1) or is_tree_balanced(
    root.left) is True and is_tree_balanced(
        ↪ root.right) is True:
        return True
    return False
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	YES ✗
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	NO ✓
StarCoder2-15B	YES ✗

Figure 20: In the easy example, the function returns True only if both conditions are met (using and) in the original version, while the transformed version incorrectly uses or, meaning it will return True if either condition is satisfied. The model predictions indicate varying responses from different LLMs regarding the correctness of the transformed function, with some models confirming its validity and others rejecting it. In the hard example, the transformed version of the code introduces a logical change in the balance condition, using an 'or' instead of an 'and' operator. Most LLMs misclassify the code's correctness except for one instance.

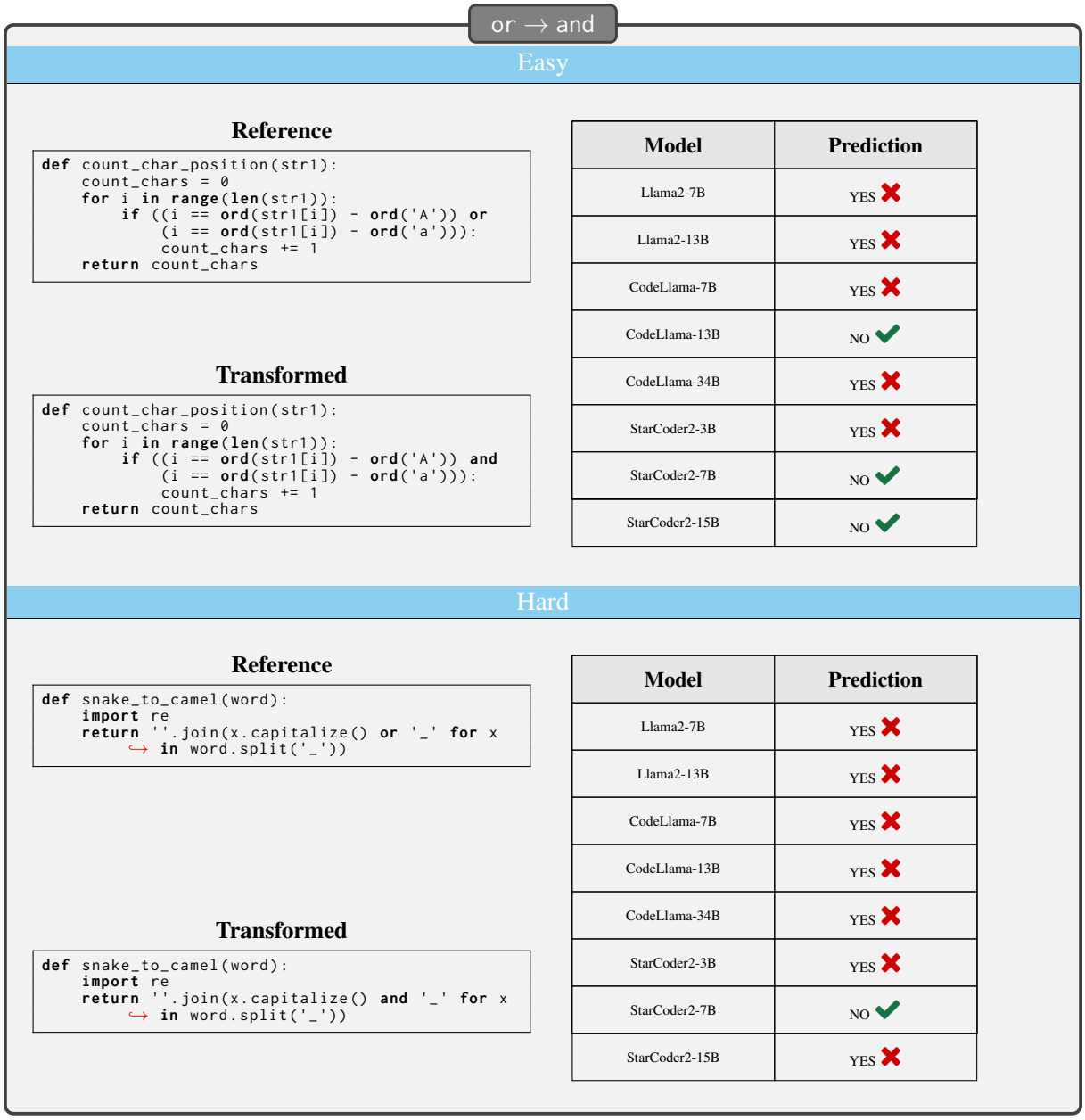

Figure 21: In the easy example, the condition checks if the index i is equal to the ASCII value of the character at that index minus the ASCII value of 'A' or 'a' in the original version. In the transformed version, the logical operator changes from 'or' to 'and', which alters the intended functionality. The model predictions indicate varying results across different LLMs, with some confirming the correctness of the original code and others failing to recognize the transformed version. In the hard example, the original implementation uses `capitalize()` to format each word after splitting the input string by underscores, while the transformed version incorrectly uses 'and' instead of 'or', which alters the intended behavior. The model predictions indicate that various LLMs struggle to determine the function's purpose, while one variant of StarCoder2 succeeds.

is → is not

Easy

Reference

```
def sort_mixed_list(mixed_list):
    int_part = sorted([i for i in mixed_list if
        ↪ type(i) is int])
    str_part = sorted([i for i in mixed_list if
        ↪ type(i) is str])
    return int_part + str_part
```

Transformed

```
def sort_mixed_list(mixed_list):
    int_part = sorted([i for i in mixed_list if
        ↪ type(i) is not int])
    str_part = sorted([i for i in mixed_list if
        ↪ type(i) is str])
    return int_part + str_part
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	YES ✗
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	NO ✓
StarCoder2-15B	NO ✓

Hard

Reference

```
def is_tree_balanced(root):
    class Node:
        def __init__(self, data):
            self.data = data
            self.left = None
            self.right = None
    def get_height(root):
        if root is None:
            return 0
        return max(get_height(root.left),
            ↪ get_height(root.right)) + 1
    if root is None:
        return True
    lh = get_height(root.left)
    rh = get_height(root.right)
    if (abs(lh - rh) <= 1) and is_tree_balanced(
    root.left) is True and is_tree_balanced(
        ↪ root.right) is True:
        return True
    return False
```

Transformed

```
def is_tree_balanced(root):
    class Node:
        def __init__(self, data):
            self.data = data
            self.left = None
            self.right = None
    def get_height(root):
        if root is not None:
            return 0
        return max(get_height(root.left),
            ↪ get_height(root.right)) + 1
    if root is None:
        return True
    lh = get_height(root.left)
    rh = get_height(root.right)
    if (abs(lh - rh) <= 1) and is_tree_balanced(
    root.left) is True and is_tree_balanced(
        ↪ root.right) is True:
        return True
    return False
```

Model	Prediction
Llama2-7B	YES ✗
Llama2-13B	YES ✗
CodeLlama-7B	YES ✗
CodeLlama-13B	YES ✗
CodeLlama-34B	YES ✗
StarCoder2-3B	YES ✗
StarCoder2-7B	YES ✗
StarCoder2-15B	YES ✗

Figure 22: In the easy example, the transformed version incorrectly filters for non-integer types instead of integers, which disrupts the intended functionality. The model predictions indicate that several LLMs fail to correctly identify the reference code's functionality, while some versions of StarCoder2 infer correctly. In the hard example, the transformed code contains a minor error in the `get_height()` function, where it incorrectly returns 0 when the root is not None. The model predictions indicate that all LLMs fail in assessing for functional equivalence.