IRCoder: Intermediate Representations Make Language Models Robust Multilingual Code Generators

Indraneil Paul[1], **Jun Luo**[2], **Goran Glavas**[3], and **Iryna Gurevych**[1]
[1] Ubiquitous Knowledge Processing Lab (UKP Lab)
Department of Computer Science and Hessian Center for AI (hessian.AI)
Technische Universität Darmstadt
[2] Huawei Noah's Ark Lab
[3] CAIDAS, University of Würzburg
www.ukp.tu-darmstadt.de

Abstract

Code understanding and generation have fast become some of the most popular applications of language models (LMs). Nonetheless, research on multilingual aspects of Code-LMs (i.e., LMs for code generation) such as cross-lingual transfer between different programming languages, language-specific data augmentation, and post-hoc LM adaptation, alongside exploitation of data sources other than the original textual content, has been much sparser than for their natural language counterparts. In particular, most mainstream Code-LMs have been pre-trained on source code files alone. In this work, we investigate the prospect of leveraging readily available compiler *intermediate representations*—shared across programming languages—to improve the multilingual capabilities of Code-LMs and facilitate cross-lingual transfer.

To this end, we first compile **`SLTrans`**, a parallel dataset consisting of nearly 4M self-contained source code files coupled with their respective intermediate representations. Next, starting from various base Code-LMs (ranging in size from 1.1B to 7.3B parameters), we carry out continued causal language modelling training on `SLTrans`, forcing the Code-LMs to (1) learn the IR language and (2) align the IR constructs with respective constructs of various programming languages. Our resulting models, dubbed **`IRCoder`**, display sizeable and consistent gains across a wide variety of code generation tasks and metrics, including prompt robustness, multilingual code completion, code understanding, and instruction following.

1 Introduction

Language models for code generation (Code-LMs) are some of the most promising tools for enhancing the productivity of software developers. They have proliferated into automating several parts of the traditional software development lifecycle including code infilling, comment generation, refactoring,

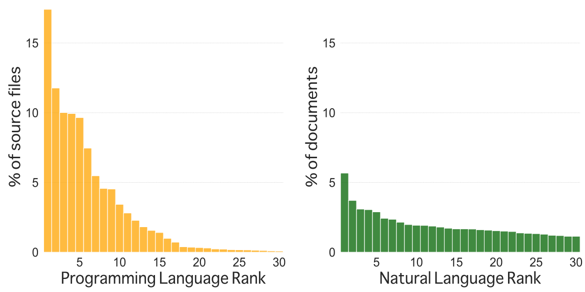

Figure 1: Comparison of the distribution of the top 30 programming languages on GitHub (left) against the top 30 natural languages in the mC4 corpus (right).

and build error prediction (Frömmgen et al., 2024; Dunay et al., 2024), inter alia. Despite a strong demand for such capabilities across all programming languages, the benchmarking of Code-LMs has largely been dominated by the most resourced languages. For instance, popular benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021) and APPS (Hendrycks et al., 2021) all test Code-LMs' competence only in `Python`: this can result in misleading conclusions on the global utility of Code-LMs. More recent transpilation-oriented benchmarks like Multipl-E (Cassano et al., 2022) and BabelCode (Orlanski et al., 2023)—that test competence on several languages—have laid bare the gaps in Code-LMs' performance across different programming languages. For instance, the state-of-the-art DeepSeekCoder's (Guo et al., 2024) code completion performance in `Bash`, the most popular shell scripting language, lags its `Python` `pass@1` performance by 30%+ points.

The problem is exacerbated by the fact that the distribution of programming languages in code corpora is far more skewed than the distribution of natural languages in standard multilingual text corpora. As an example, Ukrainian, considered to be a moderate-to-low-resource natural language (Tracey et al., 2019), comprises a higher proportion of the massively multilingual mC4 corpus (Xue et al., 2021) than `Rust` (the 13th most popular programming language) does of GitHub[1]. This relative scarcity in code corpora, however, belies `Rust's` criticality to digital systems: Rust is one of only two languages approved for use in Linux kernel development[2]. Figure 1 illustrates this problem by comparing relative distributions of natural and programming languages in respective multilingual corpora. Moreover, unlike the spread of digital content over natural languages, global code distribution over programming languages changes rapidly, reflecting sudden gains or drops in the popularity of individual programming languages. Such changes mean that consequential programming languages at some point in time, may not have been represented in the pre-training corpora of the Code-LMs. One prominent example is `HCL`, the fastest growing programming language according to GitHub[3], used for configuring production infrastructure deployments, which did not make it into common pre-training corpora for Code-LMs (Li et al., 2023).

The Case for Intermediate Code Representation. The aforementioned limitations and properties of multilingual code corpora, i.e., skewed and rapidly changing distribution over programming languages, warrant a departure from the conventional approach of pre-training on ever-larger file-level source-code corpora. Indeed, recent evidence points to tangible downstream gains from the adoption of smaller but curated or synthesized data (Gunasekar et al., 2023) as well as from grounding code generation using metadata from language toolchains[4] (Chen et al., 2023a; Gong et al., 2024). The latter, in principle, allows one to tap into more than half a century of research on programming languages and compilers and utilize views of the source code that often contain additional or more explicitly laid out information. This makes intuitive sense: skewed and fast-evolving distribution of programming languages implies that truly robust multilingual models cannot be obtained from heterogeneous source code alone; instead, some type of *code interlingua* should be leveraged to facilitate cross-lingual transfer from high- to low(er)-resource languages.

In this work, we propose *compiler intermediate representations* (IR) to be this interlingua for grounding source code understanding across heterogeneous languages, allowing for the creation of all-around stronger models. The IR are the artifacts of transformations performed by the compiler in three sequential phases: frontend, middle-end, and backend transformations. In popular cross-language compiler frameworks, the frontend IR contains language-specific constructs, whereas the backend IR contains the target platform-specific execution constructs. The middle-end IR, however, is agnostic to the source programming language and target execution platform and thus represents, we argue, an ideal shared representation for positive knowledge transfer in multilingual Code-LMs, offering both (1) a way to better semantically align constructs from heterogeneous languages and (2) an alternative (and possibly more informative) view of the source code.

Contributions and Research Questions. Our work makes the following contributions:

1) We create `SLTrans`, a parallel dataset consisting of nearly 4M pairs of self-contained source code and corresponding IR;

2) We conduct a systematic investigation of the benefits of grounding Code-LMs in IR, demonstrating sizeable and consistent empirical gains across a broad range of tasks and programming languages;

3) We create and publicly release a suite of base and instruction-tuned Code-LMs dubbed `IRCoder`, the result of continued pre-training of state-of-the-art Code-LMs, ranging in size from 1.1B to 7.3B, on a mixture of parallel data from `SLTrans` and monolingual data of source languages.

We test the effectiveness of grounding on IR, in creating all-around stronger Code-LMs by structuring our inquiry into the following research questions:

RQ1: Does training with explicit grounding via parallel source code-IR corpora provide benefits over continued pre-training on (unpaired) source code or IR alone?

RQ2: Does grounding on IR improve robustness to prompt perturbations common in human inputs?

RQ3: Does training on parallel source-IR data improve multilingual performance on code completion and understanding, with IR driving the positive knowledge transfer?

RQ4: What effect does pre-training on IR have on multilingual instruction following?

2 Related Work

We provide a concise overview of the three most pertinent lines of work: (1) high-quality pre-training data curation, (2) grounding with toolchain

[1] GitHub Language Stats
[2] Linux Kernel Lore: Rust introduction for v6.1-rc1
[3] Octoverse Report 2022
[4] Toolchain is a set of tools required to create functional software, e.g., *compiler*, *linker*, *libraries*, or *debugger*.

metadata, and (3) alignment across and cross-lingual transfer between languages.

Curated Data For Multilingual Generalization. Curating high-quality and domain-specific data with instructional value leads to more sample-efficient LM pretraining: Phi-1 (Gunasekar et al., 2023), for example, trained on as few as 7B tokens, performs on a par with models trained on hundreds of times more uncurated data. Curation alone, as Cassano et al. (2023a) show, does not suffice for multilingual Code-LMs to generalize to underrepresented and unseen programming languages. Instead, the authors resort to using a bare-bones testcase transpiler to translate synthetic testcases to the target language, validating the quality of the synthetic target language data generated this way. This finding is in line with results from (Rozière et al., 2022), where the benefits of such verification have been demonstrated for code translation.

The compiler IR that we leverage in this work is the result of several sequentially executed transformations that—inter alia—eliminate dead code, unroll loops, combine expressions, and inline subroutines and thus offer significant instructional value without the need for generating unit tests, as the transformations are guaranteed to preserve the correctness of the source code.

Grounding in Toolchain Metadata. There exists an extensive body of work that leverages the structure of the code as well as information originating from artifacts of various stages of compilation to ground code generation. Starting with compiler frontend artifacts, attempts have been made to leverage Abstract Syntax Trees (ASTs) for grounding source code understanding by linearizing them and encoding with LSTMs (Jiang et al., 2022), GNNs (Zhang et al., 2022), CNNs (Mou et al., 2016), Transformers (Guo et al., 2022), or some combination thereof (Sun et al., 2020). Other modes of reliance on ASTs include (i) using them as a search prior for graph-based decoding (Brockschmidt et al., 2019), (ii) predicting (heuristically selected) paths from the tree as an auxiliary pre-training objective (Tipirneni et al., 2024) and, (iii) leveraging them for data augmentation: heuristic generatation of meaning-preserving transformations, leveraged for contrastive learning (Jain et al., 2021; Quiring et al., 2019; Bahrami et al., 2021). Other compiler frontend artifacts such as Data Flow Graphs (DFGs) (Brauckmann et al., 2020) and Control Flow Graphs (CFGs) (Nair et al., 2020) have also been employed in grounding program understanding. Finally, there is work (Shojaee et al., 2023; Le et al., 2022) that derives the reward that guides program generation via reinforcement learning (RL) from AST, CFG, and DFG matches between the generated and reference code.

On the opposite end, compiler backend outputs have also been employed to ground Code-LMs, with compilation feedback in text form being favored by several recent efforts (Jiang et al., 2023; Chen et al., 2023b; Gou et al., 2023) to guide refining of tentative program generations. Concurrent work (Liu et al., 2023) has proposed to create an RL reward to guide generation based on the kind and severity of compilation error outputs.

Finally, several existing efforts also leverage the IR produced by the compiler middle-end during its optimization passes, with LLVM being the most frequent choice of IR choice. IRGen (Li et al., 2022) performs an exploratory study into using the IR itself as a meaning-preserving augmentation to perform contrastive learning on `C` source code. MulCS (Ma et al., 2023) reports improvements to multilingual code snippet search when the GNN encoder utilizes a custom semantic graph derived from the IR. In the work most closely related to ours, Szafraniec et al. (2023) address code translation between four languages, pre-training the translation model using a wide variety of objectives, including source code to IR translation. Their effort, however, is limited to code translation (i.e., they do not consider any other task) and parallel source-to-IR data only at the function level (i.e., short context). In this work, in contrast, we investigate the general utility (i.e., for a wide range of downstream tasks) of pre-training multilingual Code-LMs using parallel source-to-IR data, scaling additionally up the data collection effort to (i) 12 programming languages and, importantly, (ii) self-contained file-level programs, which, intuitively, allows for grounding of many more source-code concepts (e.g., those instantiated with longer code spans) in IR. Importantly, we demonstrate that standard LM training on parallel source-to-IR data alone improves the robustness and multilingual ability of Code-LMs, without any architectural interventions and training auxiliary objectives.

Cross-lingual Transfer and Alignment. Most mainstream code generation models (Li et al., 2023; Guo et al., 2024; Nijkamp et al., 2023; Roziere et al., 2023; Zheng et al., 2023), due to being pre-trained on GitHub code, are multilingual by default. Hence, they are subject to the *curse of multilinguality* i.e. the degradation of model performance on high resource languages when the number of training languages or the proportion of low resource data in the pre-training corpus of a multilingual model is scaled up. This is usually caused by negative interference between unrelated languages to which the model can only allocate a fixed capacity (Lauscher et al., 2020; Wu and Dredze, 2020) and is a well-documented phenomenon in natural language models (Conneau et al., 2020; Arivazhagan et al., 2019). Attempts to circumvent it without scaling up the model to impractical sizes have resorted to sparsity (Ansell et al., 2022; Lee and Hwang, 2023), modularity (Pfeiffer et al., 2022) and model merging (Blevins et al., 2024). While the presence of similar phenomena has been verified in multilingual Code-LMs (Orlanski et al., 2023; Athiwaratkun et al., 2023), research into cross-lingual transfer and alignment across programming languages has been rather sparse, exploring a limited set of tasks and languages (Chen et al., 2022) or introducing task specific architectural interventions (Yuan et al., 2022; Pian et al., 2023) which are hard to scale.

Wang et al. (2020) indicate that separation of model parameters into language-agnostic and language-specific subsets can result in language-specific parameters being the cause of negative interference. This, we believe, presents an opportunity to minimize such interference by means of a shared intermediate representation rather than language-specific parameters. While it is unclear what such representation would be in the case of natural languages, intermediate compiler representations make an obvious choice for programming languages. Grounding Code-LM pretraining on IR data, we believe, should also and improve generalization (including to languages unseen in pretraining) and consequently facilitate cross-lingual transfer in downstream tasks, akin to cross-lingual transfer between non-English languages by models trained on English-centric bi-texts (Gao et al., 2023; Artetxe and Schwenk, 2019). Our experiments on a large array of tasks and programming languages show that this indeed is the case.

3 SLTrans: A Source Code to LLVM IR Translation Pairs Dataset

In order to test the hypotheses we posit in Section 1, we seek to acquire parallel source-IR data for a mixture of low-, medium-, and high-resource programming languages.

Intermediate Code Representation. We utilize LLVM (Lattner and Adve, 2004) as the intermediate representation of our choice because it possesses many of the qualities we deem beneficial for our objectives. LLVM is the most prevalent IR in existing code corpora (Kocetkov et al., 2023) and one of the few frameworks that maintain a well-developed human-readable IR standard[5] rendering its syntax and semantics learnable via language modelling. Additionally, LLVM is adopted as the target IR of many compiler frontends across several programming languages[6] mainly due to the ease with which its tooling infrastructure enables upstart languages to attain general availability.

Language	Frontend	No. Samples	
		Opt-Level -Oz	Opt-Level -O3
C++	clang	2,956,611	2,897,477
C	clang	419,227	411,332
Python[7] [Codon]	codon	291,011	284,676
Rust	rustc	82,667	74,689
Haskell	ghc	61,483	59,378
Go	gollvm	55,578	42,241
Fortran	flang	35,288	31,299
D	ldc	18,111	6,125
Ruby[8] [Crystal]	crystal	13,949	5,787
Nim	nlvm	2,865	-
Swift	swiftc	2,179	1,354
Objective-C	clang	403	261
	Total:	3,939,372	3,814,619

Table 1: Breakdown of SLTrans across programming languages (with respective compiler frontends).

SLTrans Creation. Extracting LLVM IR from free-form source code in GitHub demands compilable and complete code units, the collection of which comes with several challenges. The proportion of compilable code units in free-form code is abysmally low due to the need for tracking dependencies. Many languages such as C and C++ do not have mature package management systems, which makes following dependency paths across repository boundaries virtually impossible. The problem is exacerbated by the difficulty of reliably following within-repository file-level dependencies due to aggressive de-duplication of source files during curation of language modelling code corpora

[5] https://llvm.org/docs/LangRef.html

[6] https://llvm.org/ProjectsWithLLVM/

[7] We source Python data via a Codon, which implements a statically typed subset of the Python language specification

[8] We source Ruby samples via Crystal — a statically typed and compiled derivative of the language

Figure 2: A high-level overview of our parallel data sourcing and training objective. Each source file is compiled via a corresponding LLVM frontend to obtain human-readable IR. The source and IR are then concatenated and the model is required to auto-regressively predict the tokens of one with the other in context thus aligning the constructs in the respective programming language with their analogues in LLVM.

for performance reasons (Allamanis, 2019; Lee et al., 2022): this mangles the repository structure[9]. Additionally, there are also obstacles to obtaining *complete* compilation units. Languages such as `Rust`, `Go` or `Swift` simply cannot be compiled at the file level (unless the files are self-contained) as their respective LLVM frontends operate on module or package-level compilation units. As of this writing, multi-file repository-level code language modeling is unsupported by most mainstream code models (Roziere et al., 2023; Li et al., 2023; Nijkamp et al., 2023). As a result, prior attempts at extracting parallel source-IR data have been stymied by the need to implement language-specific mechanisms to track dependency code for successful compilation (Grossman et al., 2023) and thus mostly restricted to function/snippet level code (Szafraniec et al., 2023), which is very limiting in terms of coverage of language constructs (Li et al., 2022).

We sidestep the above issues by sourcing self-contained compilation units from accepted solutions to programming contest problems (Rosetta Code, 2023; Mirzayanov, 2020; Puri et al., 2021; Caballero and Sutskever, 2021), which typically do not have cross-file dependencies. We then compile these source files into two IR flavours: size-optimized (`-Oz` opt-level equivalent) IR and performance-optimized (`-O3` opt-level equivalent) IR. We further filter only samples with IR shorter than 2500 code lines. The size-optimized IR allows for larger context windows in LLM inference and is also more uniform across languages; being used for deployment, the performance-optimized IR is more prevalent in open-domain code corpora. Collecting both enables fine-grained trade-offs between the two IR forms during language modelling. Finally, given the abundance of near-duplicates in programming contest solutions, we perform MinHash-based (Broder, 1997) de-duplication. The final dataset, dubbed `SLTrans`, consists of ca. 4M samples across 12 programming languages, totalling 26.2B tokens.[10] A breakdown of `SLTrans` is given in Table 1.

4 Experimental Setup

Data Preparation. We leverage LLVM IR to ground matching constructs across heterogeneous languages and facilitate cross-lingual transfer: as header data (along with superfluous platform, vendor, and memory layout information) does not contribute to this goal, we remove it from IR before pairing with source code. We choose the size-optimized IR 80% of the time and performance-optimized IR for 20% of the training samples.

Given our computational budget, we could afford to perform continued pretraining on IR-grounded code on approximately 1.5B tokens. Given that (u) this is substantially smaller than the overall size of `SLTrans` and (ii) acknowledging the skewed language distribution of the dataset, we sub-sample the training corpora using token-level UniMax-1 sampling (Chung et al., 2023), based on the StarCoderBase tokenizer (Li et al., 2023). We select a token budget of 600M tokens this way. We next source 200M tokens of unpaired open-domain IR

[9] A particularly common scenario is a popular repository with several forks being torn apart as the de-duplication pipeline non-deterministically selects disjoint subsets of files from different forks

[10] Based on the StarCoderBase tokenizer.

code from TheStack (Kocetkov et al., 2023), to allow Code-LMs to better learn the IR itself. Finally, to avoid catastrophic forgetting of pre-trained source-code knowledge, we spend the remaining 700M tokens of our budget on high-quality code and text data: these include math articles from the OpenWebMath dataset (Paster et al., 2023), research articles from the PeS2o dataset (Soldaini and Lo, 2023), source code sampled from language splits in TheStack present in SLTrans and single-file changing GitHub commits[11]. The breakdown of the final dataset, to which we refer with Paired, is given in Table 2.

Data Mix	CodeText	Unpaired	Paired
OpenWebMath	300M	200M	100M
PeS2o	500M	300M	200M
Git Commits	200M	150M	100M
TheStack	500M	400M	300M
Unpaired IR	-	450M	200M
Source-IR Pairs (SLTrans)	-	-	600M
Total:	1.5B	1.5B	1.5B

Table 2: Token counts for the paired, unpaired, and codetext pre-training dataset (StarCoderBase tokenizer).

Model Training. Aiming for robust findings, we test the effects of IR grounding on six different Code-LMs from three different providers, ranging in size from 1.1B to 7.3B parameters: StarCoderBase (Li et al., 2023) 1.1B, 3.1B, and 7.3B; DeepSeekCoder (Guo et al., 2024) 1.3B and 5.7B; and CodeLlama (Roziere et al., 2023) 6.7B.

We perform continued LM training for each of these models on the Paired dataset built from SLTrans. We introduce two new sentinel tokens—<s2l> and <l2s>—into the models' vocabulary and use them, respectively, for two possible directions of grounding (each sampled for 50% of training instances):

```
source_code <s2l> llvm_ir <|EOS|>
llvm_ir <l2s> source_code <|EOS|>
```

We randomly initialize the sentinel tokens' embeddings from a Gaussian distribution with the mean set to the average of all pre-trained vocabulary embeddings and retain the variance from the models' initializer configurations.

We rely on LoRA (Hu et al., 2022) for parameter-efficient continued pre-training (we set r to 256 and an α to 128), while keeping the embedding layers trainable. We resort to DeepSpeed (Rasley et al., 2020) Zero Stage-2 to accelerate our training jobs. We train with a maximum sequence length of 4096 tokens using the Adam (Kingma and Ba, 2015) optimizer (β values of (0.95, 0.99)) with a base learning rate of $1e-4$ for the LoRA modules and $4e-5$ for the embedding layers, employing a cosine schedule (culminates at 10% of the base).

[11]We source contents before and after the change, along with the commit message

Model	Base	Continued Pre-Training		
		CodeText	Unpaired	Paired (IRCoder)
StarCoderBase 1.1B	8.35 –	8.39 +0.04	8.59 +0.24	8.76 +0.41
DeepSeekCoder 1.3B	18.34 –	18.22 -0.12	18.77 +0.43	20.51 +2.17
StarCoderBase 3.1B	12.78 –	12.75 -0.03	12.97 +0.19	14.36 +1.58
DeepSeekCoder 5.7B	30.47 –	30.22 -0.25	30.44 -0.03	31.14 +0.67
CodeLlama 6.7B	21.83 –	21.94 +0.11	22.14 +0.31	24.06 +2.23
StarCoderBase 7.3B	17.94 –	17.73 -0.21	18.03 +0.09	18.46 +0.52

Table 3: **RQ1**: Multipl-E pass@1 all language average performance comparison between different continued pre-training settings.

5 Results and Discussion

RQ1: Pairing source code and IR matters. We first test whether the grounding of source code in IR, i.e., language modelling on paired source-IR instances, actually matters. To this end, we compare the performance of models trained on the Paired data against counterparts trained on (1) the unpaired concatenation of code and IR data (dubbed Unpaired) and (2) just more source-code data (referred to as CodeText). CodeText is derived from the same sources as Paired (see Section 4) but it does not contain any (paired or unpaired) LLVM IR data: instead, we simply upsample other sources to reach 1.5B tokens. The Unpaired is upsampled from all sources except the source-IR pairs from our SLTrans, i.e., compared to CodeText, it additionally samples training examples from the 450M token corpus of (unpaired) LLVM IR code from TheStack. Table 2 details the compositions of Unpaired, and CodeText corpora against our primary Paired corpus.

We benchmark Code-LLMs additionally trained on these three corpora against their base variants on the Multipl-E (Cassano et al., 2022) code completion benchmark (with the pass@1 metric), a transpiled expansion of the popular HumanEval (Chen et al., 2021) benchmark to 18 programming languages. We limit our evaluation to the subset of languages present in SLTrans: C++, D, Go, Python, Ruby, Rust and Swift. Comparison of models' performance, displayed in Table 3, brings the key

🤖 Model	ReCode		
	Format	Syntax	Function
StarCoderBase 1.1B	28.08	26.39	11.31
DeepSeekCoder 1.3B	49.61	44.88	25.13
StarCoderBase 3.1B	38.70	33.29	19.04
DeepSeekCoder 5.7B	62.37	55.43	36.73
CodeLlama 6.7B	54.50	45.23	24.49
StarCoderBase 7.3B	46.30	41.50	23.53
IRCoder 1.1B	30.18 +2.10	27.50 +1.11	12.01 +0.70
IRCoder 1.3B	49.85 +0.24	45.43 +0.55	25.75 +0.63
IRCoder 3.1B	39.78 +1.08	34.42 +1.13	18.80 -0.24
IRCoder 5.7B	65.76 +3.39	59.24 +3.82	38.66 +1.93
IRCoder 6.7B	56.41 +1.91	48.11 +2.88	25.47 +0.98
IRCoder 7.3B	46.62 +0.32	41.82 +0.32	23.76 +0.23

Table 4: **RQ2**: ReCode split average pass@1 comparison between IRCoder and the corresponding base models. For detailed perturbation-level breakdowns in the Format, Syntax, and Function splits refer to Tables 6 to 8 in the Appendix respectively.

insights: while adding unpaired IR data can bring some performance gains (compare Unpaired vs. Base and CodeText), these gains are much less pronounced than the gains we obtain by adding paired source-IR data to the training mix (Paired vs. Unpaired). These results strongly suggest that grounding of heterogeneous source code languages in the same IR accounts for the majority of performance gains, and not the mere exposure to IR. Comparison between CodeText and Base reveals that continued training on the data distribution similar to that used in the original pre-training can hurt performance: most models additionally trained on CodeText exhibit small drops in performance compared to their Base variants. This observation is in line with prior findings (Cassano et al., 2023a) and is likely the result of degradations caused by repeating data in language modeling (Allamanis, 2019).[12]

RQ2: Grounding in IR improves robustness to prompt perturbations. We next investigate how our source-IR grounding affects the perturbation robustness of Code-LMs. Such robustness is critical, as malformed and adversarial prompts have been shown to successfully lead to the generation of incorrect (Zhou et al., 2022) and insecure code (Dinh et al., 2023; Wu et al., 2023). Our intuition is that grounding on IR should reduce the vulnerability of Code-LMs to such perturbations, as IR is the result of several transformations that tend to remove the effects of minor semantic variances or even mistakes in the source code. We test our hypothesis using 5 differently seeded ReCode (Wang et al., 2023) transformations of HumanEval to measure robustness to three classes of perturbations in Python: code formatting, syntactic variation, and function name mangling.

As evidenced by the detailed results in Table 4, IRCoder displays gains across the board, with particularly significant gains in robustness against syntactic variations that are typical for human-written prompts. Interestingly, the gains for robustness to function header mangling are substantially smaller. We believe that this is the artefact of the benchmark, abundant with prompts that include headers and docstrings, which may underestimate the functional robustness of IR grounding "in the wild".

RQ3: IR grounding improves multilingual code understanding. We next test the multilingual code completion and understanding capabilities of the models after IR grounding, both in zero-shot and fine-tuning setups. For completion, we report performance on Multipl-E in terms of pass@1, pass@10, and pass@25. We test zero-shot code understanding on CodeXGLUE (Lu et al., 2021) docstring generation task, which requires models to generate a docstring description given the function code as the prompt. We measure the performance with Smoothed BLEU-4 (Lin and Och, 2004) scores w.r.t. the reference docstrings for the languages present in SLTrans: Python, Ruby, and Go.

Regarding fine-tuning, we benchmark on the Commit Chronicle (Eliseeva et al., 2023) commit message generation task. [13]. For the 8 languages present in SLTrans— C, C++, Go, Objective-C, Python, Ruby, Rust and Swift—we fine-tune the IR-grounded Code-LLMs and report the performance in terms of ROUGE-2 and ROUGE-L against the reference commit messages.

Results in Table 5 show that IRCoder significantly and consistently outperforms the base LLMs on all multilingual benchmarks. The language-level breakdown of results (in Appendix B), suggests that grounding in IR facilitates cross-lingual transfer since we observe substantial improvements for low-resource language.

The results demand one further point of discus-

[12]While the exact pre-training corpora of DeepSeekCoder and CodeLlama is unknown, their data collection pipelines promise large overlaps with our CodeText corpus.

[13]The task also indirectly tests the commonsense knowledge in various languages as it requires the commit message to be generated purely from the *diff* in the absence of the original code

Model	Multipl-E			CodeXGLUE Code-Text	Commit Chronicle		HumanEvalFixDocs	
	pass@1	pass@10	pass@25	BLEU-4	ROUGE-2	ROUGE-L	pass@1	pass@10
StarCoderBase 1.1B	8.35	13.43	16.43	10.05	12.41	33.67	12.23	17.70
DeepSeekCoder 1.3B	18.34	26.12	31.36	9.63	12.33	33.16	25.48	36.22
StarCoderBase 3.1B	12.78	19.09	22.62	10.61	14.35	33.70	28.44	42.91
DeepSeekCoder 5.7B	30.47	41.38	48.04	11.80	13.77	35.18	48.21	61.05
CodeLlama 6.7B	21.83	34.78	42.50	9.74	14.46	35.91	44.50	56.79
StarCoderBase 7.3B	17.94	27.38	34.12	10.74	15.22	37.74	40.74	55.27
IRCoder 1.1B	8.76 +0.41	14.51 +1.08	19.32 +2.89	11.41 +1.36	13.15 +0.73	35.04 +1.38	13.01 +0.78	18.00 +0.30
IRCoder 1.3B	20.51 +2.17	31.14 +5.02	37.90 +6.54	10.79 +1.16	13.12 +0.79	34.57 +1.41	27.07 +1.59	37.95 +1.73
IRCoder 3.1B	14.36 +1.58	22.58 +3.49	28.02 +5.39	11.74 +1.13	14.29 -0.06	36.81 +1.11	28.99 +0.55	42.76 -0.15
IRCoder 5.7B	31.14 +0.67	45.00 +3.62	51.29 +3.25	13.21 +1.41	14.71 +0.93	37.15 +1.97	49.79 +1.57	66.35 +4.29
IRCoder 6.7B	24.06 +2.23	39.38 +4.60	47.03 +4.53	11.15 +1.41	14.95 +0.49	36.82 +0.91	46.59 +2.09	58.74 +1.95
IRCoder 7.3B	18.46 +0.52	30.43 +3.05	38.04 +3.92	11.16 +0.42	15.88 +0.67	38.96 +1.22	44.07 +3.33	57.38 +2.11

Table 5: **RQ3** and **RQ4**: All language average performance comparison between IRCoder and the corresponding base models on multilingual tasks. For detailed language-wise breakdowns in Multipl-E results refer to Tables 9 to 11, CodeXGLUE code to text results refer to Table 12, Commit Chronicle results refer to Tables 13 and 14, and HumanEvalFixDocs results refer to Tables 15 and 16 in the Appendix.

sion. Our findings are in contrast with the findings of Orlanski et al. (2023) who show a trade-off between the performance on high and low-resource languages: we, instead, observe gains across the board with no evidence of interference even between typologically disparate programming languages. We find that the IR-grounding also substantially boosts performance on high-resource languages like C++ and Python for which the Code-LLMs have seen hundreds of billions of tokens in pre-training. This contributes to the hypothesis that, despite their large-scale pre-training, Code-LMs gain a limited understanding of higher-level concepts such as control and data flow (Hooda et al., 2024), instead resorting to superficial attributes such as identifier names for anchoring representations across languages (Ahmed and Devanbu, 2022). IR, instead, quite intuitively, does have the potential to align code representations over such concepts. For example, the single-static assignment (SSA) form used by LLVM alongside transformations such as loop vectorization and register allocation specifies the data flow explicitly; other modifications captured by IR, such as loop simplification also aid in simplifying the control flow of source code thus aiding code understanding.

RQ4: Grounding in IR improves multilingual instruction following. Finally, we test if the improvements from IR grounding extend to instruction following. To this end, we perform 3 epochs of instruction tuning on 23.5k instruction-output pairs and evaluate instruction following on the HumanEvalFixDocs (Muennighoff et al., 2023) task. The task instructs the model to fix buggy code snippets given the docstring of the correct sub-routine and tests the models' ability to correct faults such as identifier and operator misuse as well as missing or excess logic. We evaluate for SLTrans languages: C++, Go, Python, and Rust. Table 5 shows again that IR grounding brings performance gains, with the largest improvements observed for the strongest Code-LMs. This is consistent with existing work which shows that the benefits of instruction tuning are most apparent for strong base models (Muennighoff et al., 2023; Longpre et al., 2023).

6 Conclusion

In this work, we investigate the effects of grounding heterogeneous source-code to a shared intermediate representation (IR) on code understanding and generation abilities of Code-LLMs. To this end, we first create SLTrans, a 26.2B token source code-IR parallel dataset containing nearly 4M training examples. We then perform continued pretraining on the corpus that includes parallel source code-IR data from SLTrans for 6 established Code-LMs, demonstrating the IR grounding brings substantial performance gains in prompt robustness, multilingual code completion, code understanding, and instruction following, all while training on data orders of magnitude smaller than Code-LLM pre-training corpus. We hope that our encouraging results catalyze broader research efforts on the inclusion of intermediate code representations both in Code-LLM pre-training as well as in the post-hoc adaptation of pre-trained models.

Limitations

We show that compiler IR is a powerful source of cross-lingual alignment that allows for the structures in various languages to be anchored in common IR constructs. However, this is by no means a perfect process. Different frontends make disparate choices regarding how source code must be transformed to IR leading to several 'dialects' of IR that are all valid but may slightly differ. While this does not seem to get in the way of our gains, it might have an effect when our approach is extended to newer languages with less mature toolchains.

Additionally, while the middle-end LLVM IR is intended to be a target-platform agnostic representation, this constraint can sometimes be violated due to the presence of platform-specific constants, application binary interface code, and linker logic. For our purposes, this was worked around by some data cleaning and by sourcing the IR consistently from the same platform.

Thirdly, there is a risk that the IR may not be able to anchor all the constructs of a language. While in some languages like `C` and `C++` there is a strong mapping between the language constructs and LLVM ones, in others the association might be less tight. For instance, in `Rust`, the source code is first transformed to the language's own IR[14] before the LLVM framework is used. Our results indicate that this hasn't gotten in the way so far.

Finally, due to the IR being on average several times longer than the source code, there arise constraints on the types of models to which our approach can be applied. Most competitive Code-LMs have a context window of at least 4096 tokens making this largely a non-issue. However, it might pose problems in applying this method to older Code-LMs.

Ethical Risks

Our work does not directly interface with any human annotators, with our data collection and evaluation being completely automated. However, there still exists the risk of our improved model being more competent at generating malicious code. This is a prospect we haven't explicitly evaluated for. We take mitigating steps by releasing the Docker containers used in our training and evaluation jobs, to minimize the risks to downstream users employing our models and methods. As a further guardrail, we plan to release our artefacts under a non-commercial license.

[14]https://rustc-dev-guide.rust-lang.org/mir/index.html

Acknowledgements

This work has been funded by Huawei Technologies (Ireland) Co., Ltd.

References

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455.

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153.

Alan Ansell, Edoardo Ponti, Anna Korhonen, and Ivan Vulić. 2022. Composable sparse fine-tuning for cross-lingual transfer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1778–1796, Dublin, Ireland. Association for Computational Linguistics.

Naveen Arivazhagan, Ankur Bapna, Orhan Firat, Dmitry Lepikhin, Melvin Johnson, Maxim Krikun, Mia Xu Chen, Yuan Cao, George Foster, Colin Cherry, et al. 2019. Massively multilingual neural machine translation in the wild: Findings and challenges. *arXiv preprint 1907.05019*.

Mikel Artetxe and Holger Schwenk. 2019. Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond. *Transactions of the Association for Computational Linguistics*, 7:597–610.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

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

Mehdi Bahrami, NC Shrikanth, Yuji Mizobuchi, Lei Liu, Masahiro Fukuyori, Wei-Peng Chen, and Kazuki Munakata. 2021. Augmentedcode: Examining the effects of natural language resources in code retrieval models. *arXiv preprint 2110.08512*.

Terra Blevins, Tomasz Limisiewicz, Suchin Gururangan, Margaret Li, Hila Gonen, Noah A Smith, and Luke Zettlemoyer. 2024. Breaking the curse of multilinguality with cross-lingual expert language models. *arXiv preprint 2401.10440*.

Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 201–211.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative code modeling with graphs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE.

Ethan Caballero and Ilya Sutskever. 2021. Description2code dataset.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Carolyn Jane Anderson, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2023a. Knowledge transfer from high-resource to low-resource programming languages for code llms.

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

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Anton Lozhkov, Carolyn Anderson, and Arjun Guha. 2023b. Can it edit? evaluating the ability of large language models to follow code editing instructions. *arXiv preprint 2312.12450*.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 18–30.

Fuxiang Chen, Fatemeh H. Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 401–412, New York, NY, USA. Association for Computing Machinery.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint 2107.03374*.

Nuo Chen, Qiushi Sun, Jianing Wang, Xiang Li, and Ming Gao. 2023a. Pass-tuning: Towards structure-aware parameter-efficient tuning for code representation learning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 577–591, Singapore. Association for Computational Linguistics.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching large language models to self-debug. *arXiv preprint 2304.05128*.

Hyung Won Chung, Xavier Garcia, Adam Roberts, Yi Tay, Orhan Firat, Sharan Narang, and Noah Constant. 2023. Unimax: Fairer and more effective language sampling for large-scale multilingual pretraining. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, Online. Association for Computational Linguistics.

Kaustubh D Dhole, Varun Gangal, Sebastian Gehrmann, Aadesh Gupta, Zhenhao Li, Saad Mahamood, Abinaya Mahendiran, Simon Mille, Ashish Shrivastava, Samson Tan, et al. 2021. Nl-augmenter: A framework for task-sensitive natural language augmentation. *arXiv preprint 2112.02721*.

Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Large language models of code fail at completing code with potential bugs. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Maddila, Vijayaraghavan Murali, et al. 2024. Multi-line ai-assisted code authoring. *arXiv preprint 2402.04141*.

Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. 2023. From commit message generation to history-aware commit message completion. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 723–735. IEEE.

Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Dan Zheng, Satish Chandra, and

Petros Maniatis. 2024. Resolving code review comments with machine learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

Pengzhi Gao, Liwen Zhang, Zhongjun He, Hua Wu, and Haifeng Wang. 2023. Learning multilingual sentence representations with cross-lingual consistency regularization. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 243–262, Singapore. Association for Computational Linguistics.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. Ast-t5: Structure-aware pretraining for code generation and understanding. *arXiv preprint 2401.03003*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint 2305.11738*.

Aiden Grossman, Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, and Johannes Doerfert. 2023. Compile: A large ir dataset from production sources. *arXiv preprint 2309.15432*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint 2306.11644*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

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

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. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Ashish Hooda, Mihai Christodorescu, Miltos Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. 2024. Do large code models understand programming concepts? a black-box approach. *arXiv preprint 2402.05980*.

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 *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. 2022. An AST structure enhanced decoder for code generation. *IEEE ACM Trans. Audio Speech Lang. Process.*, 30:468–476.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Self-evolve: A code evolution framework via large language models. *arXiv preprint 2306.02907*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Denis Kocetkov, Raymond Li, Loubna Ben allal, Jia LI, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and Harm de Vries. 2023. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi Rui Tam, Keith Stevens, Abdullah Barhoum, Duc Nguyen, Oliver Stanley, Richárd Nagyfi, Shahul ES, Sameer Suri, David Glushkov, Arnav Dantuluri, Andrew Maguire, Christoph Schuhmann, Huu Nguyen, and Alexander Mattick. 2023. Openassistant conversations - democratizing large language model alignment. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*

(CGO 2004), 20-24 March 2004, San Jose, CA, USA, pages 75–88. IEEE Computer Society.

Anne Lauscher, Vinit Ravishankar, Ivan Vulić, and Goran Glavaš. 2020. From zero to hero: On the limitations of zero-shot language transfer with multilingual Transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4483–4499, Online. Association for Computational Linguistics.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Jaeseong Lee and Seung-won Hwang. 2023. Multilingual lottery tickets to pretrain language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9387–9398, Singapore. Association for Computational Linguistics.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, Dublin, Ireland. Association for Computational Linguistics.

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*. Reproducibility Certification.

Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2253–2265. ACM.

Wing Lian, Bleys Goodson, Eugene Pentland, Austin Cook, Chanvichet Vong, and "Teknium". 2023. Openorca: An open dataset of gpt augmented flan reasoning traces. `https://https://huggingface.co/Open-Orca/OpenOrca`.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint 2307.04349*.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. 2023. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 22631–22648. PMLR.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Yingwei Ma, Yue Yu, Shanshan Li, Zhouyang Jia, Jun Ma, Rulin Xu, Wei Dong, and Xiangke Liao. 2023. Mulcs: Towards a unified deep representation for multilingual code search. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*, pages 120–131. IEEE.

Mike Mirzayanov. 2020. Codeforces: Results of 2020 [annual report]. `https://codeforces.com/blog/entry/89502`.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1287–1293. AAAI Press.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023*

Workshop on Instruction Tuning and Instruction Following.

Aravind Ashok Nair, Avijit Roy, and Karl Meinke. 2020. funcgnn: A graph neural network approach to program similarity. In *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*, pages 10:1–10:11. ACM.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 26619–26645. PMLR.

Keiran Paster, Marco Dos Santos, Zhangir Azerbayev, and Jimmy Ba. 2023. Openwebmath: An open dataset of high-quality mathematical web text. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*.

Jonas Pfeiffer, Naman Goyal, Xi Lin, Xian Li, James Cross, Sebastian Riedel, and Mikel Artetxe. 2022. Lifting the curse of multilinguality by pre-training modular transformers. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3479–3495, Seattle, United States. Association for Computational Linguistics.

Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Metatptrans: A meta learning approach for multilingual code representation learning. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 5239–5247. AAAI Press.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 479–496. USENIX Association.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. ACM.

Rosetta Code. 2023. Rosetta code.

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

Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*.

Luca Soldaini and Kyle Lo. 2023. peS2o (Pretraining Efficiently on S2ORC) Dataset. Technical report, Allen Institute for AI. ODC-By, `https://github.com/allenai/pes2o`.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 8984–8991. AAAI Press.

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

Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. Structcoder: Structure-aware transformer for code generation. *ACM Trans. Knowl. Discov. Data*, 18(3):70:1–70:20.

Jennifer Tracey, Stephanie Strassel, Ann Bies, Zhiyi Song, Michael Arrigo, Kira Griffitt, Dana Delgado, Dave Graff, Seth Kulick, Justin Mott, and Neil Kuster.

2019. Corpus building for low resource languages in the DARPA LORELEI program. In *Proceedings of the 2nd Workshop on Technologies for MT of Low Resource Languages*, pages 48–55, Dublin, Ireland. European Association for Machine Translation.

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. 2023. 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.

Zirui Wang, Zachary C. Lipton, and Yulia Tsvetkov. 2020. On negative interference in multilingual models: Findings and a meta-learning treatment. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4438–4450, Online. Association for Computational Linguistics.

Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. 2023. Deceptprompt: Exploiting llm-driven code generation via adversarial natural language instructions. *arXiv preprint 2312.04730*.

Shijie Wu and Mark Dredze. 2020. Are all languages created equal in multilingual BERT? In *Proceedings of the 5th Workshop on Representation Learning for NLP*, pages 120–130, Online. Association for Computational Linguistics.

Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. 2021. mT5: A massively multilingual pre-trained text-to-text transformer. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 483–498, Online. Association for Computational Linguistics.

Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: continual repair across programming languages. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 678–690. ACM.

Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to represent programs with heterogeneous graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, pages 378–389. ACM.

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 2023, Long Beach, CA, USA, August 6-10, 2023*, pages 5673–5684. ACM.

Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2022. Adversarial robustness of deep code comment generation. *ACM Trans. Softw. Eng. Methodol.*, 31(4):60:1–60:30.

A Experimental Details

We employ Paged Attention (Kwon et al., 2023) via vLLM on model checkpoints loaded in half-precision for efficient inference while evaluating our models on zero-shot inference benchmarks. All our inference runs are conducted on Nvidia A100 80GB GPUs with 95% of the GPU VRAM explicitly reserved for vLLMs GPU pages. We further set aside 64GB of RAM as a CPU swap, allowing for offloading pages to the CPU during bursts of long sequences. We limit the continuous batching parameter to 32 to minimize incidents of running out of swap space.

A.1 Multipl-E

We sample $N = 50$ continuations of at most 1024 tokens for all Multipl-E runs. While the more common standard is to choose $N = 200$, in the interest of efficiency, we follow existing work (Li et al., 2023) that shows that one can obtain reliable `pass@k` estimates in as few as 20 generations. We always use nucleus sampling with p of 0.9.

Our estimates of `pass@1` emulate usage scenarios where correctness is paramount. Hence, we utilize a low temperature of 0.2. In contrast, for our `pass@10` and `pass@25` we mimic scenarios where creativity and diversity of generations are more important and hence use a higher temperature of 0.8. This practice keeps us in line with prior work (Roziere et al., 2023).

A.2 ReCode

We run three of the four ReCode evaluations using HumanEval as the base dataset. The Format sub-task scrutinizes how robust these models are to source formatting variations such as turning docstrings into comments and randomly inserting newlines. The Syntax sub-task tests models' susceptibility to syntactic variation patterns common in human-written code (Chakraborty et al., 2022) such as dead-code blocks and renamed variables. Finally, the Function sub-task tests models' robustness to conventional variations seen in function names such as inflectional variations or synonym substitutions. We follow the benchmark authors' guidance and estimate pass@1 from one greedily sampled output per prompt of at most 1024 tokens.

We ignore the Docstring sub-task as our pilot runs found the NLAugmenter (Dhole et al., 2021) transformations it uses to be an unrepresentative of the deviations found in developer prompts.

A.3 CodeXGLUE Code-to-Text

We greedily sample continuations capped at 512 tokens and measure their smoothed BLEU-4 score against the reference docstrings. The prompts per language are detailed below:

```
# Python
[source_code]
""" The goal of this function is to:

# Ruby
[source_code]
=begin The goal of this function is to:

# Go
[source_code]
/* The goal of this function is to:
```

A.4 Commit Chronicle

We randomly partition 80%, 10% and 10% of the data into train, validation, and test splits for the 8 languages present in SLTrans — C, C++, Go, Objective-C, Python, Ruby, Rust and Swift. For languages with a lot of diff samples, we cap the train split at 25,000 samples We train for 3 epochs with a maximum sequence length of 2048 tokens, using LoRA tuning with an r of 32, α of 16, and a batch size of 16. We use the ADAM optimizer with β of (0.95, 0.99) and a base learning rate of 3e-4. We employ a cosine scheduler that finishes at 10% of the base learning rate. Unlike continued pre-training, in this phase, losses are only back-propagated for the continuations.

A.5 Instruction Tuning

We collate 18k instruction-output pairs from EditPackFT (Cassano et al., 2023b), which are derived by re-formatting the file contents and commit messages of single-file edit GitHub commits. In the interest of preserving natural language ability, we also source a further 5.5k code-adjacent natural language instruction-output pairs from the OASST (Köpf et al., 2023) and OpenOrca (Lian et al., 2023) collections. We perform 3 epochs of instruction tuning on all the base and IRCoder models with a maximum sequence length of 2048 and backpropagate losses on only the continuations. We leverage LoRA tuning with an r of 32, α of 16, and a batch size of 16. We use the ADAM optimizer with β of (0.95, 0.99) and a base learning rate of 3e-4. We employ a cosine scheduler that finishes at 10% of the base learning rate. Our instruction tuning template is outlined below, with losses calculated on only the completions:

```
### Instruction:

Text Instruction
[optional_code]

### Response:
Completion <|EOS|>
```

A.6 HumanEvalFixDocs

We benchmark the instruction following ability of our models using pass@1 at temperature 0.2 and pass@10 at temperature 0.8 by sampling 20 continuations of at most 1024 tokens. Here again, the first setting is designed to mimic factual generations, and the second is to recreate more creative settings. The task consists of the buggy code followed by the correct docstring and an accompanying instruction to fix the code snippet. This information is usually cast to the models' instruction tuning template and input as a prompt as outlined below:

```
### Instruction:
Fix bugs in [function_name]

[buggy_code]

[Correct Code Docstring]

### Response:
```

B Detailed Results

For completeness, we detail the split and language-wise performance of the models on all tasks discussed in Section 5.

Model	Doc to Comments	Newline After Code	Newline After Doc	Newline Random	Line Split	Tab Indent
StarCoderBase 1.1B	23.40	29.87	31.70	27.42	27.43	28.65
DeepSeekCoder 1.3B	45.43	53.44	50.98	50.60	44.18	53.04
StarCoderBase 3.1B	34.78	39.63	40.85	37.80	40.11	39.02
DeepSeekCoder 5.7B	50.12	61.42	66.38	64.63	64.67	66.98
CodeLlama 6.7B	50.11	56.09	55.87	54.43	52.58	57.92
StarCoderBase 7.3B	40.11	48.59	47.88	44.89	47.53	48.78
IRCoder 1.1B	26.14 +2.74	30.48 +0.61	34.48 +2.78	29.21 +1.78	30.70 +3.27	30.04 +1.39
IRCoder 1.3B	42.46 -2.75	53.97 +0.53	51.44 +0.46	49.44 -1.16	46.34 +2.16	55.23 +2.19
IRCoder 3.1B	37.14 +2.36	40.24 +0.61	39.63 -1.22	40.41 +2.61	40.41 +0.30	40.85 +1.83
IRCoder 5.7B	57.31 +7.19	68.28 +6.86	68.90 +2.52	66.74 +2.11	64.98 +0.31	68.36 +1.38
IRCoder 6.7B	54.18 +4.07	55.85 -0.24	57.92 +2.05	55.48 +1.05	55.71 +3.13	59.33 +1.41
IRCoder 7.3B	40.85 +0.74	43.98 -4.61	48.06 +0.18	48.06 +3.17	49.31 +1.88	49.37 +0.59

Table 6: ReCode Format pass@1 comparison between `IRCoder` and its corresponding base models.

Model	Dead Code Insert	For While Transform	Operand Swap	Var Renaming CB	Var Renaming Naive	Var Renaming RN
StarCoderBase 1.1B	8.53	32.93	29.14	31.70	29.87	26.18
DeepSeekCoder 1.3B	17.64	52.43	50.52	50.52	50.60	47.56
StarCoderBase 3.1B	14.02	38.41	39.63	38.11	37.81	31.78
DeepSeekCoder 5.7B	23.48	64.35	61.63	64.02	60.52	58.56
CodeLlama 6.7B	16.94	52.95	51.97	54.26	51.78	43.47
StarCoderBase 7.3B	14.96	50.31	50.31	45.63	45.12	42.68
IRCoder 1.1B	10.36 +1.83	33.94 +1.01	31.70 +2.56	32.31 +0.61	30.60 +0.73	26.11 -0.07
IRCoder 1.3B	18.29 +0.65	49.96 -2.47	50.60 +0.08	49.86 -0.66	54.36 +3.76	49.51 +1.95
IRCoder 3.1B	12.19 -1.83	40.11 +1.70	41.87 +2.24	43.11 +5.00	36.93 -0.88	32.31 +0.53
IRCoder 5.7B	26.92 +3.34	66.47 +2.12	65.84 +4.21	67.68 +3.66	66.46 +5.94	62.19 +3.63
IRCoder 6.7B	18.90 +1.96	56.09 +3.14	55.65 +3.68	54.44 +0.18	54.41 +2.36	49.17 +5.70
IRCoder 7.3B	14.85 -0.11	49.77 -0.54	50.46 +0.15	49.63 +4.00	45.85 +0.73	40.36 -2.32

Table 7: ReCode Syntax pass@1 comparison between `IRCoder` and its corresponding base models.

Model	Camel Case	Butter Fingers	Swap Characters	Change Character Case	Inflectional Variation	Synonym Substitution
StarCoderBase 1.1B	10.24	11.44	13.14	10.33	12.44	10.26
DeepSeekCoder 1.3B	26.99	23.93	26.12	19.65	25.67	28.41
StarCoderBase 3.1B	20.87	18.02	19.42	17.07	20.32	18.56
DeepSeekCoder 5.7B	40.24	34.32	38.47	30.26	40.33	36.76
CodeLlama 6.7B	27.44	25.11	24.41	21.08	23.79	25.09
StarCoderBase 7.3B	26.33	25.02	22.17	19.68	25.39	22.56
IRCoder 1.1B	11.82 +1.58	11.36 -0.08	12.49 -0.65	13.02 +2.69	12.63 +0.19	10.76 +0.50
IRCoder 1.3B	29.94 +2.95	23.17 -0.76	25.67 -0.45	22.06 +2.41	26.76 +1.09	26.88 -1.53
IRCoder 3.1B	21.88 +1.01	17.67 -0.35	18.39 -1.03	15.12 -1.95	20.68 +0.36	19.07 +0.51
IRCoder 5.7B	43.86 +3.62	37.84 +3.52	37.19 -1.28	34.57 +4.31	39.91 -0.42	38.58 +1.82
IRCoder 6.7B	27.11 -0.33	25.17 +0.06	24.56 +0.15	25.78 +4.70	25.44 +1.65	24.77 -0.32
IRCoder 7.3B	26.94 +0.61	24.07 -0.95	22.51 +0.34	20.84 +1.16	25.92 +0.53	22.30 -0.26

Table 8: ReCode Function pass@1 comparison between `IRCoder` and its corresponding base models.

Model	C++	D	Go	Python	Ruby	Rust	Swift
StarCoderBase 1.1B	10.22	3.87	12.79	14.26	4.46	9.21	3.64
DeepSeekCoder 1.3B	28.21	9.77	15.87	27.91	21.21	16.46	8.94
StarCoderBase 3.1B	16.64	4.89	15.63	21.51	4.52	16.31	9.98
DeepSeekCoder 5.7B	43.44	13.65	24.64	42.67	33.43	31.79	23.79
CodeLlama 6.7B	26.72	9.67	18.69	31.13	25.28	21.43	19.87
StarCoderBase 7.3B	23.19	7.62	16.76	27.88	16.96	18.81	14.38
IRCoder 1.1B	11.10 +0.88	4.65 +0.78	11.78 -1.01	14.29 +0.03	6.34 +1.88	9.62 +0.21	3.76 +0.12
IRCoder 1.3B	31.79 +3.58	10.57 +0.80	16.17 +0.30	30.61 +2.70	24.35 +3.14	20.91 +4.45	9.14 +0.20
IRCoder 3.1B	16.87 +0.23	5.67 +0.78	17.78 +2.15	21.98 +0.47	11.46 +6.94	16.78 +0.47	9.96 -0.02
IRCoder 5.7B	45.61 +2.17	15.96 +2.41	23.77 -0.87	42.92 +0.25	34.60 +1.17	33.94 +2.15	21.17 -2.62
IRCoder 6.7B	29.12 +2.40	13.02 +3.35	19.10 +0.41	31.11 -0.02	26.28 +1.00	24.37 +2.94	25.45 +5.58
IRCoder 7.3B	23.06 -0.13	11.97 +4.35	16.81 +0.05	25.24 -2.64	19.52 +2.56	19.63 +0.82	12.99 -1.39

Table 9: Multipl-E `pass@1` comparison between `IRCoder` and its corresponding base models.

Model	C++	D	Go	Python	Ruby	Rust	Swift
StarCoderBase 1.1B	20.81	7.81	17.33	19.97	6.92	12.19	8.87
DeepSeekCoder 1.3B	34.36	16.76	18.98	41.30	37.86	18.94	14.65
StarCoderBase 3.1B	24.16	11.06	18.11	30.86	10.39	22.61	16.43
DeepSeekCoder 5.7B	54.67	22.74	28.78	61.53	46.24	42.96	32.76
CodeLlama 6.7B	45.86	15.56	23.11	52.75	45.87	28.67	31.65
StarCoderBase 7.3B	36.97	16.14	20.21	40.58	33.42	25.63	18.69
IRCoder 1.1B	21.24 +0.43	9.07 +1.18	17.84 +0.51	22.61 +2.64	9.76 +2.84	11.86 -0.33	9.17 +0.30
IRCoder 1.3B	36.42 +5.17	24.27 -0.13	18.95 +1.12	48.67 +4.36	42.39 +12.54	25.48 -0.04	21.82 +1.41
IRCoder 3.1B	29.33 +5.17	10.93 -0.13	19.23 +1.12	35.22 +4.36	22.93 +12.54	22.57 -0.04	17.84 +1.41
IRCoder 5.7B	58.51 +3.84	28.64 +5.90	28.59 -0.19	68.11 +6.58	49.58 +3.34	44.09 +1.13	37.45 +4.69
IRCoder 6.7B	51.76 +5.90	26.14 +10.58	25.48 +2.37	56.71 +3.96	49.40 +3.53	31.28 +2.61	34.89 +3.24
IRCoder 7.3B	39.24 +2.27	20.23 +4.09	19.68 -0.53	45.69 +5.31	36.88 +3.46	27.65 +2.02	23.45 +4.76

Table 10: Multipl-E `pass@10` comparison between `IRCoder` and its corresponding base models.

Model	C++	D	Go	Python	Ruby	Rust	Swift
StarCoderBase 1.1B	28.19	12.78	19.22	23.04	7.65	13.45	10.69
DeepSeekCoder 1.3B	38.44	20.71	22.79	50.68	43.26	20.13	23.49
StarCoderBase 3.1B	27.38	13.98	20.87	33.89	14.59	24.97	22.68
DeepSeekCoder 5.7B	58.59	28.32	31.11	69.14	53.76	50.84	44.55
CodeLlama 6.7B	57.65	22.86	25.73	62.43	55.96	31.95	40.93
StarCoderBase 7.3B	40.28	20.93	22.14	53.44	44.85	30.80	26.41
IRCoder 1.1B	29.79 +1.60	13.45 +0.67	20.02 +0.80	28.43 +5.39	18.79 +11.14	14.31 +0.86	10.43 -0.26
IRCoder 1.3B	40.55 +2.11	31.89 +11.18	23.47 +0.68	57.84 +7.16	52.46 +9.20	28.63 +8.50	30.44 +6.95
IRCoder 3.1B	35.18 +7.80	14.77 +0.79	23.59 +2.72	46.19 +12.30	28.72 +14.13	24.12 -0.85	23.55 +0.87
IRCoder 5.7B	62.16 +3.57	33.79 +5.47	33.86 +2.75	73.41 +4.27	55.08 +1.32	51.69 +0.85	49.04 +4.49
IRCoder 6.7B	60.08 +2.43	35.27 +12.41	26.31 +0.58	67.49 +5.06	59.88 +3.92	35.39 +3.44	44.77 +3.84
IRCoder 7.3B	47.87 +7.59	27.67 +6.74	21.87 -0.27	56.47 +3.03	49.35 +4.50	30.92 +0.12	32.12 +5.71

Table 11: Multipl-E `pass@25` comparison between `IRCoder` and its corresponding base models.

Model	Go	Python	Ruby
StarCoderBase 1.1B	10.23	12.89	7.03
DeepSeekCoder 1.3B	11.29	14.07	3.52
StarCoderBase 3.1B	10.33	12.78	8.71
DeepSeekCoder 5.7B	10.09	14.15	11.16
CodeLlama 6.7B	9.96	14.33	4.94
StarCoderBase 7.3B	9.54	13.52	9.17
IRCoder 1.1B	12.33 +2.10	12.77 -0.12	9.14 +2.11
IRCoder 1.3B	11.87 +0.58	16.62 +2.55	3.88 +0.36
IRCoder 3.1B	11.99 +1.66	13.34 +0.56	9.88 +1.17
IRCoder 5.7B	11.81 +1.72	16.28 +2.13	11.25 +0.39
IRCoder 6.7B	9.90 -0.06	15.61 +1.28	7.95 +3.01
IRCoder 7.3B	10.71 +1.17	13.49 -0.03	9.28 +0.11

Table 12: CodeXGLUE code-to-text smoothed BLEU-4 comparison between IRCoder and its corresponding base models.

Model	C	C++	Go	Obj-C	Python	Ruby	Rust	Swift
StarCoderBase 1.1B	11.76	13.14	10.55	10.81	13.73	17.44	11.13	10.74
DeepSeekCoder 1.3B	11.96	12.67	12.89	10.02	12.77	16.16	11.53	10.66
StarCoderBase 3.1B	13.19	15.74	13.19	14.36	14.57	17.35	13.81	12.56
DeepSeekCoder 5.7B	14.01	14.48	12.91	12.95	14.13	17.83	12.15	11.72
CodeLlama 6.7B	14.26	14.89	13.96	14.24	14.44	17.73	13.37	12.76
StarCoderBase 7.3B	15.06	17.01	14.56	14.68	15.01	18.38	14.06	12.97
IRCoder 1.1B	11.91 +0.15	12.99 -0.15	11.99 +1.44	12.25 +1.44	14.09 +0.36	18.91 +1.47	11.98 +0.85	11.04 +0.30
IRCoder 1.3B	12.20 +0.24	14.68 +2.01	12.85 +0.04	11.93 +1.91	13.29 +0.52	16.56 +0.40	12.45 +0.92	10.99 +0.33
IRCoder 3.1B	13.39 +0.20	14.69 -1.05	13.88 +0.69	13.95 -0.41	14.51 -0.06	17.41 +0.06	13.64 -0.17	12.81 +0.25
IRCoder 5.7B	14.56 +0.55	16.98 +2.50	14.59 +1.68	13.22 +0.27	14.02 -0.11	18.08 +0.25	14.03 +1.88	12.17 +0.45
IRCoder 6.7B	14.36 +0.10	16.06 +1.17	14.96 +1.00	13.92 -0.32	14.64 +0.20	18.51 +0.78	14.46 +1.09	12.66 -0.10
IRCoder 7.3B	15.32 +0.26	17.75 +0.74	15.76 +1.20	14.92 +0.24	15.98 +0.97	19.26 +0.88	14.98 +0.92	13.09 +0.12

Table 13: CommitChronicle ROUGE-2 comparison between IRCoder and its corresponding base models.

Model	C	C++	Go	Obj-C	Python	Ruby	Rust	Swift
StarCoderBase 1.1B	29.78	35.63	32.56	30.03	35.47	39.56	33.97	32.32
DeepSeekCoder 1.3B	31.87	35.91	33.98	30.59	34.71	34.62	32.86	30.77
StarCoderBase 3.1B	32.76	37.84	36.73	37.72	35.76	37.48	34.23	33.06
DeepSeekCoder 5.7B	32.61	36.25	35.48	32.63	36.56	40.07	34.69	33.16
CodeLlama 6.7B	35.02	37.24	36.82	33.43	33.73	39.75	34.46	33.83
StarCoderBase 7.3B	35.93	38.67	37.24	38.53	38.17	40.97	37.28	35.09
IRCoder 1.1B	32.67 +2.89	34.84 -0.79	34.79 +2.23	33.43 +3.40	35.56 +0.09	40.19 +0.63	35.21 +1.24	33.63 +1.31
IRCoder 1.3B	33.30 +1.43	36.14 +0.23	35.11 +1.13	31.58 +0.99	34.25 -0.46	39.34 +4.72	34.69 +1.83	32.14 +1.37
IRCoder 3.1B	35.29 +2.53	38.21 +0.37	36.90 +0.17	36.83 -0.89	37.01 +1.25	40.40 +2.92	36.07 +1.84	33.76 +0.70
IRCoder 5.7B	36.15 +3.54	39.39 +3.14	37.44 +1.96	34.73 +2.10	36.78 +0.22	41.38 +1.31	36.97 +2.28	34.33 +1.17
IRCoder 6.7B	35.94 +0.92	37.81 +0.57	38.11 +1.29	34.08 +0.65	37.01 +0.28	41.02 +1.27	36.41 +1.95	41.17 +0.34
IRCoder 7.3B	37.52 +1.59	40.48 +1.81	38.96 +1.72	38.59 +0.06	38.69 +0.52	43.17 +2.20	38.41 +1.13	35.82 +0.73

Table 14: CommitChronicle ROUGE-L comparison between IRCoder and its corresponding base models.

Model	C++	Go	Python	Rust
StarCoderBase 1.1B	11.26	10.07	18.87	8.70
DeepSeekCoder 1.3B	24.01	25.89	35.36	16.67
StarCoderBase 3.1B	33.87	27.18	34.92	18.89
DeepSeekCoder 5.7B	52.32	52.89	57.88	29.76
CodeLlama 6.7B	49.39	49.76	51.68	27.17
StarCoderBase 7.3B	44.37	43.77	48.33	26.48
IRCoder 1.1B	11.71 +0.45	10.76 +0.69	19.93 +1.06	9.63 +0.93
IRCoder 1.3B	26.97 +2.96	27.12 +1.23	35.21 -0.15	18.98 +2.31
IRCoder 3.1B	32.99 -0.88	25.48 -1.70	35.67 +0.75	21.82 +4.03
IRCoder 5.7B	53.94 +1.62	52.64 -0.25	58.77 +0.89	33.79 +4.03
IRCoder 6.7B	50.96 +1.57	51.33 +1.57	55.69 +4.01	28.38 +1.21
IRCoder 7.3B	48.62 +4.25	46.04 +2.27	49.52 +1.19	32.11 +5.63

Table 15: HumanEvalFixDocs `pass@1` comparison between `IRCoder` and its corresponding base models.

Model	C++	Go	Python	Rust
StarCoderBase 1.1B	16.91	14.48	26.67	12.74
DeepSeekCoder 1.3B	35.49	40.78	42.94	25.68
StarCoderBase 3.1B	45.39	44.11	54.67	27.16
DeepSeekCoder 5.7B	64.10	60.44	70.59	49.07
CodeLlama 6.7B	61.32	58.73	61.67	45.44
StarCoderBase 7.3B	55.94	58.36	59.56	47.22
IRCoder 1.1B	18.16 +1.25	13.99 -0.49	26.67 –	13,18 +0.44
IRCoder 1.3B	38.41 +2.92	43.11 +2.31	43.15 +0.23	27.11 +1.43
IRCoder 3.1B	42.06 -3.33	45.17 +0.76	55.06 +0.39	25.74 +1.58
IRCoder 5.7B	68.22 +4.12	63.47 +3.03	73.42 +2.83	56.27 +7.20
IRCoder 6.7B	61.87 +0.55	60.59 +1.86	62.04 +0.37	50.44 +5.00
IRCoder 7.3B	57.35 +1.41	59.33 +0.97	60.11 +0.55	52.71 +5.49

Table 16: HumanEvalFixDocs `pass@10` comparison between `IRCoder` and its corresponding base models.