Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs

FEDERICO CASSANO, Northeastern University, USA
JOHN GOUWAR, Northeastern University, USA
FRANCESCA LUCCHETTI, Northeastern University, USA
CLAIRE SCHLESINGER, Northeastern University, USA
ANDERS FREEMAN, Wellesley College, USA
CAROLYN JANE ANDERSON, Wellesley College, USA
MOLLY Q FELDMAN, Oberlin College, USA
MICHAEL GREENBERG, Stevens Institute of Technology, USA
ABHINAV JANGDA, Microsoft Research, USA
ARJUN GUHA, Northeastern University, USA and Roblox, USA

Over the past few years, Large Language Models of Code (Code LLMs) have started to have a significant impact on programming practice. Code LLMs are also emerging as building blocks for research in programming languages and software engineering. However, the quality of code produced by a Code LLM varies significantly by programming language. Code LLMs produce impressive results on *high-resource programming languages* that are well represented in their training data (e.g., Java, Python, or JavaScript), but struggle with *low-resource languages* that have limited training data available (e.g., OCaml, Racket, and several others).

This paper presents an effective approach for boosting the performance of Code LLMs on low-resource languages using semi-synthetic data. Our approach, called MultiPL-T, generates high-quality datasets for low-resource languages, which can then be used to fine-tune any pretrained Code LLM. MultiPL-T translates training data from high-resource languages into training data for low-resource languages in the following way. 1) We use a Code LLM to synthesize unit tests for commented code from a high-resource source language, filtering out faulty tests and code with low test coverage. 2) We use a Code LLM to translate the code from the high-resource source language to a target low-resource language. This gives us a corpus of candidate training data in the target language, but many of these translations are wrong. 3) We use a lightweight compiler to compile the test cases generated in (1) from the source language to the target language, which allows us to filter our obviously wrong translations. The result is a training corpus in the target low-resource language where all items have been validated with test cases. We apply this approach to generate tens of thousands of new, validated training items for five low-resource languages: Julia, Lua, OCaml, R, and Racket, using Python as the source high-resource language. Furthermore, we use an open Code LLM (StarCoderBase) with open training data (The Stack), which allows us to decontaminate benchmarks, train models without violating licenses, and run experiments that could not otherwise be done.

Authors' Contact Information: Federico Cassano, Northeastern University, Boston, USA, cassano.f@northeastern.edu; John Gouwar, Northeastern University, Boston, USA, gouwar.j@northeastern.edu; Francesca Lucchetti, Northeastern University, Boston, USA, lucchetti.f@northeastern.edu; Claire Schlesinger, Northeastern University, Boston, USA, schlesinger.c@northeastern.edu; Anders Freeman, Wellesley College, Wellesley, USA, af103@wellesley.edu; Carolyn Jane Anderson, Wellesley College, Wellesley, USA, ca101@wellesley.edu; Molly Q Feldman, Oberlin College, Oberlin, USA, mfeldman@oberlin.edu; Michael Greenberg, Stevens Institute of Technology, Hoboken, USA, michael@greenberg.science; Abhinav Jangda, Microsoft Research, Redmond, USA, ajangda@microsoft.com; Arjun Guha, Northeastern University, Northeastern, USA and Roblox, San Mateo, USA, a.guha@northeastern.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/10-ART295
https://doi.org/10.1145/3689735

Using datasets generated with MultiPL-T, we present fine-tuned versions of StarCoderBase and Code Llama for Julia, Lua, OCaml, R, and Racket that outperform other fine-tunes of these base models on the natural language to code task. We also present Racket fine-tunes for two very recent models, DeepSeek Coder and StarCoder2, to show that MultiPL-T continues to outperform other fine-tuning approaches for low-resource languages. The MultiPL-T approach is easy to apply to new languages, and is significantly more efficient and effective than alternatives such as training longer.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Large Language Models trained on Code

ACM Reference Format:
Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 295 (October 2024), 32 pages. https://doi.org/10.1145/3689735

1 Introduction

Large Language Models of Code (Code LLMs) are starting to have a significant impact on both professional programmers and research in programming languages and software engineering. GitHub Copilot is just one of several popular tools powered by Code LLMs [CodeWhisperer 2023; Copilot 2023; TabNine 2023]. Moreover, Code LLMs are also emerging as a building block for research [Bareiß et al. 2022; Chen et al. 2023; First et al. 2023; Joshi et al. 2023; Lemieux et al. 2023; Murali et al. 2023; Nam et al. 2024; Phung et al. 2023; Ross et al. 2023; Schäfer et al. 2024; Xia et al. 2024]. However, the quality of code produced by a Code LLM varies significantly by programming language. Models are most impressive at producing code in high-resource programming languages such as Python, JavaScript, and Java, but struggle in low-resource languages, such as Racket and OCaml [Athiwaratkun et al. 2022; Cassano et al. 2023; Zheng et al. 2023]. This puts programmers who rely on these languages at a disadvantage, since they do not receive the same benefits that Code LLMs can deliver for high-resource languages [Murali et al. 2023; Ziegler et al. 2022].

The key issue is that the performance of Code LLMs depends on the amount of language data available for training. For example, *The Stack*, which is the training set for several contemporary Code LLMs, has 64GB of Python, but only around 1GB of OCaml and 0.5GB of Scheme/Racket [Kocetkov et al. 2023].[1] As Figure 1 shows, the performance of *StarCoderBase*, an open Code LLM trained on The Stack, generally increases as the training data for the language increases.

Our goal in this paper is to investigate methods to further train, or *fine-tune*, pretrained Code LLMs to improve their performance on low-resource languages. The obvious approach is to try to find more data, but for a low-resource language, more data is hard to find by definition. For example, The Stack already includes all permissively licensed code for 358 programming languages from GitHub as of 2022, and GitHub is by far the largest repository of open-source code [Borges et al. 2016].[2] An alternative is to train longer (i.e, for more epochs) on existing data. However, in this paper, we show that *training longer on several low-resource languages is not only inefficient, but can actually hurt performance* (§3.1). Another alternative is to train models on synthetic data, which is data generated by an LLM itself. These synthetic fine-tuning datasets are effective for high-resource programming languages [Luo et al. 2024; Wang et al. 2023]. However, we show that *synthetic data does not work for low-resource languages* for the intuitive reason that LLMs generate poor quality programs in low-resource languages (§3.2).

[1]This is the volume of data that remains after files are deduplication for training [Li et al. 2023].
[2]The Stack deliberately excludes copyleft licenses and unlicensed code.

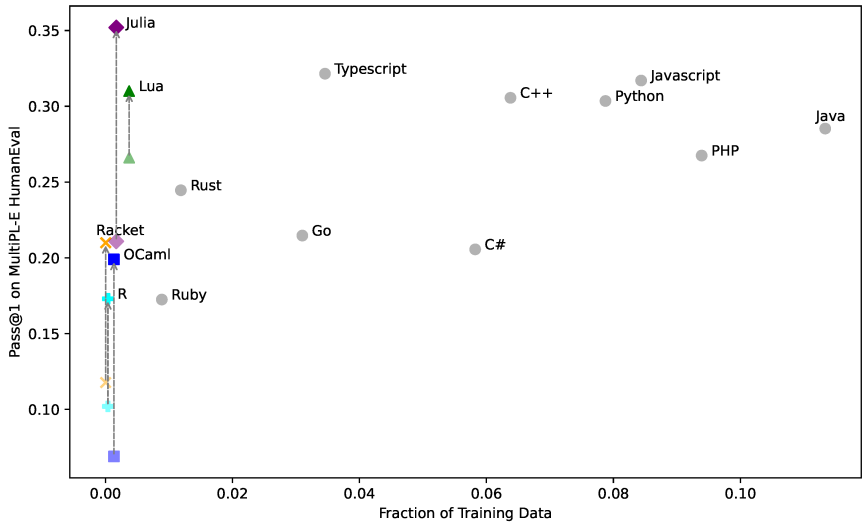


Fig. 1. The performance of StarCoderBase-15B on several languages supported by the MultiPL-E benchmark for Code LLMs, plotted against their proportion of the model's training data. Using MultiPL-T, this paper significantly improves how StarCoderBase-15B performs on several low-resource languages, as shown by the arrows. The bottom of each arrow indicates how the base model performs, and the arrowheads indicate performance after fine-tuning with MultiPL-T. We also show significant improvement on other LLMs (§5).

Our approach. In this paper, we present a new and effective approach for fine-tuning Code LLMs for low-resource programming languages that is based on generating *semi-synthetic training data*. Our approach relies on several key ingredients. 1) The large volume of training data for high-resource programming languages includes a lot of well-documented code; 2) Code LLMs are effective and efficient unit test generators, and we can check that generated tests are valid [Schäfer et al. 2024]; 3) We can compile many unit tests to a low-resource language with a simple compiler [Athiwaratkun et al. 2022; Cassano et al. 2023; Roziere et al. 2021]; 4) Code LLMs can translate code from one language to another, and although these translations may be faulty, we can filter them with the aforementioned tests, retry until tests pass, and engineer a prompt to increase the likelihood of a successful translation. Putting these four ideas together, we develop a pipeline for *transferring training data across multiple programming languages* that we call MultiPL-T.

Figure 2 gives a high-level overview of how MultiPL-T produces high-quality training data for a low-resource programming language. We use an LLM to translate code from a high-resource language (①) to a target low-resource languages (②). However, this translation is unreliable by definition: LLMs are bad at producing code in low-resource programming languages. But, we can leverage the stochasticity of LLM generation to produce several candidate translations for each item and filter out the faulty translations with synthesized test cases (④). We cannot generate tests directly from the low-resource code (since it is likely to be faulty). Instead, we generate and validate tests in the high-resource language (③), and then compile these tests to the low-resource language (④). The composition of these steps gives training data in the low-resource that passes compiled tests that also pass in the high-resource language.

The training corpora of high-resource programming languages (e.g., Python) are enormous, so we use aggressive heuristic filters to build a corpus of "high-quality" functions (①) before attempting any LLM translation. For these functions, we find that the LLM generates tests reliably

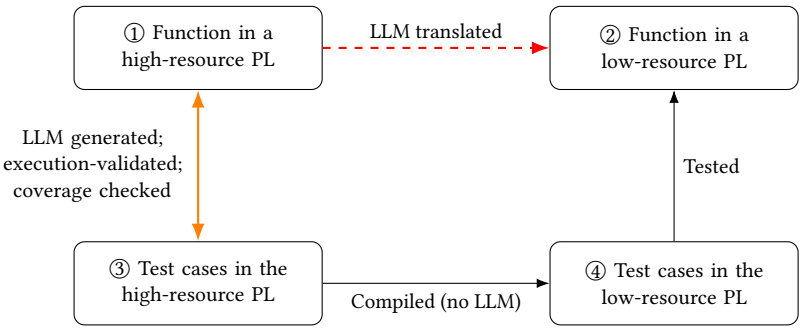


Fig. 2. A high-level overview of how MultiPL-T produces high-quality training data for a low-resource programming language. We use a Code LLM to translate a function from a high resource language (①) to a low-resource language (②). The translated code is likely to be wrong, since LLMs perform poorly on low-resource languages. However, we filter out bad translations as follows. First, we generate unit tests the original code (③). We execute these tests to ensure they succeed and also check for test coverage. Second, we compile these tests to the low-resource language (④). Finally, we filter the low-resource code (②) using the translated tests (④), only keeping those that pass tests.

(③), but some effort is needed to get high test coverage. Translating functions (②) is simply LLM prompting, but requires some lightweight type inference to be effective when the target language is typed. Finally, to compile tests to the low-resource language (④), we build on an existing toolchain ([Cassano et al. 2023]), adding support for OCaml, new prompting formats, and support for an LLM-translation task that is three orders of magnitude larger than what it originally supported.

Using training data generated by MultiPL-T, *we present fine-tuned Code LLMs that achieve state-of-the-art performance on five low-resource languages: Racket, OCaml, Lua, R, and Julia.* We focus primarily on fine-tuning the StarCoder family of Code LLMs [Li et al. 2023]. There are StarCoder models available in a variety of sizes, including a 1B parameter model that is lightweight enough to run on CPUs, and a more capable 15B parameter model, which we use as the test generator and language translator for MultiPL-T. The StarCoder models also have open training data, which allows us to compare MultiPL-T to a baseline of training longer on existing data for low-resource languages. We also present fine-tuned versions of the Code Llama 34B and 70B [Rozière et al. 2023] models, and Racket fine-tunes of the recently released DeepSeek Coder [Guo et al. 2024] and StarCoder2 models [Lozhkov et al. 2024].

Contributions. To summarize, we make the following contributions:

(1) MultiPL-T, an effective approach for generating semi-synthetic data for low-resource programming languages using test-validated translation of high-quality code in high-resource languages.
(2) Efficient fine-tuning datasets for Julia, Lua, OCaml, R, and Racket, comprising tens of thousands of documented and tested functions generated with StarCoderBase-15B.
(3) A dataset of 133,168 Python functions extracted from the Stack, where every function has natural language documentation and a validated set of generated tests with high coverage. This dataset could be used to generate fine-tuning sets for other programming languages.
(4) Fine-tuned versions of StarCoderBase 1B and 15B for Julia, Lua, OCaml, R, and Racket. For these languages, these fine-tuned models outperform prior fine-tunes of StarCoderBase on the natural language to code task.

(5) Fine-tuned versions of Code Llama 34B and 70B for Julia, Lua, OCaml, R, and Racket. For these languages, these fine-tuned models outperform prior fine-tunes of Code Llama. More significantly, this is an uncommon result where *data generated from a smaller model (StarCoderBase-15B) improves the performance of larger and better models (Code Llama 34B and 70B).*
(6) Fine-tuned versions of StarCoder2-15B and DeepSeek Coder 33B for Racket that also outperform other fine-tuned models. These are two very recently released models.
(7) A thorough evaluation that includes a) a comparison of MultiPL-T to the baseline of training further on existing data, b) an evaluation of the fine-tuning efficiency with MultiPL-T, c) results on prior multi-language benchmarks [Cassano et al. 2023], d) a new multi-language benchmark designed to exercise in-context learning, e) an evaluation of how generated code adheres to the common Racket programming style, f) the impact of data deduplication, and g) the impact of fine-tuning on the Python source data.

2 Background

In this section, we give a high-level overview of how Code LLMs are trained and evaluated. We use StarCoder as the example, since it is the model that we use for most of our work.

2.1 Training and Fine-Tuning Large Language Models of Code

A *large language model (LLM)* is a neural network trained on hundreds of gigabytes or even terabytes of data. Code LLMs are trained on source code (and often natural language documents too), which allows them to generate code from comments, comments from code, more code from code, and so on. LLM training takes significant resources: StarCoderBase was trained on approximately 800GB of code, which took three weeks on a cluster of 512 NVIDIA A100 GPUs.

The only way to build a training set of this scale is to scrape public repositories of code. There are a handful of public training sets that are based on GitHub [Felipe Hoffa 2016; Xu et al. 2022], and *The Stack* [Kocetkov et al. 2023] is a recent example. The Stack v1.2 has 3TB of permissively licensed source code for 358 programming languages. It was constructed in 2022, and has since been used to train several Code LLMs [Allal et al. 2023; Nijkamp et al. 2023; Replit 2023], including StarCoderBase. Specifically, StarCoderBase was trained on a filtered subset of The Stack consisting of 86 programming languages.

The StarCoder model family. StarCoder is a family of models that are available at several sizes [Li et al. 2023]. The largest and most capable model in the family is called StarCoderBase, which has 15B parameters. There are smaller versions of StarCoderBase that were trained on exactly the same data. To make use of limited GPU resources, we use the smallest model, StarCoderBase-1B, for most experiments in this paper. However, we also show that our results generalize to StarCoderBase-15B. There is also a model in the StarCoder family that is just named StarCoder: it is StarCoderBase-15B specialized to excel at Python.[3] This paper uses StarCoderBase-15B for translations to low-resource languages, and StarCoder-15B for Python test generation.

The Code Llama model family. The Code Llama [Rozière et al. 2023] family of models were recently released and perform better than the StarCoder models on common benchmarks. While the authors state that the training data comes from publicly accessible datasets, they do not disclose the specific datasets used, preventing us from conducting the exhaustive evaluation on Code Llama that we do with StarCoder and its training data. Moreover, the Llama license forbids using model outputs to train non-Llama models, which is why we use StarCoder for data generation. However, we train and evaluate the larger Code Llama models (34B and 70B).

[3]StarCoder fine-tunes StarCoderBase-15B on two more epochs of Python data from The Stack.

```python
def vowels_count(s):
    """Write a function vowels_count which takes a string representing a word as
    input and returns the number of vowels in the string.  Vowels in this case are
    'a', 'e', 'i', 'o', 'u'. Here, 'y' is also a vowel, but only when it is at the
    end of the given word.
    """
```

(a) Python prompt.

```ocaml
(* Write a function vowels_count which takes a string representing a word as
   input and returns the number of vowels in the string.  Vowels in this case are
   'a', 'e', 'i', 'o', 'u'. Here, 'y' is also a vowel, but only when it is at the
   end of the given word.  *)
let vowels_count (s : str) : int =
```

(b) OCaml prompt.

Fig. 3. An example prompt from a HumanEval problem and its translation to OCaml, with our extension to MultiPL-E. Not shown are doctests and hidden test cases, which are also translated to OCaml. This particular problem is hard for many LLMs because it alters the strong prior on what vowels are, by saying that *y is a vowel when it is the last letter in a word.*

Fine-tuning. After training, a model can be further trained, or *fine-tuned*, with significantly fewer resources. For example, there are several fine-tuned versions of StarCoderBase that were trained with a few days of GPU time on a modest amount of data (e.g., [Luo et al. 2024; Muennighoff et al. 2024]). Most fine-tuned versions of StarCoderBase are designed to make the model even better at high-resource languages, such as Python. In contrast, this paper presents fine-tuned versions of StarCoderBase that are significantly better at several low-resource languages.

It is common to *distill* data from a larger model (e.g., GPT-4), to fine-tune a smaller model [Gunasekar et al. 2023; Luo et al. 2024; Wei et al. 2024]. However, we show that MultiPL-T can do the reverse: we use data generated from StarCoderBase-15B to fine-tune larger models, CodeLlama-34B and 70B. This is a form of weak-to-strong supervision [Burns et al. 2024], where a smaller model is used to generate data for training a larger model, showing the scalability of MultiPL-T.

2.2 Code LLM Tasks and Benchmarking Code LLMs

A Code LLM can be prompted to perform a wide variety of tasks, including code translation (e.g., [Pan et al. 2024]), test generation (e.g., [Schäfer et al. 2024]), code mutation (e.g., [Xia et al. 2024]), code editing (e.g., [Cassano et al. 2024]), and much more. This article focuses on the *natural language to code task* (e.g., [Heidorn 1974]) for low-resource programming languages. Making Code LLMs better on other tasks is beyond the scope of this article.

Most Code LLM benchmarks for the natural language to code task, including those we use in this paper, follow the format introduced by the Codex "HumanEval" benchmark [Chen et al. 2021]. Every benchmark problem has two parts: 1) a prompt for the LLM that has a function signature and a comment, and 2) a suite of test cases that are not given to the LLM. Thus each problem is run in two steps: 1) the LLM generates a function from the prompt, and 2) the generated function is then tested with the hidden tests, and all tests must pass for the generated code to be considered correct.

The HumanEval benchmark has 164 problems for Python. However, it is possible to mechanically translate most of these problems to other programming languages (Figure 3). Translating comments and function signatures is straightforward, but some care is needed to introduce types for typed target languages. Translating test cases turns out to be easy as well, since almost all HumanEval

test cases are of the form $f(v_{in}) = v_{out}$, where v_{in} and v_{out} are first-order values. This is the approach that is taken by MultiPL-E and similar tools [Athiwaratkun et al. 2022; Cassano et al. 2023; Orlanski et al. 2023] to build polyglot benchmarks for Code LLMs. This paper utilizes MultiPL-E, which is the only benchmark to date that supports Racket, and we extend it to support OCaml for this paper.

Code LLMs appear to produce higher-quality code when their output is sampled [Chen et al. 2021]. Since sampling introduces non-determinism, we must evaluate their output by generating several samples from the same prompt. The most widely used metric for Code LLM performance is *pass@k*, which is the likelihood that the LLM produces a program that passes all hidden tests at least once from k attempts. Pass@k must be estimated from $n >> k$ samples. When $k = 1$ and there are c successes, pass@1 is the same as the pass rate (c/n). We use pass@1 as the metric for all our benchmarking experiments, which is common practice. Intuitively, pass@1 measures the ability of the Code LLM to generate a correct solution in a single attempt.

2.3 Why StarCoder?

In the rest of this paper, the majority of the work that we present uses the StarCoder family of Code LLMs for the following reasons.

(1) At the time that we started this work, StarCoder was the best-performing, permissively licensed, open Code LLM available.
(2) The StarCoder family includes a fairly small 1B parameter model, which is amenable to experiments on a budget.
(3) Although there were better closed-sourced LLMs when we started, they (a) did not support fine-tuning, (b) were significantly more expensive to use at the scale of our work, or (c) prohibited using their outputs to fine-tune other LLMs [Anthropic 2023b; Google 2023; OpenAI 2023b].
(4) StarCoder remains the only Code LLM with open training data, which we use to correlate model performance with training set size (Figure 1), evaluate further training (§3.1), and to check that our benchmarks are not in the training data. It would not have been possible to do this work with any other Code LLM.

We do fine-tune several newer Code LLMs with StarCoder-generated data to show that our approach remains useful (§5).

3 Alternatives to MultiPL-T

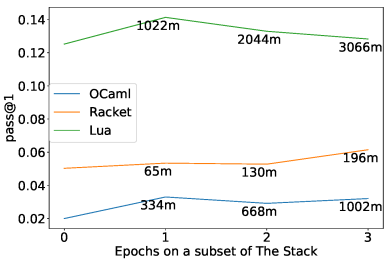
Before we present the MultiPL-T approach, we consider two simpler alternatives.

3.1 Further Training on Natural Data

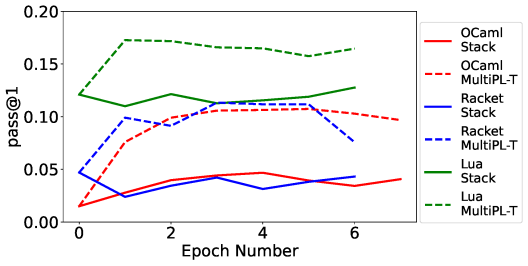
The simplest way to boost the performance of a Code LLM on a programming language is to train it further on *natural data*, which is code written by human programmers rather than code generated by other means (e.g., an LLM). This was the approach taken to create StarCoder from StarCoderBase. The latter is the base model, and the former is fine-tuned on roughly two additional epochs[4] of the Python subset of The Stack. Although this approach is effective for high-resource languages, we now show that it does not work for several low-resource languages (Figure 4).

In Figure 4a, we fine-tune three versions of StarCoderBase-1B on three more epochs of Lua, OCaml, and Racket each. This data is from The Stack. However, The Racket subset of The Stack is poor quality, so we use the Scheme subset instead.[5] The Stack has an order of magnitude more Lua

[4]An *epoch* in machine learning refers to one complete pass through the entire training dataset.
[5]The Racket subset accidentally omits the `.rkt` file extension and largely contains Racket documentation (in Scribble). Since Racket is descendant from Scheme, the Scheme subset is a more reasonable fine-tuning set.



(a) Fine-tuning on the complete language specific subsets of The Stack.



(b) Fine-tuning on subsets that are approximately the same size as the MultiPL-T fine-tuning datasets.

Fig. 4. We fine-tune StarCoderBase-1B on several epochs of language-specific data of The Stack and measure performance with MultiPL-E. In Figure 4a, we train on all data from The Stack for each language. These datasets vary in size (the labels measure their size in tokens). In Figure 4b we sample each dataset to be approximately the same size as the MultiPL-T datasets. Both approaches that use data from The Stack barely improve performance, and can even hurt performance. In contrast, fine-tuning on MultiPL-T (dashed lines) shows significant improvement.

than OCaml and Racket. Moreover, even the Racket and OCaml data in The Stack is significantly larger than the fine-tuning datasets we will develop with MultiPL-T. Therefore, these experiments are not directly comparable to each other, since they train on wildly varying amounts of data. Nevertheless, we get poor results for all: the performance of these fine-tuned models barely increases for Racket and OCaml and even decreases for Lua.

In Figure 4b, we do another experiment with The Stack that lends itself to a direct comparison with MultiPL-T. We randomly sample data from The Stack to get approximately the same volume of data that we generate with MultiPL-T. Thus, fine-tuning on these datasets will use similar computing resources as fine-tuning a model with MultiPL-T data. We use this data to fine-tune three versions on StarCoderBase-1B for six epochs, and evaluate the models at each epoch. We still get poor results with The Stack: Lua and OCaml performance barely increases and Racket performance decreases. In contrast, fine-tuning with MultiPL-T will show significant gains.

3.2 Self-Instruction for Low-Resource Programming Languages

An alternative to fine-tuning on natural data is to fine-tune on LLM-generated data [Luo et al. 2024; Wang et al. 2023]. The usual approach is to hand-select a seed dataset of programs, prompt the model with each seed to generate more programs, and iterate until a large enough dataset has been collected. This type of approach has been used successfully to generate training data for Code LLMs in high-resource languages [Luo et al. 2024]. However, it should be obvious that *these approaches presuppose that the LLM is good at generating reasonably correct and high-quality programs.* We are interested in languages that the model is bad at, so it should not be surprising that self-instruction does not work.

Illustrating self-instruction. To illustrate how self-instruction goes wrong with low-resource languages, we use StarCoderBase-15B to generate functions in Racket, mimicking the first step of self-instruction. We prompt the model with five hand-written examples (included in the supplementary material) and have it generate five more functions (Figure 5). We find that four of the five model-generated programs have bugs. This is a much higher error rate than what is evident from self-instruct datasets for high-resource languages [Chaudhary 2023; Muennighoff et al. 2024].

```
;; sum-digits-of-number: Number -> Number
;; Sums the digits of a number
(define (sum-digits-of-number num)
  (foldr +
         0
         (map string->number (string-split (number->string num) "") )))
```

(a) Splitting on the empty string produces empty strings in the list, which cannot be converted to a number.

```
;; add-rev-nums: [List-of Numbers] -> Number
;; Adds the reverse of all the numbers in a list
(define (add-rev-nums lon)
  (foldr +
         0
         (map (lambda (n)
                (string->number (list->string
                  (reverse (string->list (number->string n))))))
              lon)))
```

(b) Working function.

```
;; is-palindrome-sum?: Number -> Boolean
;; Checks if a number is a palindrome sum of its digits
(define (is-palindrome-sum? num)
  (is-palindrome? (sum-remainders
    (map char->integer (string->list (number->string num))) 10)))
```

(c) char->integer returns the ASCII code of a character, and does not parse it as a digit.

```
;; sum-palindromes: [List-of Numbers] -> Number
;; Sums the palindrome numbers in the list
(define (sum-palindromes lon)
  (foldr + 0 (filter is-palindrome? lon)))
```

(d) Utilizes what looks like a helper function, is-palindrome?, which is not defined.

```
;; is-prime?: Number -> Boolean
;; Checks if a number is prime
(define (is-prime? num)
  (cond
    ((= num 2) #t)
    ((= num 1) #f)
    (else (zero? (remainder (add-odds (repeat "2" (/ num 2))) num)))))
```

(e) The highlighted code calls two helper functions that are not defined.

Fig. 5. Faulty Racket code generated by StarCoderBase-15B when seeded with five hand-written examples.

A self-instruction experiment. In the supplementary material we present results from an experiment where we self-instruct StarCoderBase-15B on Racket and get the expected poor results.

Self-instruction and training further on existing public data do not help Code LLMs perform better on low-resource programming languages. Thus we now turn to the MultiPL-T approach.

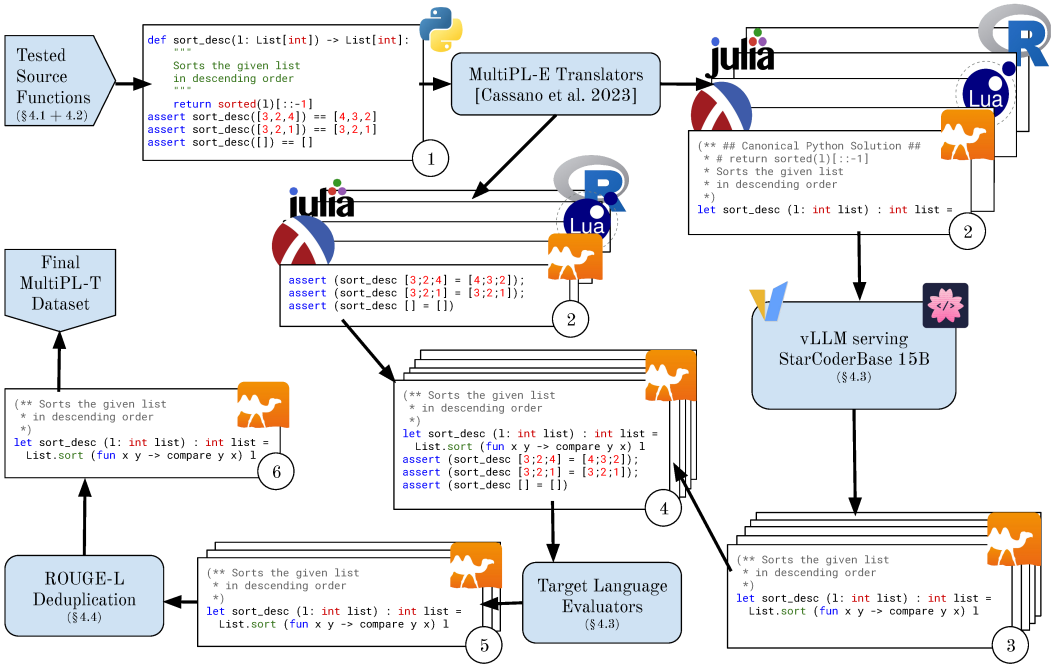


Fig. 6. The MultiPL-T pipeline for generating semi-synthetic data. Starting with a tested Python function ①, we compile the header and test cases into each target language ②. We prompt the model with the target-language header and comment to generate 50-100 candidate translations ③ (varies by language). We append the compiled test cases to the candidate translations ④ and evaluate with the target language runtime. We deduplicate all programs that pass all test cases ⑤ to build the fine-tuning dataset ⑥.

4 Our Approach

We now present the MultiPL-T approach to generating high-quality, semi-synthetic data for low-resource languages. Figure 6 depicts the MultiPL-T system, which has several stages. **1)** Given a training dataset (The Stack), we filter data from a high-resource language (Python) to select code that is amenable to automatic test generation and translation. The Stack has 60GB of Python, and translation and test generation are expensive, so we filter quite aggressively. We only select individual Python functions that have docstrings and pass a heuristic type-checker (§4.1). **2)** Given the filtered dataset, we use a Code LLM (StarCoder-15B) to generate test suites for each function. We validate the generated tests for correctness and code coverage, and find that the Code LLM can be used as a capable test generator for our purposes (§4.2). **3)** We translate each Python function to a target language L, by prompting the Code LLM to translate code. This translation may go wrong, especially because the Code LLM performs poorly on the low-resource target language. **4)** We filter the L functions (from Step 1) to only select those that pass test cases. To do so, we compile the Python test cases (from Step 2) to the language L, using the Python-to-L test case compiler from MultiPL-E. The test case compiler is a traditional compiler that does not suffer from LLM hallucinations: if it cannot compile a test case, it signals an error, and we discard the training item if too many test cases fail to compile (§4.3). The final result is thus a dataset of novel training items for the language L, which may be used to fine-tune any LLM. In §5, we discuss how we use this

Table 1. Size of the Python source dataset after each filtering step.

Filtering Step	#Functions
All functions	22,311,478
With docstrings	5,359,051
Typechecked and returns value	459,280
No TODOs and no benchmark solution	432,361
Test generation	157,767
90% line coverage from tests	133,168

data to fine-tune and evaluate several models for five different low-resource languages. The rest of this section describes the above steps in depth.

4.1 Filtering Data from a High-Resource Language for Translation and Test Generation

The first step in MultiPL-T is to filter code from a high-resource language to serve as the translation source for our semi-synthetic data. We use Python because it has the highest representation in The Stack and because MultiPL-E can compile Python function signatures and test cases to several low-resource languages. However, our approach could easily be adapted to work with other high-resource languages.

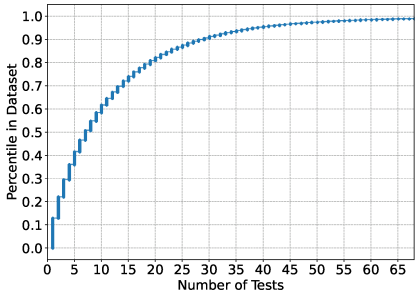
Filtering Python Functions Before Translation. The Stack has 22 million Python functions (Table 1). However, not all of these are amenable to translation and test-based validation with MultiPL-T. One could naively try to translate and generate tests for all 22M functions. However, since doing so requires GPUs, it would be prohibitively expensive. Instead, we aggressively filter the 22M functions down to ~400,000 functions using the following steps:

(1) We exclude Python functions that do not have a docstring or use non-ASCII characters. One could generalize to include functions that have an associated comment. However, we still end up with over 5M candidate functions with this simple filter.
(2) We use the Pyright [Pyright 2023] Python checker to validate that each function returns a value, uses only the Python standard library, and is thus likely type-correct. Pyright uses heuristics and makes no attempt at being sound. This does not impact MultiPL-T, since we merely use typeability as a heuristic for code quality. This narrows the 5M functions to approximately 460,000 functions.
(3) We exclude Python functions that have comments suggesting the implementation is incomplete (e.g. "TODO"). It turns out that a fair amount of code on The Stack is incomplete; these functions are not likely to be useful training data. To avoid data contamination, we filter out functions whose prompt or solution appears in widely-used Code LLM benchmarks by finding exact matches of the prompts [Austin et al. 2021; Chen et al. 2021].

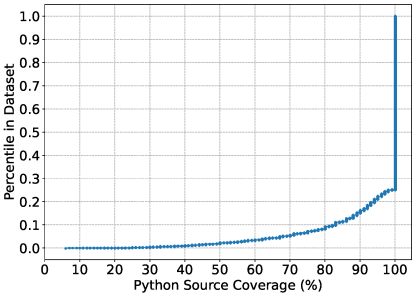
The final dataset contains 432,361 Python functions. With this narrower set of functions, we move on to the next steps that require GPUs.

4.2 Generating Python Unit Tests

The next step in MultiPL-T is to generate unit tests for each Python function. We will then compile these unit tests to target low-resource languages using the test case translators from MultiPL-E. We generate Python unit tests using the following steps.



(a) The y-axis shows the fraction of test suites with fewer than x tests.



(b) The y-axis shows the fraction of test suites with less than $x\%$ coverage.

Fig. 7. Python test suite sizes and coverage distribution among the unfiltered test generation dataset.

```
def count(word):
    """Count the number of X's in a word."""
    count = 0
    for letter in word:
        if letter == "Y":
            count += 1
    return count

assert count("XXXXXX") == 6  # First completion
assert count("YYY") == 3 # Second completion
```

Fig. 8. An example prompt to generate a unit test where the code and comment are inconsistent: the comment counts X's, but the code counts Y's. Using a Code LLM to generate tests helps expose the inconsistency: we get some tests that are based on the comment and others based on the code. We show two completions we got from StarCoderBase-15B in two successive queries. The "`assert count(`" is part of the prompt, but we show it with the completion for clarity.

(1) We prompt the Code LLM to generate assertions given the Python function, taking care to generate several independent test suites to get high coverage, and executing the tests to ensure that they pass.
(2) For each function, we measure the coverage of the aggregated test suites, discarding functions with less than 90% line coverage.
(3) We use the tests to infer basic types for the Python function, which is necessary to compute the tests to certain target languages.

Test generation. Instead of using a traditional test generator that synthesizes tests from code (e.g., [Lukasczyk et al. 2020]), we use a Code LLM to generate test cases by prompting the model to produce an assertion. The Code LLM conditions on the source code text and thus serves as a weak detector for inconsistencies between code and comments. For example, Figure 8 shows a function that would be a bad training item: the code counts Y's, but has a comment that says it counts X's. When we generate several test cases independently, we end up with tests for both X and Y, which lowers the ratio of passing tests. Conditioning on the text makes it less likely that inconsistent tests will be generated, decreasing the likelihood that functions will be filtered out of the training set due to low coverage (as described below).

```python
def rep_or_hello(x):
    """Repeat string twice, or
    produce Hello if no string given
    """
    if x is None:
        return "Hello"
    else:
        return x + x

assert rep_or_hello(None) == "Hello"
assert rep_or_hello("y") = "yy"
```

(a) Python function and generated tests.

```ocaml
(* Repeat string twice, or produce
   Hello if no string given. *)
let rep_or_hello (x : string option)
  : string =

assert rep_or_hello None == "Hello";;
assert rep_or_hello (Some "y") = "yy"
```

(b) OCaml prompt and compiled tests.

Fig. 9. An example that demonstrates the need for Python type inference when translating to OCaml. Figure 9a shows a Python function with generated test cases. We then compile both the Python function signature and the test cases to OCaml (Figure 9b). However, to know that the Python string `"y"` must compile to `Some "y"` in OCaml, we need to infer Python types.

We prompt StarCoder-15B to generate five independent test suites for each function with high temperature[6] (0.8) to get a diverse set of candidate tests. We parse each generated test suite and extract all test cases that are suitable for translation using MultiPL-E. We take the set of matching test cases and run each test in isolation in a container to verify that it passes, discarding any that fail. If no correct tests are generated, we discard the function. The result is a set of nearly 160,000 Python functions with at least one passing test case. Figure 7a shows the distribution of test suite sizes. The median number of test cases per function is 7, and the mean is 12.1. Note that we do not filter any tests after validated, so a function may be more tests that strictly necessary.

Filtering on test coverage. Given the dataset of Python functions with docstrings and test suites, our next step is to filter out functions with low test coverage. We use *line coverage* as the coverage metric and exclude all functions with less than 90% line coverage. The result is a dataset of 133,668 functions with 90% line coverage from tests.

Since we start with nearly 160,000 functions, this implies that most of the generated test suites that work have high line coverage. In fact, most functions have 100% line coverage (Figure 7b). This stringent criterion ensures that the functions in our final set are not just correct but also comprehensively tested, reinforcing the reliability of our dataset. In the filtered Python dataset, the average function has 10.3 lines of code (SD=10.7) and on average 3.6 branches (SD=4.1).

Type inference. The steps described above are sufficient to generate data for an untyped low-resource language (e.g., Racket or Lua). However, for a typed target (e.g., OCaml or Julia), we also need to infer types for two reasons.[7] Consider the case where we target OCaml. First, we rely on the LLM to generate an OCaml function body, given only a comment and the function header (`let f x =`). Without type annotations, the only signal about the desired type of `f` are the identifier names and the comment. With type annotations, the LLM is far more likely to produce a function with the expected type. Second, we need to infer Python types to compile test cases, which we illustrate in Figure 9. In this example, we have a Python function that consumes an optional string. We have two test cases, one that applies the function to `None` and the other that applies the function to a string. When we compile these tests to OCaml, we have to transform the Python argument type

[6] *Temperature* is a parameter that controls the randomness of the next-token predictions. A higher temperature results in more varied (less predictable) output, while a lower temperature produces more conservative (more predictable) results.
[7] Julia can be used with or without types, and our dataset has both typed and untyped examples.

`Union[str,None]` to the OCaml type `string option`. We can only do this type transformation after we have inferred the Python type.

Our approach to type inference is simple: we deduce types based on test cases, ignoring the function body. We extract the instance type of each argument and expected return value in each test, computing the union type between the types at the same position among tests. For example, if the test cases apply `foo(1)` and `foo(None)`, we infer `Union[int, None]` as the type of `foo`'s argument. Moreover, we simplify `Union[T, None]` to the more canonical `Optional[T]`. For example, `Union[int, int, None]` would be then simplified to `Optional[int]`. This approach to type inference can only fail if the code is non-deterministic, which does not occur in our dataset.

Following the steps above produces two datasets of Python functions—one with and one without type annotations—where every function has a docstring and a suite of unit tests that achieve high coverage. These datasets can be used to generate training data for any low-resource language.

4.3 Translation from a High-Resource to a Low-Resource Language

Given a dataset of commented Python functions with high coverage unit test suites, our next goal is to translate the dataset from Python to a target language L and use tests to validate the translation.

Translation with a Code LLM and MultiPL-E. We use a modified version of MultiPL-E [Cassano et al. 2023] to translate each Python function into an equivalent function in the target low-resource language. We construct a MultiPL-E prompt with the following three parts:

(1) *Docstring:* We turn the Python docstring into a comment in the target language. The MultiPL-E toolchain translates between different comment formats and also alters common type names in natural language using simple rules. For example, when translating from Python to OCaml, we turn "dictionary" into "association list".
(2) *Function signature:* We turn the Python function signature into a function signature in the target language. This step may involve translating types from Python into the target language if they are required.
(3) *Original Python code:* Finally, we add a comment (in the target language) that contains the original Python code. We find that this additional information increases the chance that the model generates a correct translation (§5.4.2).

Figure 6 highlights an example prompt and test suite for translating a descending sort function written in Python to OCaml in the programs labeled 1 and 2. MultiPL-E translates comments written in Python to OCaml and translates each test case and the function signature from Python to OCaml. The original Python code is added as part of the comment.

Given this prompt, we use StarCoderBase-15B to generate translations of each problem in our Python dataset. For all of our languages, we generate 50 translations[8] with high temperature (0.8), to encourage the Code LLM to produce a more diverse set of candidate solutions [Chen et al. 2021].

Checking translations with compiled tests. A Code LLM is quite likely to produce faulty translations; in our case, this is even more likely, since we specifically target languages on which the Code LLM performs poorly. We address this problem by translating test cases from Python to the target language using a simple, recursive compiler. MultiPL-E has a suite of compilers that translate simple Python assertions into assertions in 20+ other programming languages. The compilers support assertions that are simple equalities between first-order values, specifically atomic Python data and collections (lists, tuples, and dictionaries). We use these compilers to translate tests from Python to

[8]For OCaml, we generated 100 translations per problem, as the base pass rate was significantly lower than other languages.

Algorithm 1 Parallelized ROUGE-L deduplication procedure.

```
procedure Deduplicate(I, t, groupSize, rounds)        ▷ Deduplicate items (I) with threshold t
    G ← GroupByPrompt(I)                               ▷ Group items by prompt (comment)
    deduped ← DedupGroups(G,t)                         ▷ Deduplicate for each prompt
    for i ← 0 to rounds do
        G ← RandomGroups(deduped, groupSize)           ▷ Randomly group items
        deduped ← DedupGroups(G, t)                    ▷ Global deduplication
    end for
    return deduped
end procedure
procedure DedupGroups(G, t) ▷ Deduplicates item within each group in the list of groups (G).
    ℓ ← []
    for g ← G in parallel do
        keep ← [true | x ∈ g]                          ▷ Initially keep every item in the group
        for i ← 0 to |g| −1 do
            for j ← i + 1 to |g| do
                if i = j or not keep[j] then
                    continue
                end if
                a, b ←RemComments(g[i]), RemComments(g[j])
                if F-Measure(ROUGE-L(a, b)) > t then
                    keep[j] ← false    ▷ Remove an item if it is similar to others in the group.
                end if
            end for
        end for
        ℓ ← ℓ + [g[i] | keep[i]]
    end for
    return ℓ                                           ▷ A list of items (ungrouped)
end procedure
```

each target language, removing test cases that MultiPL-E does not support. If we are left with zero test cases, we discard the function entirely.

Given the set of 50 generated translations for each function, we select only those solutions that pass all tests. This may include selecting several solutions to the same problem, which is beneficial to the model in terms of learning diverse code styles.

4.4 Deduplication

We define a set of solutions as *diverse* when they differ in form. For instance, two solutions that perform identically on a test suite but differ in their implementation–one using a recursive function and the other a loop–are considered diverse. Ensuring diversity among generated and verified solutions is crucial for teaching fine-tuned models a variety of syntactic and semantic features of the target programming language. Furthermore, redundant or similar solutions may diminish the effectiveness of a dataset [Lee et al. 2022].

Just resampling with high temperature does not guarantee diverse solutions: the LLM may still produce nearly identical solutions (e.g., with a few variables renamed). To address this, we employ a deduplication algorithm based on ROUGE-L [Lin and Och 2004]. ROUGE-L is a metric of text summarization quality, and quantifies the syntactic overlap between two pieces of text with a score

ranging between 0 and 1 where 1 indicates the highest similarity. We use 0.6 as the similarity threshold for discarding duplicates. Before comparing a pair of solutions, we remove all comments from the code, as it may introduce noise in the deduplication process.

Running ROUGE-L on all pairs of items is prohibitively expensive. Instead, we use a heuristic that is amenable to parallelization (Algorithm 1). We apply ROUGE-L to deduplication items in fixed-size groups (we use size 200). Initially, the groups are solutions to the same prompt, as this group is likely to have many duplicates. We then randomly regroup items and run grouped deduplication again. The number of rounds of deduplication is proportional to the total number of items: more rounds increase the likelihood that duplicates will be removed. Ultimately, this results in a set of diverse, accurate, and semantically equivalent solutions for each prompt.

5 Evaluation

In this section, we use MultiPL-T to fine-tune a variety of different models of varying sizes. We demonstrate state-of-the-art results on standard benchmarks for the natural language to code task (§5.2), novel qualitative and quantitative evaluation (§5.3), and ablations showing the significance of various design decisions in MultiPL-T(§5.4).

5.1 Experimental Setup and Implementation

Training hyperparameters. We fine-tune all models with a sequence length of 2,048 tokens. StarCoderBase-1B is fine-tuned for seven epochs with batch size 8, with learning rate 3×10^{-5}, 10 steps warmup, and cosine learning rate decay. For StarCoderBase-15B, CodeLlama-34B, and CodeLlama-70B we make these configuration changes: ten epochs, batch size 32, and learning rate 2×10^{-5}.

Estimated computing resources used. The work for this article was done over several months using V100 (32GB), A100 (80GB), and H100 (80GB) NVIDIA GPUs, as they were available across several clusters and servers. We estimate that we spent approximately 550 days of A100 (80GB) GPU time with the following breakdown:

- Training: ~3,444 hours fine-tuning several versions of CodeLlama-70B, CodeLlama-34B, StarCoderBase-15B, and StarCoderBase-1B. These include the models presented in this section and in §3.
- Evaluation: ~310 hours running benchmarks, which include MultiPL-E and the new in-context learning benchmark (§5.3.1).
- Dataset Generation: ~9,984 hours generating the MultiPL-T training sets. This was the most significant use of resources but is reusable for future model development. Using models larger than StarCoderBase-15B for generation would take up significantly more resources.

Training and evaluation tools. We experimented with several technologies during development. The final MultiPL-T pipeline uses vLLM [Kwon et al. 2023] for inference, DeepSpeed ZeRO to fine-tune larger models [Rajbhandari et al. 2020], Transformers [Wolf et al. 2020], and MultiPL-E [Cassano et al. 2023] with several modifications, such as supporting OCaml.

5.2 Evaluation on Standard Benchmarks

Most Code LLM benchmarks target high resource programming languages, mostly Python. To evaluate our fine-tuned LLMs for low-resource languages, we use and extend MultiPL-E, which is the benchmark that was used to evaluate StarCoder, Code Llama, Stability.ai's StableCode, and several other Code LLMs. As described in §2.2, MultiPL-E is a benchmark for the natural language to code task. We report MultiPL-E results in the standard way: using the pass@1 metric, which

Table 2. MultiPL-E pass@1 scores for 1B, 15B, 34B, and 70B parameter models before and after fine-tuned on MultiPL-T data. The fine-tuned models show significant improvement.

Language	StarCoderBase-1B		StarCoderBase-15B		CodeLlama-34B		CodeLlama-70B	
	Base	Fine-tuned	Base	Fine-tuned	Base	Fine-tuned	Base	Fine-tuned
OCaml	1.5	9.7	6.9	19.9	18.3	27.4	23.2	29.3
Racket	4.7	11.3	11.8	21.0	15.9	29.1	21.9	33.1
R	5.4	8.9	10.2	17.3	18.2	25.5	23.0	28.5
Julia	11.3	15.6	21.1	35.2	31.8	43.5	41.9	44.5
Lua	12.1	17.3	26.6	31.0	38.1	43.9	41.7	44.9

is simply the *mean pass rate* on the benchmark, where a passing completion must successfully execute all tests.

Choosing baselines and comparisons. Before presenting results, we discuss how we believe fine-tuned models should be evaluated.

It is well known that larger models perform better because they can learn more complex patterns [Hoffmann et al. 2022]. Moreover, there is a recent trend of building smaller models that outperform larger models by training them on far more data [de Vries 2023]. For example, StarCoder2-15B is the same size as StarCoderBase-15B, but is trained on 400% more data and performs 50% better on MultiPL-E [Lozhkov et al. 2024].

Therefore, it trivial to achieve an impressive fine-tuning result by simply starting with a newer, better model. We argue that the right way to evaluate a fine-tuning approach is to do all of the following:

(1) Compare the new fine-tuned model to the base model. This allows us to ask, *how does a fine-tuning approach improve baseline performance for a particular model?*
(2) Compare the new fine-tuned model to other fine-tunes of the same model. Fine-tuning on a large dataset is likely to do better than fine-tuning on a smaller dataset. But, this allows us to ask, *How efficient is one fine-tuning approach compared to another?*
(3) Fine-tune a variety of base models. This allows us to ask, *does the fine-tuning approach generalize to different model sizes and families?*

Thus a fine-tuning approach should improve baseline performance on several different model families and across several model sizes, and we will show that this holds for MultiPL-T.

5.2.1 Fine-Tuning StarCoderBase and Code Llama. Fine-tuning StarCoderBase and Code Llama with MultiPL-T-generated data on our target languages improves performance on MultiPL-E across the board (Table 2). For each model, we fine-tune a separate model for Julia, Lua, OCaml, R, and Racket. We checkpoint and evaluate the models at each epoch and report the peak performance. It is *not* a goal of this paper to maximize MultiPL-E scores. In fact, the next section suggests that it would be easy to improve some of these scores by either training longer on the data we already have or by letting MultiPL-T generate more data. Beyond the generally improved performance, we draw several other conclusions:

(1) Racket and OCaml, which are the lowest-resource languages that we evaluate, show the largest relative gains. For example, the fine-tuned versions of both models have more than double the score of their base models, with particularly large gains for OCaml. Even the fine-tuned 1B models perform comparably to the base 15B models at producing correct solutions in these languages.

Table 3. MultiPL-E pass@1 scores for two recently released models on Racket. The fine-tuned models are significantly better, even though the MultiPL-T datasets are from an older model.

Language	StarCoder2-15B		DeepSeekCoder-Base-33B	
	Base	Fine-tuned	Base	Fine-tuned
Racket	22.41	29.7	23.3	36.6

(2) Lua obtains relative gains of 42% for 1B and 17% for 15B. However, these gains are significant and put the fine-tuned models' Lua performance on par with the base models' performance on the highest-resource languages. For example, StarCoderBase-15B achieves 30.6 on MultiPL-Python, and our fine-tuned model achieves 31.0 on MultiPL-Lua.
(3) Julia also shows a significant relative gain of 67% for 15B, achieving a score on MultiPL-Lua that exceeds the base model's MultiPL-Python scores.
(4) We also fine-tune and evaluate CodeLlama-34B and CodeLlama-70B, where we see significant improvements as well.
(5) In contrast to distillation, where a larger model produces training data for a small model, the MultiPL-T approach allows a smaller model to improve the performance of much larger models. Specifically, we use data generated by StarCoderBase-15B and validated using the MultiPL-T approach to improve the performance of models that are nearly 5x larger.

The supplementary material includes an evaluation with the MBPP benchmark. On MBPP, we show even larger relative gains with MultiPL-T.

5.2.2 Fine-Tuning StarCoder2 and DeepSeek Coder. When this work was done, Code Llama was the best-performing base Code LLM with open model weights. However, two new base models were recently released: DeepSeek Coder [Guo et al. 2024] and StarCoder2 [Lozhkov et al. 2024]. To show that MultiPL-T remains relevant, we fine-tune DeepSeekCoder-33B and StarCoder2-15B on our Racket dataset. The aforementioned models are the two top-performing base models of their size at the time this article was written. In both cases, we find that the MultiPL-T approach continues to work. Both models show significant improvement in their Racket capabilities (Table 3).

The datasets presented in this article are part of the StarCoder2 pretraining corpus as one of several sources of "(Leandro's) High Quality" data. All we can conclude is that they did not hurt performance on the target languages and that further fine-tuning still help (contrast with §3.1).

5.2.3 Summary. Using MultiPL-T, we have fine-tuned models from four different model families and a wide range of model sizes (1B to 70B parameters). We find that our fine-tuned models outperform base models across the board, with the most significant gains for the lowest-resource languages and small to mid sized models.

The largest models (70B) do not show as much as the smallest models. However, our datasets are generated with a much smaller model (15B), and it is unusual that we can use a smaller model to improve the performance of a much larger model [Burns et al. 2024].

At the time of writing, our fine-tuned version of DeepSeek Coder outperforms every fine-tuned model on the BigCode Models Leaderboard for Racket [Allal 2024]. The leaderboard models fine-tuned on an order of magnitude more data than ours, as well as models that distill proprietary models such as GPT-4. Thus we conclude that for low-resource languages, MultiPL-T is significantly more data efficient than alternative approaches to fine-tuning.

Table 4. The pass@1 scores of the original models and fine-tuned models on our new benchmark that uses user-defined types, higher-order functions, helper functions, and non-standard libraries. The scores suggest that the 15B model may have overfit when fine-tuned on OCaml. However, the other models do the same or better after fine-tuning.

Language	StarCoderBase-1B		StarCoderBase-15B	
	Base	Fine-tuned	Base	Fine-tuned
OCaml	33.0	33.7	50.6	42.9
Racket	19.3	22.7	28.4	41.3
Lua	26.3	46.9	48.7	51.3

5.3 New Tasks and Qualitative Evaluation

The previous section uses MultiPL-E for evaluation, which has a very particular format (§2.2) and uses unit tests to test correctness. Code LLMs are much more flexible and there are acceptability criteria that cannot be captured with unit tests or specific formats. We address these in this section.

5.3.1 Evaluating In-Context Learning. A limitation of the MultiPL-T datasets is that every training item is a single function without any other context: they may use standard libraries, but cannot depend on other functions, classes, or third-party libraries. Thus it is plausible that fine-tuning a model on MultiPL-T data will make it overfit to this format.[9] Moreover, conventional Code LLM benchmarks, including MultiPL-E, will not expose this problem, because the benchmark tasks largely involve generating standalone functions that use only standard libraries.

To determine if this kind of overfitting is a problem, we construct a new, multi-language benchmark with fourteen problems. We decided to manually construct each problem as opposed to sourcing them from repositories to ensure that the problems were similar across languages, and more importantly, that they were not part of the training data. Every problem has hidden test cases and a prompt that exercises the model's ability to use user-defined types, higher-order functions, helper functions, or external libraries (Table 5). We manually translate these prompts into idiomatic OCaml, Racket, and Lua.

We evaluate StarCoderBase 1B and 15B on this new benchmark (Table 4). The results suggest that *the 15B OCaml-tuned model may have overfit to the MultiPL-T format.* However, *all other fine-tuned models do the same or better.* We speculate that fine-tuning on a mix of natural and MultiPL-T data will decrease the likelihood of overfitting.

5.3.2 Evaluating Coding Style. A potential limitation of MultiPL-T is that it may negatively impact the style of generated code, since the training items are translated from Python. We study this issue in Racket using a qualitative evaluation process. We developed a Racket style grading rubric (Table 6) based on the Racket style guide and our experience teaching and grading Racket programming assignments. The rubric outlines grading items and their corresponding deductions. Several deductions are designed for style problems that arise in code written by beginning students, such as needlessly long lines. Others penalize Lisp style, such as using *car* and *cdr* instead of *first* and *rest*. Finally, the largest deduction is for using imperative iteration when a simple functional solution is possible.

We use our rubric to grade the HumanEval solutions produced by StarCoderBase-1B before and after fine-tuning on MultiPL-T Racket data. We grade only the 35 problems that both models can solve at least once (thus we omit several problems solvable only after fine-tuning). Since we generate

[9]Recall that the base models have been pretrained on natural code that is not constrained to the MultiPL-T format. Thus appropriate fine-tuning should not eliminate their ability to generate code that is not in the MultiPL-T format.

Table 5. The 14 problems that exercise in-context learning.

Problem	Description
Add Left of NumTree	Context: a type for a tree of numbers. Problem: add the numbers on the left branches.
Add Subway Station	Context: a subway system represented as a graph. Problem: add connections between stations.
Map Over Android Phones	Context: a type for phone models. Problem: a mapping function that only applies to Android phones.
Electrify Instruments	Context: a type hierarchy for several musical instruments with a flag that indicates if they are electric. Problem: mark instruments as electric.
Collatz Depth	Context: empty. Problem: requires a solution to see how far the given number is from converging on 1 in the Collatz sequence.
Mirror a Tree	Context: a type for trees. Problem: tree reflection.
Double Do It	Context: 2 of 3 helper functions. Problem: define the third helper and the primary function.
Points and Lines	Context: types for point and lines types Problem: Manhattan distance.
Series	Context: a type that represents a series with its current value and update function. Problem: define a function that updates to the next value in the series.
Shapes	Context: several shape types and functions/methods. Problem: define a function that uses the helpers.
Social Time	Context: types that define calendar events. Problem: calculate time spent on a particular kind of event.
Decode Message	Context: imports a common cryptography library. Problem: Decode AES encrypted message.
Verify Source	Context: imports a common cryptography library. Problem: verify signature.
Start AES	Context: imports a common cryptography library. Problem: generate an AES key and encrypts it.

several solutions for each problem, we select the most common working solution, and in the case of ties, we select the one with the best style. We grade 70 Racket programs in total: 35 produced by the base model and 35 after fine-tuning. To avoid bias, we use two graders and anonymized the selection and grading process by assigning random IDs to each program and tracking their provenance on a hidden spreadsheet. The two graders have substantial Racket teaching experience and we find minimal discrepancy between them: their scores differed by more than 1 point for just 15 out of the 70 candidate programs, and only differed by 3–4 points for two programs.

We compute the mean overall score for the base model and MultiPL-T model over the 35 HumanEval problems. We find that the base model achieves a style score of 89.5% and the fine-tuned model 85.2%. In other words, the mean grade for a base model program is 13.4/15 while for a fine-tuned model it is 12.8/15. Thus fine-tuning leads to a slight decrease in our Racket style score.

We inspect the 17 programs that scored higher for the base model than the fine-tuned model. We find that the fine-tuned model is more likely to use nested *if* expressions in these programs, as well as performing iteration where recursion is available. Conversely, we inspect the eight programs

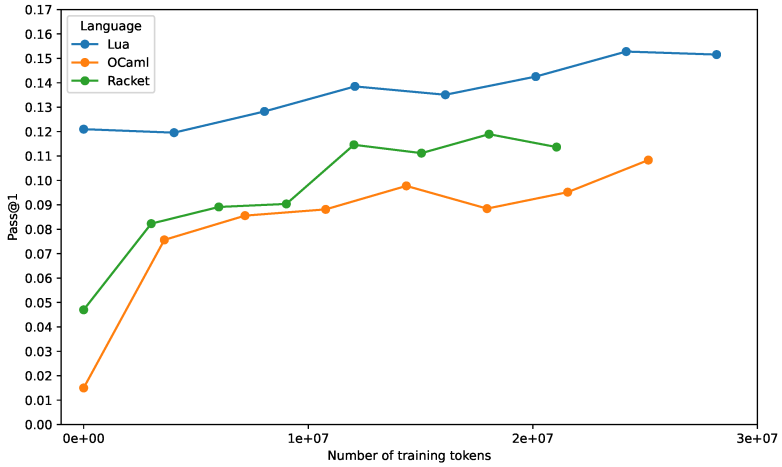


Fig. 10. We fine-tune three versions of StarCoderBase-1B on 25k MultiPL-T generated training items. The y-axis measures performance on MultiPL-E and the x-axis counts the number of tokens. The points on the line mark epochs. Racket is far less verbose than Lua or OCaml, and thus has fewer tokens at each epoch. Epoch 0 is the base model.

Table 6. Our Racket style rubric for grading generated programs. The maximum score for a program is 15 points. Items are grouped into categories according to the type of error they entail.

Category	Grading Item	Max Deduction
Text	Dangling parentheses	−0.5
	Line is too long	−1
	Using `car`/`cdr`	−0.5
	`cond` with round brackets	−0.5
Definitions	`let` expression not at beginning of function body	−1
	Nesting `define` or `let` expressions	−2
	Unnecessary use of `let*` or `letrec`	−1
	Defining useless local variables	−1
	Not defining helpers or variables for reused code	−1
Conditionals	Nested `if` expression instead of `cond`	−2
	Using (`if COND #t #f`)	−1
Traversal	Using iteration when recursion is available	−3

that scored higher in the MultiPL-T model and find that the base model is more likely to use direct recursion with *car/cdr* while the fine-tuned model uses Racket list abstractions. Figure 11 shows the breakdown of the kinds of deductions assigned per problem to each model, where deductions are averaged among graders. Figure 12 shows a graded pair where the base model scores better on coding style than the fine-tuned model, and the supplementary material shows a graded pair where the converse is true: the fine-tuned model produces a cleaner solution than the base model. Coding style aside, in both cases the lower quality solution is much more verbose than necessary.

The results of our evaluation are consistent with how we generated our training data from Python code. Overall, although *fine-tuning slightly decreases the model's ability to generate idiomatic Racket code, it also significantly increases its ability to generate correct Racket code* as we showed

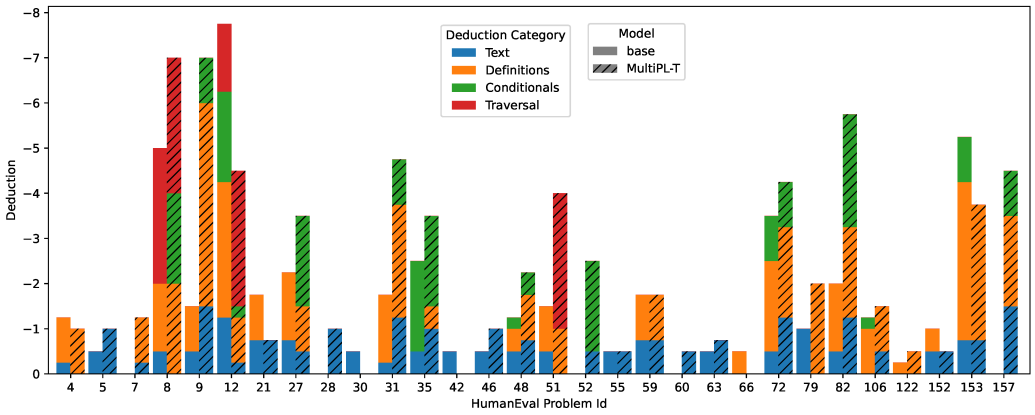


Fig. 11. Results of our Racket style evaluation for 30 out of 35 selected HumanEval problems. Omitted problems have no deductions and both models scored 15/15 points. Deductions for each item category are averaged across graders for each problem. Fine-tuning has a small negative impact on coding style.

```
(define (rolling_max numbers)
  (define (max_in_list list)
    (foldl max 0 list))
  (define (max_in_list_with_index list)
    #| body omitted |#)
  (let loop ((numbers numbers) (max_numbers '()))
    (if (empty? numbers)
      max_numbers
      (let* ((number (car numbers)) (max_number (max_in_list max_numbers)))
              (loop (cdr numbers)
              (append max_numbers
                      (list (if (> number max_number) number max_number)))))))))
```

(a) A solution from the fine-tuned model. The `max_in_list_with_index` function is redundant.

```
(define (rolling_max numbers)
  (define (max_list numbers)
    (if (null? numbers)
        '()
        (cons (apply max numbers) (max_list (cdr numbers)))))
  (reverse (max_list (reverse numbers))))
```

(b) A better solution from the base model.

Fig. 12. Two working solutions to the same HumanEval-Racket problem produced by the base and MultiPL-T fine-tuned model. In this case, the base model produces higher-quality code.

earlier. Moreover, *both models perform well on the Racket style rubric*, which suggests that the trade-off is minimal.

5.4 MultiPL-T Ablations

The final part of our evaluation are ablations to demonstrate the efficacy of different parts of the MultiPL-T pipeline. This section showcases results for a significant number of fine-tuned models. To manage costs, we use StarCoderBase-1B for the majority of the experiments.

Table 7. The pass@50 performance of the basic prompt compared to the prompt with the original Python code for OCaml, Racket, and Lua.

Language	Basic	With Python Solution
OCaml	26.1	23.9
Racket	34.7	56.8
Lua	51.4	68.5

5.4.1 Fine-Tuning Efficiency. We now investigate how the performance of StarCoderBase-1B on MultiPL-E varies during fine-tuning. On three languages (Lua, OCaml, and Racket), we fine-tune StarCoderBase-1B for seven epochs of MultiPL-T data and evaluate at each epoch. However, the datasets that we have generated are imbalanced in the number of training items (Table 8). To better balance the training sets, we randomly sample 25,000 items from each dataset to get similarly-sized fine-tuning sets for each language.[10]

Table 8. Dataset sizes.

Language	Size
R	37,592
Racket	40,510
OCaml	43,401
Julia	45,000
Lua	48,194

In Figure 10, we see that performance increases substantially after a single epoch of MultiPL-T data for the lowest resource languages (Racket and OCaml). However, a higher-resource language (Lua) requires more data to realize even modest gains. These results are what one would expect: *lower-resource languages enjoy easy and efficient gains from fine-tuning with MultiPL-T data.*

On the other hand, MultiPL-T requires more computing resources to generate data for low-resource languages, so the overall efficiency gap between languages is narrower than the figure suggests: it does not show the cost of generating training data. Nevertheless, *MultiPL-T data only needs to be generated once for a given language and can then be reused to train many models* (§5.2).

5.4.2 Translation Versus Generation. MultiPL-T uses StarCoderBase-15B to translate training items from Python to low-resource languages. The success rate of this translation is dependent on both the quality of the pretrained model (which is fixed) and the quality of the prompt (which we design). We tried several prompt variations during development and eventually settled on a prompt that includes the original Python code in a comment (§4.3).

Doing a complete ablation with all five languages and ~133,000 functions would be prohibitively expensive. Instead, we run an experiment with a random sample of 1,000 source Python functions. We use the LLM to translate these 1,000 functions to OCaml, Racket, and Lua with two different prompt formats: with and without the original Python source. We generate 50 candidates for each prompt.[11] We evaluate performance using pass@50, which is the likelihood that the model produces at least one correct solution in 50 attempts.

As shown in Table 7, *adding the original Python to the prompt substantially increases the likelihood of a successful translation to Racket and Lua, but slightly decreases the likelihood of a successful translation to OCaml.* We can only speculate about why this happens: Python may be misleading the model and OCaml seems further removed from Python than Racket or Lua.

5.4.3 Testing Is Critical for MultiPL-T. MultiPL-T relies on tests to validate the LLM-translated training items. We reason that since we are translating to programming languages on which the

[10]However, small differences remain: Lua is more verbose than OCaml, so 25,000 training items for Lua is more data than 25,000 training items for OCaml.

[11]Thus this small experiment still requires 300,000 generations from the LLM and takes about 1/2 a day on an A100 GPU.

Table 9. Without test-based validation, the LLM-translated training items are poor quality and fine-tuning is not effective. The performance of the 15B model decreases slightly compared to the base model. The performance of the 1B model increases slightly, but is far worse than the result we get with the tested items.

Model	Baseline	MultiPL-T	No Validation
StarCoderBase-1B	4.7	11.3	8.6
StarCoderBase-15B	11.8	21.0	11.6

LLM performs poorly at synthesis, it is also likely to perform poorly at translation. But, to validate this claim, we fine-tune models on Racket data without test validation. We get poor results as expected (Table 9). This shows that the effort we take to generate, validate, and compile tests is essential for MultiPL-T.

5.4.4 The Impact of Deduplication. Prior work has shown that data duplication decreases performance while increasing training time [Allal et al. 2023; Lee et al. 2022]. To demonstrate its impact in our work, we do an experiment on the dataset where it has the most impact—where deduplication discards the most data—which is for Lua. This is to be expected because the number of nearly duplicate functions will be higher when the data generating LLM translates functions with a higher success rate. Of the languages we target, our data generating LLM (StarCoderBase15-B) has the highest pass rate on Lua.

Before deduplication, the Lua dataset has 1.4M functions, but after deduplication we are left with 48K (reported in Table 8). For this experiment, we fine-tune StarCoderBase-1B on a single H100 GPU on the undeduplicated dataset, with exactly the same hyperparameters we used earlier (§5). We find the following:

(1) Without deduplication, the fine-tuned StarCoderBase-1B gets a **17.1** pass@1 score on MultiPL-E and takes **12 hours** to train.
(2) With deduplication, the fine-tuned StarCoderBase-1B gets **17.3** pass@1 on MultiPL-E and takes less than **30 minutes** to train.

The two pass@1 scores are very close. But, without deduplication, training takes significantly longer, because of the increase in dataset size.

The impact would be more significant with a larger model. For example, it took us 8 hours on 8xH100 GPUs to fine-tune StarCoderBase-15B with a deduplicated dataset. Without deduplication, fine-tuning would take several days. Similarly, fine-tuning the 33B and 70B models would take even longer. Thus deduplication is a way to manage the time and cost of training large models.

5.4.5 Targeting the Source Language. The complete MultiPL-T toolchain is designed to produce training data for a target low-resource language from a source language (Python). However, it is also possible to directly evaluate the first part of the MultiPL-T toolchain, which builds a high-quality dataset for the source language (Python). In this section, we fine-tune select LLMs on Python using the dataset of 133,668 Python functions that are documented, tested, typeable, and filtered on test coverage (§§ 4.1 and 4.2).

We fine-tune and evaluate StarCoderBase-1B directly on this dataset of Python functions. The base model gets a **15.1** pass@1 score on Python, whereas the fine-tuned model gets **15.0** pass@1.[12] Thus, it is clear that fine-tuning directly on this subset of Python does not produce the same improvements that we see on the low-resource languages after translation.

We hypothesize that this occurs for the following reason: StarCoderBase-1B has already been trained on three epochs of this Python data. We are not fine-tuning the model on novel data, thus

[12]We observe this score on epoch 3. On epochs 1–7, the score varies between 10.7 and 15.0.

there is little new for the model to learn, and this is reflected in the unchanged pass@1 score. In contrast, when we translate this data to a low-resource language using the full MultiPL-T pipeline, we are effectively creating novel data for the model to learn.

6 Discussion

We have shown that MultiPL-T is an effective and efficient method for generating semi-synthetic training data for low-resource programming languages. In this section, we discuss the implications of extending MultiPL-T in various ways.

Generalizing to other programming languages. We hope it is clear to the reader that the MultiPL-T approach is straightforward to generalize to more programming languages. The language-specific work involves 1) translating comments and function signatures into an appropriate prompt, and 2) writing a compiler that can translate simple assertions from the source to the target. MultiPL-E already supports both these steps for 20+ programming languages, several of which are low-resource, including D, Bash, MATLAB, Haskell, and Perl. So, generating fine-tuning sets for these languages may just be a matter of running the MultiPL-T pipeline for a few days on GPUs.

Some combinations of source and target languages may be more effective than others. For example, consider building semi-synthetic training data for Rust. We could directly leverage the datasets in this paper and utilize a Code LLM to translate Python to Rust. However, we speculate that it would be more effective to modify MultiPL-T to support a source language closer to Rust, such as C++.

Generalizing to other LLMs. Although this paper focuses on fine-tunes of the Code Llama and StarCoder family of Code LLMs, our datasets could also be used to fine-tune other LLMs. We fine-tune DeepSeek Coder and StarCoder2 on Racket, and we expect our results will generalize to the other languages.

Regenerating MultiPL-T data. The MultiPL-T pipeline in this paper uses StarCoderBase-15B. It should be clear that using a more capable model would improve conversion rates, and thus produce better results. A proprietary model such as GPT-4 would likely produce the best results, but doing so would violate its terms of use [OpenAI 2023b]. An unusual result in this paper is that MultiPL-T can use a weaker model (StarCoderBase-15B) to improve much better models (CodeLlama-70B, DeepSeekCoder-34B, and StarCoder2-15B) on low-resource languages.

"No-resource" languages. MultiPL-T targets low-resource languages, but it is unlikely to work as-is for "no-resource" languages that StarCoderBase is not trained on at all. It may be possible to cleverly prompt the model with enough information about a no-resource language to bootstrap data generation. But, doing so efficiently may be challenging.

Composability with self-instruct. Although we have argued that self-instruct is unlikely to succeed on a low-resource language, self-instruct and MultiPL-T could be composed together in a natural way: one could generate a high-quality dataset of instructions in a high-resource language, and then use MultiPL-T to translate them to a low-resource language. Given the effectiveness of WizardCoder [Luo et al. 2024] and Magicoder [Wei et al. 2024] at Python, this composition seems likely to succeed.

Other kinds of benchmarks. A lot of effort has been put into evaluating the Python programming abilities of LLMs, but there are far fewer benchmarks for low-resource languages. For example, there are Python benchmarks for specialized tasks, such as data science [Lai et al. 2023], and with prompts authored by specific populations, such as beginning programmers [Babe et al. 2024]. We

need to develop these kinds of benchmarks for low-resource languages to truly understand the capabilities and limitations of Code LLMs.

7 Related Work

Code translation with language models and unit tests. A number of projects use language models to translate code between programming languages and test that the generated translations are correct by compiling working unit tests from one language to another. TransCoder-ST [Roziere et al. 2021] and CMTrans [Xie et al. 2024] use these techniques to generate training data for a code translation model between Java, Python, and C++, whereas MultiPL-E [Cassano et al. 2023], MBXP [Athiwaratkun et al. 2022], and BabelCode [Orlanski et al. 2023] translate Code LLM benchmarks from Python to 10+ programming languages. A distinguishing feature of MultiPL-T is that it employs an off-the-shelf pretrained Code LLM (StarCoder) to both generate test cases and translate code to low-resource languages. When the aforementioned papers were written, the best open Code LLMs were far less capable than StarCoder: they were trained on fewer programming languages using far less training data, and they were an order of magnitude smaller. Thus we believe the MultiPL-T approach would have likely failed. Although capable closed models were available, they were either rate-limited or prohibitively expensive for the scale of data generation that MultiPL-T needs. For example, Cassano et al. [2023] report that they used a commercial model during a free beta period, but it would have cost $37,000 with equivalent released models. MultiPL-T's data generation requires an order of magnitude more queries, making it prohibitively expensive to use with commercial models at 2023 prices.

Multi-lingual language models of code. Training models on a dataset of code written in programming languages similar to the target language, known as transfer learning, is a commonly studied method for improving model performance on a specific programming language [Ahmed and Devanbu 2022; Baltaji et al. 2024; Katzy et al. 2023]. Chen et al. [2022] explore this method in the context of low-resource languages by pretraining and fine-tuning small encoder-only Transformers with different programming languages. They then evaluate their performance on code search and summarization tasks for Ruby, a language with a dataset 10x smaller than that of Python. Their findings suggest that multilingual pretraining can improve performance at generating code in low-resource languages. However, this strategy may falter with languages that significantly differ in syntax or semantics, such as Racket. For example, Baltaji et al. [2024] employ a similar methodology to train models on 41 different programming language pairs, noting that Scheme, the predecessor of Racket, benefits less from transfer learning compared to other languages. In such scenarios, data generation strategies like MultiPL-T are deemed more likely to achieve success. Moreover, these studies predominantly rely on the BLEU score and other syntax-based metrics for evaluating correctness, which have been criticized for their inadequacy in code generation tasks [Chen et al. 2021]. In our work, we utilize test-based validation of both our training data and evaluation metrics to ensure the correctness of generated code, which is a more reliable measure of model performance.

Our work begins with the observation that Code LLM performance can vary significantly by language, and this has been observed repeated in prior work. Most closely related to MultiPL-T are benchmarks for the natural language to code task [Athiwaratkun et al. 2022; Cassano et al. 2023; Orlanski et al. 2023]. However, similar trends occur when Code LLMs are used for other tasks, such as fuzzing and translation [Pan et al. 2024; Xia et al. 2024].

Instruction tuning. To get an LLM to perform a desired task, the user must prompt it in the right way. There are several techniques for *instruction tuning* LLMs to better follow natural, human-written instructions. One approach uses human annotators to write sample instructions and give

feedback on a large number of model generations [Ouyang et al. 2022], but this is expensive and requires significant resources. A cheaper approach is to have a capable LLM *self-instruct* to generate instructions from a relatively small set of human-written seed instructions [Wang et al. 2023]. Evol-Instruct uses an LLM to create variations of instructions [Luo et al. 2024]. These techniques have been used to create datasets for instruction-tuning Code LLMs [Chaudhary 2023; Luo et al. 2024; Muennighoff et al. 2024; Wei et al. 2024]. These datasets focus on high-resource languages, and, as we show in §3.2, they are unlikely to succeed for low-resource languages.

Training on high-quality data. Training on high-quality data is an effective way to reduce both the size of an LLM and the volume of training data needed, while maintaining performance. Gunasekar et al. [2023] achieves high HumanEval scores on a small model with a modest amount of "textbook quality" training data. This includes both natural and synthetic data generated by a more capable model. Their work targets Python, and we argue in §3 that the approach is less likely to succeed with low-resource languages.

Proprietary Code LLMs. At the time of writing, there are proprietary LLMs that perform better at programming tasks than the open models we build upon [Anil et al. 2023; Anthropic 2023a; OpenAI 2023a]. However, most of these models only support inference (i.e., running the trained model) and not training or fine-tuning. Even when fine-tuning is possible, because these models are trained on closed data sets, we would not be able to compare MultiPL-T to the natural baseline of training longer on existing data. Moreover, a significant limitation arises from the proprietary licensing constraints of these models. Many of their licenses expressly forbid the use of generated data to train other models [Anthropic 2023b; Google 2023; OpenAI 2023b].

Even though, many proprietary models forbid using their generated data to train other models, there are models that do so. These models are *distillations* of much larger proprietary models, and it is not possible to compare them to the natural baseline, which is training longer on existing data, which for these models is the proprietary OpenAI training set. The strength of the MultiPL-T approach is not that it does better in an absolute sense, but that fine-tuning with MultiPL-T data is better than training longer on existing data. We show that this holds for StarCoderBase and its training set, and can only do so because both are open.

8 Conclusion

In the last few years, Code LLMs have rapidly made their way into more and more programming tools. However, the quality and reliability of Code LLMs is highly language-dependent: they can be remarkable on high-resource programming languages, but are far less impressive at working with low-resource languages [Cassano et al. 2023]. It is possible that in the near future, a large number of programmers will expect LLM-based technology to just work, just as many programmers today expect syntax highlighting, continuous analysis, or type-based completion in their programming environments. We hope that MultiPL-T—a methodology for generating large-scale, high-quality fine-tuning datasets in low-resource languages—will help low-resource languages compete in a world where many developer tools rely on Code LLMs.

The MultiPL-T fine-tuning data (and code) are also open: they are constructed from the StarCoder training data (The Stack) and augmented by StarCoder itself. We deliberately do not use a more capable proprietary model to fine-tune an open model. This allows us to demonstrate that fine-tuning on MultiPL-T data is more effective and efficient than training longer on existing data. We evaluate MultiPL-T in several other ways, including a new benchmark that exercises in-context learning and a qualitative evaluation of coding style. When compared to other fine-tunes of the same base model, MultiPL-T achieves state-of-the-art results for Julia, Lua, OCaml, R, and Racket.

Data-Availability Statement

All code, data, and models from this paper are available with open licenses. *Code:* available on GitHub and archived on Zenodo; *Datasets:* available on Hugging Face; and *Models:* available on Hugging Face. In particular:

(1) A guide to reproduce the results in this article is available at doi.org/10.5281/zenodo.12453932.
(2) The MultiPL-T datasets are available at doi.org/10.57967/hf/2941 with links to several models fine-tuned on these datasets.

Acknowledgements

We thank Loubna Ben Allal, Harm de Vries, Joydeep Biswas, Matthias Felleisen, Shriram Krishnamurthi, and Leandro von Werra for helpful conversations. Thanks to Leandro von Werra for including the MultiPL-T datasets in the StarCoder2 training corpus. We also thank the OOPSLA reviewers for their feedback. We especially thank the artifact evaluation committee for their patience reviewing our complex artifact, as well as the AEC chairs (Guillaume Baudart and Sankha Narayan Guria) for facilitating review. This work used computing resources provided by Northeastern Research Computing, New England Research Cloud, and Joydeep Biswas (UT Austin). Federico Cassano was affiliated with Roblox for most of his work on this paper. This work is partially supported by the National Science Foundation (SES-2326173, SES-2326174, and SES-2326175).

References

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1443–1455. https://doi.org/10.1145/3510003.3510049

Loubna Ben Allal. 2024. Big Code Models Leaderboard. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: Don't Reach for the Stars!. In *Deep Learning for Code Workshop (DL4C)*.

Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. 2023. PaLM 2 Technical Report. arXiv:2305.10403 [cs.CL]

Anthropic. 2023a. Model Card and Evaluations for Claude Models. https://www-files.anthropic.com/production/images/Model-Card-Claude-2.pdf Accessed: August 17, 2023.

Anthropic. 2023b. Terms of Service. https://console.anthropic.com/legal/terms Accessed: August 17, 2023.

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, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2022. Multi-Lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations*.

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* (2021).

Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q. Feldman, and Carolyn Jane Anderson. 2024. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. In *Findings of the Association for Computational Linguistics*.

Razan Baltaji, Saurabh Pujar, Louis Mandel, Martin Hirzel, Luca Buratti, and Lav Varshney. 2024. Learning Transfers over Several Programming Languages. arXiv:2310.16937 [cs.CL]

Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. arXiv:2206.01335 [cs]

Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344. https://doi.org/10.1109/ICSME.2016.31

Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeffrey Wu. 2024. Weak-to-Strong Generalization: Eliciting Strong Capabilities With Weak Supervision. In *International Conference on Machine Learning (ICML)*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering (TSE)* 49, 7 (2023), 3675–3691.

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. 2024. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions. In *International Workshop on Large Language Models for Code (LLM4Code)*.

Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.

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 *IEEE/ACM International Conference on Program Comprehension (ICPC)*. Association for Computing Machinery, 401–412. https://doi.org/10.1145/3524610.3527917

Le Chen, Xianzhong Ding, Murali Emani, Tristan Vanderbruggen, Pei-Hung Lin, and Chunhua Liao. 2023. Data Race Detection Using Large Language Models. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*. Association for Computing Machinery, New York, NY, USA, 215–223. https://doi.org/10.1145/3624062.3624088

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 arXiv:2107.03374* (2021).

CodeWhisperer. 2023. ML-powered Coding Companion – Amazon CodeWhisperer – Amazon Web Services. https://aws.amazon.com/codewhisperer/.

Github Copilot. 2023. Github Copilot Your AI pair programmer. https://github.com/features/copilot

Harm de Vries. 2023. Go smol or go home. https://www.harmdevries.com/post/model-size-vs-compute-overhead/.

Felipe Hoffa. 2016. GitHub on BigQuery: Analyze All the Open Source Code. https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code.

Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (6–8). San Fransisco, CA, USA.

Google. 2023. Generative AI Terms of Service. https://policies.google.com/terms/generative-ai Accessed: August 17, 2023.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. arXiv:2306.11644 [cs.CL]

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. https://doi.org/10.48550/arXiv.2401.14196 arXiv:2401.14196 [cs]

George E. Heidorn. 1974. English as a Very High Level Language for Simulation Programming. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. Association for Computing Machinery, New York, NY, USA, 91–100.

https://doi.org/10.1145/800233.807050

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katherine Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack William Rae, and Laurent Sifre. 2022. An empirical analysis of compute-optimal large language model training. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.).

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 4 (June 2023), 5131–5140. https://doi.org/10.1609/aaai.v37i4.25642

J. Katzy, M. Izadi, and A. Deursen. 2023. On the Impact of Language Selection for Training and Evaluating Programming Language Models. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Los Alamitos, CA, USA, 271–276. https://doi.org/10.1109/SCAM59687.2023.00038

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The Stack: 3 TB of Permissively Licensed Source Code. In *Deep Learning for Code Workshop (DL4C)*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *International Conference on Machine Learning (ICML)*.

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 *Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 8424–8445. https://doi.org/10.18653/v1/2022.acl-long.577

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

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

Chin-Yew Lin and Franz Josef Och. 2004. Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*. Barcelona, Spain, 605–612. https://doi.org/10.3115/1218955.1219032

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. arXiv:2402.19173 [cs.SE]

Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *Proceedings of the 12th Symposium on Search-based Software Engineering (SSBSE 2020, Bari, Italy, October 7–8) (Lecture Notes in Computer Science, Vol. 12420)*. Springer, 9–24. https://doi.org/10.1007/978-3-030-59762-7_2

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *The Twelfth International*

Conference on Learning Representations.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. OctoPack: Instruction Tuning Code Large Language Models. In *International Conference on Learning Representations (ICLR)*.

Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan. 2023. CodeCompose: A Large-Scale Industrial Deployment of AI-assisted Code Authoring. arXiv:2305.12050 [cs.SE]

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 1–13. https://doi.org/10.1145/3597503.3639187

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv:2305.02309 [cs.LG]

OpenAI. 2023a. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

OpenAI. 2023b. Terms of Service. https://openai.com/policies/terms-of-use Accessed: August 17, 2023.

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 *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 26619–26645.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. Curran Associates, Inc.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models While Translating Code. In *IEEE/ACM International Conference on Software Engineering (ICSE) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639226

Tung Phung, José Pablo Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Kumar Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *ArXiv* abs/2302.04662 (2023).

Pyright. 2023. Static Type Checker for Python. https://github.com/Microsoft/pyright

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Replit. 2023. Replit Code v1.3. https://huggingface.co/replit/replit-code-v1-3b.

Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces* (Sydney, NSW, Australia) *(IUI '23)*. Association for Computing Machinery, New York, NY, USA, 491–514. https://doi.org/10.1145/3581641.3584037

Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. Adaptive Test Generation Using a Large Language Model. *IEEE Transactions on Software Engineering (TSE)* 50, 1 (2024). https://doi.org/10.1109/TSE.2023.3334955

TabNine. 2023. AI Assistant for Software Developers | Tabnine. https://www.tabnine.com/.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Model with Self Generated Instructions. In *Annual Meeting of the Association of Computation Linguistics (ACL)*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. In *International Conference on Machine Learning (ICML)*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics,

38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639121

Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. 2024. Data Augmentation for Code Translation with Comparable Corpora and Multiple References. In *Findings of EMNLP*.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Deep Learning for Code Workshop (DL4C)*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. Association for Computing Machinery, 5673–5684. https://doi.org/10.1145/3580305.3599790

Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/3520312.3534864

Received 2024-04-05; accepted 2024-08-18