Building A Proof-Oriented Programmer That Is 64% Better Than GPT-4o Under Data Scarcity

Dylan Zhang[1,‡], **Justin Wang**[2], **Tianran Sun**[3]
[1]University of Illinois Urbana-Champaign,
[2]University of Chicago,
[3]Shanghai Jiaotong University,
[‡]Project Lead
Correspondence: shizhuo2@illinois.edu

Abstract

Existing LMs struggle with proof-oriented programming due to data scarcity, which manifest in two key ways: (1) a lack of sufficient corpora for proof-oriented programming languages such as F*, and (2) the absence of large-scale, project-level proof-oriented implementations that can teach the model the intricate reasoning process when performing proof-oriented programming. We present the first on synthetic data augmentation for project level proof oriented programming for both generation and repair. Our method addresses data scarcity by synthesizing basic proof-oriented programming problems for proficiency in that language; incorporating diverse coding data for reasoning capability elicitation and creating new proofs and repair data within existing repositories. This approach enables language models to both synthesize and repair proofs for function- and repository-level code. We show that our fine-tuned 14B parameter model, PoPilot, can exceed the performance of the models that outperforms GPT-4o in project-level proof-oriented programming by 64% relative margin, and can improve GPT-4o's performance by 54% by repairing its outputs over GPT-4o's self-repair.

1 Introduction

In an era where software vulnerabilities can risk tremendous damages[1] and compromise national security [2], ensuring the correctness and safety has become an urgent priority. Proof-oriented programming integrates formal verification into software development, enabling mathematically rigorous correctness guarantees. Proof-oriented programming languages such as F*(Swamy et al., 2011) and Dafny(Microsoft, 2024a) support this paradigm by providing expressive type systems and precise specification mechanisms. These capabilities enable static verification of program correctness without the need for extensive test suites or runtime execution. These languages like F* allow developers to write programs alongside formal proofs, providing strong guarantees about functional correctness and security properties, and famous real-world systems including Firefox, the Linux kernel, Tezos blockchain, and Azure Cloud have employed formally verified components.However, despite decades of research, the adoption of proof-oriented programming remains limited due to the high cost of proof construction and the steep learning curve of formal methods.

On the other hand, the rapid advancements in large language models (LLMs) have transformed many areas of software development(Austin et al., 2021; Chen et al., 2023; Nijkamp et al., 2022; Xia and Zhang, 2022; Xia et al., 2023; Jin et al., 2023; Jimenez et al., 2024). Yet, applying LLMs to proof-oriented programming presents fundamental challenges (Chakraborty et al., 2024). Proof-oriented programming is highly distinct from conventional coding paradigms, demanding complex formal reasoning of program semantics over long contexts, which current models struggle with (Loughridge et al., 2024). A major bottleneck is extreme data scarcity, which manifests in two key ways: (1) a lack of diverse, high-quality corpora in proof-oriented programming languages such as F* to teach the model the syntax and semantics of that language (F* constitutes of only 0.002% in Stack-v2 (Lozhkov et al., 2024)), and (2) an absence of large-scale project-level verification data, which involves highly complex and context-dependent formal reasoning. As a result, even state-of-the-art LMs fail to generalize effectively to proof construction and verification tasks (Loughridge et al., 2024;

[1]On July 19, 2024, an abnormal update distributed by the cybersecurity company CrowdStrike caused issues for a large number of its global customers' Windows operating systems, resulting in blue screen errors.

[2]U.S. Cybersecurity and Infrastructure Security Agency has urged the future software development to ensure memory safety.

Findings of the Association for Computational Linguistics: ACL 2025, pages 23101–23118

Microsoft, 2024b).

In this work, we introduce a data-centric post-training recipe designed to bridge the gap between general-purpose coding LLMs and repository-level proof-oriented programming in F*. We systematically address data scarcity and adaptation challenges through three key strategies:

1. **Enhancing General Programming Capabilities with Diverse Code Data:** inspired by existing works showing the benefit of diversification (Zhang et al., 2024; Dong et al., 2023; Chen et al., 2024), we train the model beyond the immediate focus of formal verification in F* but over diverse programming tasks to enable its code-reasoning and instruction-following capabilities.

2. **Learning Basic Proof-Oriented Programming via Synthetic Tasks:** We synthesize function-level basic programming and property-proving problems in F*, allowing LLMs to learn fundamental verification patterns in a controlled setting without introducing complex inter-dependencies.

3. **Synthetic Augmentation for Proof Synthesis and Repair:** Beyond training basic property-proving problems and existing repositories, we curate novel synthetic repository-level problem solving and proof repair data to teach LLMs how to complete and correct project-level proofs with more complex and longer contextual dependencies.

Following the recipe above, we transform LLMs into specialized verification assistants capable of both synthesis and repair for **P**roof-**o**riented **P**rogramming, which we call PoPilot. PoPilot is the first project-level formal verification specialist LLM trained on synthetic instruction-tuning data. Notably, strategies **2** and **3** require generating problems using existing language models that has limited knowledge and low accuracy on F*. However, we could leverage F* solver to obtain data with correctness guarantee for generation tasks, and retrieve error messages to craft repair datasets[3] Our experiments demonstrate that PoPilot demonstrates strong capacities to perform proof-oriented programming on a project level, leading to a remarkable margin of 64% over GPT-4o and can boost GPT-4o's performance to 54% by repairing a randomly chosen failed attempt.

[3]The correctness can be determined by running the solver without test cases as in conventional programming languages.

2 Background

Formal verification is a process of examining the correctness of the operation of software programs by mathematical proof (Grout, 2008). F* is an SMT-solver based proof-oriented language that enables convenient verification through execution by F* compiler. However, F*, like many other languages used for formal proofs (verified Rust, Rocq(Coq), Lean etc.), are comparatively low resource. The popular open coding data corpus Stack-v2 (Lozhkov et al., 2024) containing over 3B files in 600+ programming and markup languages, F* has only 29.6k entries (less than 0.002%), much fewer than common programming languages like Python (80.6M entries, 2.95%), Java (223M entries, 8.17%). This scarcity implies both the lack of knowledge of the off-the-shelf pre-trained checkpoints and the insufficiency of existing resources to further train the model.

Additionally, F* is a dependently-typed language, where type definitions depend on values. This allows for more precise specification of program properties and invariants but also introduces complexity due to the need for intricate computations to determine type equality and detailed type reasoning (Chakraborty et al., 2024). Programmers also often need to go back and forth while writing F* code. Therefore, enhancing the model's ability to repair code is crucial for both iterative improvement of automatic proof synthesis and better assistance to human programmers.

In reality, the challenge of proof-oriented programming is further exacerbated by the cross-repository dependencies of the code. Proof-oriented programming often spans multiple repositories, especially in the large-scale formal verification of software systems, where the project contains different components relying on verifying properties from other repositories. This introduces huge challenges in dependency and environment management, as the proofs account for resolving inter-repository specification and module openings beyond the single repository (Chakraborty et al., 2024). This property, together with the type-dependent nature of the programming language, makes the task difficult to learn for model and even human expertise.

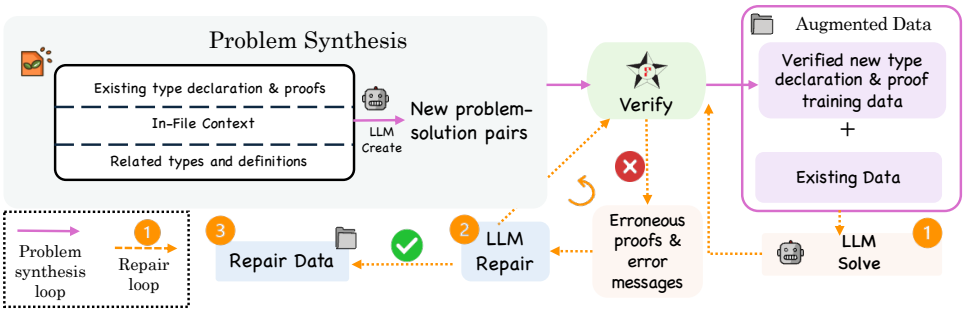

Figure 1: Illustration of repository-level data generation pipeline.

3 Function-Level Dataset Collection

In this section, we describe how we synthesize diverse function-level programming tasks from existing open-source code snippets. We focus on three types of tasks: natural language to code tasks, proof-oriented code completion tasks, and code repair tasks. To evaluate the generated data, we rely on F*'s feature to automatically verify code correctness via execution. In addition, the dataset is diversified by incorporating instruction-tuning data from other languages.

3.1 Data Synthesis

In this section, we describe our method for synthesizing high-quality F* training data. To generate diverse programming tasks, we construct instruction data from existing open-source code snippets inspired by OSS-Instruct (Wei et al., 2024b). We select code models that outperform non-code models in terms of accuracy and quality for F* code generation and prompt them to create high-quality tasks from a variety of code examples.

3.1.1 Task generation

We included three types of tasks in our dataset: natural language based tasks, proof-oriented code completion tasks and code repair tasks. Below, we describe each task in detail.

Natural Langugage To Code Tasks We use code examples from the F* source code repository and GitHub OCaml (Geng, 2022) as our seed corpus.The code snippets extracted from these source codes are diverse, correct and practical, inspiring LLM to generate high-quality and varied instructions and responses. The response code generated by the model is also required to be self-contained to facilitate the verification of code correctness and the generation of other types of tasks.

Proof-oriented code completion tasks In proof-oriented code completion problems, we choose verified and self-contained code snippets from the responses models generated in all tasks. We then ask the model to generate code completion problems that require proving a specific property or function related to the given snippet in order to obtain problems and corresponding responses related to proofs. This part of data, with greater complexity, enhances the model's ability to write proofs in F* and perform reasoning. Verified response codes can also be reused as snippets for future code completion tasks.

Instruction:
In the provided F* code snippet, we have a function `list_with_length` that takes a list of integers and returns a tuple containing the original list and its length. Your task is to write an F* program that proves the following property:
Property: For any list of integers `l`, the length of the list returned by `list_with_length l` is equal to the length of the original list `l`.
Here is the code snippet for reference:

Code Snippet:
```fstar
module ListUtils
open FStar.List.Tot
let list_with_length (l: list int): (list int * int) =
  let len = FStar.List.length l in
      (l, len)
```

Response:
```fstar
module ListUtils
open FStar.List.Tot
let list_with_length (l: list int): (list int * int) =
    let len =
FStar.List.length l in
        (l, len)
// Lemma to prove the property
val lemma_length_preserved: l:
list int -> Lemma (requires
True) (ensures snd
(list_with_length l) =
FStar.List.length l)
let lemma_length_preserved l =
    assert (snd
(list_with_length l) =
FStar.List.length l)
```

Figure 2: Function-level Proof-oriented programming example.

Code repair tasks. It is difficult to generate correct code on the first try for many programming tasks, even for human. Instead of discarding the flawed code, they typically analyze the results and modify it to fix any errors (Chen et al., 2023). Therefore, It is crucial for LLMs to have the capability to repair code as well. Given the complexity of writing F* programs, which often requires iterative refinement, models trained on synthetic datasets should be capable of repairing erroneous code effectively. Code repair pairs in our dataset

are generated by prompting the model to fix the given incorrect code response, supplying both the erroneous code and the execution log obtained after verification.

The verified instruction-response pairs are added to our dataset.

3.1.2 Execution-based data evaluation and dataset filtering

As an SMT-guided language, F* can directly and accurately receive correctness feedback from execution, enabling the identification of discrepancies between code behavior and formal specifications. Therefore, no extra effort such as test cases and LLM judges is required for verifying F* data. Specifically, to verify the response codes, we put the code snippet within the F* environment and execute them, sparing little effort. Data that successfully compile and run are retained, while incorrect ones and their associated error messages are stored for inclusion in the repair dataset. This verifiability is a favorable property of F* that we could leverage to obtain data quality signals for free, whereas in general domains, the instruction tuning data is largely unverifiable or relies on heuristics (Wei et al., 2024a).

3.2 Diversification

Diverse instructions can better enhance an LLM's ability to generalize to new tasks (Wei et al., 2021), as well as improve its comprehension and adherence to instructions (Chen et al., 2024; Zhang et al., 2024; Dong et al., 2023). This is particularly crucial in our case since the F* community is smaller than common programming languages with scarcer source codes and more sparse documentation. Therefore, we integrate diverse instruction-tuning data pairs in other languages besides synthetic F* data to supplement our dataset and enhance model's capability. This general approach is applicable beyond F*, as it can be extended to other programming languages with limited available data, helping to improve model performance in low-resource scenarios.

4 Project-Level Dataset Synthesis

In this section, we describe how we synthesize more project-level proof generation problems from existing repositories. Project-level verification involves generating or repairing proofs for definitions embedded within complete verification repositories, where correctness depends not only on local function logic but also on broader module-level context—including previously established lemmas, type invariants, module imports, and shared assumptions. Our goal is to generate new problem-solution pairs based on existing programming contexts where the **problem** consists of a type declaration, and the **solution** is the correct F* definition (a proof) that satisfies the given type declaration (See examples in B).

4.1 Generating New Problems

We start the problem-solution pairs generation from a seed dataset, in which each instance has the following structure:

Definition An initial problem-solution pair of a definition in the context.

Context Required context information from existing repositories, such as opened premises and pre-defined definitions, and selected premises which are likely to be used in the body of the definitions(Yang et al., 2023).

Examples A set of semantically similar problem-solution pairs retrieved from the same context using the similarity between types in their embedding space(Chakraborty et al., 2024).

We prompt the language model to create new definitions or prove new properties based on the context in the seed dataset. The detail is listed as follows:

Generation Prompt Curation: We structure the prompt with relevant premises and pre-defined definitions, providing essential context for generating new definitions and proofs. (Prompt see C.3). To guide the model in following F* conventions, we retrieve multiple example definitions with varied structures from the same context while ensuring diversity and discouraging direct copying.

Definition Generation: For each unique context in the seed dataset, we apply the predefined prompt template and sample multiple candidate problem-solution pairs using two LLMs.

Data Filtering: To maintain quality and prevent redundancy, we apply de-duplication by (1) computing sequence similarity and filtering out overly similar definitions, including those resembling reference examples, and (2) remove any generated definitions that overlap with the test set to prevent data leakage and ensure a fair evaluation.

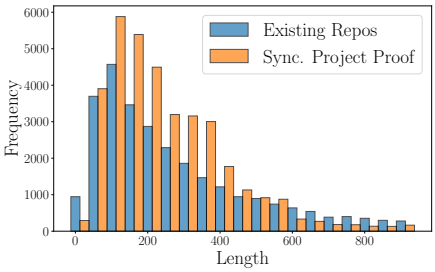

Figure 3: Length Comparison between Generated Definitions vs Existing Definitions.

These steps expand the dataset with diverse problem-solution pairs while maintaining real-world F* relevance and validation feasibility. Figure 3 shows that the problem-solution pair augmentation is effective for simpler definitions but struggles with longer and more complex ones, reflecting the long-tailed distribution observed in both real-world and synthesized datasets. This alignment suggests that our synthetic data captures the difficulty distribution in real F* development.

4.2 Creating Repair Data

In this section, we generate new problem-solution pairs for the proof repair task. The **problem** consists of a type declaration, an incorrect proof, and the corresponding error message from the F* compiler, while the **solution** is the corrected proof. To construct this dataset, we combine rule-based data synthesis with LLM-generated repair data.

4.2.1 Synthetic Repair Data

We generate a synthetic mutation dataset from the F* dataset by randomly modifying ground-truth solutions at the abstract syntax tree level. If a mutation causes type checking to fail, we treat it as a synthetic error and use it to train a model to recover the original solution. Mutations include omitting parts, replacing arguments with underscores, and modifying control structures (e.g., removing branches or let-definitions). These errors mimic those commonly made by human F* programmers, but we avoid mutating identifiers.

4.2.2 Repair Data From The Model

Since Section 4.1 expands the problem set, we now prompt the model to solve these problems within their given contexts, collecting incorrect proofs and their corresponding error messages. Correct answers for the repair task are retrieved either from the original correct proofs or by prompting LLMs to generate new valid proofs.

Repair Problems Generation: We combine the seed dataset problems with the generated ones and prompt the LLM to solve them. Solutions are validated (3.2), and incorrect proofs with error messages are collected from the compiler as repair problems.

Obtaining Correct Repairs: (1) We prompt LLMs to solve the repair problems directly, though zero-shot performance is often limited. (2) Alternatively, we reuse the original correct proofs from the definition generation task as repair solutions.

Data Filtering: (1) Duplicate repair problems are removed by filtering identical incorrect proofs. (2) Each definition generation problem contributes at most three repair problems to prevent redundancy. (3) Correct answers appearing in the test set are removed to ensure fair evaluation.

We can see that the dominating model-generated errors is *identifier not found* and *syntax error*. While syntax errors reflect model's limited understanding of F* grammar, identifier not found errors indicate deeper semantic and type-related challenges that are characteristic of F* language.

5 Function-Level Experiments

We present our experiments on the function-level synthetic data mixtures in this section.

5.1 Setup

Dataset Our dataset primarily comprises synthetic F* data, along with selected data from datasets of other programming languages (CodeAlpaca, Evol-Instruct, Deepseek-Prover-V1, RunbugRun, stack-exchange-preferences). CodeAlpaca collects synthetic coding instructions from ChatGPT following self-instruct, including diverse programming problems and languages (Chaudhary, 2023). Evol-Instruct also contains high-quality synthetic natural language to code data (Wei et al., 2024b). The Deepseek-Prover-V1 (Xin et al., 2024) dataset includes extensive Lean 4 proof data to enhance theorem-proving capabilities in LLMs. RunbugRun provides an executable dataset for automated program repair on commonly used programming languages (Prenner and Robbes, 2023). We conduct experiments to test the impact of the composition of the data and different mixing ratios on F* code.

Model	Pass@1	+Repair
QWEN-2.5-CODER-7B-INSTRUCT	0.25	0.30
QWEN-2.5-CODER-14B-INSTRUCT	0.50	0.55
QWEN-2.5-CODER-32B-INSTRUCT	0.48	0.58
QWEN-2-72B-INSTRUCT	0.34	0.43
DEEPSEEK-CODER-33B-INSTRUCT	0.29	0.38
DEEPSEEK-CODER-V2-LITE-INSTRUCT	0.43	0.53
DEEPSEEK-V3	0.66	0.78
LLAMA-3.1-70B	0.21	0.27
LLAMA-3.3-70B-INSTRUCT	0.17	0.26
GPT-4O	0.60	0.70
Fine-tune Data Mixture		
54K F* Only	0.42	0.47
+ Evol	0.52	0.56
93K F* Only	0.48	0.52
+ DSP-V1	0.52	0.54
+ DSP-V1 + Evol + CodeAlpaca + RBR	0.58	0.62
+ DSP-V1 + Evol + CodeAlpaca + RBR (14B)†	**0.74**	**0.77**
- F NL2Code*	0.48 (-)	0.52(-)

Table 1: Performance comparison across different models and fine-tuning data mixtures. **F* only**: synthetic F* data, Evol: 80K (54K F*) / 50K (93K F*) Magicoder-Evol-Instruct data, **DSP-V1**: 20K Deepseek-Prover-V1 data, **CodeAlpaca**: 15K CodeAlpaca data, **RBR**: 15K RunBugRun data.
†: Adopting Qwen2.5-Coder-14B as base model.

5.2 Synthetic Data Generation Setup

We select code LLMs demonstrating relatively superior capability in generating F* code: Qwen2.5-Coder-32B-Instruct, Qwen2.5-Coder-14B-Instruct (Hui et al., 2024b), CodeLlama-13b-Instruct-hf (Roziere et al., 2023), DeepSeek-Coder-V2-Lite-Instruct (Liu et al., 2024), DeepSeek-R1-Distill-Qwen-32B (Guo et al., 2025). Different prompt templates are adopted in generating different tasks of data. All promptings are done in zero shot. We use a temperature of 0.7 to generate data.

Evaluation Dataset and Metrics We sample 2,000 instructions from synthesized F* dataset as hold-out test set, equally covering all 3 problem types described above. All evaluations are done on the same test set. During evaluation, the model first generates an initial response. If incorrect, we prompt the same model again with the erroneous code and the error message for repair. We record both the initial code generation accuracy and the overall accuracy after a single repair attempt. The response codes are generated with T=0.1.

5.3 Results

Comparing Against Powerful LLMs Compared with 6 popular open-source LLMs: Qwen-2.5-coder-7B-instruct, Qwen-2.5-coder-14B-instruct, Qwen-2.5-coder-32B-instruct (Hui et al., 2024a), Qwen-2-72B-instruct (Hui et al., 2024b), DeepSeek-Coder-V2-Lite-Instruct (Liu et al., 2024), LLaMa-3.1-70B, our model achieves the best performance in both generation and repair within the F* framework. The results in Table 1 demonstrate that in the initial generation, our model significantly outperforms non-code models such as LLaMA-3.1-70B and Qwen-2-72B in terms of accuracy. At the same time, the accuracy of the generation also surpasses that of code models with larger parameter sizes such as Qwen-2.5-coder-32B-instruct, indicating that our instruction-tuning dataset is highly effective in enhancing the model's ability to generate F* data. Our initial generation accuracy is also comparable to GPT-4o, which is generally challenging given the size of its base model parameters.

Benefits of data diversity and the effect of different data mixtures As shown in Table 1, data diversity has a positive impact on the model's performance. When more diverse language data (e.g. data from Evol-Instruct & Deepseek-Prover-V1) is added to the F* synthetic dataset, the model's accuracy on the F* validation set significantly improves, regardless of the amount of original F* synthetic dat (2) Within a certain range, more diverse data leads to better model performance. (3) More high-quality synthetic data indeed leads to better model performance, which suggests that for languages like F*, where the model's knowledge is still limited, increasing the amount of high-quality language-specific fine-tuning data is beneficial for improving the model's performance.

6 Project-Level Proof Synthesis

In this section, we present the experiments on project-level proof synthesis tasks.

6.1 Training

Training Dataset Both the definition generation and repair datasets are formatted using predefined prompt templates, which are listed in Appendix C. The prompt settings for definition generation and repair are listed in A.2.

Next we integrated the formatted existing and repository-level definition generation data, model-generated and synthetic proof repair data, and the mixed synthetic Function-Level data, and prepared different data mixtures to dine the model, allowing us to explore the influence of each dataset and

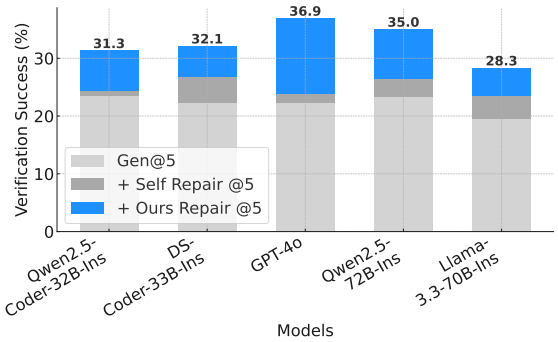

Figure 4: PoPilot repairing failed outputs for state-of-the-art models.

develop a best training corpus.

We detail the training set-up for PoPilot in A.3.

6.2 Validation

After expanding the problem-solution pairs for both definition proof and repair tasks, we apply execution-based validation (detail in 3.1.2) to filter out incorrect solutions and store error messages for the repair dataset and enhance the diversity of the data. Since our data generation process involves multiple LLMs and stages, and diversity is limited by the fixed number of seed contexts and reference examples, we apply an additional de-duplication step. Specifically, we limit each unique type declaration to at most two instances in the repair dataset, ensuring diversity while minimizing redundancy.

6.3 PoPilot Auto-pilots and Co-pilots

In this section, we evaluate the supervised fine-tuned models using different training data mixtures as well as baseline models on the task of proof generation and proof repair. Our evaluation data is a random-sampled, held-out test set with 1K repository -level definition proof problems. The validation process consists of the following three stages:

Definition Generation and Self-Repair: The model generates F* proofs given a type declaration and context (see example prompt in C), sampling either 5 times followed by 5 repair attempts (Sample-5 + Repair-5) or 10 times directly (Sample-10). If at least one proof compiles, the problem is solved. Otherwise, for the Sample-5 group, failed attempts and error messages are stored for repair, where the model randomly selects an incorrect response and generates 5 additional repair attempts to fix the incorrect proofs. This setup allows us to compare whether self-repairing incorrect generations improves success rates over simply increasing the sampling budget.

Repair Using Outputs from Other Models: To further assess model performance, we use incorrect proofs generated by a baseline model as input to our fine-tuned model. This allows us to evaluate whether fine-tuning enhances the model's ability to generalize and improve proof repair across different model outputs.

6.4 Result

We evaluate fine-tuning performance using **Generate@5** and **Repair@5** metrics. Generate@5 represents the correctness rate when sampling the model five times on the proof generation task, while Repair@5 measures the number of additional questions correctly repaired after sampling five repair attempts on incorrect solutions from the previous stage. As shown in Table 2, fine-tuning QWEN2.5-CODER-14B on our generated dataset outperforms five larger state-of-the-art coding LLMs in both repository-level proof generation and self-repair tasks.

Proof Repairing Ability: We use the model fine-tuned with the data mixture Existing Repository-Level Definitions + Synthetic Project Proofs + All Repair dataset (see Table 2) to repair incorrect outputs from the baseline models. The results in Figure 4 demonstrate its effectiveness in repairing out-of-distribution incorrect proofs. Notably, our fine-tuned model surpasses all baseline large models in correcting their own incorrect answers, bringing a significant improvement in the total number of correct solutions. This suggests a possible application where a fine-tuned, smaller model serves as a debugging assistant for larger models, allowing for potentially efficient co-serving of these powerful models to efficiently adapt to the domain of proof oriented programming.

Data Mixture Analysis: Table 2 shows how different fine-tuning data mixtures impact proof generation and repair. (1) Augmenting the existing repository-level set with model-generated definitions significantly improves Generate@5 and Repair@5 accuracy, demonstrating the effectiveness of our generated definitions. (2) Rule-based synthetic repair data alone does not enhance repair performance, suggesting that rule-based errors may not accurately reflect the error situations the model would have encountered. (3) Model-generated re-

Baseline Models	Generate@5	Repair@5	Gen+Rep (Total 10)	Generate@10
QWEN2.5-CODER-7B-INSTRUCT	23.4	0.2	23.6	26.3
QWEN2.5-CODER-14B-INSTRUCT	24	0.4	24.4	26.9
QWEN2.5-CODER-32B-INSTRUCT	24.2	2.5	26.7	27.1
DEEPSEEK-CODER-V2-LITE-INSTRUCT	24.4	0.7	25.1	25.1
DEEPSEEK-V3	18.7	3.6	22.3	28.6
DEEPSEEK-CODER-33B-INSTRUCT	22.3	4.6	26.9	28.8
GPT-4O	22.2	1.7	23.9	23.8
QWEN2.5-72B-INSTRUCT	23.4	3.0	26.4	25.8
LLAMA-3.3-70B-INSTRUCT-TURBO	19.6	3.9	23.5	21.6
LLAMA-3.1-70B	19.3	2.3	21.6	22.4
Data Mixture				
Existing Repos	30.7	1.0	31.7	35.3
+ Syn. Project Proof	32.2	2.2	34.4	36.2
+ Func + Syn. Project Proof	32.8	2.7	35.5	37.8
+ Syn. Project Proof + Syn. Repair	32.7	0.7	33.4	37.5
+ Syn. Project Proof + Model Repair	33.1	4.2	37.3	37.2
+ Syn. Project Proof + All Repair	**34.0**	4.7	38.7	38.0
POPILOT	33.0	**6.4**	**39.4**	**38.5**

Table 2: Performance comparison of baseline models and fine-tuning data configurations. **Existing Repos**: 30K existing repository level definition + proofs from the seed dataset; **Syn. Project Proof** : 30K model generated new definitions + proofs as described in 4.1; **Func**: synthetic simple questions mixed with other datasets in 5.2; **Syn. Repair**: 30K synthetic repair data in 4.2.1, **Model Repair**: 30K model-generarted repair data in 4.2.2; **All Repair**: Syn. Repair + Model Repair; **PoPilot**: Existing Repos + Syn. Project Proof + All Repair + 180K mixed function-level coding data used to finetune the best performance in Table 1

pair data improves both repair accuracy (from 32.2 to 33.1) and generation accuracy (from 2.7 to 4.2), indicating that language-model-produced errors and repairs can better capture real-world failure patterns than the synthetic ones, that are largely synthetic, generated by mutating ASTs of the program. (4) PoPilot trained with **Existing Repos + Syn. Project Proof + All Repair + Mixed Function-Level Data** – our most diverse mixture of data points – gives the highest Gen+Rep (39.4) and Generate@10 (38.5) across the board, confirming that diverse data improves both proof generation and repair. The function-level synthetic data for F* and mixture from other languages well complement the repository-level F* verification and repair data, and boosts the model's performance.

7 Related Work

Language Models For Code Large language models (LLMs) have advanced in code generation (Chen et al., 2021; Austin et al., 2021), program repair (Xia and Zhang, 2022; Xia et al., 2023; Jin et al., 2023), and software engineering tasks like issue fixing (Jimenez et al., 2023) and testing (Deng et al., 2023). Open-source models (e.g., Qwen2.5-Coder (Hui et al., 2024b), Deepseek-Coder (Guo et al., 2024)) and closed-source models (e.g., GPT-4o (Hurst et al., 2024)) undergo **pre-training** on large-scale code datasets (Radford, 2018; Nijkamp et al., 2022), followed by **post-training** via instruction fine-tuning (Muennighoff et al., 2023; Roziere et al., 2023; Luo et al., 2023; Chaudhary, 2023) or reinforcement learning (Ouyang et al., 2022b; Bai et al., 2022). While these models excel in common languages like Python and C++, proof-oriented languages such as F* (Swamy et al., 2011) remain underrepresented, limiting their effectiveness in proof synthesis.

Language Models For Formal Proof Formal theorem proving and proof repair offer an appealing domain for unlocking the reasoning potential of LLMs, with proofs being easier to verify rigorously without hallucination(Yang et al., 2023), in both mathematical theorem proving and formal program verification. Currently, Language models have shown capability in formal languages such as Isabelle(Jiang et al., 2022; Wang et al., 2023a; Zhao et al., 2023) and Lean (Polu et al., 2022; Han et al., 2021; Yang et al., 2023). Researchers have also explored various approaches to optimize automated theorem proving using large language models: employing retrieval-augmented assistance(Yang et al., 2023), improving search efficiency by dynamically allocating computational resources (Wang et al., 2023b), predicting the progress of proofs(Huang et al., 2025), employing LLMs as copilots that as-

sist humans in proving theorems(Song et al., 2024; Kozyrev et al., 2024), and introducing synthetic data during training(Wang and Deng, 2020; Xin et al., 2024; Lin et al., 2025; Wu et al., 2024). However, most of these efforts focus on mathematical domains rather than repository-level software verification, which is addressed by PoPilot.

Synthetic Data for Instruction Tuning Instruction fine-tuning improves LLMs' ability to follow instructions and relies on high-quality datasets (Zhou et al., 2024; Wang et al., 2022). Since human-annotated datasets are costly (Ouyang et al., 2022a; Köpf et al., 2024; Zheng et al., 2024), recent methods focus on LLM-generated instruction data (Wang et al., 2022; Gunasekar et al., 2023; Wang et al., 2024; Xu et al., 2024). Self-Instruct (Wang et al., 2022) pioneered this approach, later extended by Alpaca (Taori et al., 2023) and Code Alpaca (Chaudhary, 2023). Evol-Instruct (Xu et al., 2023; Ahn et al., 2024) and Code Evol-Instruct (Luo et al., 2023) introduced multi-stage generation for better instruction diversity, though risks of reinforcing biases remain (Yu et al., 2024). OSS-INSTRUCT (Wei et al., 2024b) and SelfCodeAlign (Wei et al., 2024a) mitigate this by leveraging open-source data, while MultiPL-T (Cassano et al., 2024) enables cross-lingual instruction transfer.

8 Conclusion

In this work, we propose a synthetic data recipe for instruction-tuning code language models to become proficient proof-oriented programmers in F* under extreme data scarcity. By synthesizing function-level F*, diversifying with other programming languages and tasks and generating new verification tasks on a repository level, we build a powerful PoPilot that outperforms powerful language models, even GPT-4o with only 14B parameter. We further show that PoPilot can work together with existing code LMs to improve their proof-oriented programming capabilities by large margins. More broadly, we present a variable path for low-resource programming languages and verification tools, lowering the barrier to adopting formal verification in real-world software development.

Limitations This work focuses on the programming language of F*, but did not experiment with other languages due to their lack of well-established evaluation suite.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. *arXiv preprint arXiv:2402.00157*.

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

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.

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. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708.

Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. 2024. Towards neural synthesis for smt-assisted proof-oriented programming. *arXiv preprint arXiv:2405.01787*.

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

Hao Chen, Abdul Waheed, Xiang Li, Yidong Wang, Jindong Wang, Bhiksha Raj, and Marah I Abdin. 2024. On the diversity of synthetic data and its impact on training large language models. *arXiv preprint arXiv:2410.15226*.

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*.

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

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang,

Zheng Yuan, Chang Zhou, and Jingren Zhou. 2023. How abilities in large language models are affected by supervised fine-tuning data composition. *arXiv preprint arXiv:2310.05492*.

Allen Geng. 2022. ocamlgithub. https://huggingface.co/datasets/AllenGeng/ocamlgithub.

Ian Grout. 2008. *Digital Systems Design with FPGAs and CPLDs*. Newnes, USA.

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

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. 2021. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jian Hu, Xibin Wu, Weixun Wang, Dehao Zhang, Yu Cao, et al. 2024. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*.

Suozhi Huang, Peiyang Song, Robert Joseph George, and Anima Anandkumar. 2025. Leanprogress: Guiding search for neural theorem proving via proof progress prediction. *arXiv preprint arXiv:2502.17925*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024a. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024b. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656.

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi Rui Tam, Keith Stevens, Abdullah Barhoum, Duc Nguyen, Oliver Stanley, Richárd Nagyfi, et al. 2024. Openassistant conversations-democratizing large language model alignment. *Advances in Neural Information Processing Systems*, 36.

Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. Coqpilot, a plugin for llm-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2382–2385.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. 2025. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

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

Microsoft. 2024a. The dafny programming and verification language. https://dafny.org.

Microsoft. 2024b. F*: A proof-oriented programming language. https://fstar-lang.org.

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

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

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. 2022a. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022b. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2022. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*.

Julian Aron Prenner and Romain Robbes. 2023. Runbugrun–an executable dataset for automated program repair. *arXiv preprint arXiv:2304.01102*.

Alec Radford. 2018. Improving language understanding by generative pre-training.

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

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2024. Towards large language models as copilots for theorem proving in lean. *arXiv preprint arXiv:2404.12534*.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. *ACM SIGPLAN Notices*, 46(9):266–278.

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

Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, et al. 2023a. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*.

Haiming Wang, Ye Yuan, Zhengying Liu, Jianhao Shen, Yichun Yin, Jing Xiong, Enze Xie, Han Shi, Yujun Li, Lin Li, et al. 2023b. Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12632–12646.

Mingzhe Wang and Jia Deng. 2020. Learning to prove theorems by learning to generate theorems. *Advances in Neural Information Processing Systems*, 33:18146–18157.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Zifeng Wang, Chun-Liang Li, Vincent Perot, Long T Le, Jin Miao, Zizhao Zhang, Chen-Yu Lee, and Tomas Pfister. 2024. Codeclm: Aligning language models with tailored synthetic data. *arXiv preprint arXiv:2404.05875*.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024a. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024b. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. *arXiv preprint arXiv:2410.15700*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE.

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. 2024. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*.

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

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024. Magpie: Alignment data synthesis from scratch by prompting aligned llms with nothing. *arXiv preprint arXiv:2406.08464*.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612.

Yue Yu, Yuchen Zhuang, Jieyu Zhang, Yu Meng, Alexander J Ratner, Ranjay Krishna, Jiaming Shen, and Chao Zhang. 2024. Large language model as attributed training data generator: A tale of diversity and bias. *Advances in Neural Information Processing Systems*, 36.

Dylan Zhang, Justin Wang, and Francois Charton. 2024. Only-if: Revealing the decisive effect of instruction diversity on generalization. *arXiv preprint arXiv:2410.04717*.

Xueliang Zhao, Wenda Li, and Lingpeng Kong. 2023. Decomposing the enigma: Subgoal-based demonstration learning for formal theorem proving. *arXiv preprint arXiv:2305.16366*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *Preprint*, arXiv:2309.11998.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36.

A Experiment Setting

A.1 Function-level Data Experiment Setting

We finetune Qwen2.5-Coder-7B on our dataset for one epoch using one NVIDIA A100-40GB GPU. The initial learning rate is set at 5e-6 while the batch size is set at 256. During finetuning, we adopt the OpenRLHF (Hu et al., 2024) library and modules, applying LoRA (Hu et al., 2021) with a rank of 32 and an alpha of 32.

A.2 Prompt Setting

For **definition generation prompt**, we will provide the type declaration, set the max prompt length to 4096 tokens, and use all the opened modules and premises, 15 selected premises the model may use, and 10 retrieved related examples. For the **repair prompt**. the problem also includes the incorrect solution and its corresponding error message to guide the model in learning error correction, but excludes the selected premises since (Chakraborty et al., 2024) discovers that the premises has limited effect on proof repairing. We also use fewer related examples in repair prompts to limit the prompt length

A.3 Model-generate Data Experiment Setting

We train Qwen2.5-Coder-14B using supervised fine-tuning for one epoch using 4507 × NVIDIA A100-40GB GPU, with learning rate508 1e-5 and batch size 64. We adopt OpenRLHF (Hu509 et al., 2024), applying LoRA (Hu et al., 2021) with510 a rank of 32 and an alpha of 32.

B Problem-Solution Pair Examples

Example 1

```
val clientCertTypeExtension_serializer:
    LP.serializer
    c   lientCertTypeExtension_parse

let clientCertTypeExtension_serializer =
    LP.serialize_vlarray
    1 255 certificateType_serializer
    1 255 ()
```

Example 2

```
val
clens_uncompressedPointRepresentation32_x:
LL.clens
uncompressedPointRepresentation32
  uncompressedPointRepresentation32_X

let clens_uncompressedPointRepresentation32
_x
: LL.clens
uncompressedPointRepresentation32
uncompressedPointRepresentation32_X = {
  LL.clens_cond = (fun _ -> True);
  LL.clens_get = (fun x -> x.x);
}
```

C Prompt Templates

C.1 Definition Generation Prompt Example

```
You are tasked with F* code generation.
You will be given a type declaration,
and you need to write a definition for
it.
## Type declaration:
val tau: Prims.unit -> Tac unit

 1. Write the definition that satisfies the
    above type.
 2. Start the definition with ``` let tau ``` .
 3. Only write in F* code.
 4. Add END token after completing the
    definition.

## Possibly useful premises:

 FStar.Tactics.Effect.raise
 FStar.Pervasives.reveal_opaque
 FStar.Tactics.Effect.get
 FStar.Tactics.Effect.tactic
 FStar.Pervasives.Native.snd
 FStar.Pervasives.Native.fst
 FStar.Monotonic.Pure.
    elim_pure_wp_monotonicity
 FStar.Tactics.Types.issues
 FStar.Pervasives.dfst
 FStar.Pervasives.dsnd
 FStar.Tactics.Effect.tac_return
 FStar.Monotonic.Pure.
    elim_pure_wp_monotonicity_forall
 FStar.Tactics.Effect.tac
 FStar.Monotonic.Pure.
    intro_pure_wp_monotonicity
 Prims.l_True

## Already opened files and delared modules

 open FStar
 open FStar.Pervasives
 open Prims
 open FStar.Tactics.V2

## Related types and definitions

val tau: Prims.unit -> Tac unit
let tau () : Tac unit =
    apply_lemma (`refl)
val tau: Prims.unit -> Tac unit
let tau () : Tac unit =
    let * = implies*intro () in
    let * = implies*intro () in
    let * = implies*intro () in
    let b = implies_intro () in
    var_retype b; // call retype,
    get a goal `p == ?u`
    let pp = `p in
    let rr = `r in
    grewrite pp rr; // rewrite p to q,
    get `q == ?u`
    trefl (); // unify
    apply_lemma (`l); //prove (p == q),
    asked by grewrite
    let e = cur_env () in
    match vars_of_env e with
    | [_;_;_;b] ->
        let t = type_of_binding b in
        let t = norm_term [] t in
        // contains uvar redexes.
        if FStar.Order.ne
        (compare_term t rr)
      then fail "binder was not retyped?"
        else ();
        apply_lemma (`l2);
        assumption' ();
        qed ()
    | _ ->
        fail "should be impossible"


Write your response below.
```

C.2 Repair Prompt Example

````
You are tasked with F* code generation.
You will be given a type declaration,
and an incorrect student solution.
You need to produce a correct solution.

## Type declaration:

val clientHelloExtension_e_session
_ticket_clens:
    LL.clens clientHelloExtension_
    e_session_ticket
  sessionTicket

 1. Write the definition
    that satisfies the above type.
 2. Start the definition
    with
    ```
let clientHelloExtension
_e_session_ticket_clens
    ``` .
 3. Only write in F* code.
 4. Add END token after completing
    the definition.



## Already opened files and delared modules

 open MiTLS.Parsers.SessionTicket
 open Prims
 open FStar.Bytes
 open MiTLS.Parsers
 open FStar.Pervasives
 open FStar



## Related types and definitions

val newSessionTicketExtension_clens'
_session_ticket:
LL.clens newSessionTicketExtension
  newSessionTicketExtension_e_default
let
newSessionTicketExtension_clens'
_session_ticket
: LL.clens newSessionTicketExtension
newSessionTicketExtension_e_default =
LL.clens_dsum_payload
newSessionTicketExtension_sum
(LL.Known
(known_extensionType_as_enum_key
Session_ticket))

val newSessionTicketExtension_clens'
_client_certificate_type:
LL.clens newSessionTicketExtension
  newSessionTicketExtension_e_default
let newSessionTicketExtension_clens'
_client_certificate_type :
LL.clens newSessionTicketExtension
newSessionTicketExtension_e_default
= LL.clens_dsum_payload
newSessionTicketExtension_sum
(LL.Known (known_extension
Type_as_enum_key
Client_certificate_type))



## Student Solution

@@ Student F* Code
```fstar
open FStar
open Prims
open FStar.Pervasives
open MiTLS.Parsers
open MiTLS.Parsers
open FStar.Bytes
module U8=FStar.UInt8
module U16=FStar.UInt16
module U32=FStar.UInt32
module U64=FStar.UInt64
module LP=LowParse.Spec.Base
module LS=LowParse.SLow.Base
module LSZ=LowParse.SLow.Base
module LPI=LowParse.Spec.AllIntegers
module LL=LowParse.Low.Base
module L=FStar.List.Tot
module B=LowStar.Buffer
module BY=FStar.Bytes
module HS=FStar.HyperStack
module HST=FStar.HyperStack.ST
open MiTLS.Parsers.SessionTicket
open MiTLS.Parsers.
ClientHelloExtension_e_session_ticket
#push-options "--initial_fuel 2 -
-max_fuel 8 --initial_ifuel 1
--max_ifuel 2 --smtencoding.elim_box false
--smtencoding.nl_arith_repr boxwrap
--smtencoding.l_arith_repr boxwrap
````
````

```
--smtencoding.valid_intro true
--smtencoding.valid_elim false
--z3rlimit 5 --z3rlimit_factor
1 --z3seed 0"

#restart-solver
val
clientHelloExtension_e
_session_ticket_clens
:LL.clens
clientHelloExtension_e_session_ticket
// Error Range Start - Line 27
  sessionTicket
// Error Range End - Line 27
let
clientHelloExtension_e
_session_ticket_clens
:LL.clens sessionTicket =
  {
    LL.clens_cond = (fun _ -> True);
    LL.clens_get
    =
    (fun (x:
   clientHelloExtension_e_session_ticket)
    -> (x <: sessionTicket))
  }

@@ Error Message
  - Expected type "Type"; but
  "LL.clens sessionTicket"
  has type "t2: Type -> Type"


  - Expected type "Type";
  but "LL.clens sessionTicket"
  has type "t2: Type0 -> Type"


  - Expected type
  "LL.clens sessionTicket";
  but "LL.Mkclens (fun _ -> l_True)
  (fun x -> x <: sessionTicket)" has type
  "LL.clens
  clientHelloExtension_e_session_ticket
  sessionTicket"


  - Expected type
  "LL.clens
  clientHelloExtension_e_session_ticket
  sessionTicket";
  but "LL.Mkclens (fun _ -> l_True)
    (fun x -> x <: sessionTicket)
    <: LL.clens sessionTicket"
    has type "LL.clens
    sessionTicket"


Write your response below.
```

C.3 New Definition Prompt Example

```
You are tasked with generating F* code.
You will be given some premises,
opened modules and some example type
declarations and definitions. Your
goal is to write a different type
declaration and a corresponding
definition that satisfies the
type declaration.

You can use the information
provided in the following sections
to help you construct the new
type declaration and definition.
Here's how you can use each section:

Possibly useful premises:

This section lists modules, types,
or functions that might be helpful.
Consider incorporating them into
your definition if appropriate.

Already opened files and declared
modules:

The modules listed here
are already opened in the context.
You can use definitions from these
modules directly without needing
to prefix them with the module name.

Example type declarations and
definitions:

This shows some examples
of how a definition satisfying a
similar type declaration can be
written. Each example is delimited
using "```". Use this as a reference
for the structure and style,
but do not use the examples
definitions in your answer.
```

Use the information provided to
write one new type declaration
and a definition that satisfies
this new type declaration.
Only write in F* code, you don't
need to provide any explanation or
example.
Start your new type declaration
and definition with "val" and "let"
respectively, and add END after
completing the definition.
You should only use the premises
from "Possibly useful premises".

Possibly useful premises:

FStar.Tactics.Effect.raise
FStar.Pervasives.Native.fst
FStar.Pervasives.Native.snd
FStar.Tactics.Types.issues
FStar.Tactics.Effect.get
FStar.Pervasives.dfst
FStar.Pervasives.dsnd
GradedMonad.monoid_nat_plus
GradedMonad.st
FStar.Pervasives.st_post_h
FStar.Pervasives.reveal_opaque
FStar.Issue.mk_issue
FStar.Pervasives.st_post_h'
FStar.Pervasives.st_pre_h
FStar.Monotonic.
Pure.elim_pure_wp_monotonicity

Already opened files
and declared modules

```
open FStar.Pervasives
open FStar
open Prims
```

Example definitions

```
val st_monad (s: _)
: monad (st s)
instance st_monad s
: monad (st s) =
{
   return = (fun
   #a (x:a) ->
   (fun s -> x, s));
   bind   =
   (fun #a #b (f: st s a)
   (g: a -> st s b) (s0:s) ->
           let x, s1 = f s0 in
           g x s1);
}
```

```
val monad_functor (#m: _)
(d: monad m) : functor m
instance monad_functor #m
(d : monad m) : functor m =
  { fmap = (fun #_ #_ f x
  -> bind #m x (fun xx
  -> return #m (f xx))); }
```

```
val FStar.DM4F.MonadLaws.st
= s: Type -> a: Type -> Type
let st (s:Type) (a:Type)
= s -> Tot (a * s)
```

```
[@@ FStar.Tactics.
Typeclasses.tcinstance]
val opt_monad:monad option
instance opt_monad :
monad option =
{
   return = (fun #a
   (x:a) -> Some x);
   bind = (fun #a
   #b (x:option a)
   (y: a -> option b) ->
         match x with
         | None -> None
         | Some a -> y a)
}
```

Write your response below.

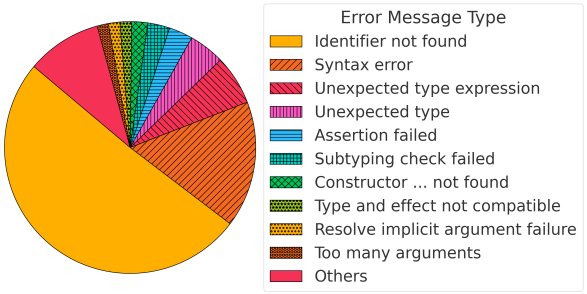

Figure 5: Distribution of Top 10 Error Types of Model-Generated Repair Data.

Model	Gen@5	sample1-on-5	sample5-on-1
OUR BEST MODEL	34	+1.7	+4.7
QWEN2.5-CODER-32B	23.5	+0.8	+7.8
DS-CODER-33B	22.3	+4.6	+9.8

Table 3: Comparison of repair sampling strategies: **sample1-on-5** repairs each incorrect solution once, while **sample5-on-1** repairs the same incorrect solution multiple times.

D Ablation Study

D.1 Repair Sampling Strategy Ablation study

In the self-repair experiment, we tested an alternative strategy: sampling 5 incorrect solutions and attempting to repair each once (totaling 5 repair attempts). However, this approach performed significantly worse than the Sample 1 & Repair 5 strategy, so we chose the latter in the following experiments. The comparison of both strategies on three models is shown in Table 3.

We hypothesize that proof repair remains a challenging task, even for our fine-tuned model. Allowing multiple repair attempts on the same problem increases the chances of success, leading to better overall accuracy.

D.2 Fine-tune on smaller model

We used the same data of PoPilot to fine-tune a Qwen/Qwen2.5-Coder-7B model, the result is shown in table 4 with comparison to PoPilot and the baseline models.

Given the small size of PoPilot, the performance is reasonably good. However, the repair capability is still minor, so we will just report PoPilot in our main part.

Baseline Models	Generate@5	Repair@5	Gen+Rep (Total 10)	Generate@10
QWEN2.5-CODER-32B-INSTRUCT	23.5	0.8	24.3	27.1
DEEPSEEK-CODER-33B-INSTRUCT	22.3	4.6	26.9	28.8
GPT-4O	22.2	1.7	23.9	23.8
QWEN2.5-72B-INSTRUCT	23.4	3.0	26.4	25.8
LLAMA-3.3-70B-INSTRUCT-TURBO	19.6	3.9	23.5	21.6
Data Mixtures (*Qwen/Qwen2.5-Coder-7B*)				
Existing Repos	12.9	3.6	16.5	18.5
+ Syn. Project Proof	14.2	3	17.2	19.8
+ Func + Syn. Project Proof	14	3.7	17.7	18.7
+ Syn. Project Proof + Syn. Repair	13.9	3.8	17.7	18.7
+ Syn. Project Proof + Model Repair	15.2	4.4	19.6	20.8
+ Syn. Project Proof + All Repair	15	4.7	19.7	21.2
POPILOT-SMALL	21.9	3.9	25.8	29.2
POPILOT	**33.0**	**6.4**	**39.4**	**38.5**

Table 4: Performance comparison of the small model