Towards Neural Synthesis for SMT-Assisted Proof-Oriented Programming

Saikat Chakraborty[§], Gabriel Ebner[§], Siddharth Bhat[†*], Sarah Fakhoury[§],
Sakina Fatima[‡*], Shuvendu Lahiri[§], Nikhil Swamy[§]
[§]Microsoft Research, Redmond, WA, USA
[†]University of Cambridge, Cambridge, UK, [‡]University of Ottawa, Ottawa, ON, Canada
{saikatc, gabrielebner, sfakhoury, shuvendu, nswamy}@microsoft.com
[†]sb2743@cam.ac.uk, [‡]sfati077@uottawa.ca

Abstract—**Proof-oriented programs mix computational content with proofs of program correctness. However, the human effort involved in programming and proving is still substantial, despite the use of Satisfiability Modulo Theories (SMT) solvers to automate proofs in languages such as F*.**

Seeking to spur research on using AI to automate the construction of proof-oriented programs, we curate a dataset of 600K lines of open-source F* programs and proofs, including software used in production systems ranging from Windows and Linux, to Python and Firefox. Our dataset includes around 32K top-level F* definitions, each representing a type-directed *program and proof synthesis* problem—producing a definition given a formal specification expressed as an F* type. We provide a program-fragment checker that queries F* to check the correctness of candidate solutions. We believe this is the largest corpus of SMT-assisted program proofs coupled with a reproducible program-fragment checker.

Grounded in this dataset, we investigate the use of AI to synthesize programs and their proofs in F*, with promising results. Our main finding in that the performance of fine-tuned smaller language models (such as Phi-2 or StarCoder) compare favorably with large language models (such as GPT-4), at a much lower computational cost. We also identify various type-based retrieval augmentation techniques and find that they boost performance significantly. With detailed error analysis and case studies, we identify potential strengths and weaknesses of models and techniques and suggest directions for future improvements.

Index Terms—**Proof Oriented Programming, AI for Proofs, Trustworthy AI programming**

I. Introduction

The recent excitement around AI-assisted programming has been tempered by concerns around the trustworthiness of AI-generated code [1]. Languages that offer static guarantees can help reduce some of these concerns, e.g., having AI generate safe Rust code rather than C eliminates the risk of AI-introduced memory safety issues. Taking this line of thinking to its limit, using AI to generate code in *proof-oriented languages* which allow programs to be augmented with specification and proofs of correctness could eliminate trust in AI-generated code, so long as the specification can be audited to match a human's intent. Conversely, proof-oriented languages often require a high-level of human expertise—AI assistance could help make them easier to use.

[*] Work done as interns at Microsoft Research

Recognizing the potential dual benefit (i.e., trustworthy AI programming & easier program proof) many researchers have begun investigating using AI to synthesize proofs. However, most of the prior work has focused on using AI with tactic-based proof assistants, such as Coq, Lean, and Isabelle [2]–[5], including projects like CoqGym [6], which builds models based on hundreds of Coq projects containing more than 70,000 tactic scripts, and LeanDojo [7], which uses math-lib [8], a large corpus of formalized mathematics in Lean. AI automation for proof-oriented programming languages like F* [9], Dafny [10], Viper [11], Verus [12] and others has received comparatively less attention. The prior work [13]–[18] has been limited by the availability of data, focusing instead on small, hand-crafted problem sets numbering in the few hundreds. This is unfortunate, since proof-oriented languages may be close to the limit of symbolic automation provided by SMT solvers and AI automation could open the door to further automation that has remained out of reach. Additionally, to achieve the promise of trustworthy AI programming, we believe it is essential to develop AI for *program and proof synthesis* rather that only on mostly mathematical tactic-based proofs.

Towards this end, our paper makes the following three major contributions:

1. A new dataset: Aiming to spur research in AI-assisted proof-oriented programming, our first main contribution is FStarDataSet, a dataset of F* programs and proofs extracted from 2060 source files, representing about 600K lines of source code, drawn from 8 open source projects. The dataset provides around 32K top-level F* definitions, coupled with tooling that allows each definition to be checked in isolation. We believe this is the largest dataset of SMT-assisted proof-oriented programs and we envision for it to be a live, evolving data set, with new projects added to it over time. Indeed, even in the period of preparing this paper, FStarDataSet has grown to include 4 additional projects reaching 940K lines of source code.[1] Although we currently focus on F*, we hope for the dataset to also grow to include data from other proof-oriented languages. We describe the dataset in detail in §III.

[1]For the latest version of the dataset and results, we refer to the online appendix [19] of this paper.

2. A benchmark problem: Grounded in this data, we design a straightforward benchmark: *given a type as a formal specification, synthesize a program that has that type*. Each of the 32K definitions in FSTARDATASET yields its own synthesis problem, where the type of the definition is the "goal" type, and a technique is expected to synthesize a definition that the F* compiler attests is goal-type-correct. In F*, types are rich enough to capture a variety of specifications, ranging from simple types as in other functional programming languages, to dependently typed specifications that capture functional correctness properties of a program, i.e., types can represent arbitrary propositions. Dually, programs in F* contain computational content (e.g., logic for reversing a list), but they can also be interpreted as proofs. As such, our benchmark can be seen as an instance of a type- or specification-directed program & proof synthesis problem. We present a simple and objective taxonomy to classify benchmark instances, distinguishing simply typed, dependently typed, and fully specified proofs, corresponding roughly to the precision of specifications.

3. Designing and evaluating neural synthesis techniques: We apply a (by now) standard regime of prompting large language models (LLMs) to generate solutions, backed by retrieval augmentation and fine-tuning techniques specific to our setting. In particular, we construct a prompt by augmenting the goal type with related types and definitions from the program context, based on various embedding models. In §V, we evaluate the performance of off-the-shelf large language models, including GPT-3.5 [20] and GPT-4 [21], as well as fine-tuned smaller models include Phi2-2.7B [22], Orca2-7B [23], and StarCoder-15B [24], with the following main takeaways.

- Fine tuned smaller models can match or outperform large language models (§V-A).
- Different classes of problems are solved with varying degrees of success, with common error modes differing between pretrained and fine-tuned models (§V-B).
- Leveraging the contexts as well as retrieval augmentation significantly boosts the quality of results (§V-C).

Based on our results, we are optimistic about for the future of AI-assisted proof-oriented programming. Researchers building verified systems in proof-oriented languages have reported writing around 3-5 lines of proof for each line of verified code [25], [26], a considerable overhead, despite strong symbolic automation from SMT solvers. For SMT-assisted proof-oriented programs, we provide the first, substantial empirical evidence that LLMs trained on corpora like FSTARDATASET, and prompted with retrieval augmentation, can automate somewhere between a third and a half of proofs. That said, our approach focuses on synthesizing program and proof fragments given their specifications: finding the right modular decomposition to prove a program correct, with the right specifications and auxiliary lemmas, is not yet within the scope of the techniques we explore. §VI provides further discussion and analysis. We release FSTARDATASET in `https://huggingface.co/datasets/microsoft/FStarDataSet`. The source code for training and evaluation is available at `https://github.com/microsoft/PoPAI-FStar`.

II. BACKGROUND

We start by providing some general background on F*, adapted from its online manual. F* is a dependently typed programming language and proof assistant. It encourages *proof-oriented programming*, a paradigm in which one co-designs programs and proofs which attest various aspects of a program's correctness, including its input/output behavior together with its side effects, if any.

The core design philosophy of F* is that the type of a program is a specification of its runtime behavior. Many terms can have the same type and the same term can have many types, e.g., `e : int` states that the term or program fragment `e` reduces to an integer; and `e : nat` states that `e` reduces to a non-negative integer, where $\texttt{nat} = \texttt{x:int}\{\texttt{x} \geq 0\}$ is a *refinement* of type `int`. When proving a program `e` correct, one specifies the properties one is interested in as a type `t` and then tries to convince F* that `e` has type `t`, i.e., deriving `e : t`.

Many dependently typed languages have the same core design, however F* is distinctive in that in addition to several built-in type-checking algorithms, it uses the Z3 SMT solver [27] to try to automatically prove various obligations that arise during type-checking. For example, under the hypothesis that `x : even`, proving that `x + 2 : even`, where $\texttt{even} = \texttt{x:int}\{\texttt{x \% 2 == 0}\}$, results in a query to Z3 of the form $\forall(\texttt{x:int}).\ \texttt{x \% 2 == 0} \implies \texttt{(x + 2) \% 2 == 0}$. While in general the queries F* presents to Z3 are undecidable, in practice Z3 successfully automates a large number of the queries it receives. However, when Z3 proof search fails, the programmer must write additional lemmas and other proofs hints to decompose the proof into smaller pieces that can be checked by F* and Z3.

The concrete syntax of F* is based on other languages in the ML family, including OCaml and F#. Shown below is a recursive implementation of Quick Sort, together with its specification and proof of correctness. The type of `sort` states that for any total order `f` on elements of type α, given an input list `l:list` α, `sort` is a total function (i.e., it always terminates) returning a list `m:list` α that is sorted according to `f` and where `m` is a permutation of `l`. The predicates like `total_order`, `sorted`, `is_permutation` etc. are other auxiliary definitions in scope. The implementation of `sort` mixes the computational behavior (i.e., partitioning the list based on a pivot, recursively sorting the partitions, combining and returning them) with proof steps that attest to the correctness of the code with respect to its specification.[2]

[2]Many F* examples, including the ones shown here, are similar to program proofs in languages like Dafny or Verus. However, F* also allows other styles of higher order and dependently typed definitions that are not expressible in other SMT-assisted languages. We refer the reader to the F* manual for a full discussion of similarities and differences.

```
let rec sort (f:total_order_t α) (l:list α)
: Tot (m:list α{ sorted f m ∧ is_permutation l m })
      (decreases (length l))
= match l with
    | [] → []
    | pivot :: tl →
        let hi, lo = partition (f pivot) tl in
        let res = append (sort f lo) (pivot :: sort f hi) in
    (* <proof> *)
        partition_mem_permutation (f pivot) tl;
        append_count lo hi; append_count hi lo;
        sorted_concat f (sort f lo) (sort f hi) pivot;
        append_count (sort f lo) (sort f hi);
        permutation_app_lemma pivot tl (sort f lo) (sort f hi);
    (* </proof> *)
        res
```

For example, the annotation **decreases** (length l) indicates that the recursion terminates because the length of the list input decreases at each recursive call. Additionally, the source lines delimited by <proof> comment tags are calls to F⋆ *lemmas*, auxiliary definitions that prove certain properties. For instance, the auxiliary lemma append_count is shown below. Its type states that every call to append_count l m x guarantees the postcondition described in the **ensures** clause; or, equivalently, the type claims the universal property ∀l m x. count x (append l m) = count x l + count x m. The proof of this property is by induction on the list l, expressed as a recursive function.

```
let rec append_count (l m:list α) (x:α)
: Lemma (ensures (count x (append l m) = count x l + count x m))
= match l with
   | [] → ()
   | hd :: tl → append_count tl m x
```

Program and proof fragments like sort, append_count etc. each constitute a *top-level definition* in an F⋆ program. Each definition yields a type-directed synthesis problem, i.e., given a goal type such as the first two lines of append_count shown above, can an LLM generate a type-correct definition. To help the LLM succeed, we aim to augment the goal type with related information, e.g., the definitions of symbols such as count, append etc. We evaluate the performance of various retrieval augmentation strategies and LLMs on this task.

III. A Corpus of Proof-Oriented Programs

FStarDataSet is an archive of source code, build artifacts, and metadata assembled from eight (8) different F⋆-based open source projects on GitHub, summarized below, with a focus on libraries that provide secure low-level software and the tools that enable their proofs.

- FStar: The F⋆ compiler itself, including its standard libraries and examples.
- Karamel: A transpiler from a subset of F⋆ called Low* to C, including libraries to work with a model of C types and control structures, e.g., for- and while-loops [28].
- EverParse: A parser generator for binary formats [29], used in various large scale systems, e.g., the Windows kernel [30]
- HACL*: A library of verified cryptographic algorithms [26], including ValeCrypt [31], a library of verified assembly code, as well as EverCrypt, a cryptographic provider [32], including code deployed in Linux, Firefox, and Python.
- miTLS-F*: A partially verified reference implementation of the TLS protocol [33].
- EverQuic-Crypto: A verified implementation of header and packet protection for the QUIC protocol [34].
- Merkle-tree: A verified, incremental Merkle tree, designed for use in Azure CCF, a confidential computing system [32].
- Steel: A concurrent separation logic library, with proofs of data structures and concurrency primitives [35].

The dataset will be available publicly as an open source repository referencing the other projects as sub-modules, including a given version of F⋆ itself. All the projects are built with the same version of F⋆ and the Z3 SMT solver, resulting in a single archive with all the 2,060 source files and a build artifact for each of them, i.e., F⋆'s .checked files. Each checked file is accompanied by record of metadata about its contents: for each top-level element (*e.g.,*, the definition of a function, or a type, or a type signature), the metadata records its dependences, the compiler settings used to verify them, etc.

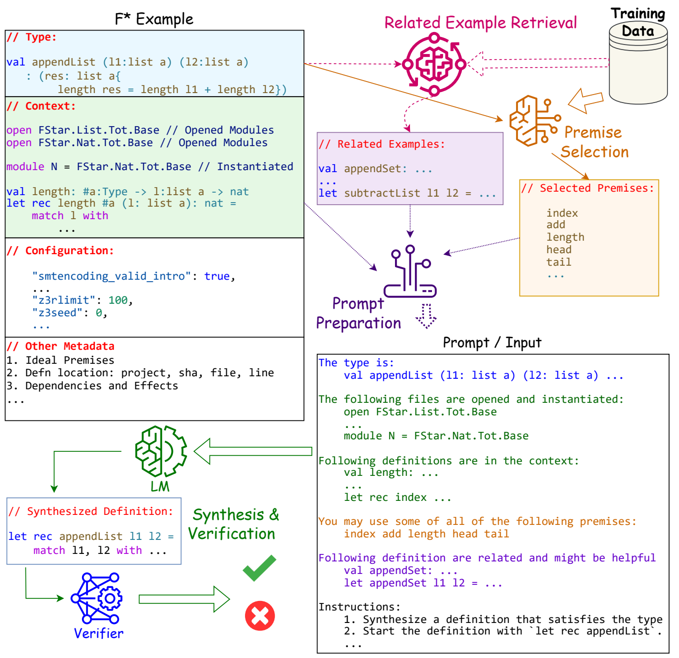
Reproducibility & evolution. We aim to strike a balance between reproducibility of the dataset, while also encouraging the data set to grow and change along with projects it references. Referencing the precise commit hashes of all the referenced projects as sub-modules allows any version of the dataset to be fully reproducible. The results reported in this paper focus on version 1 of the dataset, a snapshot from November 2023, provided as an anonymous supplement.

A type-checking harness. To enable using the data to validate program-proof synthesis attempts, we provide scripts that enable launching F⋆ and initializing it so that it is capable of checking each top-level definition independently. Once initialized in this way, the F⋆ process can be repeatedly queried using an interactive JSON-based API to type-check program fragments—in F⋆, type-checking amounts to program verification. However, replaying the F⋆ type-checker on top-level definitions in the dataset is not perfect, e.g., due to sensitivity to small changes in the search heuristics used by SMT solvers. We use the type-checking harness to identify those top-level definitions whose proofs can be checked in isolation, finding that 90% of them meet this criterion. In all our experiments, we focus on this re-checkable fragment of the dataset.

During the development of the harness, we faced similar challenges as reported by LeanDojo [7, A.2], mainly: (a) ensuring names are resolved correctly by opening the right namespace in the right order, (b) ensuring that definitions to be checked cannot refer to their original version in the library, that they cannot use their original definition implicitly via Z3, (c) verifying that solutions do not use escape hatches such as admit () or **assume** that are intended for interactive use, and (d) that all files can be loaded in a consistent state even though the bare dataset has clashing file names and conflicting options. Challenges (b) and (c) are hard to notice during development because they only result in false positives, i.e., the checker incorrectly claiming that a solution is correct. The biggest development effort was related to

TABLE I: Summary statistics of the FSTARDATASET.

Metric	Train	Valid	Test Intra-project	Test Cross-project
Number of Definitions	22779	1541	5965	1769
Number of Projects	6	6	6	2
Number of Files	1216	72	306	126
Avg. num of lines	8.66	13.63	11.40	7.45
Avg. num of tokens	92.16	157.26	124.32	60.32
# Simply Typed	6736	434	1248	149
# Dependently Typed	12047	764	3111	1431
# Proofs	3996	343	1606	189



```
// Type:

val appendList (l1:list a) (l2:list a)
   : (res: list a{
        length res = length l1 + length l2})
// Context:

open FStar.List.Tot.Base // Opened Modules
open FStar.Nat.Tot.Base // Opened Modules

module N = FStar.Nat.Tot.Base // Instantiated

val length: #a:Type -> l:list a -> nat
let rec length #a (l: list a): nat =
    match l with
        ...
// Configuration:

    "smtencoding_valid_intro": true,
    ...
    "z3rlimit": 100,
    "z3seed": 0,
    ...
// Other Metadata
1. Ideal Premises
2. Defn location: project, sha, file, line
3. Dependencies and Effects
...
```

```
// Related Examples:

val appendSet: ...
...
let subtractList l1 l2 = ...
```

```
// Selected Premises:

    index
    add
    length
    head
    tail
    ...
```

```
The type is:
    val appendList (l1: list a) (l2: list a) ...

The following files are opened and instantiated:
    open FStar.List.Tot.Base
    ...
    module N = FStar.Nat.Tot.Base

Following definitions are in the context:
    val length: ...
    ...
    let rec index ...

You may use some of all of the following premises:
    index add length head tail

Following definition are related and might be helpful
    val appendSet: ...
    let appendSet l1 l2 = ...

Instructions:
    1. Synthesize a definition that satisfies the type
    2. Start the definition with `let rec appendList`.
    ...
```

```
// Synthesized Definition:

let rec appendList l1 l2 =
    match l1, l2 with ...
```

Fig. 1: Overview of our experimental approach for synthesizing F* definitions.

efficiently supporting (b): we added a feature to F* that allows the client to partially load a compiled `.checked` file until we reach the definition to be checked; this ensures that exactly the same definitions and modules are in scope that were accessible to the original definition. After deduplication, removal of auto-generated code, data type definitions, and all proofs that are fully automated by SMT, the dataset contains 32,054 top-level definitions.

Partitions of the dataset. We split FSTARDATASET into four sets: *training*, *validation*, *intra-project test*, and *cross-project test*. Each source file is in exactly one of these sets. The first three sets are taken exclusively from the F*, Karamel, EverParse, HACL*, Merkle-tree, and Steel projects, while the "*cross-project test*" (we refer as "*cross-project*" hereafter) exclusively contains definitions from EverQuic-Crypto and miTLS-F*. The training set is closed under the dependence relation, *i.e.,* a file in the training set also has all its dependencies in the same set. As such, the training set contains files close to the roots of the dependence graphs for each project. In contrast, files in the intra-project test set (refers to as "*intra-project*" hereafter) may depend on files from other sets. Partitioning according to the dependence relation ensures that our trained models have not been contaminated with any information that depends on the test set. Files in cross-project set are from projects whose files are not in any of the other sets. This allows us to do a generalization evaluation across projects of the trained models, while again minimizing the potential for dataset contamination.

A Taxonomy. We classify the definitions in the dataset by their type, providing three classes corresponding loosely to the theoretical complexity of synthesizing a type-correct definition.

- *Simply typed* definitions have simple types such as `int` $\rightarrow$ `int`. Types that include refinements but no dependences are also considered simply typed, e.g., `nat` $\rightarrow$ `nat`. Definitions that are type polymorphic, e.g., `f`:($\alpha\rightarrow\beta$) $\rightarrow$ `list` $\alpha\rightarrow$ `list` β, are also considered simply typed. Types in this class could be expressed in many other general-purpose programming languages. Synthesizing definitions in this class is non-trivial—many symbolic synthesis techniques focus exclusively on simple types, while also excluding type polymorphism or non-dependent refinements [36]. Nevertheless, types in this class may admit many simple solutions. For example, given a goal type of `nat` $\rightarrow$ `nat`, a solution such as λ _ $\rightarrow$ 42 would succeed.
- *Proof* definitions have types that represent propositions, e.g., the type of `append_count` shown in §II. Such types are so precise that F*'s logic allows proving that all well-typed terms of a given proposition type are semantically equal. As such, this is the most challenging class, since there is at most one semantically unique solution and as synthesizing a solution amounts to proving a theorem in one go.
- *Dependently typed* definitions describe all cases whose types are not in the other two classes. Dependent types can encode complex specifications, e.g., the type of `sort` shown in §II is a detailed functional correctness property. Therefore definitions in this class in general have a higher theoretical complexity than the simply typed class. However there can still be multiple semantically different solutions to a dependently typed goal. For example, an implementation of merge sort can be given the same type as `sort`.

Table I shows some statistics of FSTARDATASET version 1, the basis of all experiments and analyses in this paper.

IV. Neural Program & Proof Synthesis for F*

Figure 1 shows a high level overview of the program and proof synthesis technique we develop in this paper. The top-left of the figure shows an instance of a top-level definition from FSTARDATASET, containing the dependent type t_i of a function `appendList`, other metadata, and, importantly, information about its "file context", i.e., all the top-level elements preceding `appendList` in the file in which it appears. To synthesize definitions for t_i, first (at top right) we retrieve related examples from the training data and select premises (*i.e.,* ingredient definitions) that are likely to be used in the definition body. At the center of the figure, combining t_i with its file context, related examples, selected premises, we create a prompt p_i. We use the prompt and the ground truth definition from the training data ($\mathcal{D}_T$) to train models. Finally, at bottom left, for testing and evaluation, we present the same kind of prompt to our finetuned LLMs, and evaluate the correctness of definitions with the type-checker harness. Note, we do not feedback type-checking results to the LLM for repair, an enhancement we are considering for the future.

Related Examples Retrieval. Inspired by successes of providing language models with related examples (colloquially known as Retrieval Augmented Generation, RAG) [37]–[39], we provide related examples as input to the model for better generalization. To find related examples, we calculate the similarities between the input type and the types of training examples. Our intuition is that if two examples' types are very close, there will be similarities between their definitions. We compute the similarity between types in their embedding space. Concretely, we embed the types into a fixed dimensional vector space using an LLM-based embedding model, `text-embedding-ada-002` [40], and compute the cosine similarity between t_i and $t_x \in \mathcal{D}_T$. We use the t_x and d_x (*i.e.,* definition of t_x from $\mathcal{D}_T$) as related examples.

Premise Selection. Following Yang *et al.* [7], we also provide the models with other definitions (premises) which are likely to be used in the body of the definitions. We fine-tune an embedding model based on `all-MiniLM-L6-v2` [41]. During inference, given a type as input the fine-tuned premise selection model produces ranked list of premises based on their likelihood of being used in the definition. We also explore fine-tuning alternative embedding models in Section VI-D.

To fine-tune an embedding model $e(\cdot)$, we take for every definition g in the training set the definitions $p_1, \ldots, p_n$ whose name occurs in the definition's value but not in its type and then randomly sample $q_1, \ldots, q_n$ definitions whose name occurs in neither the value nor the type but are in scope at the definition's location. As training objective we use the mean-square error $\frac{1}{2n}\sum_{i=1}^{n}\left(e(g)^T e(q_i)\right)^2 + \left(e(g)^T e(p_i) - 1\right)^2$, which aims to align the goal g to the ground-truth premises $p_1, \ldots, p_1$, and making the embedding of the goal orthogonal to the negative premises $q_1, \ldots, q_n$. For evaluation, the premises are then ranked by cosine distance to the goal. Note that, "Premise Selection" allowed us to fine-tune model as we already know the ground truth premises from the dataset. In contrast, there is no straightforward notion of ground truth for related example, hence we use embeddings from an LLM that supposedly understands its input.

Prompt Preparation. We prepare the input to the LLM (colloquially known as a prompt), by combining the "goal" type, the file context, selected premises, retrieved related examples, together with some fixed instructions for the model. We reserve a fixed number of tokens in the prompt for the file context, premises, and related examples. The file context contains opened and instantiated modules in the file together with other definitions in the file up to the point of the goal type, preferring definitions that are closer to the goal to fit in the fixed token budget for the file context. Likewise, for related examples and premises, we rank them according to descending similarity values, including as many as fit into their respective token budgets. We organize different components within a natural language prompt, *i.e.,* each component of the prompt is preceded by a brief description of what the component is. We represent each of the related examples with its type and definition. For each of the selected premises, we only use the *short name* of that premise in the prompt. While the type and definition of a premise contain more important information about it, we experimentally observe that using other information actually hurts the ultimate performance. Our conjecture is that since the number of allowed tokens for the prompt is limited, including additional information with each premise allows fewer premises to be included in the prompt, which leads to important premises being excluded from the prompt altogether.

Finally, the prompt ends with a set of instructions including the prefix of the definition. Figure 1 shows an example prompt at bottom right. We experiment on the impact of different components (context, retrieved examples, and selected premises) in §V-C.

Training. We train different language models of varying sizes to synthesize definitions from prompts. In particular, we trained Phi-2, Orca-2, and StarCoder with 2.7B, 7B, and 15.5B parameters, respectively. The input to the models is the prompt constructed as described above and the target response is set to be the definition from the dataset itself. We use standard practices for training, including low rank optimization [42], parameter efficient model tuning [43], and quantized model training, enabling us to deploy the models with GPUs with smaller memory size, for both both training and inference. We train Phi-2 model for 10K steps, Orca-2 model for 5K steps, and StarCoder for 2K steps, requiring roughly similar training time. The training objective is to maximize the probability of the definition given the prompt, which is realized by Cross Entropy Loss. We retain the model checkpoint which resulted in the least loss in the validation dataset.

Synthesis & Verification. Having the prompt designed, we synthesize the definition for a given type using the fine-tuned (also pretrained) LMs. While generating, we use temperature based sampling. For each example, we generate k definitions (S_k) using the LMs. We evaluate each of these generated definitions using the type-checking harness. In the case of a verification (type check) failure, the checker return the error code with error message. We define the evaluation metric, *verify@k* as the percentage of examples in the evaluation data for which there is at least one verified definition in the k generated solutions by the LM.

V. Empirical Evaluations

A. Performance of Different Models

In this section, we evaluate the ability of language models to synthesize F* programs and proofs, including of state-of-the-

TABLE II: Comparison between Prompting different models and finetuning for synthesizing F* definitions. Combined Prompted* shows the result when we combine the synthesized definitions from all prompted experiments (5), Combined Finetuned† corresponds to the combined results from finetuned experiments (3), and Comb. Prompted & Finetuned‡ shows the results across all the experiments (8).

Strategy	Model	verify@k (intra-project)			verify@k (cross-project)		
		k = 1	k = 5	k = 10	k = 1	k = 5	k = 10
Prompted	GPT-3.5	13.51	25.20	29.81	6.67	14.41	18.54
	GPT-4	19.41	31.90	36.38	13.00	23.57	28.49
	Phi-2 (2.7B)	0.30	1.26	2.20	0.11	0.62	1.75
	Orca-2 (7B)	2.63	6.39	8.55	0.45	2.20	3.84
	StarCoder (15.5B)	1.89	7.14	12.27	0.73	3.79	6.90
Combined Prompted*		26.34	40.02	45.47	16.56	29.96	36.18
Finetuned	Phi-2 (2.7B)	17.28	27.75	31.10	8.59	16.62	20.97
	Orca-2 (7B)	14.90	26.02	30.61	7.74	14.92	18.37
	StarCoder (15.5B)	**27.95**	**39.50**	**43.98**	**16.73**	**27.64**	**32.90**
Combined Finetuned†		32.86	44.17	48.52	20.97	32.28	37.20
Comb. Prompted & Finetuned‡		38.76	50.49	55.34	27.13	40.14	45.56

* : verify@k interprets as verify@5k. †: verify@k interprets as verify@3k, and ‡: verify@k interprets as verify@8k.

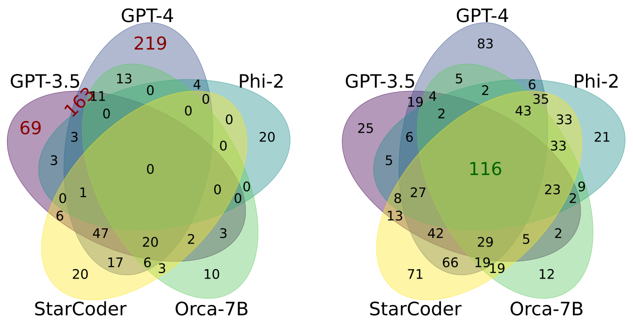
art LLMs such as GPT-3.5 and GPT-4, which are available for inference through APIs, as well as fine-tuned small and medium-sized language models deployed on local machines, posing the following research question:

RQ1. Can language models effectively synthesize F* definitions given the specification?

Experimental Setup. We evaluate all models using the same prompt, as described in §IV. We divide the experiments into two solution strategies: (i) prompting and (ii) fine-tuning. Prompting involves giving a pre-trained language model a task and assessing whether or not it is able to solve it. Fine-tuning, on the other hand, involves further refinement of a model parameters based on a task-specific dataset. For prompting LLMs, we use the OpenAI Python API. For prompting smaller language models, namely Phi-2, Orca-2, and StarCoder, we fine-tune these models (from the HuggingFace library [44]) as described in §IV and use the fine-tuned model checkpoints for inference. We set the same token limits for all models: 500 tokens for context, 400 for related examples, 300 for selected premises, and 500 for generated definitions.

Results. Table II shows the results of different strategies across different models. We focus on the verify@k metric defined in §IV, but also report a combined verify@nk metric which considers an example solved if the instance was solved by *any* of n models under the verify@k metric.

Both GPT-3.5 and GPT-4 solve a significant fraction of problems in the verify@10 metric, though GPT-4 is better. Their success is likely due to F* code being part of their training data. Both models also perform better on intra-project examples than on cross-project examples. When constructing prompts, we only retrieve related examples from the training data split, which specifically does not include any examples from the cross-project set—as such, the cross-project examples benefit less from retrieval augmentation.



(a) Prompted Models **(b)** Finetuned Models

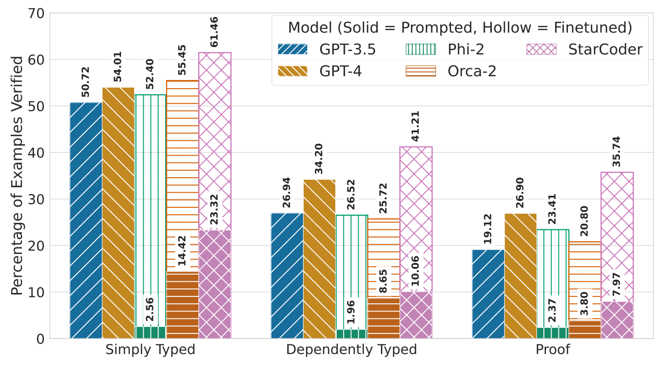
Fig. 2: Venn Diagram showing intersection of examples from Cross-Project examples solved by different models. The GPT-* performance in Fig. (b) are from prompting respective models.

The performance of prompted smaller models (Phi-2, Orca-2, and StarCoder) is significantly worse. Among these models, we do observe a positive correlation between the model size and performance, with the 15.5B parameter StarCoder performing better than Orca-2 (7B parameters), which in turn is better than Phi-2 (2.7B parameters). Their relative performance may possibly also be attributed to their pre-training data. Unlike Phi-2 and Orca-2, StarCoder's pre-training data contains OCaml code from GitHub; the syntactic similarity between OCaml and F* could also contribute to the relatively higher success of prompted StarCoder.

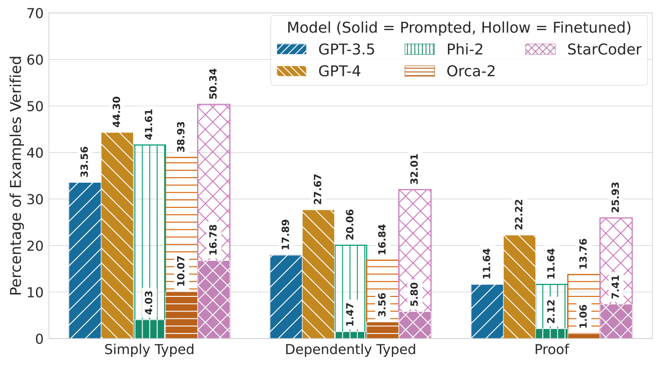
When we fine-tune these models, their performance significantly improves. For example, fine tuning Phi-2 improves its verify@10 score by more than a factor of 10. In fact, fine-tuned Phi-2 slightly outperforms GPT-3.5, while fine-tuned StarCoder also outperforms GPT-4. The combined verify@nk results suggest that one could use one or more cheaper, fine-tuned models for synthesis at first, falling back to the larger models only when the smaller models fail.

Fine-tuning equips a model with project-specific code idioms to improve performance. However, we argue that fine-tuning also yields benefits transferable across projects. Figure 2 shows the intersection of successes in the *cross-project examples* between different models in both prompted and fine-tuned settings. As Figure 2a shows, 451 examples are correctly solved by GPT-3.5 and GPT-4 exclusively, which could not be solved by prompting any other models; other models exclusively solved only 53 examples. In contrast, after fine-tuning (Figure 2b), prompted GPT models could only exclusively solve 137 problems, the rest 314 are solved by at least one fine-tuned model. On the other hand, 165 examples are exclusively solved by the smaller models that neither GPT-4 nor GPT-3.5 can solve. We believe this demonstrates the effectiveness of fine-tuning beyond specific projects they are trained on.

Result 1: Language models are helpful in automating the synthesis of definitions and proofs in F*. Fine-tuning smaller models can match or outperform the performance of



(a) Intra Project evaluation



(b) Cross Project evaluation

Fig. 3: Verify@10 across different types of examples for different models.

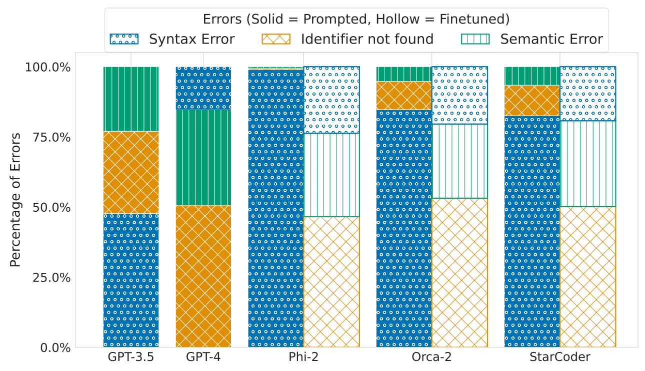


Fig. 4: Different errors spawned from the generated definitions by different models, presented as percentage of all errors.

large language models. Despite being significantly smaller, the fine-tuned Phi-2 (2.7B) model slightly outperforms GPT-3.5 by up to 5%. In addition, StarCoder (15.5B) outperforms the most advanced GPT-4 by up to 21%. Further, our evaluation shows the generalizability of these fine-tuned models on unseen projects.

B. Understanding models' successes and failures

In this section we focus on the following research question:

RQ2. What are the models' success rate across different problem types and what kinds of errors do they produce?

Experimental Setup. From §III, recall that we divide the problems in our dataset into three categories, *i.e.,* (i) simply typed definitions, (ii) dependently typed definitions, and (iii) proofs. We report on the performance of models in each of these categories. We also analyze the types of errors reported by the type-checking harness on synthesized code. We divide the errors into three broad categories (i) Syntax errors: where the F* could not parse the generated code, (ii) Identifier not found errors: where the definition was parsed, but one or more identifiers could not be resolved, and (iii) Semantic errors: the definition is syntactically valid and has no unresolved identifiers, but the type-checker could not verify the code against the goal type.

Results. Figure 3 shows the accuracy of different models in different classes of problems under the verify@10 metric. Across all configurations, models are most successful at simply typed definitions. For dependently typed definitions, we observe slightly lower performance across all models compared to simply typed definitions and performance on proof definitions is lower still. This is in keeping with our expectations: as described in §III, our taxonomy is meant to roughly capture the theoretical complexity of the problems and the model performance appears to validate this view. As in §V-A, performance on intra-project examples across all categories is also systematically better than on cross-project examples. In all cases, fine-tuned StarCoder performs the best.

We also examined specific successes from fine-tuned StarCoder to give a qualitative sense of the kinds of problems it is able to solve. We give two representative examples.

Our first example is dependently typed, for `FStar.OrdSet.where`, a function that filters and ordered set `s` to those elements that satisfy a condition `c`. The specification provided in the prompt is very detailed and includes functional correctness. The model is able to synthesize the type-correct definition shown, a fairly typical functional program. To put this in perspective, compared to AI automation for tactic-based proofs, our work exploits an LLMs ability to generate *programs*, and relies on F*'s dependent types and SMT-based automation to certify synthesis results.

```
let rec where #a #f (s:ordset a f) (c:condition a)
  : Pure (ordset a f)
      (requires ⊤)
      (ensures λ (z:ordset a f) →
          (as_list #a z == FStar.List.Tot.Base.filter c
           (as_list s)) ∧ (∀ x. mem x z = (mem x s && c x)) ∧
          (if size z > 0 && size s > 0 then
           f (head s) (head z) else true))
  = match s with
    | [] → empty
    | x::q →
      let z = where q c in
      if c x then insert' x z else z
```

Our next example is from EverQuic-Crypto, a cross-project proof problem. The lemma is about the property of a binary formatted header, using functions from the EverParse library. The preceding file context for this example contains a similar proof and the related examples includes an example from the EverParse library. In this case, the model has successfully adapted a related proof from the file context. Indeed, program proofs often contain many similar elements, tedious for a human to write, but models are adept at identifying common patterns and adapting them accordingly.

```
let serialize_header_is_retry
  (short_dcid_len: short_dcid_len_t)
  (h: header' short_dcid_len)
: Lemma (
    let s = LP.serialize (serialize_header short_dcid_len) h in
    Seq.length s > 0 ∧
    (is_retry h ⟺
        (LPB.get_bitfield (U8.v (Seq.index s 0)) 7 8 == 1 ∧
        LPB.get_bitfield (U8.v (Seq.index s 0)) 4 6 == 3)
    )
) = serialize_header_eq short_dcid_len h;
  let tg = first_byte_of_header short_dcid_len h in
  let x = LPB.synth_bitsum'_recip first_byte tg in
  LP.serialize_u8_spec x;
  let s = LP.serialize (serialize_header short_dcid_len) h in
  assert (Seq.index s 0 == x);
  assert (is_retry h ⟺ (
    LPB.get_bitfield (U8.v (Seq.index s 0)) 7 8 == 1 ∧
    LPB.get_bitfield (U8.v (Seq.index s 0)) 4 6 == 3
  ))
```

Turning our attention to errors, Figure 4 plots three different class of errors made by models as a percentage of the total erroneous solutions they generated. Interestingly, for GPT-3.5, the largest error class is “Syntax error”, whereas for GPT-4 it is “Identifier not found”—GPT-4 seems to be better at F* syntax than GPT-3.5. For the smaller models, when we prompt them, most of the errors are syntax errors. This is not surprising, since none of them have been trained on F*. When we fine-tune the models, the largest error class is “Identifier not found”, indicating that models often hallucinate identifiers. For example, the following definition is generated by StarCoder:

```
let eqList_ok (#a: Type) (d: deq a) :
    Lemma (decides_eq #(list a) (Raw.eqList d.raw.eq)) =
    let open FStar.Classical in
    let lemma_eq (xs ys: list a)
        : Lemma (Raw.eqList d.raw.eq xs ys) =
          FStar.List.Tot.lemma_eq_intro (
              Raw.eqList d.raw.eq
          ) xs ys; () in
    Classical.forall_intro (lemma_eq)
```

While seemingly syntactically correct, this definition uses the symbol `lemma_eq_intro`, which is not in scope. Adapting recent techniques to guide models based on lightweight static analyses (e.g., based on the identifiers in scope) is a promising direction for the future [45], [46].

The last, but not the least, category of error is Semantic error, which is a collection of many different errors. Notable among these include `Type Error`: a value or identifier with incompatible type is used and `Z3 Solver Error`: Z3 cannot prove the SMT query. Program repair techniques, both search-based [47] and LLM-assisted [2], may help reduce some of these errors, another direction for the future.

Result 2: Model performance follows our taxonomy, with simply typed problems being most commonly solved, then dependently typed, and finally proofs. Models are able to generate type-correct functional programs with complex dependent types, and to generate proofs by adapting similar examples. For most of the pre-trained models, the major source of errors are syntax errors, while after fine-tuning, “Identifier not found” consist of the majority of errors, followed by semantic errors.

TABLE III: Impact of different sources of information in the finetuned Phi-2 model.

Experiment Name	Auxiliary Information: Context	RE	Premise	verify@10: Intra	Cross
Phi2$_{full}$	✓	✓	✓	31.10	20.97
Phi2$_{-ctx}$	✗	✓	✓	21.89	10.97
Phi2$_{-re}$	✓	✗	✓	25.92	21.82
Phi2$_{-pre}$	✓	✓	✗	31.15	20.35
Phi2$_{ideal}$	✓	✓	Ideal	35.84	23.74

RE = Related example definition from training set

C. Impact of components of the prompt

Recall from §IV, a prompt contains the (1) local file context; (2) related examples retrieved from the training data; and, (3) selected premises by the premise selection model. In this section, we evaluate the impact of each of these three components on model performance.

RQ3. How do the different prompt components impact the effectiveness of language models?

Experimental Setup. We take as a baseline for our experience the performance a fine-tuned Phi-2 model (*Phi2*$_{full}$) that has access to all three prompt components. We fine-tune three other versions of Phi-2, each time dropping one of three three prompt components: for the *Phi2*$_{-pre}$ model, we drop the selected premises from the prompt; *Phi2*$_{-ctx}$, drops file context information; and *Phi2*$_{-re}$, drops the related examples. In addition, since we know the ideal premises used in the ground truth definition, we fine-tune another version of *Phi2*$_{full}$, where instead of the selected premised, we used the actual premises (*Phi2*$_{ideal}$). *Phi2*$_{ideal}$ is not realistic, but it serves as a roof-line for premise selection. We evaluate these variants on verify@10 as well on the numbers of errors each of these models produce. We also experiment with different ways to combine and present these prompt components to the model. Note that, since these experiments entail a large number of fine-tuning runs, we chose to focus on the Phi-2 model, as it is the most cost-effective. We believe our findings should generalize to other models.

Results. Table III summarizes our results. The baseline *Phi2*$_{full}$ model performance is as in Table II. When we remove the related examples from the prompt (*Phi2*$_{-re}$) and fine-tune a model, for intra-project, the performance drops by ∼5 percentage points. Interestingly, without the related examples, the performance in cross-project examples increases slightly. Since we extract the related examples from other projects in the training set, and there is little to no helpful examples in the *Phi2*$_{full}$ model for the cross-project examples. Thus we conjecture, the related examples from the same project helps the model most.

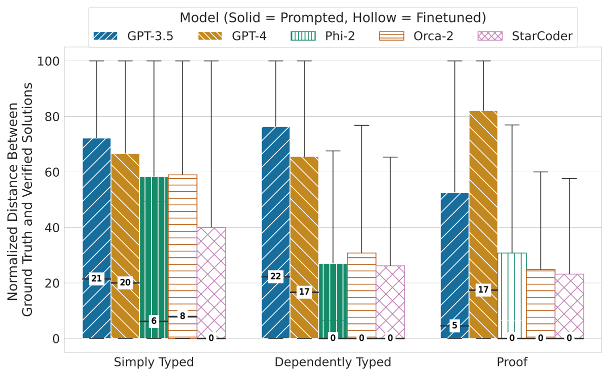


Fig. 5: Normalized Levenshtein distance between verified solutions and ground truth across different classes.

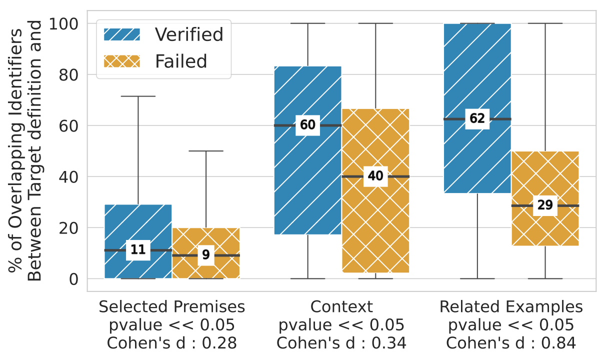


Fig. 6: Overlap between identifiers in ground truth and identifiers in each prompt component

On the other hand, in *Phi2$_{\text{-}ctx}$*, we observe the performance dropping significantly from the *Phi2$_{full}$* model for both intra-project and cross-project examples. Such a drop is not surprising, since even for a human user, the file context has the most relevant information about the target definition, e.g., in the `FStar.OrdSet.where` example shown in §V-B, the local file context *defines* the type `ordset a f` as an ordered list, crucial information for synthesizing a solution.

Finally, when we remove the selected premises (*Phi2$_{\text{-}pre}$*), the performance remains very close to the *Phi2$_{full}$* model. This is surprising, since prior work [7] demonstrates that premise selection improves the performance of proof synthesis. To investigate further, we fine-tuned *Phi2$_{ideal}$*, where instead of using premise selection model, we use the ideal premises. The results reveal that, with ideal premises, the performance does improve significantly from *Phi2$_{full}$* model to 35.84% in intra-project and 23.74%in cross-project. Additionally, the number of "Identifier not found" errors with *Phi2$_{ideal}$* reduces significantly to 22149 from 30108 such errors in *Phi2$_{full}$*. As such, *Phi2$_{ideal}$* suggests that a premise selection model that better approximates the ideal premises can provide a significant improvement—see §VI for more discussion.

Result 3: File context and related examples are very important sources of information for synthesizing definitions in F⋆. Without the context, the performance of a model drops down by up to 29.6% and up to 16.55% without the related examples. An ideal premise selection roof line suggests that it may also provide significant improvements, however our current premise selection model does not significantly impact performance.

VI. Further Analysis & Discussion

This section takes a closer look at some of the results from §V. We report on additional experiments and explore potential for future improvements.

A. Syntactic Evaluation of Model Generated Definitions

In addition to the semantic verify@k metric of the prior section, we also evaluate how close a model's generated solutions are to the ground truth solution (g). We calculate the normalized edit (Levenshtein) distance between tokenized g and solution v. For every problem which has a non-empty set of verified candidate solution (VS), we define the distance as $\min_{v \in VS} lev\_dist(g, v)$. The *lev_dist* calculates the edit distance between g and v and normalized it *w.r.t.* the number of tokens in g. Figure 5 shows the distribution of distances. Prompted LLMs such as GPT-3.5 and GPT-4 generate solutions that are further away from the ground truth compared to fine-tuned models.

B. Impact of maximum tokens on model performances

TABLE IV: Performance of different state-of-the-art LLMs with increasing model capacity for 100 randomly sampled examples.

Model	Max number of tokens	verify@10
GPT-3.5	2,048 16,000	29 35
GPT-4	2,048 16,000	37 50
StarCoder (finetuned)	2,048	45

Throughout the paper, for a fair comparison, we restrict the maximum number of tokens for different models to 2048 tokens. However, some LLMs accommodate much larger prompts, *e.g.,* `gpt-3.5-turbo-16K` with 16K. While experimenting on these larger context models are expensive, we evaluate GPT-3.5 and GPT-4 models allowing maximum 10K tokens for context, 3K for related examples, and 2K for selected premises, and 1K maximum generation length. To keep the cost of experiment tractable, we randomly sample 100 problems for evaluation. As Table IV shows, we observe that, with increased capacity, both GPT-3.5 and GPT-4 improves by a large margin compared smaller capacities. While GPT-4 with 16K token capacity achieves 50% verify@10, fine-tuned StarCoder achieves 45% with only 2K tokens. Further experiments with GPT-4 on the entire data set would be more definitive, though the cost is prohibitive.

C. Definition ingredients in the prompt and its impact

Following our observation that that majority of the errors are due to hallucinated identifiers, we investigated how much each

TABLE V: Evaluation of alternative models for the premise selection. An asterisk (*) indicates a fine-tuned model.

Model	Intra-Project		Cross-Project	
	MAP	NDCG	MAP	NDCG
Pythia 70M	0.15	0.37	0.14	0.37
Pythia 160M	0.15	0.38	0.13	0.37
all-MiniLM-L6-v2	0.15	0.38	0.16	0.40
OpenAI Ada	0.19	0.42	0.19	0.43
Pythia 70M*	0.33	0.55	0.15	0.40
Pythia 160M*	0.34	0.56	0.13	0.38
all-MiniLM-L6-v2*	0.32	0.54	0.18	0.43

of the prompt components help the model generate correct identifiers. We calculated the percentage of overlap between the identifiers used in the ground truth and different prompt components and investigate whether this correlates with model performance. Figure 6 shows that when there is a higher overlap between an information modality and the ground truth identifiers, the models are more likely to generate verified definitions (with statistical significance). While in theory, it is impossible to know the name of the identifiers that will be used in a definition a priori, the analysis suggests that better retrieval augmentation could boost performance.

D. Evaluating Different Premise Selection Models.

We compared four embedding models for premise selection, before and after fine-tuning: the small 22M parameter model `all-MiniLM-L6-v2` from sentence transformers [41], the `text-embedding-ada-002` model [40] from OpenAI, as well as two generic transformer models from the Pythia [48] family with 70M and 160M parameters. Comparing the Mean Average Precision (MAP) and Normalized Cumulative Discounted Gain (NDCG) (Table V), without fine-tuning the Ada model achieves roughly 25% higher performance. Surprisingly, the off-the-shelf Pythia models are competitive with `all-MiniLM-L6-v2` even though they are not trained on any contrastive objective. After fine-tuning, all of the models have comparable performance irrespective of model size. However, the training does not generalize well and fine-tuning only barely improves performance on the *cross-project* set.

E. Alternative Prompt Preparation

TABLE VI: Different Input Formatting.

Input Type	verify@10	
	Intra-Project	Cross-Project
NL Prompt	**31.10**	**20.97**
Structured	30.76	19.45
Completion	10.54	12.21

In addition to presenting a natural language prompt (NL prompt) (see §IV for details), we investigated alternative prompting strategies. In particular, we created a structured format format similar to the format proposed by Gupta *et al.* [49]. In this version, every input modality is surrounded by tags, e.g., `<context>` and `</context>` surrounds the file context; `<related>` and `</related>` surrounds the related examples. In another version, we created a completion style where all the prompt components are concatenated with a separator token `<|end_of_text|>`, the default padding token of Phi-2, without any description of each of the components. Natural language prompts are slightly better that structured prompts, while the completion-style prompt performs substantially worse. We conjecture that since the Phi-2 model is pre-trained mostly on natural language, wrapping the prompt components with natural language helps during fine-tuning; the structured tag formats also help the model distinguish the components. Such a distinction is lost in the completion style prompt.

VII. Threats & Limitations

Data contamination. The training data for GPT-3.5 and GPT-4 is not publicly known, but it is very likely that it intersects the intra-project and cross-project test sets because those are taken from repositories publicly hosted on GitHub. On the other hand, despite maintaining file level separation between train and test sets, we observe that there are a small number of clones (*i.e.,* examples with different names, yet same definitions) between train and test set. In particular, we identified 343 such clones in intra-project test set and 7 in the cross-project test set. While these solutions are clones, the context they are in are different. Hence, the prompt to the language model (see Section IV for prompt preparation) are different for these. While the related examples retrieval method are in general capable of finding these clones, we argue that such a setup is not unrealistic for real development, where developer often reuse code written elsewhere.

Hints about the definition problem statement. The synthesis problem in this paper requires the model to synthesize a definition for a given type. However, there are some implicit hints about the definition that are present in the types, and context. For instance, consider the example `sort` in §II, the type contains an effect **`decreases`** `(length l)`, which implicitly tells the model that there will be an induction on the length of the `l`. While these are hints potentially helping the model, we argue that these are the hints that a developer writing F* code may already know.

Preciseness of specifications. We consider a problem to be solved if the solution type-checks. For problems in the proof class, all solutions are equivalent, so no further inspection is necessary for a successful proof. However, in other cases, specifications may only be partial and, although type-correct, a solution may still be subject to inspection to confirm if it matches a user's under-specified intent.

Non-deterministic nature of LLMs. Large Language Models, especially those for which we do not have access to the model weights (*e.g.,* GPT-3.5/GPT-4) often go through regular updates. Hence, it it very difficult, if not impossible, to reproduce some of the results from those LLMs. In contrast, the fine-tuned model, being smaller in size, will be accessible to those seeking to reproduce our results.

VIII. Related Work

In the past 15 years, an increasing number of software systems have been proven correct, both using interactive theorem provers like Coq and Isabelle/HOL, as well using SMT-assisted proof-oriented languages like F* and Dafny. Software proofs require expertise and can be quite verbose. A common metric used to evaluate proof effort is a proof:code ratio, i.e., the number of lines of proof for each line of executable code. This ratio is typically higher in interactive proof assistants than in SMT-assisted proof-oriented languages. For example, sel4 in Isabelle/HOL [50] incurs a proof:code ratio of 20:1 despite the use of automated theorem provers integrated with Isabelle/HOL, e.g., Sledgehammer [51]. Meanwhile, Ironclad in Dafny [25] reports a ratio of 5:1, and EverCrypt in F* [32] reports a ratio of 3:1, through the use of SMT-solvers and programs that mix computational content and proofs. Improved automation through the use of AI in both settings could help lower these overheads.

In the context of interactive theorem provers, machine-learning has been used to improve premise selection in theorem provers [52], [53]. Predictive models based on existing proofs to guide proof search have been used in TacticToe [54], TacTok [55]. GPT-f [56] uses expert iteration and HTPS [3] online reinforcement learning to improve the model by self-learning based on previous proof attempts. Baldur [2] uses an LLM-based synthesis and fine-tuned repair model to complete full proofs for Isabelle/HOL theorems. LeanDojo [7] and LEGO-Prover [57] integrate retrieval augmentation to find relevant existing theorems and proofs from the library. Draft-sketch-proof [58] introduced the use of informal sketches as an intermediate step, which has also been used in [59], [60].

In the context of SMT-assisted program verifiers, recent approaches leverage LLMs for synthesizing loop invariants [13]–[16]. Closest to our work, Rakib *et al.* [18] leverage LLMs along with retrieval augmentation to generate both functional specification and code with proofs from natural language in Dafny. However, the specifications generated may be too weak or incorrect and requires a human to audit them; this makes it subjective and difficult to treat as a benchmark problem set unlike FStarDataSet where specifications are drawn from ground truth. In addition, prior works on automating proof-oriented programming [18], [61] are evaluated on much smaller benchmarks with less than 200 mostly introductory algorithmic tasks. In contrast with FStarDataSet we provide a benchmark with more than 32K reproducible F* programs and proofs that form part of production software deployed in real-world.

Additionally, many works use LLMs for general-purpose programming tasks [62]–[65]. Recent approaches [66], [67] even attempt at formalizing functional specifications for languages such as python—without access to static verifiers, these functional specifications often are used as runtime assertions.

IX. Conclusion

Aiming to enhance both the trustworthiness of AI-generated code and to ease program proof, we investigate AI-based program and proof synthesis backed by a program verifier. While prior work has predominantly focused on tactic-based proof, we investigate AI automation for F*, a proof-oriented language with SMT-based automation. To fuel future research in this direction, we introduce the FStarDataSet, a public benchmark dataset of F* programs and proofs. We expect FStarDataSet to evolve and grow: indeed in recent weeks, after our experiments, it has grown to include four more projects, reaching 940K lines of F* code and proofs.

On a type-based program and proof synthesis task, we evaluate several state-of-the-art LLMs such as GPT-3.5 and GPT-4, fine-tune smaller language models such as StarCoder. Our findings suggest that LLMs, when trained on appropriate datasets and prompted effectively with necessary information through retrieval, can automate a significant portion programs and proof. However, beyond boosting synthesis performance, many interesting problems remain, notably around specification formulation and modular decomposition of proofs, exciting areas for further exploration.

Overall, with this paper, we contribute to the ongoing discourse on trustworthy AI programming and lay a foundation for advancements in AI-assisted proof-oriented programming.

References

[1] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.

[2] E. First, M. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1229–1241.

[3] G. Lample, T. Lacroix, M.-A. Lachaux, A. Rodriguez, A. Hayat, T. Lavril, G. Ebner, and X. Martinet, "Hypertree proof search for neural theorem proving," *Advances in Neural Information Processing Systems*, vol. 35, pp. 26 337–26 349, 2022.

[4] A. Thakur, Y. Wen, and S. Chaudhuri, "A language-agent approach to formal theorem-proving," *arXiv preprint arXiv:2310.04353*, 2023.

[5] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner, "Generating correctness proofs with neural networks," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2020, pp. 1–10.

[6] K. Yang and J. Deng, "Learning to prove theorems via interacting with proof assistants," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6984–6994.

[7] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[8] T. mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 367–381. [Online]. Available: https://doi.org/10.1145/3372885.3373824

[9] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270. [Online]. Available: https://www.fstar-lang.org/papers/mumon/

[10] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*. Springer, 2010, pp. 348–370.

[11] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A verification infrastructure for permission-based reasoning,” in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2

[12] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, “Verus: Verifying rust programs using linear ghost types,” *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. [Online]. Available: https://doi.org/10.1145/3586037

[13] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, “Finding inductive loop invariants using large language models,” *arXiv preprint arXiv:2311.07948*, 2023.

[14] S. Chakraborty, S. K. Lahiri, S. Fakhoury, M. Musuvathi, A. Lal, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy, “Ranking llm-generated loop invariants for program verification,” *arXiv preprint arXiv:2310.09342*, 2023.

[15] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, “Can large language models reason about program invariants?” 2023.

[16] C. Liu, X. Wu, Y. Feng, Q. Cao, and J. Yan, “Towards general loop invariant generation via coordinating symbolic execution and large language models,” *arXiv preprint arXiv:2311.10483*, 2023.

[17] C. Sun, Y. Sheng, O. Padon, and C. Barrett, “Clover: Closed-loop verifiable code generation,” *arXiv preprint arXiv:2310.17807*, 2023.

[18] M. Rakib Hossain Misu, C. V. Lopes, I. Ma, and J. Noble, “Towards ai-assisted synthesis of verified dafny methods,” *arXiv e-prints*, pp. arXiv–2402, 2024.

[19] S. Chakraborty, G. Ebner, S. Bhat, S. Fakhoury, S. Fatima, S. Lahiri, and N. Swamy, “Towards neural synthesis for smt-assisted proof-oriented programming (author’s preprint),” *arXiv:2405.01787*, 2024. [Online]. Available: https://arxiv.org/abs/2405.01787

[20] R. OpenAI, “Gpt-4 technical report. arxiv 2303.08774,” *View in Article*, vol. 2, 2023.

[21] OpenAI, “Gpt-4 technical report,” 2023.

[22] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, “Textbooks are all you need,” *arXiv preprint arXiv:2306.11644*, 2023.

[23] A. Mitra, L. Del Corro, S. Mahajan, A. Codas, C. Simoes, S. Agarwal, X. Chen, A. Razdaibiedina, E. Jones, K. Aggarwal *et al.*, “Orca 2: Teaching small language models how to reason,” *arXiv preprint arXiv:2311.11045*, 2023.

[24] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

[25] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, “Ironclad apps: End-to-End security via automated Full-System verification,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 165–181. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[26] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, “HACL*: A verified modern cryptographic library,” in *ACM Conference on Computer and Communications Security*. ACM, 2017, pp. 1789–1806. [Online]. Available: http://eprint.iacr.org/2017/536

[27] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 337–340.

[28] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, “Verified low-level programming embedded in F*,” *PACMPL*, vol. 1, no. ICFP, pp. 17:1–17:29, Sep. 2017. [Online]. Available: http://arxiv.org/abs/1703.00053

[29] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, “Everparse: Verified secure zero-copy parsers for authenticated message formats,” in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC’19. USA: USENIX Association, 2019, p. 1465–1482.

[30] N. Swamy, T. Ramananandro, A. Rastogi, I. Spiridonova, H. Ni, D. Malloy, J. Vazquez, M. Tang, O. Cardona, and A. Gupta, “Hardening attack surfaces with formally proven binary format parsers,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI ’22), June 13–17, 2022, San Diego, CA, USA*, 2022. [Online]. Available: https://www.fstar-lang.org/papers/EverParse3D.pdf

[31] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, “A verified, efficient embedding of a verifiable assembly language,” *PACMPL*, no. POPL, 2019. [Online]. Available: https://github.com/project-everest/project-everest.github.io/raw/master/assets/vale-popl.pdf

[32] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Beguelin, “Evercrypt: A fast, verified, cross-platform cryptographic provider,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 983–1002.

[33] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella Béguelin, and J. K. Zinzindohoue, “Implementing and proving the TLS 1.3 record layer,” *IEEE Security & Privacy*, 2017.

[34] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou, “A security model and fully verified implementation for the ietf quic record layer,” in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1162–1178.

[35] A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martínez, D. Merigoux, and T. Ramananandro, “Steel: Proof-oriented programming in a dependently typed concurrent separation logic,” in *25th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Aug. 2021. [Online]. Available: https://www.fstar-lang.org/papers/steel/

[36] P.-M. Osera and S. Zdancewic, “Type-and-example-directed program synthesis,” *SIGPLAN Not.*, vol. 50, no. 6, p. 619–630, jun 2015. [Online]. Available: https://doi.org/10.1145/2813885.2738007

[37] Z. Jiang, F. F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu, Y. Yang, J. Callan, and G. Neubig, “Active retrieval augmented generation,” *arXiv preprint arXiv:2305.06983*, 2023.

[38] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[39] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Retrieval augmented code generation and summarization,” *arXiv preprint arXiv:2108.11601*, 2021.

[40] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy, J. Heidecke, P. Shyam, B. Power, T. E. Nekoul, G. Sastry, G. Krueger, D. Schnurr, F. P. Such, K. Hsu, M. Thompson, T. Khan, T. Sherbakov, J. Jang, P. Welinder, and L. Weng, “Text and code embeddings by contrastive pre-training,” *CoRR*, vol. abs/2201.10005, 2022. [Online]. Available: https://arxiv.org/abs/2201.10005

[41] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: https://arxiv.org/abs/1908.10084

[42] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” *arXiv preprint arXiv:2106.09685*, 2021.

[43] Y. Mao, L. Mathias, R. Hou, A. Almahairi, H. Ma, J. Han, W.-t. Yih, and M. Khabsa, “Unipelt: A unified framework for parameter-efficient language model tuning,” *arXiv preprint arXiv:2110.07577*, 2021.

[44] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, “Huggingface’s transformers: State-of-the-art natural language processing,” *arXiv preprint arXiv:1910.03771*, 2019.

[45] L. A. Agrawal, A. Kanade, N. Goyal, S. Lahiri, and S. Rajamani, “Monitor-guided decoding of code lms with static analysis of repository context,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[46] Y. Wei, C. S. Xia, and L. Zhang, “Copiloting the copilots: Fusing large language models with completion engines for automated program repair,” in *Proceedings of the 31st ACM Joint European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 172–184.

[47] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[48] S. Biderman, H. Schoelkopf, Q. G. Anthony, H. Bradley, K. O'Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff *et al.*, "Pythia: A suite for analyzing large language models across training and scaling," in *International Conference on Machine Learning*. PMLR, 2023, pp. 2397–2430.

[49] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "Grace: Language models meet code edits," ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1483–1495. [Online]. Available: https://doi.org/10.1145/3611643.3616253

[50] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[51] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending sledgehammer with SMT solvers," *J. Autom. Reason.*, vol. 51, no. 1, pp. 109–128, 2013. [Online]. Available: https://doi.org/10.1007/s10817-013-9278-5

[52] D. Kühlwein, J. C. Blanchette, C. Kaliszyk, and J. Urban, "Mash: Machine learning for sledgehammer," in *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, ser. Lecture Notes in Computer Science, S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., vol. 7998. Springer, 2013, pp. 35–50. [Online]. Available: https://doi.org/10.1007/978-3-642-39634-2_6

[53] M. Mikula, S. Antoniak, S. Tworkowski, A. Q. Jiang, J. P. Zhou, C. Szegedy, L. Kucinski, P. Milos, and Y. Wu, "Magnushammer: A transformer-based approach to premise selection," *CoRR*, vol. abs/2303.04488, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.04488

[54] T. Gauthier, C. Kaliszyk, J. Urban, R. Kumar, and M. Norrish, "Tactictoe: Learning to prove with tactics," *J. Autom. Reason.*, vol. 65, no. 2, pp. 257–286, 2021. [Online]. Available: https://doi.org/10.1007/s10817-020-09580-x

[55] E. First, Y. Brun, and A. Guha, "Tactok: Semantics-aware proof synthesis," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–31, 2020.

[56] J. M. Han, J. Rute, Y. Wu, E. W. Ayers, and S. Polu, "Proof artifact co-training for theorem proving with language models," *CoRR*, vol. abs/2102.06203, 2021. [Online]. Available: https://arxiv.org/abs/2102.06203

[57] H. Xin, H. Wang, C. Zheng, L. Li, Z. Liu, Q. Cao, Y. Huang, J. Xiong, H. Shi, E. Xie *et al.*, "Lego-prover: Neural theorem proving with growing libraries," *arXiv preprint arXiv:2310.00656*, 2023.

[58] A. Q. Jiang, S. Welleck, J. P. Zhou, T. Lacroix, J. Liu, W. Li, M. Jamnik, G. Lample, and Y. Wu, "Draft, sketch, and prove: Guiding formal theorem provers with informal proofs," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/pdf?id=SMa9EAovKMC

[59] S. Welleck, J. Liu, X. Lu, H. Hajishirzi, and Y. Choi, "Naturalprover: Grounded mathematical proof generation with language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 4913–4927, 2022.

[60] Y. Huang, X. Lin, Z. Liu, Q. Cao, H. Xin, H. Wang, Z. Li, L. Song, and X. Liang, "Mustard: Mastering uniform synthesis of theorem and proof data," *arXiv preprint arXiv:2402.08957*, 2024.

[61] J. Yao, Z. Zhou, W. Chen, and W. Cui, "Leveraging large language models for automated proof synthesis in rust," *arXiv preprint arXiv:2311.03739*, 2023.

[62] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[63] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "Natgen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 18–30.

[64] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," *CoRR*, vol. abs/2208.05950, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.05950

[65] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[66] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Formalizing natural language intent into program specifications via large language models," *CoRR*, vol. abs/2310.01831, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.01831

[67] D. Key, W.-D. Li, and K. Ellis, "I speak, you verify: Toward trustworthy neural program synthesis," 2022.