

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY


DafnyBench

Citation

Loughridge, Chloe Rose. 2024. DafnyBench. Bachelor's thesis, Harvard University Engineering and Applied Sciences.

Link

https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37379278

Terms of use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material (LAA), as set forth at

https://harvardwiki.atlassian.net/wiki/external/NGY5NDE4ZjgzNTc5NDQzMGIzZWZhMGFlOWI2M2EwYTg

Accessibility

https://accessibility.huit.harvard.edu/digital-accessibility-policy

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story

DafnyBench

Chloe Loughridge

March 2024

1 Abstract

It is said that "program testing can be used to show the presence of bugs, but never to show their absence" [31]. As Large Language Models are increasingly used to generate code, it is important that we verify the absence of bugs in their outputs. For this reason, we introduce DafnyBench, a novel benchmark for evaluating Large Language Models' abilities to write Dafny code, which is a verification-aware programming language. With over 1,000 sample programs, DafnyBench represents the largest benchmark of its kind to date. We argue this benchmark represents an important piece of infrastructure for an emerging field: AI-assisted formal verification. The benefits of having Large Language Models write verification-aware code cuts in two directions: the Dafny verifier provides a rich stream of feedback and clear learning signal for what constitutes correct code with respect to a set of pre- and post- conditions (helping machine learning research efforts), and also greater prevalence of Dafny code generally leads to a higher amount of safe software in the world.

Contents

1 Abstract **1**

2 Acknowledgements **4**

3 Introduction **5**

4 Contributions **5**

5 Background of Key Concepts **5**
5.1 Language Models 5
5.2 Verification-Aware Languages 8

6 Related Work **10**
6.1 Benchmarks 10
6.1.1 Formal Reasoning Benchmarks 10
6.1.2 (Unverified) Programming Benchmarks 11
6.1.3 Formally Verified Programming Benchmarks 13
6.2 Algorithms 13
6.2.1 Algorithms for Proof Writing (Formal Reasoning) 14
6.2.2 Algorithms for General Reasoning 15
6.2.3 Algorithms for (Unverified) Programming 16
6.2.4 Algorithms for Verified Programming 17

7 Benchmark Construction and Design **17**
7.1 Methods of Construction 18
7.1.1 Rationale Behind Language Choice 18
7.1.2 Sourcing Ground Truth Programs 18
7.1.3 Suggested Training Data 19
7.2 Task Design 19
7.2.1 Fill_body 20
7.2.2 Fill_annotations 21
7.3 Evaluation Design 22

8 Proposed Algorithms for Success on this Benchmark **23**

9 Limitations **26**

10 Future Extensions to this Benchmark **27**

11 Use Cases and Impacts **28**

12 Conclusion **29**

13 Appendix **30**
13.1 A: The Minhash Deduplication Algorithm 30
13.2 B: GPT-4 spot-check on Benchmark Tasks 32
13.3 C: Samples of HumanEvalX Problems Translated to Dafny . . . 32
13.3.1 HumanEvalX 4 32
13.4 D: Source Repositories for Benchmark Code 33
13.5 E: Examples of Hacks and Failure Modes our Benchmark Evaluation will Catch 35
13.5.1 Modification of Formal Specification 35
13.5.2 Cheat Via Assume False 36
13.5.3 Cheat Via :Verify False 37
13.6 F: Mathematics Behind Training on Crossentropy 38
13.7 G: Direct Preference Optimization 38
13.7.1 Theory of Direct Preference Optimization 38
13.7.2 Dataset of Positive/Negative Triples Used for Fine Tuning Phind-Code-Llama 39
13.8 H: Theory of SMT Solvers 40
13.8.1 Simple Examples for Intuition 41

2 Acknowledgements

First and foremost I would like to thank Professor Amin for being the most inspiring mentor a student could ask for. I would also like to thank Professor Max Tegmark for his advice and guidance.

Wendy Sun and Federico Cassano provided invaluable advice and support as well.

Also, thank you to the many peers and friends who have made my academic experience this year great.

Of course, thank you to my parents for everything.

Finally, I would like to thank GPT-4 for doing a lot of the tedius formatting behind this thesis document. Thanks to Claude3 Opus, too, which helped with some parts of code analysis. (Please spare me in the robot revolution.)

3 Introduction

GPT-4 has recently captivated the global audience with its sophisticated text generation capabilities. However, its eloquence may obscure its capacity for logical reasoning. A particularly noteworthy application of GPT-4 is in code generation. Developers increasingly use GPT-4 models to write their code. Despite GPT's proficiency in producing code that meets the criteria for a significant portion of benchmark problems, such as those in the HumanEvalX [14] dataset, its ability to consistently and effectively reason about its own code remains questionable. This raises concerns about the security and reliability of the code it generates, highlighting a relatively unexplored aspect of Large Language Models' capabilities.

This thesis introduces the most extensive benchmark to date for algorithms that have been formally verified in Dafny. By doing so, it aims to pioneer the emerging field of applying Large Language Models to verification-aware programming. The benchmark challenges models to not only create algorithmic implementations that adhere to a specific, formal specification but also to verify that these implementations meet the given specifications (which in this iteration of the benchmark is always given). This requirement pushes models to elucidate the rationale behind the code they generate and the properties of that code.

Unlike preceding benchmarks, this thesis organizes verification-aware programming tasks into two distinct categories: (1) simple single-method problems and (2) harder multi-step problems. Segmenting model performance by these categories allows for a more precise analysis of models' reasoning abilities and weaknesses. Additionally, this benchmark incorporates the first comprehensive compilation of publicly available, compilable Dafny code from GitHub. To enhance the dataset further, we manually convert some programs from Python to Dafny, using benchmarks like HumanEvalX [14] and MIPS [38]. In addition to establishing a benchmark, this thesis proposes novel algorithms designed to improve performance on generating Dafny code.

4 Contributions

Please find the supporting code for my proposed benchmark here: `https://github.com/ChloeL19/DafnyBench/tree/Thesis2024`

5 Background of Key Concepts

5.1 Language Models

Since computers were created, humans have dreamed of imbuing them with natural language understanding: when Alan Turing formalized the Turing Machine, he also formalized the notion of a Turing Test, which would determine whether artificial intelligence had been achieved based on whether a computer could converse with other humans at a human level [63]. Deep learning model

architectures for natural language processing– dubbed Large Language Models (LLMs)– are manifestations of this long held desire, and arguably some of them have passed the Turing Test. As part of passing the Turing Test, transformer models like ChatGPT-4 show evidence of being able to reason. How do these models work?

The key model architecture powering ChatGPT-4 is called the transformer [43]. It is the most successful architecture configuration for natural language processing so far in a sequence of related model architectures. Tracing this lineage provides some intuition for why transformer architectures work so well.

Modeling human language means modeling a dynamic system: if we think about words as occupying different timesteps in a sentence, then the probability of a word occupying timestep t depends on the probability of a word occupying timestep $t-1$ and likely all the words from $t-1, t-2, \ldots t-k$ for some value k. In other words, language is full of patterns, and these allow us to guess things like the missing letter in the following word with high confidence:

$$H \quad e \quad l \quad l \quad __$$

Why do patterns occur? Patterns indicate the presence of some correlated signals in space or time [49]. From an entropy perspective, correlations reduce the number of states a system can have, and therefore reduce the required amount of information needed for transmitting a message (I don't need a full five bytes to communicate the word "hello" with high confidence, as demonstrated above, because of English spelling rules). Generating believable human-sounding language can therefore be framed as finding the proper set of signals that will let us generate bytes of language in recognizable patterns. For these reasons, if a set of observations contains patterns, then the space of generating signals for these observations will likely be smaller than the observation space itself (i.e. learning to generate human-sounding English means learning the smaller set of rules/signals/features governing English grammar, spelling, and vocabulary usage). Therefore, the story of deep learning's evolution has in large part been a story about removing human-coded features from models and instead asking models to develop their own compressed representations of the world, which is what models like ChatGPT-4 do [1].

As mentioned initially, the patterns in language are dynamic: they contain autoregressive patterns that depend on values of previous timesteps. How can we bias a deep learning model towards learning functions that depend on previous timesteps of input? The Recurrent Neural Network (RNN) architecture was one attempt to do this.

However, in practice, this architecture suffers from vanishing and/or exploding gradients during the backpropagation learning process, and empirically these models only learn functions that depend on at most 10 steps of previous input [54].

[1] For example, GPT3 represents a compression of all the text on the internet captured in the The Pile (∼886 GB [6]) into a model of size ∼350GB [64].

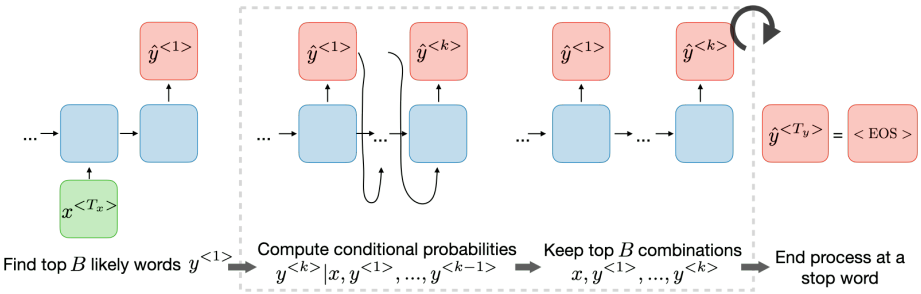


Figure 1: Basic RNN Architecture. Source: Afshine Amidi and Shervine Amidi, *Recurrent Neural Networks cheat sheet*, 2020.

The LSTM network design introduces forgetting gates that in theory allow the model to carry more relevant bits of information forward across a greater number of timesteps without encountering numerical issues.

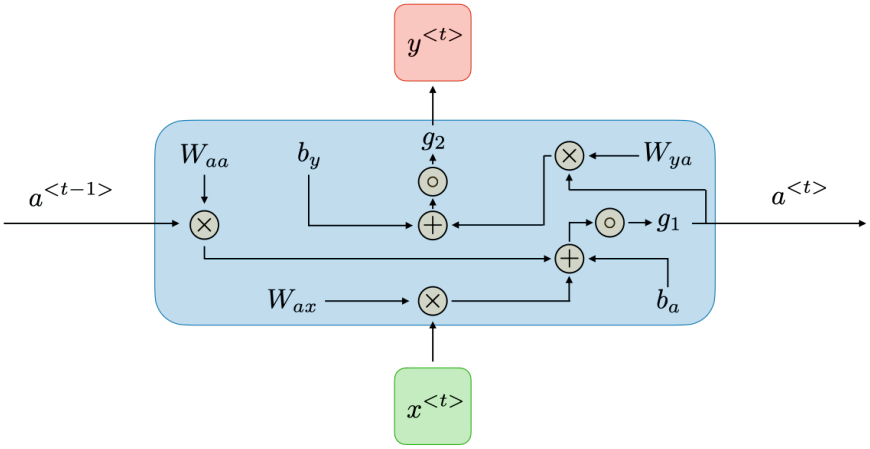


Figure 2: Basic LSTM Architecture. Source: Afshine Amidi and Shervine Amidi, *Recurrent Neural Networks cheat sheet*, 2020.

Empirically, LSTMs have been found to learn patterns across more than 1000 timesteps [54]. However, the number of operations required to learn a relationship between two points in the input sequence grows linearly with the length of the sequence, making it challenging to learn long-term dependencies between data points. Transformers use attention to solve this problem, and effectively reduce the required number of operations for computing relationships between data points in their context window to a constant number of operations [58].

We note that GPT3 is simpler still in terms of design: it uses only stacked decoder layers, no encoder layers. It is trained to predict the next likeliest

token of text given a longer previous sequence of text. It does this by learning to reduce the cross-entropy errors of its next-token prediction. In appendix F, we show how this is equivalent to learning the underlying distribution of the text.

GPT3 and its successors also rely on later stages of training in order to give outputs that respond so well to our instructions. This later stage of training is generally referred to as "fine tuning" the model. OpenAI notably used reinforcement learning from human feedback to tweak the weights of the underlying transformer model in accordance with a model of human preferences [15]. Later in this thesis, we discuss whether a successful algorithm on this benchmark could employ a similar strategy by learning the compiler's "preferences".

Transformers like GPT3 are not bottlenecked by human engineers' abilities to construct relevant hidden signals (i.e. features) for language. On the other hand, this means GPT3 is not easily interpretable, which can lead to alarming consequences in certain situations. For example, since we cannot easily characterize the features and circuits [2] behind GPT3's code generation abilities, we cannot easily explain why it outputs the code it does, or why it might output code with nefarious bugs. To this end, asking the LLM to output verification-aware code may be a way to control its outputs without understanding fully how those outputs are generated. For this reason, our presented benchmark may also be a way to distinguish trustworthy model outputs from untrustworthy model outputs.

5.2 Verification-Aware Languages

In this thesis, we specifically propose evaluating language model outputs in the domain of verification-aware programming languages.

One core piece of theory powering the development of verification-aware languages is the idea that programs are proofs. This is a direct consequence of the Curry-Howard correspondence, a foundational principle that establishes a deep isomorphism between computer programs and mathematical proofs. According to this principle, writing a program corresponds to proving a theorem; the program's type acts as the theorem's statement, and the program itself serves as the proof. For example, a function typed as mapping integers to integers in a programming language can be seen as a proof of the proposition that for all inputs, an integer implies an integer [5].

Building on this theoretical foundation, the study of automated theorem proving has evolved significantly since its inception in the 1950s [57], leading to the development of sophisticated first-order logic solvers like Z3 (which underpins Dafny) and CVC4. These solvers strive to tackle the inherently NP-hard challenge of theorem proving, as detailed in the supplementary material on SMT solvers [Appendix H].

[2] Neel Nanda, one of the pioneers of the nascent field of mechanistic interpretability, defines features as "the variables inside the model" and circuits as "the algorithms learned to compute features" [40].

Languages such as Coq, Lean, Isabelle [3], and Dafny exemplify the evolution of programming languages towards environments where programming and mathematical theorem proving are not just parallel activities but deeply integrated processes. Coq and Lean enable the explicit formulation and proof of mathematical theorems within the programming process. These platforms facilitate the creation of both verified software, where the correctness of algorithms is rigorously established, and extensive formal mathematics, with proofs verified by the computer.

Coq and Lean are examples of interactive theorem provers, which require that the user write out tactics in order to complete proofs. On the other hand, Dafny relies on Hoare logic to prove that functions and methods adhere to given pre- and post- conditions; implementing algorithms in Dafny feels much more akin to regular programming. To get a sense of the style in each language, consider the following examples. In the first example, we implement and prove the correctness of a function in Coq that calculates the length of an input list. In the second example, we implement and prove the correctness of a method in Dafny that returns the length of an input sequence.

In Coq, to write our proof of correctness, we stitch together tactics like 'simpl' and 'rewrite IHs' to prove our desired program property.

```
Inductive seq : nat -> Set :=
  | niln : seq 0
  | consn : forall n : nat, nat -> seq n -> seq (S n).

  Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
    match s with
    | niln => 0
    | consn i _ s' => S (length i s')
    end.

  Theorem length_corr : forall (n : nat) (s : seq n), length n s =
      n.
  Proof.
    intros n s.

    (* reasoning by induction over s. Then, we have two new goals
       corresponding on the case analysis about s (either it is
       niln or some consn *)
    induction s.

      (* We are in the case where s is void. We can reduce the
         term: length 0 niln *)
      simpl.

      (* We obtain the goal 0 = 0. *)
      trivial.

      (* now, we treat the case s = consn n e s with induction
         hypothesis IHs *)
```

[3] Coq, Lean, and Isabelle feature in the top list of languages that have been used to formalize a significant number of theorems in the "Formalizing 100 Theorems" challenge `https://www.cs.ru.nl/~freek/100/`

```
      simpl.

      (* The induction hypothesis has type length n s = n.
         So we can use it to perform some rewriting in the goal: *)
      rewrite IHs.

      (* Now the goal is the trivial equality: S n = S n *)
      trivial.

    (* Now all sub cases are closed, we perform the ultimate
       step: typing the term built using tactics and save it as
       a witness of the theorem. *)
  Qed.
```

Now for the Dafny example:

```
method find_length<T>(l: seq<T>) returns (len: nat)
    ensures len == |l|
{
    len := 0;
    for i := 0 to |l|
        invariant i == len
    {
        len := len + 1;
    }
}
```

To verify our method in Dafny, which allows us to write in an imperative programming style, we need to state the property we want to verify using an 'ensures' clause at the beginning of the method and then provide a compiler hint to the verifier in the form of an 'invariant' clause within the for loop. Qualitatively, then, writing in Dafny feels more like writing in a prevalent, readable language like Python.

6 Related Work

DafnyBench fills a unique niche in the current landscape of benchmarks, and based on the algorithms that do well on related benchmarks, DafnyBench has a strong shot at being solved within the next five years.

6.1 Benchmarks

6.1.1 Formal Reasoning Benchmarks

MiniF2F: this benchmark assesses models' reasoning capabilities by requiring them to complete proofs in the formal languages Metamath, Lean, Isabelle, and HOL Light. It consists of 488 problem statements drawn from the AMC, AIME, and IMO. [75]

Archive of Formal Proofs (AFP): this is an online repository of formal proofs for the Isabelle proof assistant. It contains over one million lines of code. [10]

MATH dataset: This dataset contains 12,500 challenging competition math problems with step by step solutions for each (all in natural language). [26]

PISA: PISA contains 138,000 Isabelle proofs from the Isabelle Standard Library and Archive of Formal Proofs. [2]

NaturalProofs: NaturalProofs contains ∼15,000 definitions, proofs, and theorem statements sourced from ProofWiki. [61]

GSM8k: This dataset contains ∼8,500 high quality linguistically diverse grade school math problems. [16]

CoqGym: CoqGym contains 71,000 human-written proofs in Coq. [69]

LeanDojo: LeanDojo provides a programmatic interface for interacting with the proof environment, and also establishes a benchmark consisting of 98,734 theorems and proofs extracted from Lean's math library. [70]

6.1.2 (Unverified) Programming Benchmarks

Most current programming benchmarks, including HumanEvalX, Mostly Basic Python Programs (MBPP), and APPS, evaluate code generated by models against a finite set of test cases. While Large Language Models have demonstrated effectiveness on simpler datasets involving single-function programming problems, they face greater challenges with more complex, multi-function solutions, like those in APPS.

To give a visceral sense of the difference in problem difficulty, first consider this solution to HumanEvalX problem 22 implemented in Python [76]:

```
def filter_integers(values: List[Any]) -> List[int]:
    return [x for x in values if isinstance(x, int)]
```

Now consider this expected solution to APPS problem 0 implemented in Python [25]:

```
for _ in range(int(input())):
    n = int(input())
    mass = []
    zo = 0
    oz = 0
    zz = 0
    oo = 0
    ozs = []
    zos = []
    ozss = set()
    zoss = set()
    for j in range(n):
        k = input()
        mass.append(k)
        if k[0] == '0' and k[-1] == '1':
            zoss.add(k)
            zos.append(j + 1)
```

```
            zo += 1
        elif k[0] == '1' and k[-1] == '0':
            ozss.add(k)
            ozs.append(j + 1)
            oz += 1
        elif k[0] == '0' and k[-1] == '0':
            zz += 1
        else:
            oo += 1
    if zz and oo and not oz and not zo:
        print(-1)
        continue
    else:
        if zo > oz:
            print((zo - oz) // 2)
            ans = []
            need = (zo - oz) // 2
            i = 0
            while need:
                zzz = mass[zos[i] - 1][len(mass[zos[i] - 1]) - 1::
                    -1]
                if zzz not in ozss:
                    ans.append(zos[i])
                    need -= 1
                i += 1
            print(*ans)
        else:
            print((oz - zo) // 2)
            ans = []
            need = (oz - zo) // 2
            i = 0
            while need:
                zzz = mass[ozs[i] - 1][len(mass[ozs[i] - 1]) - 1::
                    -1]
                if zzz not in zoss:
                    ans.append(ozs[i])
                    need -= 1
                i += 1
            print(*ans)
```

The most established benchmarks for (unverified) code generation include:

APPS: contains 10,000 problems ranging from introductory programming problems to hard programming challenges [25].

HumanEvalX: contains ∼165 programming problems with natural language descriptions, implemented in Python, Go, Java, C++, and Javascript [76, 14].

MBPP: Mostly Basic Python Problems contains 974 programming tasks designed to be solved by entry-level programmers [7].

LiveCodeBench: LiveCodeBench mitigates data contamination issues by constantly scraping newly published code problems from weekly competitions held on sites like Leetcode and Codeforces. [28]

SWEBench: SWEBench provides 2,294 problems drawn from real Github issues. [30]

6.1.3 Formally Verified Programming Benchmarks

The two existing benchmarks specifically geared towards formally verified programming are both written in Dafny. These are called Clover and dafny-synthesis, and contain 66 and 153 samples respectively. As stand-alone benchmarks, Clover and dafny-synthesis do not accurately represent the most challenging or realistic Dafny programming scenarios. For example, all of the examples in Clover are single-method Dafny programs, and sometimes the formal specification for the program directly repeats the implementation of the algorithm. For example, consider the method 'updateElements' below.

```
method UpdateElements(a: array<int>)
  requires a.Length >= 8
  modifies a
  ensures old(a[4]) +3 == a[4]
  ensures a[7]==516
  ensures forall i::0 <= i<a.Length ==> i != 7 && i != 4 ==> a[i]
      == old(a[i])
{
  a[4] := a[4] + 3;
  a[7] := 516;
}
```

Here, the ensures clauses lay out very explicit hints for constructing the algorithm solution.

Dafny-synthesis has some multi-step problems with helper functions, which Clover does not have. However, dafny-synthesis translates MBPP problems using GPT-4, which may skew the distribution of represented problems away from real-world Dafny problems that may be too hard for GPT-4 to verify [37].

In terms of evaluation, Clover proposes a sophisticated three-way consistency check between code, docstrings, and annotations. Their checker achieves an 87% acceptance rate of correct implementations on the Clover benchmark while rejecting all incorrect implementations [55]. The authors note that equivalence checking with natural language is currently weak, but can hopefully be improved upon [56]. In light of this, we intentionally avoid incorporating any sort of dependence on natural language equivalence checking into DafnyBench.

6.2 Algorithms

Research on algorithms has primarily focused on enabling models to write proofs, demonstrate general reasoning or planning abilities, or write Python code, with little attention to the intersection of these tasks: verified programming.

6.2.1 Algorithms for Proof Writing (Formal Reasoning)

Since the beginning of the 2020s, Large Language Models have dominated existing formal reasoning benchmarks. In 202l, Ilya Sutskever and a colleague tested whether transformer-based language models could demonstrate reasoning abilities by seeing whether a transformer model could prove theorems in Metamath. Indeed, in their paper titled GPT-f, the researchers found that generative pre-training and fine-tuning on mathematical data (like that found in arxiv) boosted model performance at proving theorems in Metamath above previous state of the art level on the benchmark at the time. They also found that performance generally scaled with model size. Finally, they unlocked a potentially unlimited stream of training data: they found that training a value function on proofs generated by the prover improved model performance [47].

A year later, Welleck et al eked out improved results using NaturalProver, which involves prompting the model with relevant proof references and lexically constraining the model output. The references and constraints applied to GPT3 prevented detrimental hallucinations and outperformed other methods on their NaturalProofGen benchmark [62]. In 2022, Ilya et al reclaimed state of the art on the MiniF2F dataset by implementing expert iteration with a transformer model. Expert iteration involves training a model on its own previous training trajectories in order to achieve continuous improvement. The training trajectories typically become more complex as the model becomes stronger, creating a natural sort of easy to hard "curriculum" for the model during the training process [46].

In 2024, GPT-4 self-verified with compiler feedback holds the SOTA result on the MATH dataset. The authors achieve this result by prompting GPT-4 to self-verify using its code interpreter. If the code verification returns a false result, the model automatically rectifies its errors. The researchers find that more iterations with the code verifier lead to increased levels of accuracy on the MATH dataset, ultimately boosting performance from 53.9% response rate to a 84.3% response rate [78].

Notably in 2024, AlphaGeometry has achieved significant success in solving International Mathematical Olympiad (IMO) geometry problems. It does this by synthesizing millions of theorems and proofs across different levels of complexity and using this as training data for a Large Language Model that then acts as a guide for a symbolic deduction engine. The proof process is initiated by running a deductive logic solver until it either finds the final proof state or exhausts itself. If the latter occurs, the neural model will suggest its most likely auxiliary construction, which in the context of geometry proofs may look like constructing a midpoint for a certain segment. The deductive logic solver then runs again, and the entire process iterates like this. Ultimately, this method performs near the level of IMO gold medalists on IMO geometry problems [57].

2024 state of the art (SOTA) results on the MiniF2F benchmark follow from an insight on LLMs' abilities for autoformalization: Autoformalization with Large Language Models finds that language models already have an innate ability to translate natural language to formal statements. Out of 150 natural

language mathematical statements, LLMs out of the box can correctly translate 35 to formal statements. The authors note this ability unlocks potential for generating more training data for neural-guided proof solvers [68]. The authors then leverage this insight to achieve state of the art on the MiniF2F benchmark. They do this by training Thor on autoformalized theorems. Thor trains a transformer model from scratch on the Github and Arxiv subset of The Pile dataset, and then preprocesses all propositions it wants the model to prove before querying the model using a format similar to GPT-f. This preprocessing step involves using a pre-existing hammer method for premise selection, which language models typically struggle with on their own. This method boosts the model's success rate on the PISA dataset from 39% to 57% [29]. With extra training on the data produced by autoformalizing additional theorems with Large Language Models, Thor achieves state of the art on the MiniF2F benchmark, solving 35.2% of the problems at a pass@1 rate [68].

6.2.2 Algorithms for General Reasoning

Multiple papers have attempted to increase the general reasoning ability and consistency of Large Language Model outputs by experimenting with various prompting techniques: for example, multi-shot prompting (in which multiple similar prompts and responses are prepended to the model prompt) has been shown to significantly boost Large Language Model performance across a wide variety of tasks like those documented in Google's BigBench benchmark. [53, 9]. Similarly, Chain of Thought prompting, in which models are prompted with examples and logical explanations for completing those examples, has been shown to significantly boost models' reasoning performance. [60]. Tree of Thought approaches build upon Chain of Thought approaches by allowing models to explore multiple reasoning paths and backtrack when necessary. [71]. Belief graphs paired with truth maintenance algorithms have been shown to boost the logical consistency of LLM outputs, as shown by the paper titled Language Models with Rationality [32]. The StaR model attempts to bootstrap reasoning with reasoning by using few multishot chains of reasoning to have a model generate rationales for a much larger set of problems; for incorrect outputs the models are encouraged to try again to create a rationale using the correct answer; the model is fine-tuned on the correct rationales it has generated thus far; the process repeats. This method empirically emulates a 30x larger model fine-tuned on CommonsenseQA [73].

Often, reasoning over multi-step problems well means having a way to effectively evaluate intermediate reasoning steps: at a non-terminal step in the reasoning process, is the model on the right track? Let's Verify Step by Step trains a process-supervised reward model to score each intermediate step of GPT-4's reasoning process using labels collected in the publicly released PRM800k dataset [34]. AlphaZero, which plays Go at a superhuman level, also trains a value function to score intermediate board states in order to guide the planning algorithm choosing next moves in the game [52]. Arora et al. learn a verifier that predicts which actions are valid or not in a given state, and enable GPT-2 to

perform effectively in the Blocksworld domain [41]. Relatedly, Lovon-Melgarejo et al. investigate how to generalize models' single-hop reasoning capabilities to multi-hop reasoning capabilities [35].

Larger models are capable of self-reflecting on their own outputs. Reflexion [51] and LATS [77] harness this capability to propose more general planning frameworks for LLMs. LATS specifically couches self-reflection in a larger tree-search algorithm for planning with LLMs.

6.2.3 Algorithms for (Unverified) Programming

Program synthesis has existed as a problem before the era of deep learning, and naturally classical approaches have attempted to tackle this problem before the advent of LLMs. Recently, researchers have investigated the consequences of merging neural-based approaches enabled by Large Language Models with traditional techniques. For example, BUSTLE uses learning algorithms to prioritize partial programs in a bottom-up program synthesis scheme (in which programs are assembled based on input and output examples by piecing together smaller, valid expressions in a language) [8]. Kevin Ellis takes the idea of assembling partial programs into useful programs a step further: he introduces a wake-sleep algorithm for library learning, which learns a recognition model for identifying useful program fragments and uses this to combine partial programs during a wakeful neurally guided search phase. Then, during the sleep phase, the recognition model trains on the tasks it solved during waking and the tasks it "fantasized" during waking (i.e., the partial programs it created that did not actually solve an intended task). In the third phase before returning to waking, the programs assembled thus far are refactored– common pieces of the programs are pulled out and consolidated into library functions of the domain-specific language the model is learning [21, 22]. Similarly, Lilo learns libraries of interpretable code for solving problems, and documents each library function with a natural language docstring [23]. AlphaCode trains a large transformer model on coding data from Github, and then fine-tunes this model on gold-level competitive coding data. During inference time, AlphaCode outputs many program samples that it then filters and clusters based on program behavior [33].

Other methods for program synthesis with deep learning lean more heavily on providing appropriate reward signals to Large Language Models in order to synthesize correct code. For example, fine-tuning foundation models like GPT-3 [13] and Llama [4] have produced derivative models that do well on code synthesis benchmarks like HumanEval. StaRCoder has been trained from scratch 80+ programming languages [3]. The Big Code Models leaderboard lists about 50 Large Language Models trained specifically for the code generation task [3]. Variants on vanilla fine-tuning LLMs on code-related text include training models on the execution dynamics of code. For example, LEVER trains a verifier to evaluate the correctness of a program based on natural language input, the program itself, and its execution results. Program candidates sampled from an LLM are then re-ranked according to this verifier score and the LLM's logit scores [41]. The CodeT5 model utilizes a transformer architecture that employs

denoising sequence-to-sequence pretraining, and another pretraining step that asks the model to identify which tokens in a chunk of code are identifiers and recover them when masked [59]. Coarse-Tuning Models of Code with Reinforcement Learning Feedback propagates compiler signals to the language model in a similar way to how Reinforcement Learning from Human Feedback propagates human feedback signals to the underlying model [27].

We can also layer inference time algorithms on top of model output for boosting performance. For example, Parsel decomposes problem statements into subtasks that a language model implements and verifies independently [72]. Planning with Large Language Models for Code Generation uses a planning algorithm to do look-ahead search and guide transformer outputs for code. The planner generates candidate programs and tests them on public test cases [74]. Recent works have also explored incorporating multiple agents into the code-generation process. For example, AgentCoder holds the current State of the Art position on the HumanEval benchmark, and uses a collaborative system of agents: one agent specializes in writing code, another agent specializes in designing test cases, and a third agent specializes in executing the test cases and writing feedback to the code writer agent [**?**].

6.2.4 Algorithms for Verified Programming

Compared to the categories above, considerably fewer algorithms have been implemented for solving formally verifiable programming problems that reflect practical programming challenges as opposed to esoteric mathematical theorems or unverified programming problems. Orenes-Vera et al explore GPT-4's ability to write SystemVerilog Assertions for processor designs, and notably their generated assertions uncover a bug in the CVA6 core that eluded prior works [44]. More generally, Verified Multistep Tree Search Using Large Language Models and Monte Carlo Tree Search uses a monte-carlo tree search planning algorithm to guide LLMs' output as they solve multi-step reasoning problems in Coq, Lean, and Dafny. A diagram of this planning algorithm is included below [12].

We note that under the VMCTS setup here, running VMCTS on Category 1 problems in our DafnyBench benchmark, which only have a single Dafny method, should be equivalent to strict sampling from an underlying LLM (since each child in the VMCTS monte carlo tree is a complete lemma or method).

7 Benchmark Construction and Design

We now discuss the benchmark design that will allow us to further evaluate the aforementioned algorithms in the formally verified programming context.

7.1 Methods of Construction

7.1.1 Rationale Behind Language Choice

Why Dafny? We choose to focus this benchmark on Dafny for three main reasons:

1. **Gap:** Relatively robust infrastructure exists for Lean, Coq, and Isabelle, but not so much for Dafny, even though there have been very recent pushes to expand the prevalence of Dafny on the internet [11].
2. **Practical value:** Dafny feels much more similar to writing in Python, the most widely used language in the software development world [45], versus Isabelle, Lean, or Coq. A 2023 study finds that Dafny users typically write as much auxiliary code for verification to implementation as programmers do for implementation versus unit testing (in a ratio of 1.14) [31].
3. **Compile-time verification:** This is a stricter form of verification, and more important for mission-critical situations in which we want guarantees on our code before we execute it on specific inputs, and we want those guarantees to extend to all possible algorithm inputs. Verification solutions like Deal [1], while they work in Python, are weaker in some sense because they verify at runtime, not compile-time.

7.1.2 Sourcing Ground Truth Programs

In total, our DafnyBench benchmark contains ∼1063 ground truth stand-alone Dafny programs that compile. These ground-truth problems have been secured from four main sources:

1. First, we scraped all publicly available GitHub files published on the site in December 2023. At this point in time, the vast majority of these files were presumably written by humans. The relevant files were returned from the GitHub API using the `language:Dafny` search command. We then deduplicated these files using a minhash deduplication script written by Chenghao Mou. Minhash deduplication is described in Appendix A. The deduplication process reduced the number of `.dfy` files from ∼15,000 to ∼5,000. We then attempted to verify each of these remaining files using the `dafny verify` command with a local installation of Dafny 4.3.0, and we removed any files that contained only commented lines. Additionally, we removed all of the files from the Clover repository, which had already been formatted as benchmark files and would be selectively merged with our benchmark later. Lastly, we removed all Dafny files that did not have function or method keywords (using a regex script). This cut our number of usable Dafny files to 847. We retain the repositories from which each of these files came so as to give credit where it is due and also give researchers an opportunity to mitigate data contamination (models run on our benchmark should ideally not be trained on data from the repositories listed in Appendix D).

2. Secondly, Chloe and Professor Amin manually translated a collection of 10 HumanEvalX problems into Dafny (Appendix C). For most of these problems, we asked GPT-4 to first give its best attempt at translation, and then we manually added correct invariants and assert statements with guidance from the VSCode Dafny Extension. Additionally, Chloe and Anish Mudide from MIT collaborated to prove that a non-standard python implementation of bit addition with length 10 binary strings was functionally equivalent to adding two length 10 bitvectors in Dafny (`BinaryAddition.dfy` in the benchmark).

3. Third, we ported the ground-truth examples given by the Clover benchmark into our DafnyBench with permission from the Clover authors. These constitute 66 files.

4. Lastly, we ported the MBPP-DFY-153 `.dfy` files from Md's dafny-synthesis repository. We clearly cite each of these files. This boosts our total file count by another 153 files.

Overall, the mean ground-truth program in our dataset has 1.96 methods, 0.99 functions, and 0.98 lemmas. This places the mean complexity of our examples at a level higher than Clover, which only has one stand-alone method per example.

We roughly divide our benchmark samples into two categories. Category 1 problems are most similar to Clover– they contain only one stand-alone method, and for this reason are simpler. Category 2 problems are everything else involving more than one method, and some number of helper functions and lemmas. We expect benchmark solutions to completely solve Category 1 related tasks before completely solving all Category 2 related tasks. Please see the full list of ground truth program characteristics here).

7.1.3 Suggested Training Data

We save the remaining ∼4,000 `.dfy` files from our deduplicated GitHub scrape as training data suggestions for researchers who would like to build models for this benchmark. While none of these files compile as stand-alone files, a majority of them likely still represent valid Dafny code with dependencies distributed across files. For this reason, they likely still contain valuable signals for fine-tuning methods.

7.2 Task Design

At the time of this writing, we have fully implemented two tasks: the *fill_body* task and the *fill_annotations* task.

For reference in the following task description, please keep in mind this ground truth implementation of `FindMaxWord.dfy`.

```
/*
HumanEvalX 158
```

```
Write a function that accepts a list of strings. The list contains different words.
Return the word with maximum number of unique characters. If multiple strings have maximum number of unique
characters, return the one which comes first in lexicographical order.
*/

function numUniqueChars(s: string): nat {
    |set c | c in s|
}

method FindMaxWord(words: seq<string>) returns (maxWord: string)
    ensures maxWord in words || |words| == 0
    ensures forall w :: w in words ==> numUniqueChars(w) <= numUniqueChars(maxWord)
{
    if |words| == 0 {
        maxWord := "";
        return;
    }
    var word := words[0];
    maxWord := words[0];
    var maxUniqueChars := |set c | c in maxWord|;
    assert maxUniqueChars == numUniqueChars(maxWord);
    for i := 0 to |words|
        invariant maxWord in words
        invariant maxUniqueChars == numUniqueChars(maxWord)
        invariant numUniqueChars(word) <= numUniqueChars(maxWord)
        invariant (forall w :: 0 <= w < i ==> numUniqueChars(words[w]) <= numUniqueChars(maxWord))
    {
        word := words[i];
        var uniqueChars := set c | c in word;
        var count := |uniqueChars|;
        assert count == numUniqueChars(word);
        if count > maxUniqueChars || (count == maxUniqueChars && word < maxWord) {
            maxUniqueChars := count;
            maxWord := word;
            assert maxWord == word;
        }
        assert maxUniqueChars >= count;
        assert maxUniqueChars == numUniqueChars(maxWord);
    }
    assert forall w :: w in words ==> numUniqueChars(w) <= numUniqueChars(maxWord);
}
```

7.2.1 Fill_body

In this task, we take the ground-truth programs and remove the bodies of any lemmas and methods in these program files. We replace the removed bodies with `/* TODO */` tags. For example, consider `FindMaxWord_fillbody.dfy`:

```
/*
```

```
HumanEvalX 158
Write a function that accepts a list of strings.
The list contains different words.
Return the word with maximum number of unique characters.
If multiple strings have maximum number of unique
characters,
return the one which comes first in lexicographical order.
*/

function numUniqueChars(s: string): nat {
    |set c | c in s|
}

method FindMaxWord(words: seq<string>) returns (maxWord: string)
    ensures maxWord in words || |words| == 0
    ensures forall w :: w in words ==> numUniqueChars(w) <=
    numUniqueChars(maxWord)
{
    /* TODO */
}
```

Listing 1: FindMaxWord_fillbody.dfy

In order to successfully complete this task, a model must synthesize an appropriate algorithm and insert adequate compiler hints (assert/invariant statements) for making it compile.

7.2.2 Fill_annotations

In this task, we take a ground-truth program and remove all of its compiler hints (i.e., all of the assert and invariant statements in the body of the code). We do not demarcate from where these hints have been removed, i.e., we do not insert `/* TODO */` after we remove each compiler hint, since this would make the task overly easy. For example, consider `FindMaxWord_fillanno.dfy`:

```
/*
HumanEvalX 158
Write a function that accepts a list of strings. The list contains
    different words.
Return the word with maximum number of unique characters. If
    multiple strings have maximum number of unique
characters, return the one which comes first in lexicographical
    order.
*/

function numUniqueChars(s: string): nat {
    |set c | c in s|
}

method FindMaxWord(words: seq<string>) returns (maxWord: string)
    ensures maxWord in words || |words| == 0
    ensures forall w :: w in words ==> numUniqueChars(w) <=
        numUniqueChars(maxWord)
{
    if |words| == 0 {
        maxWord := "";
```

```
        return;
    }
    var word := words[0];
    maxWord := words[0];
    var maxUniqueChars := |set c | c in maxWord|;
    for i := 0 to |words|
    {
        word := words[i];
        var uniqueChars := set c | c in word;
        var count := |uniqueChars|;
        if count > maxUniqueChars || (count == maxUniqueChars &&
            word < maxWord) {
            maxUniqueChars := count;
            maxWord := word;
        }
    }
}
```

We note that many of the `_fillanno.dfy` files still verify with their compiler hints removed. Therefore, the final score on this benchmark task is the difference between the number of verifiable `.dfy` files and the number of verifiable `.dfy` files the model produces. Note in this case that scores can be negative if the model breaks files that could originally be verified.

7.3 Evaluation Design

Setting the design barrier for what the benchmark evaluation should tolerate is not entirely trivial. Do we punish weaker open-source models for formatting errors? For example, code-phind-llama sometimes places only one method per "`dafny`" block in its raw output even if the full problem solution involves multiple methods and helper functions. In general, models may express correct thoughts in a fragmented way, and it's tempting to make the benchmark evaluation code account for everything. The most generalizable solution, however, is also the simplest solution: we expect submissions to the benchmark to account for quirks of model formatting, and we expect attempted solutions to be submitted as `.dfy` files that the benchmark treats as continuous pieces of code. To this end, our benchmark evaluation script simply checks whether a solution `.dfy` file verifies within 10 minutes using the `dafny verify` command. We also check for model hacks, the primary ones in Dafny being the insertion of {`:verify false`} and `assume false` in Dafny solutions. Both of these lines cause the dafny verifier to pass the code without rigorously checking adherence to the provided formal specifications. Finally, we only pass a submitted benchmark solution if it contains executable Dafny code (i.e., it is not just a file with a bunch of comments) and if it leaves the formal specification of the ground-truth program unmodified. Here, we define the formal specification as the set of all function, method, and lemma signatures with all of their associated `ensures` and `requires` clauses (and the helper functions used to define those ensures and requires clauses). For example, the formal specification of `FindMaxWord.dfy` can be expressed as follows:

```
    formal_spec = [
        """function numUniqueChars(s: string): nat {
            |set c | c in s|
        }""",
        """method FindMaxWord(words: seq<string>) returns (maxWord:
             string)
            ensures maxWord in words || |words| == 0
            ensures forall w :: w in words ==> numUniqueChars(w) <=
                 numUniqueChars(maxWord)
        {"""]
```

See Appendix E for examples of hacks and failure modes that this evaluation scheme will catch.

8 Proposed Algorithms for Success on this Benchmark

Broadly, successful approaches to this benchmark will likely contain elements from successful approaches to related benchmarks. Based on the related work section above, these include:

1. Pretraining on Dafny-related data, which may be synthetically generated[4].
2. Expert iteration with curriculum learning on Category 1 problems in the benchmark, and then Category 2 problems.
3. Output restrictions such that LLMs only output syntactically valid Dafny expressions.
4. Inference-time planning algorithms like that inspired by Verified Monte-Carlo Tree Search (VMCTS) [12].
5. Trained value functions that are effective at estimating the value of partial programs and partial Dafny proof implementations.

From initial small-scale experiments, it is clear that a successful implementation on this benchmark must master the syntax and then the semantics of the Dafny language. Such an implementation may be conceivable with existing open-source models provided that we give stronger controls for avoiding syntax errors. One simple experiment points in this direction: when we run code-phind-llama on a subset of DafnyBench, it solves 12 out of the 27 problems in the subset. One place it consistently fails is in defining for and while loops in the Dafny style. Consider the following phind-code-llama attempt at implementing 'arrayProduct':

[4]Fuzzing techniques generate a lot of potentially useful synthetic Dafny data as a byproduct. For example, Huang et al. outline a method for automatically fuzzing the Dafny compiler, and this pipeline could be adopted and modified by researchers looking to create synthetic training data for their algorithms on this benchmark. https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/2223-ug-projects/fuzz-d-Random-Program-Generation-for-Testing-Dafny.pdf

```
method arrayProduct(a: array<int>, b: array<int>) returns (c: array<int>)
 requires a.Length==b.Length
 ensures c.Length==a.Length
 ensures forall i:: 0 <= i < a.Length ==> a[i] * b[i] == c[i]
{
 var c := new int[a.Length];
 for (var i := 0; i < a.Length; i++)
   c[i] := a[i] * b[i];
 return c;
}
```

If we instead fine-tune phind-code-llama using Direct Preference Optimization (see Appendix G) on successful and failed completions to DafnyBench problems outside of this subset, we boost its performance to solving 17 problems out of the same 27 problems. A lot of this improvement is because of greater syntax control, as evidenced in this successful attempt to implement 'arrayProduct':

```
method arrayProduct(a: array<int>, b: array<int>) returns (c: array<int>)
  requires a.Length==b.Length
  ensures c.Length==a.Length
  ensures forall i:: 0 <= i< a.Length==> a[i] * b[i]==c[i]
{
  c := new int[a.Length];
  var i: int := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length
    invariant forall j :: 0 <= j < i ==> a[j] * b[j] == c[j]
  {
    c[i] := a[i] * b[i];
    i := i + 1;
  }
}
```

Syncromesh and Outlines both offer promising alternative solutions to DPO and other fine-tuning based approaches. The aforementioned libraries allow users to place regex constraints on Large Language Model outputs. One can imagine leveraging Syncromesh in particular for ensuring language models output only valid expressions in the Dafny language. For example, it is possible to write something like the following in order to force the language model to only output "line number [x]" followed by an assert or invariant statement:

```
class DafnyActionCompletionEngine:
    def __init__(self, current_program):
        self._current_program_lines = current_program.split('\n')

    def complete(self, prefix: str) -> list[regex.regex]:
        # Will allow:
        # '' -> "// On line <L>, add assert"
```

```
        #     | "// On line <L>, add invariant"
        if not prefix:
            line_numbers = ('(' +
                            '|'.join(map(str, range(len(self.
                                _current_program_lines)))) +
                            ')')
            return regex.compile(f'// On line {line_numbers}, add (
                assert|invariant) ')
            # return regex for choosing line and action type
        else:
            # Any of the allowed characters, up to 80 of them.
            # Here, we can limit it to syntactically valid
                expressions
            # that also only use valid variable names.
            # Since the prefix is in the line number, we can
            # limit it to variables that have been declared
            # up to the current point.
            #return regex.compile('[()><%0-9a-zA-Z\\[\\]:]{0,80};\n
                ')
            return regex.compile('[^;]{0,80};\n')

    def is_complete(self, prefix: str) -> bool:
        return prefix.endswith(';\n')
```

Based on initial experiments, any planning-based approaches for solving this benchmark will likely need to unlock sources of higher granularity feedback. By this we mean that effective models likely need feedback after each generated line of code instead of after each full method, function, or lemma. VMCTS is no better than straight sampling on Category 1 problems, for example, because we do not run the verifier until we have a complete expression that can verify (i.e. after we completely generate a section of code enclosed by curly braces, which corresponds to a complete lemma or complete method implementation).

One approach for achieving higher granularity in feedback may involve treating the model action space as the edit space, not linear token generation space. This way, the verifier will be able to give relevant feedback to every edit of a pre existing base program.

Another key observation from smaller experiments on subsets of this benchmark is that models tend to "overprove". Open source models struggle to recognize when they can offload work to Dafny's automated SMT solver. For example, consider the following model attempt. The model's efforts in the last two lines are completely unnecessary because 'lemma sum_plus' verifies with an empty body!

```
function sum(s: seq<int>, n: nat): int
    requires n <= |s|
{
        if |s| == 0 || n == 0 then
                0
        else
                s[0] + sum(s, n-1)
}
```

```
lemma sum_plus(s: seq<int>, i: nat)
    requires i < |s|
    ensures sum(s, i) + s[i] == sum(s, i+1)
{
    if |s| == 0 {
    } else {
```

As we brainstorm solutions for this benchmark, we can't discount the power of prompt engineering. We may observe significant traction on this benchmark simply from rephrasing how the questions are posed to the language model.

Finally, we are interested in research that demonstrates how certain reasoning abilities from models may translate between benchmarks implemented in different languages. To this end, it may be the case that models which do particularly well on the MiniF2F benchmark also do well on DafnyBench, for example. We welcome the evaluation of such models on DafnyBench. Conversely, we are curious if high-performing models on DafnyBench do better on HumanEval or even APPS.

9 Limitations

This benchmark does not assess a model's competence in translating natural language into concise formal specifications. Arguably, this conversion is a more demanding and crucial skill we seek from language models: the capacity to validate, beyond merely verifying code. The pivotal question is whether a model can assist in identifying the essential properties an algorithm must fulfill. Currently, evaluating this ability presents significant challenges. The Clover project stands as a prominent example in this area, highlighting the complexity of translating natural language descriptions into formal specifications that can be effectively used for validation.

The evaluation of models' capability to generate formal specifications might be enhanced by integrating the process with the creation of positive and negative test cases for each Dafny implementation. This approach proposes a reward system where models are evaluated based on the number of positive test cases their formal specifications support and the number of negative test cases they successfully reject. However, this method introduces a new challenge: ensuring the test cases accurately reflect the comprehensive meaning intended in the natural language descriptions. The consistency and validity of these test cases become critical, raising questions about the methods used to generate and verify them.

Data contamination emerges as another significant limitation. The issue of scraping data from platforms like GitHub introduces risks of incorporating inaccurate or misleading information into the training data. A potential mitigation strategy could involve the generation of synthetic data, designed to provide a more controlled and relevant set of examples for model training. This approach,

however, would need careful implementation to ensure the synthetic data adequately represents the complexity and variety of real-world scenarios.

While the benchmark encompasses increasingly complex multi-step programs, the tasks are designed to fit on a single page, avoiding the intricacies associated with distributed files or the integration of external libraries.

A subset of the tasks within this benchmark suffers from under-specification, lacking comprehensive formal specifications. These instances primarily test the model's ability to generate self-consistent formal specifications, proofs and code, which may not necessarily align with the broader, more specific requirements of real-world applications.

For example, we have this *fill_body* task in the benchmark:

```
method GCD2(a: int, b: int) returns (r: int)
requires a >= 0 && b >= 0
decreases b
//ensures gcd(a,b)
{/* TODO */ }
// r == gcd(a,b)
```

If the model implements an algorithm that simply decrements b but doesn't find a greatest common divisor between a and b, then this implementation will pass the verifier. [5].

10 Future Extensions to this Benchmark

This benchmark can notably be expanded in several key ways.

First, manual community contributions. Indeed, this thesis aims not to be a standalone project but rather a launching point for further momentum in the community. For all those souls who enjoy translating random Python problems into Dafny code, may this repository now serve as a receptacle and celebration point for their efforts. Perhaps it's time to spin out a variant on the Advent of Code? (The Advent of Translating Python to Dafny?) Naturally, however, the rate of contributions of this form will approach a low limit point (motivation to manually write Dafny code seems low to begin with, based on existing GitHub data). Regular scrapes of GitHub could reduce the relative amount of data contamination in the benchmark—less models are likely to be pretrained on more recent Dafny code. As a rough proxy for the amount of data we could expect to add to this benchmark in coming years, consider the growth of Dafny language files on GitHub over the past three months. This suggests an average rate of approximately 1,630 Dafny files per month. If we assume about 1/15th of these are unique and compilable as stand-alone files, then we would expect to grow our benchmark by approximately 0.11% each month.

Second, translation from Python code to Dafny code could ameliorate data leakage and reduce the percentage of trivial, under-specified examples. However,

[5] It's worth noting though that GPT-4 still solves this problem in the way we would expect: `https://chat.openai.com/g/g-JAUZ1i49Q-dafny-assistant`

careful filtering is necessary to avoid overly simplistic specifications that do not contribute meaningful challenges to the benchmark.

Third, we can explore the use of fuzzing systems to generate more verifiable Dafny programs, although this approach might diverge from a "human distribution" of Dafny programs, which is more reflective of real-world use cases.

Fourth, leveraging outputs from models that perform well on this initial benchmark could help in generating more ground-truth programs. This method allows for bootstrapping out of a data chicken-and-egg problem. Importantly, we encourage researchers to submit failed attempts on this benchmark as "incorrect ground-truth" for each Dafny program, enriching the dataset with valuable negative examples.

Further innovation could involve providing access to underlying SMT solver states, echoing the work of LeanDojo, to enhance the understanding and analysis capabilities for researchers.

Finally, making the benchmark friendlier for curriculum learning by further segmenting the difficulty levels of tasks and programs could facilitate more nuanced and progressive training approaches, particularly after several models submit results on the benchmark.

11 Use Cases and Impacts

LLMs fluent in verification-aware programming languages could make writing Dafny code easier, and therefore more widespread. This could improve software quality generally in open-source projects and in industry, especially for security-critical applications.

Formally verified languages could play a role in the interpretability of neural networks, as explored by Professor Max Tegmark. Professor Tegmark theorizes it may be possible to represent large parts of transformer models as Python (or Dafny) code. It would be useful and exciting to prove properties about the algorithms these models use for processing their inputs [38].

Formally verifiable code may improve AI safety control schemes proposed by Buck Schlegeris et al at Redwood Research, who worry that we may need meta systems for throwing away backdoored code from untrustworthy models. Verification-aware languages could be integrated into pipelines for controlling the outputs of intentionally subversive models. [24]. Generally, asking models to write verification-aware code can shift the AI safety field's perspective from thinking about unsafe models to thinking about specfic unsafe model outputs. It could still be possible to harness the benefits of potentially untrustworthy but powerful language models while still guaranteeing a certain level of trustworthiness for their outputs.

12 Conclusion

Based on the related works section, many techniques that have shown progress in the mathematical theorem proving domain should be transferable to designing algorithms that perform well on this benchmark. The main contribution of this thesis, then, is refocusing the field of neuro-symbolic systems on a use-case that may have particularly important ramifications for safety: verification-aware programming in an industry-friendly language.

This endeavor marks a pivotal shift, emphasizing the importance of safety and reliability in LLM-guided software development—a concern growing in parallel with the complexity and ubiquity of LLM generated software in daily and critical infrastructures. By marrying the strengths of neuro-symbolic approaches with the rigors of formal verification, this work paves the way for more resilient, error-resistant software systems capable of meeting the high stakes of contemporary technological demands.

As we stand on the cusp of this research frontier, it is imperative to acknowledge that while this thesis delineates a clear direction, the journey towards fully realizing the potential of neuro-symbolic systems in enhancing software safety and reliability is ongoing. Future efforts should aim to refine these systems, ensuring they are accessible, efficient, and seamlessly integrable into the software development lifecycle. Additionally, explorations into how these advancements can be harmonized with current industry practices will be crucial for their adoption and impact.

13 Appendix

13.1 A: The Minhash Deduplication Algorithm

We can think about deduplicating a set of files by finding groups of "similar" files and then choosing only one file representative from each group to form our final deduplicated set of files. To do this, we can use the Jaccard similarity metric to decide whether one document is a duplicate of another.

The Jaccard similarity metric provides a way to quantify the similarity of two sets. It is defined as [65]:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In the application to code files, we could consider each file to be a set of n-grams, where an n-gram is defined as a sequence of n adjacent symbols in a particular order [67], and then apply the Jaccard score as a similarity metric for our files. To directly calculate this Jaccard score, we would need to run string comparison on every n-gram, which would have time complexity $O(n * m^2)$ if we have n n-grams each with max length m characters. This turns out to be an inefficient method for representing each code file as a set. Instead, the minhash deduplication algorithm approximates the Jaccard simiarity between two documents by shingling the documents and comparing the minhash representation of each set of shingles (i.e. we compare fingerprints of documents instead of full documents). The minhash representation of a document is a way to represent a text document as a set of numbers that is faithful to the structure of its content but with a fixed set size that is smaller than the total number of n-grams in the document (i.e. the minhash representation of the document is a form of numerical fingerprint of the document). Below we provide the pseudocode for the minhash algorithm used, based entirely on the script here [39]:

```
function minhash_deduplication(documents, num_permutations,
    threshold):
    # Preprocess the documents
    for each document in documents:
        tokenize the document into n-grams (shingles)
        hash each n-gram using a hash function (e.g., xxHash or SHA
            -1)
        store the hashed n-grams in a set

    # Generate permutations
    for i from 1 to num_permutations:
        generate random coefficients a and b
        create a permutation function: (a * x + b) % prime_modulus

    # Create minhash signatures
    signatures = []
    for each document in documents:
        signature = []
        for each permutation function:
            min_hash = INFINITY
            for each hashed n-gram in the document:
```

```
                permuted_hash = apply permutation function to
                    hashed n-gram
                min_hash = min(min_hash, permuted_hash)
            append min_hash to signature
        append signature to signatures

    # Perform Locality-Sensitive Hashing (LSH)
    # We use 250 permutations, so to achieve Jaccard similarity
        threshold of 0.5
    # We really only need one band (i.e. one hash table)
    num_bands = choose number of bands
    rows_per_band = num_permutations / num_bands
    candidate_pairs = []
    for each band:
        create an empty hash table
        for each document signature:
            band_signature = subset of signature for the current
                band
            hash_bucket = hash(band_signature)
            add document to the corresponding hash bucket
        for each hash bucket:
            if number of documents in the bucket > 1:
                generate all pairs of documents in the bucket
                add pairs to candidate_pairs

    # Use a union-find datastructure to track groups of duplicates
    duplicates = UnionFind()
    for each band:
        for each row in hashtable:
            for each hash_bucket:
                if size(hash_bucket) <= 1:
                    continue
                else:
                    cluster_id = min(hash_bucket)
                    for x in hash_bucket:
                        duplicates.union(x, cluster_id)

    # Perform deduplication
    deduplicated_documents = []
    for each document in documents:
        if duplicates.find_root(document) = document:
            add document to deduplicated_documents

    return deduplicated_documents
```

We note that the probability two files have the same min hash value under the same hash function is equivalent to their Jaccard similarity. Concretely, for file A and file B:

$$P(minh_i(A) = minh_i(B)) = J(A, B)$$

Here, $minh_i()$ denotes taking the minimum hash value under hash function h_i. This makes sense because, assuming negligible hash collision, $P(minh_i(A) = minh_i(B))$ is equivalent to the probability that the first n-gram hash of A under h_i is equal to the first n-gram hash of B under h_i. If h_i is a good hash function, then it uniformly distributes the hash values of the original n-gram hashes over

the range of h_i. Let's call the number of n-grams with equivalent hashes c; let a represent the number of n-grams from A with smaller hash values than the hash value of corresponding n-gram from B; let b represent the reverse of the previous category. Then $P(minh_i(A) = minh_i(B)) = \frac{c}{a+b+c}$, given the uniformity of h_1. We note that $\frac{c}{a+b+c} = \frac{|A \cap B|}{|A \cup B|} = J(A, B)$.

13.2 B: GPT-4 spot-check on Benchmark Tasks

GPT-4 with Reflection solving $DoAlgebra_fillbody.dfy : https : //chat.openai.com/share/9a250315-1ef2-460b-9fd2-a941e7e64141$

GPT-4 with reflection solving $DoAlgebra_fillanno.dfy : https : //chat.openai.com/g/g-JAUZ1i49Q-dafny-assistant/c/34137160-d8d9-41c5-a7cc-c8714215ac5c$

13.3 C: Samples of HumanEvalX Problems Translated to Dafny

13.3.1 HumanEvalX 4

Python:

```
def mean_absolute_deviation(numbers: List[float]) -> float:
    mean = sum(numbers) / len(numbers) return sum(abs(x - mean) for
         x in numbers) / len(numbers)
```

Dafny:

```
/*
HumanEvalX 4
For a given list of input numbers, calculate Mean Absolute
    Deviation around the mean of this dataset.
Mean Absolute Deviation is the average absolute difference between
    each element and a centerpoint (mean in this case):
MAD = average | x - x_mean |
*/

function abs(x: real): real
{
  if x < 0.0 then -x else x
}

function sum(numbers: seq<real>): real
{
    if |numbers| == 0 then
        0.0
    else
        numbers[0] + sum(numbers[1..])
}

function mean(numbers: seq<real>): real
    requires numbers != []
{
    sum(numbers) / (|numbers| as real)
}
```

```
26
27 function mapf<A,B>(f: A -> B, xs: seq<A>): seq<B>
28     ensures |xs| == |mapf(f, xs)|
29     ensures forall i :: 0 <= i < |xs| ==> (mapf(f, xs))[i] == f(xs[
           i])
30 {
31     if |xs| == 0 then
32         []
33     else
34         [f(xs[0])] + mapf(f, xs[1..])
35 }
36
37
38 function mean_absolute_deviation(numbers: seq<real>):real
39     requires numbers != []
40 {
41     if |numbers| > 1 then
42         mean(mapf<real,real>(x => abs(x - mean(numbers)), numbers))
43     else
44         0.0
45 }
46
47 lemma sum_nonnegative(numbers: seq<real>)
48     requires forall i :: 0 <= i < |numbers| ==> numbers[i] >= 0.0
49     ensures sum(numbers) >= 0.0
50 {
51 }
52
53 lemma mean_nonnegative(numbers: seq<real>)
54     requires numbers != []
55     requires forall i :: 0 <= i < |numbers| ==> numbers[i] >= 0.0
56     ensures mean(numbers) >= 0.0
57 {
58     sum_nonnegative(numbers);
59 }
60
61 lemma MeanAbsoluteDeviationNonnegative(numbers: seq<real>)
62     requires numbers != []
63     ensures mean_absolute_deviation(numbers) >= 0.0
64 {
65    mean_nonnegative(mapf<real,real>(x => abs(x - mean(numbers)),
          numbers));
66 }
```

13.4 D: Source Repositories for Benchmark Code

Here is the first set in the list of repositories to avoid training on. More can be found in the benchmark repository.

microsoft/Ironclad
Consensys/evm-dafny
Consensys/eth2.0-dafny
dafny-lang/libraries
lemmy/lets-prove-blocking-queue

vmware-labs/verified-betrfs
aws/aws-encryption-sdk-dafny
mit-pdos/daisy-nfsd
Consensys/deposit-sc-dafny
secure-foundations/veri-titan
namin/dafny-sandbox
Mondego/dafny-synthesis
franck44/evm-dis
wilcoxjay/notes
ChuyueSun/Clover
Consensys/distributed-validator-formal-specs-and-verification
secure-foundations/veribetrkv-osdi2020
dafny-lang/Dafny-VMC
Consensys/dafny-sc-fmics
tchajed/protocol-verification-fa2023
GLaDOS-Michigan/verification-class
Consensys/qbft-formal-spec-and-verification
Consensys/DafnyCrypto
Consensys/dafny-training
dafny-lang/compiler-bootstrap
tchajed/dafny-syntax-tutorial
vladstejeroiu/Dafny-programs
dijkstracula/smrt
monadius/dafny_examples
dafny-lang/dafny-reportgenerator
secure-foundations/veri-datalog
angelhof/penn-cis673-hw-dafny
boogie-org/BoogieV
franck44/yul-dafny
seanpm2001/SNU_2D_ProgrammingTools_IDE_Dafny
jonhnet/vmware-verification-2023
HristoStaykov/sbft-formal-verification-model
priyasiddharth/seaurchin
zhuzilin/dafny-exercises
Eggy115/Dafny
StephRMcIntyre/cs245-verification
Nathan-Fenner/bbchallenge-dafny-deciders
joaopascoalfariafeup/DafnyProjects
Nangos/verified-algorithms
secure-foundations/ironsync-osdi2023
seanpm2001/Learn-Dafny
cwatson1998/cis-6730-hw
shirimo/thesis_files
Lystora/M2
kumom/dafny-fatnode
Consensys/WrappedEther.dfy

nathand99/SENG2011
unboxedtype/EVF
paraverifierdafny/paraverif_dafny
juletx/MFDS
ArthurSudbrackIbarra/Trab1-Metodos-Formais
andricicezar/sat-solver-dafny
DavePearce/BettingContract

13.5 E: Examples of Hacks and Failure Modes our Benchmark Evaluation will Catch

13.5.1 Modification of Formal Specification

```
/*
HumanEvalX 158
Write a function that accepts a list of strings. The list contains different words.
Return the word with maximum number of unique characters. If multiple strings have maximum number of unique
characters, return the one which comes first in lexicographical order.
*/

function numUniqueChars(s: string): nat {
    |set c | c in s|
}

method FindMaxWord(words: seq<string>) returns (maxWord: string)
    ensures maxWord in words || |words| == 0
{
    if |words| == 0 {
        maxWord := "";
        return;
    }
    var word := words[0];
    maxWord := words[0];
    var maxUniqueChars := |set c | c in maxWord|;
    assert maxUniqueChars == numUniqueChars(maxWord);
    for i := 0 to |words|
        invariant maxWord in words
        invariant maxUniqueChars == numUniqueChars(maxWord)
        invariant numUniqueChars(word) <= numUniqueChars(maxWord)
        invariant (forall w :: 0 <= w < i ==> numUniqueChars(words[w]) <= numUniqueChars(maxWord))
    {
        word := words[i];
        var uniqueChars := set c | c in word;
        var count := |uniqueChars|;
        assert count == numUniqueChars(word);
        if count > maxUniqueChars || (count == maxUniqueChars && word < maxWord) {
            maxUniqueChars := count;
            maxWord := word;
            assert maxWord == word;
```

```
        }
        assert maxUniqueChars >= count;
        assert maxUniqueChars == numUniqueChars(maxWord);
    }
    assert forall w :: w in words ==> numUniqueChars(w) <=
        numUniqueChars(maxWord);
}
```

13.5.2 Cheat Via Assume False

```
/*
HumanEvalX 158
Write a function that accepts a list of strings. The list contains
    different words.
Return the word with maximum number of unique characters. If
    multiple strings have maximum number of unique
characters, return the one which comes first in lexicographical
    order.
*/

function numUniqueChars(s: string): nat {
    |set c | c in s|
}

method FindMaxWord(words: seq<string>) returns (maxWord: string)
    ensures maxWord in words || |words| == 0
    ensures forall w :: w in words ==> numUniqueChars(w) <=
        numUniqueChars(maxWord)
{
    assume false;
    if |words| == 0 {
        maxWord := "";
        return;
    }
    var word := words[0];
    maxWord := words[0];
    var maxUniqueChars := |set c | c in maxWord|;
    assert maxUniqueChars == numUniqueChars(maxWord);
    for i := 0 to |words|
        invariant maxWord in words
        invariant maxUniqueChars == numUniqueChars(maxWord)
        invariant numUniqueChars(word) <= numUniqueChars(maxWord)
        invariant (forall w :: 0 <= w < i ==> numUniqueChars(words[
            w]) <= numUniqueChars(maxWord))
    {
        word := words[i];
        var uniqueChars := set c | c in word;
        var count := |uniqueChars|;
        assert count == numUniqueChars(word);
        if count > maxUniqueChars || (count == maxUniqueChars &&
            word < maxWord) {
            maxUniqueChars := count;
            maxWord := word;
            assert maxWord == word;
        }
        assert maxUniqueChars >= count;
```

```
        assert maxUniqueChars == numUniqueChars(maxWord);
    }
    assert forall w :: w in words ==> numUniqueChars(w) <=
        numUniqueChars(maxWord);
}
```

13.5.3 Cheat Via :Verify False

```
/*
HumanEvalX 158
Write a function that accepts a list of strings. The list contains
    different words.
Return the word with maximum number of unique characters. If
    multiple strings have maximum number of unique
characters, return the one which comes first in lexicographical
    order.
*/

function numUniqueChars(s: string): nat {
    |set c | c in s|
}

method {:verify false} FindMaxWord(words: seq<string>) returns (
    maxWord: string)
    ensures maxWord in words || |words| == 0
    ensures forall w :: w in words ==> numUniqueChars(w) <=
        numUniqueChars(maxWord)
{
    if |words| == 0 {
        maxWord := "";
        return;
    }
    var word := words[0];
    maxWord := words[0];
    var maxUniqueChars := |set c | c in maxWord|;
    assert maxUniqueChars == numUniqueChars(maxWord);
    for i := 0 to |words|
        invariant maxWord in words
        invariant maxUniqueChars == numUniqueChars(maxWord)
        invariant numUniqueChars(word) <= numUniqueChars(maxWord)
        invariant (forall w :: 0 <= w < i ==> numUniqueChars(words[
            w]) <= numUniqueChars(maxWord))
    {
        word := words[i];
        var uniqueChars := set c | c in word;
        var count := |uniqueChars|;
        assert count == numUniqueChars(word);
        if count > maxUniqueChars || (count == maxUniqueChars &&
            word < maxWord) {
            maxUniqueChars := count;
            maxWord := word;
            assert maxWord == word;
        }
        assert maxUniqueChars >= count;
        assert maxUniqueChars == numUniqueChars(maxWord);
    }
```

```
    assert forall w :: w in words ==> numUniqueChars(w) <=
        numUniqueChars(maxWord);
}
```

13.6 F: Mathematics Behind Training on Crossentropy

The Kullback-Leibler (KL) divergence between two probability distributions provides a way to measure the distance between those two probability distributions. We can write the expression for KL divergence between discrete distributions P and Q explicitly as the following [66].

$$KL(P||Q) = -\sum_{x \in X} P(x)log(Q(x)) - P(x)log(P(x))$$

This can intuitively be described as the least number of bits required to describe information from distribution P if you believe the distribution is Q minus the number of bits to describe information from the distribution if you know the actual distribution P. Clearly, KL divergence is always non-negative because the only way to achieve the most compressed encoding of information from a distribution is by knowing the exact underlying distribution P. In other words, $-P(x)log(Q(x)) \geq -P(x)log(P(x))$.

We can minimize the KL distance between two distributions by minimizing $\sum_{x \in X} -P(x)log(Q(x))$ which is called the cross entropy term. Therefore, training a model to minimize crossentropy is equivalent to training the model to match the underlying distribution of its input data.

13.7 G: Direct Preference Optimization

13.7.1 Theory of Direct Preference Optimization

Direct Preference Optimization aims to provide a closed-form method for performing reinforcement learning from human feedback (RLHF) on a Large Language Model. This allows us to nudge model weights in roughly the same direction RLHF would, but under a computationally faster scheme. In RLHF, a foundation model is usually first fine-tuned on a downstream task of interest (for example, code generation, using data scraped from GitHub). Then the model is prompted with prompt x_i to generate pairs of outputs y_{1i}, y_{2i} for each i. If we let π^{SFT} denote our fine-tuned foundation model, then we can write $(y_{1i}, y_{2i}) \sim \pi^{SFT}(y_i|x_i)$. We then ask a human evaluator to rank their preferences for y_{1i} versus y_{2i}. If the human ranks $y_{1i} > y_{2i}$ then we say y_{1i} is the preferred model output. We assume these preferences are generated by a latent reward model $r^*(y_i, x_i)$, which we aim to learn. Given r^*, the Bradley-Terry model says that we can model human preferences according to

$$P(y_{1i} > y_{2i}|x_i) = \frac{exp(r^*(y_{1i}, x_i))}{exp(r^*(y_{1i}, x_i)) + exp(r^*(y_{2i}, x_i))}$$

So knowing the reward function can help us decide which model output will be preferred by humans. If we have access to a dataset of $\{x_i, y_{1i}, y_{2i}\}$ pairs, then we can parameterize an estimate of r^* as r_σ and learn r_σ through maximum likelihood estimation on the binary cross entropy problem of choosing the preferred output over the rejected output. Then, we use the learned reward function to update the weights of π^{SFT}. First we let $\pi^{REF} = \pi_\theta = \pi^{SFT}$ and then we maximize $max_{\pi_\theta} E_{x_i \sim D, y_i \sim \pi_\theta(y|x_i)}[r^*_\sigma(y_i, x_i)] - \beta D_{KL}[\pi_\theta(y|x_i)||\pi^{REF}(y|x_i)]$.

Several papers have shown that the optimal solution to the previous KL-constrained reward maximization problem is:

$$\pi_*(y|x_i) = \frac{1}{Z(x_i)} \pi_{REF} exp(\frac{1}{\beta} r_\sigma(y, x_i))$$

Here $Z(x_i) = \sum_y \pi_{REF} exp(\frac{1}{\beta} r_\sigma(y, x_i))$. This is a computationally expensive function to estimate. Instead, we can flip the problem on its head and solve for r_σ in terms of the policy functions. Doing this, we get $r_\sigma(y, x_i) = \beta log(\frac{\pi_\theta(y,x_i)}{\pi_{REF}(y,x_i)}) + \beta log(Z(x))$. We can recognize that increasing the probability $P(y_{1i} > y_{2i})$ under the Bradley-Terry model is equivalent to increasing $\frac{1}{1+exp(\beta \frac{\pi_\theta(y_{1i},x_i)}{\pi_{REF}(y_{1i},x_i)} - \beta \frac{\pi_\theta(y_{2i},x_i)}{\pi_{REF}(y_{2i},x_i)})}$. Intuitively, this makes sense because the latter quantity is the logistic function applied to the difference in reward values for y_{1i} versus y_{2i}, and as mentioned previously we can frame the preference ranking problem as a binary cross entropy problem.

The resulting loss function for DPO optimization then becomes the negative log of the previous expression, which we can optimize if given access to the original version of the foundation model we want to further optimize. By decomposing the terms of the gradient of this quantity, it becomes clear that DPO updates the parameters of the underlying foundation model so as to decrease the likelihood of lower-preference outputs and increase the likelihood of higher-preference outputs while increasing the weight of this update if the reward estimate is wrong.

All of this explanation has been paraphrased from the DPO paper [48].

13.7.2 Dataset of Positive/Negative Triples Used for Fine Tuning Phind-Code-Llama

Here is an example triple from the phind-code-llama fine tuning dataset:

```
{"prompt":
"method Abs(x: int) returns (y: int)
    ensures x>=0 ==> x==y
    ensures x<0 ==> x+y==0
{",

"chosen":
"method Abs(x: int) returns (y: int)
    ensures x>=0 ==> x==y
```

```
    ensures x<0 ==> x+y==0{
        if x >= 0 {
            y := x;
        } else {
            y := -x;
        }
    }‘‘‘",

"rejected":
"method Abs(x: int) returns (y: int)
    ensures x>=0 ==> x==y
    ensures x<0 ==> x+y==0
{
    // complete this method
}"}
```

13.8 H: Theory of SMT Solvers

"SMT" solvers are solvers for Satisfiability Modulo Theories, meaning that they solve satisfiability problems using knowledge about the theory in which they originate. [17, 42] In contrast, a (short for Satisfiability) solver does not know any theory and attempts to solve satisfiablity problems using only knowledge about the logical connectives $\wedge$, $\vee$, and $\neg$. [19, 20, 42] SMT solvers can often arrive at satisfying assignments faster that SAT solvers because they can limit the search space by relying on their specialized pre-existing knowledge about particular theories that rule out sections of the search space that are known to be incorrect. For example, many SMT solvers perform well solving linear inequalities since they come with specialized search routines that know about the laws of the theory of linear arithmetic: $\langle +, -, <, \leq, = \rangle$. For more complicated theories, SMT solvers will use a combination of heuristics tuned to each theory. [19, 18]

Some recent advancements in SMT solvers include the model-based quantifier instantiation technique of Z3, a state-of-the-art SMT solver. [17] Model-based quantifier instantiation allows Z3 to prove statements that go beyond propositional or zeroth-order logic to proving some first-order statements, which include the quantifiers $\forall$ and $\exists$ over elements in the domain of discourse. [50]

Due to their reliance on theory-specific heuristics for performance, and the recent trend towards supporting first-order logic, SMT solvers could stand to benefit greatly from advances in AI/ML systems, which are adept and making generalizations from a known corpus. [58] In particular, $\forall$-instantiation could be effectively guided using a neural-network-learned generalization from a subset of true sentences from the active theory as a heuristic, with $\exists$-instantiation being a corollary as the natural dual.

13.8.1 Simple Examples for Intuition

Problem: Is there a combination of truth values for variables A, B, and C that satisfies the following Boolean formula?

$$(A \vee \neg B) \wedge (\neg A \vee C)$$

This is a SAT problem because it involves purely Boolean variables and operations. [20] The solver needs to determine if there exists an assignment of TRUE or FALSE to A, B, and C such that the entire formula evaluates to TRUE. For example, assigning $A = TRUE$, $B = FALSE$, and $C = TRUE$ would satisfy the formula, making it satisfiable. [36]

Problem: Is there an assignment of integers to x and y such that the following conditions are satisfied?

1. $x + y > 5$
2. $x < 3$
3. y is even

Explanation: This is an SMT problem because it involves variables (x and y) that are not just Boolean but integer-valued, and it combines these with arithmetic operations and relations. An SMT solver would need to determine if there exist integer values for x and y that meet all the given conditions. In this case, $x = 2$ and $y = 4$ would be a solution, as it satisfies all the conditions: $2 + 4 > 5$, $2 < 3$, and 4 is even.

References

[1] Verification - limitations. https://deal.readthedocs.io/basic/verification.html#limitations. Accessed: 2024-03-30.

[2] Pisa: Isabelle proofs dataset. https://aitp-conference.org/2021/abstract/paper_17.pdf, 2021.

[3] Big code models leaderboard. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard, 2022.

[4] Code llama: Fine-tuning llama for code generation. https://huggingface.co/Phind/Phind-CodeLlama-34B-v2, 2022.

[5] Curry–howard correspondence — wikipedia, the free encyclopedia, 2024. [Online; accessed 28-March-2024].

[6] The pile (dataset). https://en.wikipedia.org/wiki/The_Pile_(dataset), 2024. Accessed: 2024-03-28.

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[9] BIG bench authors. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023.

[10] Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 3–17, Cham, 2015. Springer International Publishing.

[11] AWS Security Blog. Aws security profile: Rustan leino, senior principal applied scientist. https://aws.amazon.com/blogs/security/aws-security-profile-rustan-leino-senior-principal-applied-scientist/, July 2019. Accessed: 2024-03-30.

[12] David Brandfonbrener, Sibi Raja, Tarun Prasad, Chloe Loughridge, Federico Cassano, Jianang Yang, Simon Henniger, William E. Byrd, Robert Zinkov, and Nada Amin. Verified multi-step synthesis using large language models and monte carlo tree search, 2023.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[14] 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. Evaluating large language models trained on code, 2021.

[15] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2023.

[16] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for dpll (t). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006.

[19] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[20] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[21] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[22] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning, 2020.

[23] Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code, 2024.

[24] Ryan Greenblatt, Buck Shlegeris, Kshitij Sachan, and Fabien Roger. Ai control: Improving safety despite intentional subversion, 2023.

[25] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[26] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021.

[27] Abhinav Jain, Chima Adiole, Swarat Chaudhuri, Thomas Reps, and Chris Jermaine. Coarse-tuning models of code with reinforcement learning feedback, 2023.

[28] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

[29] Albert Q. Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers, 2022.

[30] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, Karthik Narasimhan, et al. Swe-bench: Can language models resolve real-world github issues?, 2023.

[31] Rui Abreu João Pascoal Faria. Case studies of development of verified programs with dafny for accessibility assessment. *arXiv preprint arXiv:2301.03224*, 2023.

[32] Nora Kassner, Oyvind Tafjord, Ashish Sabharwal, Kyle Richardson, Hinrich Schuetze, and Peter Clark. Language models with rationality, 2023.

[33] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[34] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023.

[35] Jesus Lovon-Melgarejo, Jose G. Moreno, Romaric Besançon, Olivier Ferret, and Lynda Tamine. Can we guide a multi-hop reasoning language model to incrementally learn at each single-hop? In Nicoletta Calzolari, Chu-Ren Huang, Hansaem Kim, James Pustejovsky, Leo Wanner, Key-Sun Choi, Pum-Mo Ryu, Hsin-Hsi Chen, Lucia Donatelli, Heng Ji, Sadao Kurohashi, Patrizia Paggio, Nianwen Xue, Seokhwan Kim, Younggyun Hahm, Zhong He, Tony Kyungil Lee, Enrico Santus, Francis Bond, and Seung-Hoon Na, editors, *Proceedings of the 29th International Conference on Computational Linguistics*, pages 1455–1466, Gyeongju, Republic of Korea, October 2022. International Committee on Computational Linguistics.

[36] Joao Marques-Silva. Practical applications of boolean satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE, 2008.

[37] Iris Ma James Noble Md Rakib Hossain Misu, Cristina V. Lopes. Towards ai-assisted synthesis of verified dafny methods. *Proc. ACM Softw. Eng.*, 1(FSE), 2024.

[38] Eric J. Michaud, Isaac Liao, Vedang Lad, Ziming Liu, Anish Mudide, Chloe Loughridge, Zifan Carl Guo, Tara Rezaei Kheirkhah, Mateja Vukelić, and Max Tegmark. Opening the ai black box: program synthesis via mechanistic interpretability, 2024.

[39] Chenghao Mou, Chris Ha, Kenneth Enevoldsen, and Peiyuan Liu. Chenghaomou/text-dedup: Reference snapshot, September 2023.

[40] Neel Nanda. Open problems in mechanistic interpretability: A whirlwind tour. Online Video, Mar 2023. `https://www.youtube.com/watch?v=ZSg4-H8L6Ec`.

[41] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution, 2023.

[42] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[43] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa

Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry

Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[44] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using llms to facilitate formal verification of rtl, 2023.

[45] Orient Software. Most popular programming languages. `https://www.orientsoftware.com/blog/most-popular-programming-languages`, 2023. Accessed: 2023-09-30.

[46] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[47] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[48] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2023.

[49] Sharad Ramanathan. Lecture 13 notes in applied physics 286. Lecture notes distributed in Applied Physics 286 at Harvard University, Fall 2023. Accessed: 2024-03-27.

[50] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in smt. In *Automated Deduction–CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings 24*, pages 377–391. Springer, 2013.

[51] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[52] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[53] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W. Kocurek, Ali Safaya, Ali Tazarv, Alice Xiang, Alicia Parrish, Allen Nie, Aman Hussain, Amanda Askell, Amanda Dsouza, Ambrose Slone, Ameet Rahane, Anantharaman S. Iyer, Anders Andreassen, Andrea Madotto, Andrea Santilli, Andreas Stuhlmüller, Andrew Dai, Andrew La, Andrew Lampinen, Andy Zou, Angela Jiang, Angelica Chen, Anh Vuong, Animesh Gupta, Anna Gottardi, Antonio Norelli, Anu Venkatesh, Arash Gholamidavoodi, Arfa Tabassum, Arul Menezes, Arun Kirubarajan, Asher Mullokandov, Ashish Sabharwal, Austin Herrick, Avia Efrat, Aykut Erdem, Ayla Karakaş, B. Ryan Roberts, Bao Sheng Loe, Barret Zoph, Bartłomiej Bojanowski, Batuhan Özyurt, Behnam Hedayatnia, Behnam Neyshabur, Benjamin Inden, Benno Stein, Berk Ekmekci, Bill Yuchen Lin, Blake Howald, Bryan Orinion, Cameron Diao, Cameron Dour, Catherine Stinson, Cedrick Argueta, César Ferri Ramírez, Chandan Singh, Charles Rathkopf, Chenlin Meng, Chitta Baral, Chiyu Wu, Chris Callison-Burch, Chris Waites, Christian Voigt, Christopher D. Manning, Christopher Potts, Cindy Ramirez, Clara E. Rivera, Clemencia Siro, Colin Raffel, Courtney Ashcraft, Cristina Garbacea, Damien Sileo, Dan Garrette, Dan Hendrycks, Dan Kilman, Dan Roth, Daniel Freeman, Daniel Khashabi, Daniel Levy, Daniel Moseguí González, Danielle Perszyk, Danny Hernandez, Danqi Chen, Daphne Ippolito, Dar Gilboa, David Dohan, David Drakard, David Jurgens, Debajyoti Datta, Deep Ganguli, Denis Emelin, Denis Kleyko, Deniz Yuret, Derek Chen, Derek Tam, Dieuwke Hupkes, Diganta Misra, Dilyar Buzan, Dimitri Coelho Mollo, Diyi Yang, Dong-Ho Lee, Dylan Schrader, Ekaterina Shutova, Ekin Dogus Cubuk, Elad Segal, Eleanor Hagerman, Elizabeth Barnes, Elizabeth Donoway, Ellie Pavlick, Emanuele Rodola, Emma Lam, Eric Chu, Eric Tang, Erkut Erdem, Ernie Chang, Ethan A. Chi, Ethan Dyer, Ethan Jerzak, Ethan Kim, Eunice Engefu Manyasi, Evgenii Zheltonozhskii, Fanyue Xia, Fatemeh Siar, Fernando Martínez-Plumed, Francesca Happé, Francois Chollet, Frieda Rong, Gaurav Mishra, Genta Indra Winata, Gerard de Melo, Germán Kruszewski, Giambattista Parascandolo, Giorgio Mariani, Gloria Wang, Gonzalo Jaimovitch-López, Gregor Betz, Guy Gur-Ari, Hana Galijasevic, Hannah Kim, Hannah Rashkin, Hannaneh Hajishirzi, Harsh Mehta, Hayden Bogar, Henry Shevlin, Hinrich Schütze, Hiromu Yakura, Hongming Zhang, Hugh Mee Wong, Ian Ng, Isaac Noble, Jaap Jumelet, Jack Geissinger, Jackson Kernion, Jacob Hilton, Jaehoon Lee, Jaime Fernández Fisac, James B. Simon, James Koppel, James Zheng, James Zou, Jan Kocoń, Jana Thompson, Janelle Wingfield, Jared Kaplan, Jarema Radom, Jascha Sohl-Dickstein, Jason Phang, Jason Wei, Jason Yosinski, Jekaterina Novikova, Jelle Bosscher, Jennifer Marsh, Jeremy Kim, Jeroen Taal, Jesse Engel, Jesujoba Alabi, Jiacheng Xu, Jiaming Song, Jillian Tang, Joan Waweru, John Burden, John Miller, John U. Balis, Jonathan Batchelder, Jonathan Berant, Jörg

Frohberg, Jos Rozen, Jose Hernandez-Orallo, Joseph Boudeman, Joseph Guerr, Joseph Jones, Joshua B. Tenenbaum, Joshua S. Rule, Joyce Chua, Kamil Kanclerz, Karen Livescu, Karl Krauth, Karthik Gopalakrishnan, Katerina Ignatyeva, Katja Markert, Kaustubh D. Dhole, Kevin Gimpel, Kevin Omondi, Kory Mathewson, Kristen Chiafullo, Ksenia Shkaruta, Kumar Shridhar, Kyle McDonell, Kyle Richardson, Laria Reynolds, Leo Gao, Li Zhang, Liam Dugan, Lianhui Qin, Lidia Contreras-Ochando, Louis-Philippe Morency, Luca Moschella, Lucas Lam, Lucy Noble, Ludwig Schmidt, Luheng He, Luis Oliveros Colón, Luke Metz, Lütfi Kerem Şenel, Maarten Bosma, Maarten Sap, Maartje ter Hoeve, Maheen Farooqi, Manaal Faruqui, Mantas Mazeika, Marco Baturan, Marco Marelli, Marco Maru, Maria Jose Ramírez Quintana, Marie Tolkiehn, Mario Giulianelli, Martha Lewis, Martin Potthast, Matthew L. Leavitt, Matthias Hagen, Mátyás Schubert, Medina Orduna Baitemirova, Melody Arnaud, Melvin McElrath, Michael A. Yee, Michael Cohen, Michael Gu, Michael Ivanitskiy, Michael Starritt, Michael Strube, Michał Swedrowski, Michele Bevilacqua, Michihiro Yasunaga, Mihir Kale, Mike Cain, Mimee Xu, Mirac Suzgun, Mitch Walker, Mo Tiwari, Mohit Bansal, Moin Aminnaseri, Mor Geva, Mozhdeh Gheini, Mukund Varma T, Nanyun Peng, Nathan A. Chi, Nayeon Lee, Neta Gur-Ari Krakover, Nicholas Cameron, Nicholas Roberts, Nick Doiron, Nicole Martinez, Nikita Nangia, Niklas Deckers, Niklas Muennighoff, Nitish Shirish Keskar, Niveditha S. Iyer, Noah Constant, Noah Fiedel, Nuan Wen, Oliver Zhang, Omar Agha, Omar Elbaghdadi, Omer Levy, Owain Evans, Pablo Antonio Moreno Casares, Parth Doshi, Pascale Fung, Paul Pu Liang, Paul Vicol, Pegah Alipoormolabashi, Peiyuan Liao, Percy Liang, Peter Chang, Peter Eckersley, Phu Mon Htut, Pinyu Hwang, Piotr Miłkowski, Piyush Patil, Pouya Pezeshkpour, Priti Oli, Qiaozhu Mei, Qing Lyu, Qinlang Chen, Rabin Banjade, Rachel Etta Rudolph, Raefer Gabriel, Rahel Habacker, Ramon Risco, Raphaël Millière, Rhythm Garg, Richard Barnes, Rif A. Saurous, Riku Arakawa, Robbe Raymaekers, Robert Frank, Rohan Sikand, Roman Novak, Roman Sitelew, Ronan LeBras, Rosanne Liu, Rowan Jacobs, Rui Zhang, Ruslan Salakhutdinov, Ryan Chi, Ryan Lee, Ryan Stovall, Ryan Teehan, Rylan Yang, Sahib Singh, Saif M. Mohammad, Sajant Anand, Sam Dillavou, Sam Shleifer, Sam Wiseman, Samuel Gruetter, Samuel R. Bowman, Samuel S. Schoenholz, Sanghyun Han, Sanjeev Kwatra, Sarah A. Rous, Sarik Ghazarian, Sayan Ghosh, Sean Casey, Sebastian Bischoff, Sebastian Gehrmann, Sebastian Schuster, Sepideh Sadeghi, Shadi Hamdan, Sharon Zhou, Shashank Srivastava, Sherry Shi, Shikhar Singh, Shima Asaadi, Shixiang Shane Gu, Shubh Pachchigar, Shubham Toshniwal, Shyam Upadhyay, Shyamolima, Debnath, Siamak Shakeri, Simon Thormeyer, Simone Melzi, Siva Reddy, Sneha Priscilla Makini, Soo-Hwan Lee, Spencer Torene, Sriharsha Hatwar, Stanislas Dehaene, Stefan Divic, Stefano Ermon, Stella Biderman, Stephanie Lin, Stephen Prasad, Steven T. Piantadosi, Stuart M. Shieber, Summer Misherghi, Svetlana Kiritchenko, Swaroop Mishra, Tal Linzen, Tal Schuster, Tao Li, Tao Yu, Tariq Ali, Tatsu Hashimoto, Te-Lin Wu, Théo Desbordes, Theodore Rothschild,

Thomas Phan, Tianle Wang, Tiberius Nkinyili, Timo Schick, Timofei Kornev, Titus Tunduny, Tobias Gerstenberg, Trenton Chang, Trishala Neeraj, Tushar Khot, Tyler Shultz, Uri Shaham, Vedant Misra, Vera Demberg, Victoria Nyamai, Vikas Raunak, Vinay Ramasesh, Vinay Uday Prabhu, Vishakh Padmakumar, Vivek Srikumar, William Fedus, William Saunders, William Zhang, Wout Vossen, Xiang Ren, Xiaoyu Tong, Xinran Zhao, Xinyi Wu, Xudong Shen, Yadollah Yaghoobzadeh, Yair Lakretz, Yangqiu Song, Yasaman Bahri, Yejin Choi, Yichi Yang, Yiding Hao, Yifu Chen, Yonatan Belinkov, Yu Hou, Yufang Hou, Yuntao Bai, Zachary Seid, Zhuoye Zhao, Zijian Wang, Zijie J. Wang, Zirui Wang, and Ziyi Wu. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, 2023.

[54] Ralf C. Staudemeyer and Eric Rothstein Morris. Understanding lstm – a tutorial into long short-term memory recurrent neural networks, 2019.

[55] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation, 2023.

[56] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *ICLR 2024 Conference*, 2024.

[57] Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625:476–482, 2024.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[59] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[60] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[61] Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. Naturalproofs: Mathematical theorem proving in natural language. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[62] Sean Welleck, Jiacheng Liu, Ximing Lu, Hannaneh Hajishirzi, and Yejin Choi. Naturalprover: Grounded mathematical proof generation with language models, 2022.

[63] Wikipedia contributors. Alan turing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Alan_Turing`, 2024. Accessed: 2024-03-27.

[64] Wikipedia contributors. Gpt-3 — wikipedia, the free encyclopedia, 2024. [Online; accessed 28-March-2024].

[65] Wikipedia contributors. Jaccard index — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Jaccard_index, 2024. Accessed: 2024-03-27.

[66] Wikipedia contributors. Kullback-leibler divergence. https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence, 2024. Accessed: 2024-03-27.

[67] Wikipedia contributors. N-gram — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/N-gram, 2024. Accessed: 2024-03-27.

[68] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. https://paperswithcode.com/paper/autoformalization-with-large-language-models, 2022.

[69] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants, 2019.

[70] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models, 2023.

[71] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[72] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions, 2023.

[73] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022.

[74] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation, 2023.

[75] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics, 2022.

[76] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.

[77] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2024.

[78] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification, 2023.

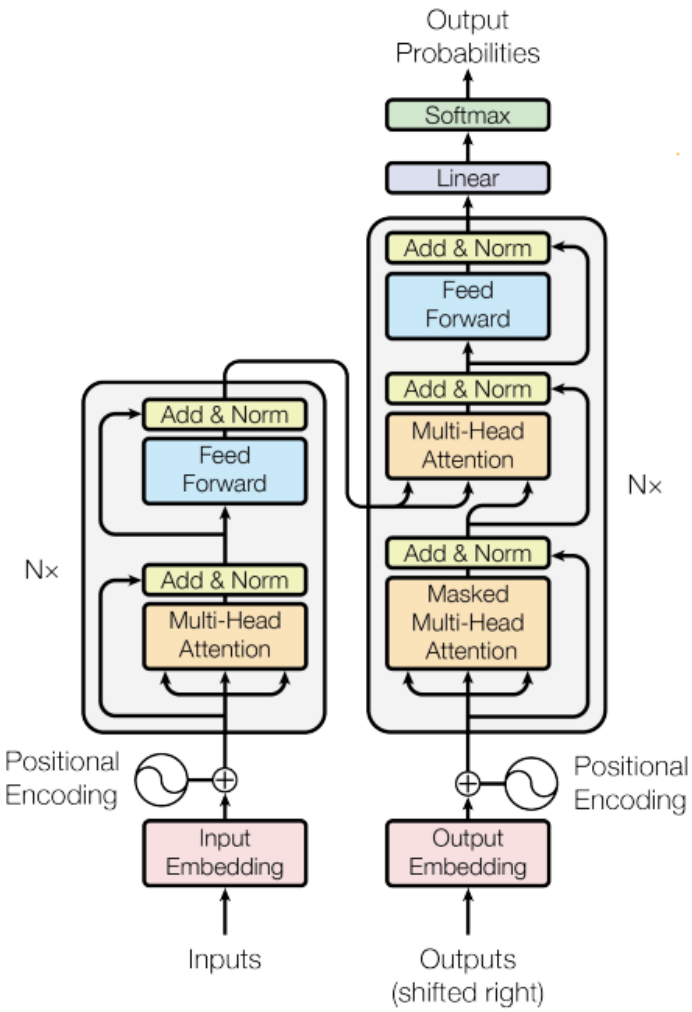


Figure 3: Basic Transformer Architecture. Source: Attention is All You Need

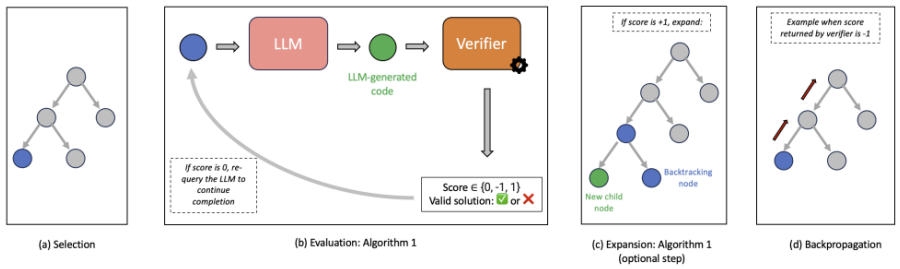


Figure 4: VMCTS algorithm. Source: Brandfonbrener et al, *Verified Multi-Step Synthesis using Large Language Models and Monte Carlo Tree Search*, 2024.

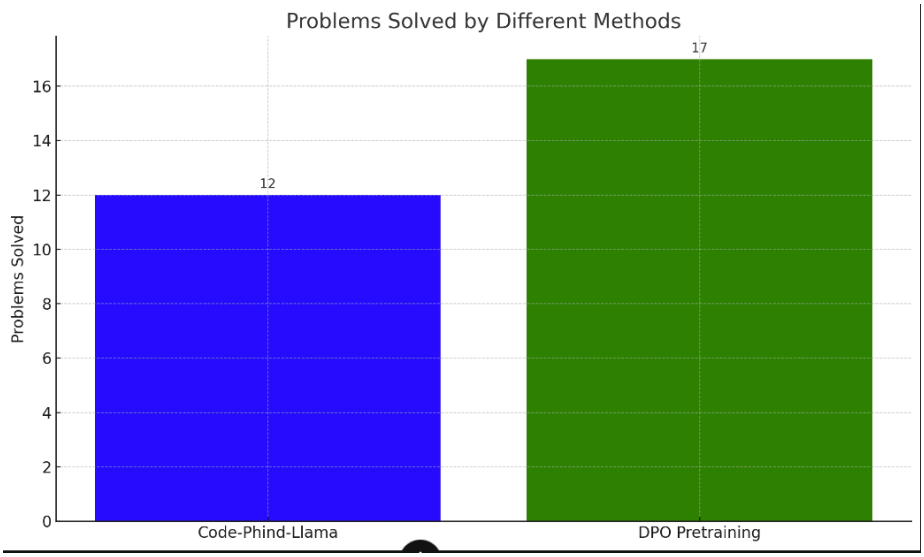


Figure 5: Simple Direct Preference Optimization Fine Tuning Results with Code Phind Llama

2024.

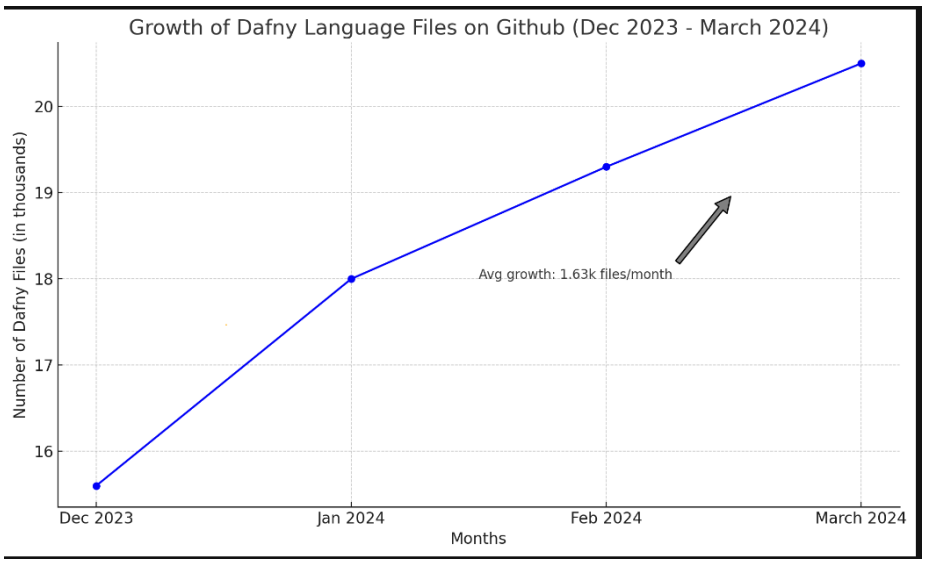


Figure 6: Growth in Dafny Files on Github over the past Months