Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Large Language Models for Verified Programs

Master's Thesis

Omkar Zade

April 11, 2024

Advisors: Prof. Dr. Peter Müller, Jingxuan He, Nicolas Klose

Department of Computer Science, ETH Zürich

Abstract

Large Language Models (LLMs) are increasingly used to generate code. However, due to their probabilistic nature, they provide no formal guarantees on the correctness or safety of the generated code. The use of automated program verification can achieve such guarantees, but requires the programmer to manually provide specifications—an expensive and difficult task. In this work, we leverage LLMs to automatically infer these specifications, taking a step forward in reducing the manual effort required for program verification and increasing trust in LLM generated programs. We design prompting and fine-tuning based solutions to infer memory safety specifications for Python programs. Moreover, we contribute a dataset of verified programs that serves as a benchmark, and evaluate our approaches on this dataset, demonstrating their effectiveness and limitations.

Acknowledgements

First, I would like to thank my supervisors, Nicolas Klose and Jingxuan He, for their invaluable guidance throughout the project. They were always encouraging and supported me with any resources I needed along the way.

I express my sincere thanks to Prof. Dr. Peter Müller for giving me the opportunity to work on this very interesting topic.

Finally, I would like to thank my mother and father for their constant support throughout my studies and believing in me.

Contents

Chapter 1

Introduction

1.1 Motivation

Large language models (LLMs) are seeing widespread adoption in generating software artifacts. While LLM powered code assistants like Copilot [2] promise increased developer productivity, they provide no guarantees of correctness or safety of the generated code.

Deductive program verification is a static analysis technique to prove that a program satisfies a formal specification—this provides much stronger guarantees than testing. Recent developments in the field have made it possible to efficiently prove complex properties of real world programs including functional correctness, memory safety and absence of data races. However, these specifications need to be provided manually, and require expertise and time on part of the programmer.

In this thesis, we leverage language modeling and code synthesis capabilities of LLMs, combined with feedback from an automatic verifier to automatically infer program specifications, thus easing the burden on the programmer. Our solution, in combination with existing code LLMs, has the potential to increase trust in generated programs.

Concretely, we target the problem of inferring specifications for Python programs. Python is widely used for scripting, prototyping, data science but also increasingly deployed in critical applications, making it a good target for verification. Code LLMs are multilingual, however have been fine-tuned and benchmarked more extensively on Python programs.

Nagini [3] is a state-of-the-art verifier for statically-typed Python programs, based on the Viper [5] verification infrastructure. It has an expressive specification language and performs modular verification, allowing verification to scale to large projects. Hence, Nagini will serve as the specification language and verification oracle.

Specifically, we focus on inferring *memory safety* specifications, as proving memory safety is a prerequisite to prove more advanced properties like functional correctness. For example: to prove that a program sorts the linked list, it must be first proved that the result is a valid linked list.

We approach the problem as follows. As there are not many existing verified programs in Nagini, we implement a dataset of 50 methods across three data structures and verify them for memory safety in Nagini. We equip this dataset with a metric to create a *benchmark* to evaluate LLMs on the verification task.

There are two primary ways to utilize LLMs for a downstream task: *prompting* and *fine-tuning*. We explore both avenues for our task.

We develop a prompting-based solution to verify an unseen method given examples of verified methods. Further, we develop an algorithm to incrementally build the set of verified programs from scratch, demonstrating real-world applicability of our solution. Our method is implemented as a generic framework that can be easily extended to different models.

Next, we fine-tune an open-source code LLM for the verification task. We shall see that the 50-method dataset is insufficient for fine-tuning, and propose a method to use it to generate a much larger training dataset.

Finally, we evaluate our approaches quantitatively and qualitatively, demonstrating their effectiveness and shortcomings, and propose directions for future work.

1.2 Contributions

The main contributions of the thesis are outlined below:

- We design a dataset of verified programs in Nagini which serves as a benchmark for evaluation and root of training dataset generation for fine-tuning.
- We develop the technique of *prompting with errors* and an algorithm to incrementally build the set of verified programs from scratch.
- We fine-tune an open-source code LLM for the verification task, and propose a method to generate a much larger training dataset.
- We conduct extensive evaluation of our approaches, demonstrating their effectiveness and shortcomings.

1.3 Outline

The rest of the thesis is organized as follows. In Chapter 2, we present an introduction to program verification in Nagini, a brief overview on LLMs, and related work in the broader field of using machine learning techniques in formal efforts.

Chapter 3 presents our approach to the problem, including the dataset design, the prompting-based solution, and the fine-tuning approach.

In Chapter 4, we present our results and the insights revealed by qualitative evaluation.

Finally, we conclude the thesis in Chapter 5, proposing directions for future work.

Chapter 2

Background

This chapter gives an overview of program verification in Nagini, Large Language Models (LLMs), and related work in the area of using learning-based approaches for formal verification.

2.1 Program Verification in Nagini

Program Verification is a technique to prove that a program satisfies a formal specification. Nagini is an automatic verifier for statically-typed Python programs. Given a Python program with Nagini annotations, Nagini returns verification success or a Python-level error message. Although the verification itself is automatic, the user needs to provide the specifications manually. These include pre- and post-conditions of methods, loop invariants, and additional ghost statements. Nagini can verify functional correctness, memory safety, termination and other properties. In order to specify and verify memory safety Nagini supports permission based reasoning for heap location, based on Implicit Dynamic Frames [10], a variant of Separation Logic.

As the thesis focuses on inferring specifications to verify memory safety, we present a brief overview of various constructs in the Nagini specification language that are relevant to our work.

2.1.1 The specification language

Nagini specifications are written as calls to special contract functions imported from the `nagini.contracts` library. Calls to contract functions are ignored (i.e. *no-op*) at runtime and are only used by the verifier. We shall describe the relevant constructs in the Nagini specification language by the means of a running example—a function to reverse a singly linked list as shown in listing 2.2.

Permissions Every heap location (field) is associated with a permission. Permissions are created when the field is assigned to for the first time, for example in the constructor. Methods may only access fields for which they have the acquired permissions and write to fields for which they have exclusive permissions. Permission assertions are expressed using the `Acc` function.

Predicates Predicates allow us to parameterize and abstract over assertions, and are used to define permissions for recursive heap data structures like linked lists and trees. For example, The `is_list` predicate asserts exclusive access to the entire list pointed to by its argument.

Listing 2.1: is_list predicate

```
@Predicate
def is_list(head: Node) -> bool:
    return (
        Acc(head.val)
        and Acc(head.next)
        and Implies(head.next is not None, is_list(head
            .next))
    )
```

Unfolding and Folding In order to use the permissions in a predicate, it needs to be unfolded. Unfolding a predicate exchanges the predicate instance for its body. Folding a predicate does the opposite—if the permissions specified in the body are available in the current program state, they are taken away and the predicate instance becomes available.

Pre- and post-conditions Preconditions specify an obligation on the caller of the method whereas post-conditions specify the guarantees provided by the method. For a verified method, if the method precondition holds before the call (and if the method terminates), the assertions specified in the post-condition will hold after the call.

Permission transfer also happens via pre- and post-conditions. Permissions required in the pre-condition are lost by the caller and made available to the callee. Permissions specified in the post-condition are transferred back to the caller when the callee returns.

Pre- and post-conditions are specified using the `Requires` and `Ensures` annotations respectively. In our example, the pre-condition of method `reverse` requires exclusive access to the entire list pointed to by `head`, as denoted by the `is_list` predicate. The post-condition guarantees that the *returned list* satisfies the `is_list` predicate and returns these permissions back to the

caller. Note that after the call, permission to `is_list(head)` are lost in the caller.

Loop Invariants A loop invariant is an assertion that is true before and after every iteration of the loop. Moreover, it specifies the permissions transferred (1) from the enclosing context to the first loop iteration, (2) from one loop iteration to the next, and (3) from the last loop iteration back to the enclosing context. Inside a loop body, heap locations may only be accessed if the required permissions have been explicitly transferred from the surrounding context to the loop body via the loop invariant. It is specified using the `Invariant` contract function.

In our example, at every iteration of the loop, `prev` points to the reversed list so far, and `ptr` points to the remaining list. As we traverse the list, `is_list(ptr)` is unfolded, making `ptr.next` accessible. The next pointer is updated to point to the reversed list `prev`. Finally, prev is advanced and `is_list(prev)` is folded, thus preserving the invariant.

Listing 2.2: Reverse a linked list

```
1  def reverse(head: Optional[Node]) -> Optional[Node]:
2      """Reverses the list and returns the new head."""
3      Requires(Implies(head is not None, is_list(head)))
4      Ensures(Implies(Result() is not None, is_list(
           Result())))
5      prev = None  # type: Optional[Node]
6      ptr = head  # type: Optional[Node]
7      while ptr is not None:
8          Invariant(Implies(ptr is not None, is_list(ptr)
               ))
9          Invariant(Implies(prev is not None, is_list(
               prev)))
10         Unfold(is_list(ptr))
11         tmp = ptr.next
12         ptr.next = prev
13         prev = ptr
14         Fold(is_list(prev))
15         ptr = tmp
16     return prev
```

2.1.2 Modular verification

Nagini performs modular verification i.e. every method is verified in isolation assuming the dependent methods are verified. Only the specifications of the dependent methods are considered rather than the implementation, while verifying the method in question.

As we shall see, this allows us to perform modular inference when a method has dependencies and allows for our Incremental Verification approach to scale to real-world programs.

2.2 Large Language Models

Large Language Models (LLMs) are deep learning models that are trained on large amounts of textual data with the objective of understanding and generating natural language. State-of-the-art LLMs are based on the Transformer architecture [11]. The heart of the Transformer is the self-attention mechanism, that enables the model to capture long-range dependencies in training data. LLMs have been shown to perform well on a variety of natural language processing tasks such as text summarization, translation, and question answering but also code-related tasks such as code completion and code generation from natural language prompts.

2.2.1 Inference

LLMs are trained to predict the next *token* (a numerical representation of a word / word fragment) given a sequence of tokens. That is, they predict the probability distribution over the token vocabulary $P(x_t|x_1, \ldots, x_{t-1})$. For open-ended generation, x_t is fed back in to the model to predict x_{t+1} and so on. In practice, the model can attend to a fixed number of tokens, called the *context window* or *token limit*. Hence, the predicted probability distribution is in fact $P(x_t|x_{t-\Delta}, \ldots, x_{t-1})$.

Decoding strategies and temperature The process of choosing the next token from the predicted distribution is called decoding. The simplest decoding strategy is greedy decoding, where the token with the highest probability is chosen. However, sampling based-approaches [12] are often used to generate diverse outputs. Among other parameters, the sampling *temperature* controls the randomness of the sampling process—a higher temperature means more random or 'creative' outputs while a temperature of zero is equivalent to greedy decoding.

2.2.2 Training

Training LLMs is typically done in two stages: pre-training and instruction-tuning. Pre-training is done on Internet scale text corpus and gives the LLM broad understanding of language. Then, LLMs are fine-tuned on task-specific tokens.

Pre-trained LM Pre-training is done with the next-token prediction (so-called Causal LM) objective:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t | x_{t-\Delta:t-1}) \tag{2.1}$$

This is usually the most time and resource intensive process.

Few-shot prompting / In-context learning GPT-3 [1] demonstrated an emergent property of model scale: the ability to solve a new task *without* gradient updates. This is called in-context learning or few-shot prompting. The LLM is provided with input-output pairs that demonstrate a task. At the end of the prompt, a new input is given, and the LLM is asked to make a prediction by conditioning on the prompt. In-context learning has shown to be competitive on several NLP benchmarks with models specifically trained on labeled data for the task.

2.2.3 Fine-tuning

A fundamental limitation of prompting is the context-window that the LLM can use during inference. Hence, LLMs are also fine-tuned on domain-specific data to specialize them for a downstream task. This is achieved by initializing from the weights of the pre-trained model and continuing the training on task-specific tokens using the Causal LM objective in equation 2.1.

LLMs for code Although training data for pre-trained LLMs (e.g. GPT, Llama2) also consists of code, state-of-the-art code LLMs are further fine-tuned on code tokens (e.g. Codex, CodeLlama).

Benchmarks Performance of code LLMs is commonly reported on e.g. HumanEval benchmark [2]. The benchmark consists of problem description and few unit-tests. The metric reported is *pass@k*. Output is sampled from the model k times, and if any of the samples passes the test cases, the problem is considered solved. *pass@k* is the fraction of correctly solved problems in this manner.

2.3 Related work

In this section, we survey learning-based approaches to formal verification, including but not limited to LLMs.

Code2Inv [9] addresses the problem of inferring loop invariants using a reinforcement learning approach. The solution creates a structured representation of the loop invariant (as a tree of disjunctions), and infers it in multiple steps using feedback from the verifier.

In Lemur [14], the authors use LLMs as an oracle to successively propose and repair program invariants, based on interaction with a program verifier. The authors have formalized this interaction as a set of deduction rules to obtain some theoretical results about soundness of the framework. Lemur instantiated for GPT-4 outperforms Code2Inv on a C loop invariant inference benchmark.

LeanDojo [15] is a work in the area of interactive theorem proving that tackles the problem of inferring applicable tactics/premises. The authors propose an LLM based to suggest the next tactic based on the current proof state and premises retrieved from a math library.

Chapter 3

Approach

This chapter details our approach. First, we introduce the root dataset which serves as a basis for both prompting and fine-tuning. This is followed by a semi-formal description of the problem statement. Then the rest of the chapter covers our prompting and fine-tuning based approaches.

3.1 Dataset design

The (root) dataset consists of 50 examples of verified programs across 3 data-structures / predicates: list, tree, and lseg:

1. *list*. Null-terminated singly linked list with 18 methods e.g. insert, remove, reverse, merge_sort, etc. (complete list in Appendix A.1.1)
2. *tree*. Binary search tree with 11 methods including insert, contains, height, etc. All methods assume the sortedness property. (complete list in Appendix A.1.2)
3. *lseg*. Linked list segment with the lseg predicate and 21 methods which include most methods from the list predicate and additional methods as described below. (complete list in Appendix A.1.3)

The methods are the ones commonly found in standard library implementations of these data structures, verified for memory safety. We made sure to have a good mix of recursive and iterative methods. To facilitate qualitative evaluation, we classified the methods into three difficulty levels: easy, medium, and hard. The predicate definitions and the classification can be found in Appendix A.

Most methods for the list predicate are recursive. The lseg predicate allows us to write iterative counterparts of these methods. For example, `count_iter` shown in listing 3.1. The `join` method, with the specification shown in listing 3.2 combines two list segments into one:

Listing 3.1: count_iter method that preserves the lseg predicate.

```
def count_iter(head: Optional[Node]) -> int:
    """Counts the number of nodes in the list."""
    Requires(lseg(head, None))
    Ensures(lseg(head, None))
    cnt = 0
    ptr = head  # type: Optional[Node]
    Fold(lseg(head, ptr))
    while ptr is not None:
        Invariant(lseg(head, ptr))
        Invariant(lseg(ptr, None))
        ...
        # Increment count
        # Unfold and advance ptr
        # Extend the lseg(head, ptr) predicate using 'join'
    join(head, ptr, None)
    return cnt
```

Listing 3.2: join: combines two list segments.

```
def join(a: Optional[Node], b: Optional[Node], c: Optional[Node]) -> None:
    """Join two list segments."""
    Requires(lseg(a, b) and lseg(b, c))
    Ensures(lseg(a, c))
...
```

Each element of the dataset is a tuple of the form

$$\mathcal{D} = \{(P_u^i, e^i, P_v^i)\}_{i=1}^{N}$$

Where P_u^i is the unverified program, e^i is the verification error, and P_v^i is the verified program. An example is shown in Figure 3.3.

3.2 Problem statement

We introduce some notation that we will use in this and following sections and define the problem statement as follows: given an unverified program in Python P_u (with a docstring comment describing the method's intended behavior) infer the verified program P_v' that:

1. Asserts pre- and post-condition for memory safety that match the ground truth specifications in P_v.

```
def join_lists(head1: Optional[Node], head2: Optional[
    Node])
    -> Optional[Node]:
    if head1 is None:
        return head2
    if head2 is None:
        return head1
    head1.next = join_lists(head1.next, head2)
    return head1
```

Figure 3.1: P_u: An unverified program

```
Verification failed: Method call might fail. There
    might be insufficient permission to access head1.
    next. at line 7.17
```

Figure 3.2: e: Verification error from Nagini for P_u

```
def join_lists(head1: Optional[Node], head2: Optional[
    Node])
    -> Optional[Node]:
    Requires(Implies(head1 is not None, is_list(head1))
        )
    Requires(Implies(head2 is not None, is_list(head2))
        )
    Ensures(Implies(Result() is not None, is_list(
        Result())))
    if head1 is None:
        return head2
    if head2 is None:
        return head1
    Unfold(is_list(head1))
    head1.next = join_lists(head1.next, head2)
    Fold(is_list(head1))
    return head1
```

Figure 3.3: P_v: The ground truth verified program

2. Contains the necessary Nagini annotations i.e. Fold, Unfold, Invariant, etc. required to verify the program.
3. Is otherwise unchanged from P_u.

We provide a brief justification of the above constraints. Constraints 1 and 2 collectively ensure that the inferred specifications by the model are correct. That is, we are okay with P'_v being not necessarily identical to P_v, as long as the pre- and post-conditions are the same. This is because there might be several ways to verify the same program. For example `Unfolding` might be used in a conditional statement instead of a combination of `Unfold` and `Fold`. Similarly, if `a` and `b` are aliases at a program point, it does not matter, for example, whether `Fold(is_list(a))` or `Fold(is_list(b))` is used.

Constraint 3 ensures the program itself is unchanged. However, we need to allow for a simple transformation. Consider the unverified program:

```
def height(node: Optional[TreeNode]) -> int:
"""Returns the height of the tree rooted at node"""
if node is None:
    return 0
return max(height(node.left), height(node.right))
```

The verified program must introduce an additional variable:

```
...
    Unfold(tree(node))
    h = 1 + max(height(node.left), height(node.right))
    Fold(tree(node))
    return h
```

In our experiments, the model successfully performs such transformations. We observed in our experiments that the modifications made by the model to the original program are none to minimal. However, we propose automating this aspect (by defining rules on which kind of transformations are admissible) as a future work item.

The docstring comment gives the model a better chance of inferring the correct pre- and post-conditions. Indeed, it happens that constraints 2 and 3 are met however the pre- or post-condition is not the one intended. As an example, the model may produce the following program that verifies:

Listing 3.3: A false positive verification

```
def subtree(root: Optional[TreeNode], key: int) ->
   Optional[TreeNode]:
    """Returns the subtree rooted at node with the
       given key if it exists, None otherwise
    Permissions to the rest of the tree are leaked"""
    Requires(Implies(root is not None, tree(root)))
```

```
    Ensures(Implies(root is not None, tree(root)))
    # method body and Nagini annotations omitted
    ...
```

In this case the result of the method is unusable (after the method returns, the callee does not get permissions to the subtree rooted at `key`). The intended post-condition (i.e. the one in the ground truth verified program) is:

```
Ensures(Implies(Result() is not None, tree(Result())))
```

Again, we discard such successful verifications as false positives.

3.3 Prompting

The primary way to interact with an LLM is through a prompt. Hence, our goal is to effectively encode the verification task in the prompt and guide the model to generate the verified program, possibly in several steps and attempts.

First, we start with an easier problem: given examples of verified programs $\{(P_u, e, P_v)\}$, can the model generate the verified program for an unseen unverified program?

Next, we move on to the more challenging problem of building up the set of verified programs from scratch.

Note that in both cases, we treat each predicate separately (when trying to infer a program for `list` we only consider other examples from `list` predicate for inclusion in few-shot prompt). This is in contrast with fine-tuning, where the model is trained on examples from all predicates.

3.3.1 Few-shot prompting

Let $(P_u, e) \in \mathcal{D}$ be the unverified program of interest. Let $E_{\text{fs}} \subset \mathcal{D} \setminus (P_u, e)$ be the set of examples to be included in the few-shot prompt. The initial input to the model is then constructed from E_{fs}, (P_u, e) and additionally a *system prompt*. The exact format varies from model to model, but the general structure is shown in Listing 3.4.

The system prompt sets the overall behavior of the model. By default, it's something along the lines of `You are a helpful assistant`. But we can get much better performance out of the model by including instructions relevant to the task in the system prompt [6]. Before diving into the algorithm, we describe our design of the system prompt.

Listing 3.4: General structure of the few-shot prompt.

```
-----------------------------------
          System prompt
-----------------------------------
-----------------------------------
Input:  P_u^0, e^0
Output: P_v^0
...
Input:  P_u^{N-1}, e^{N-1}
Output: P_v^{N-1}
-----------------------------------
Input: P_u, e
```

System prompt

Our system prompt is composed of several fixed and variable (i.e. predicate dependent) components:

Task description and formatting instructions With the default system prompt, the model usually generates an explanation of the error and a proposed fix, usually with some Markdown wrapping. Hence, we specify in the system prompt to only return the verified program without any explanation or wrapping, so we can directly extract the program snippet from the response.

Available Nagini constructs We provide a list of available Nagini constructs that the model can use, each with an example demonstrating their syntax informally. The examples are predicate dependent.

Predicate definition This is also a variable part of the system prompt. Based on the data structure we are verifying, we plug in the corresponding predicate definition.

These three components constitute System Prompt 1. The complete prompt can be found in Appendix B.

Algorithm

We present the algorithm in Algorithm 1 and explain it's high level execution. The model is prompted with *prompt* initialized as described above. The model generates a verification attempt P'_v which is then verified with Nagini. If the verification succeeds, P'_v is returned. If the verification fails, the generated program P'_v and the verification error e' is added back to the prompt the model is prompted again. This process is repeated for n levels of error messages. We call n the error depth.

The *prompt* is reinitialized and the above process is repeated for k *attempts*, increasing the temperature according to a schedule in each attempt.

Algorithm 1 TryVerify$_{k,n}$: Prompting with errors

Input: System prompt S, unverified program P_u, verification error e, few-shot examples E_{fs}
Output: Verified program P'_v or None

```
1:  for i = 1 to k do
2:    prompt ← construct_prompt(S, (P_u, e), E_fs)
3:    for j = 1 to n do
4:      P'_v ← model(prompt, temperature_k)
5:      e' ← verify(P'_v)
6:      if e' is None then
7:        return P'_v
8:      end if
9:      prompt ← extend_prompt(prompt, P'_v, e')
10:   end for
11: end for
```

We describe how the algorithm generalizes to different models and provide some implementation details.

As described earlier, models have different prompt formats: some are chat based (e.g. GPT-4, GPT-3.5) whereas some are completion based (e.g. CodeLlama, GPT-Instruct) The `construct_prompt` function constructs the prompt specific to the model's prompt format. Similarly, the `extend_prompt` function takes care of extending the prompt with the verification attempt and the error message. (We have instantiated the algorithm for several models: GPT-3.5, GPT-4, GPT-Instruct, open source models hosted on Together AI, pre-trained models and finally also our fine-tuned model.)

The call to the model on line 4 abstracts away the details of the model's API and the parameters (e.g. temperature) it takes.

The `verify` function on line 5 checks the verification attempt with Nagini. In order to do so, it combines the verification attempt with the top level declarations i.e. the data structure and predicate definition, and the (verified) dependencies of P'_v and returns the verification error.

Finally, the algorithm naturally introduces the metric that we use for evaluation:

Definition 3.1 (verify@(k, n)) *verify@(k, n) is the fraction of examples in the evaluation set under consideration for which* TryVerify$_{k,n}$ *succeeds.*

3.3.2 Incremental verification

The few-shot prompting approach in the previous section assumes access to a dataset of verified programs for the given predicate. In practice, this is might not be the case. Here, we are interested in starting from scratch and incrementally building the set of verified programs. The idea is to get some easy examples right, thus *bootstrapping* the verification process and use them as few-shot examples to successively verify more complex examples.

This approach could be used to verify a large codebase from scratch. As Nagini performs modular verification, incremental verification would start by inferring specifications for simple methods successively verifying dependent methods leading up to the complete program.

However, without access to few-shot examples, the model struggles to verify even the simplest programs. To remediate this, we add further details about the verification task to the system prompt. The next paragraph describes these additions, based on our experimentation with GPT-4.

Inadequacy of System Prompt 1 With the System Prompt 1 described in subsection 3.3.1, the model struggles to bootstrap: it tries to infer functional specifications rather than for memory safety. It makes numerous syntax errors. To address this, we add to the system prompt basic examples demonstrating the verification task for an *unrelated* predicate.

We observe that it starts getting 'easy' to 'medium' examples right, but still struggles with iterative examples. We alleviate this by adding semantics of the Nagini constructs and detailed explanation of `Invariant` to the system prompt.

Finally, some methods become easier to verify using `Unfolding` rather than `Unfold/Fold`. As an example, consider the program snippet in Listing 3.5: a slightly convoluted version of the program `reverse` from Listing 2.2. In this version, we check whether the argument `head` is a singleton list as a base case, before entering the loop. After the return statement on line 5, the model often fails to fold the `is_list(head)` predicate. Even though the model may infer the loop invariant on line 10 correctly, the missing `Fold` causes it to not hold on entry. Moreover, the model does not seem to respond to the resulting verification error.

If the model verifies the base case using `Unfolding` instead,

```
1 if Unfolding(is_list(head), head.next) is None:
2     return head
3 ...
```

these kinds of errors are avoided. Hence, we add a minimal example demonstrating the equivalence of the two constructs.

Listing 3.5: reverse method with an additional check.

```
...
Unfold(is_list(head))
if head.next is None:
    Fold(is_list(head))
    return head
### missing Fold(is_list(head)) here
prev = None  # type: Optional[Node]
ptr = head  # type: Optional[Node]
while ptr is not None:
    Invariant(Implies(ptr is not None, is_list(ptr)))
...
```

We summarize the additions necessary to make incremental verification work below:

System Prompt 2

System Prompt 2, in addition to System Prompt 1, includes:

Semantics of Nagini constructs We briefly explain the meaning of unfolding and folding a predicate, loop invariant and permission transfer from outer context to loop body via the invariant.

Examples of verified programs for an unrelated predicate This example helps demonstrate various constructs (e.g. `Implies`) that are already required to verify even 'easy' methods. Moreover, we show with help of an example the equivalence of Unfolding and Unfold/Fold.

Both the above additions are *fixed* components of the system prompt. The complete System Prompt 2 can be found in Appendix B.1.2.

Methods with dependencies

There are two ways to handle methods with dependencies. We can either evaluate the method in isolation, assuming all dependencies are correctly verified (i.e. use the ground truth verified methods). Or, we can choose to only verify the dependent method once all its dependencies are verified, and use the model-verified versions of the dependencies. We adopt the latter approach. Below we present the algorithm for incremental verification.

Algorithm

Let V be the set of verified programs, initially empty. Let U be the set of unverified programs, initially all examples in the dataset $\mathcal{D}$ (for a given predicate). The algorithm is shown in Algorithm 2.

Algorithm 2 Incremental verification

Input: System prompt S, unverified programs U, parameters k, n
Output: Verified programs V

1: $V \leftarrow \emptyset$
2: **while** $U \neq \emptyset$ **and** V was updated **do**
3: **for all** $P_u, e \in U$ **do**
4: **if not** all dependencies of P_u are in V **then**
5: continue
6: **end if**
7: $P_v \leftarrow \textsc{TryVerify}_{k,n}(S, P_u, V)$
8: **if** P_v is not None **then**
9: $V \leftarrow V \cup \{P_v\}$
10: $U \leftarrow U \setminus \{P_u\}$
11: **end if**
12: **end for**
13: **end while**

Note that because the algorithm tries to be exhaustive, it is not cheap: if U contains N examples, the algorithm makes $\mathcal{O}(N^2)$ calls to $\textsc{TryVerify}_{k,n}$ (which itself makes up to $k \times n$ calls to the model) in the worst case. That is, every time a new method is verified, all other methods in U are tried again with the updated few-shot examples V.

Sensitivity to the order of methods in U

Although the algorithm tries to be exhaustive, it is not exhaustive enough. Its performance also depends on the *order* of examples in U. Let V be the currently accumulated verified programs at an execution point. Let $\{u_1, u_2\} \subset U$ be the unverified methods, that the model can verify using V as few-shot examples. That is, $\textsc{TryVerify}(S, u, V)$ would succeed for both u_1 and u_2. Assume also that given V, the model would produce a `Unfolding` based verification for u_1 and `Unfold/Fold` based verification v_2 for u_2.

Now, if the algorithm picks u_2 before u_1, we have added another method with `Unfold/Fold` based verification to V. As V grows, the effect of few-shot examples starts dominating the system prompt. Hence, it may happen that after $V \leftarrow V \cup \{v_2\}$, the model also generates a `Unfold/Fold` based solution for u_1, further reinforcing this tendency. However, when the model would encounter the method shown in Listing 3.5, it would fail to verify as described earlier.

Hence, the order in which the methods are picked from U also affects performance. (That is, the set operations on line 3 and 10 in Algorithm 2 are in fact list operations) Therefore, in the evaluation, we report the average

performance over three random orderings of U.

3.4 Supervised fine-tuning

In this section, we turn our attention to fine-tuning an open-source LLM on the program verification task. Unlike in case of prompting, our goal here is to obtain the verified program 'zero-shot', given the unverified program and verification error. We still allow error depth and a fixed number of attempts. That is, the inference algorithm for the fine-tuned model is $\textsc{TryVerify}_{k,n}(S, (P_u, e), \emptyset)$.

We chose the open source model CodeLlama-7B [8] (the *base* model from the CodeLlama family) as the pre-trained model.

3.4.1 Training dataset

The dataset $\mathcal{D}$ described in the previous section would fall short as a training dataset for this purpose. Firstly, fine-tuning requires a large number of examples due to the large parameter size of the pre-trained model to be effective. More importantly, $\mathcal{D}$ does not contain intermediate (partially) unverified programs.

To address both issues, we propose an algorithm that exponentially increases the number of examples and contains partially unverified programs.

We use $\mathcal{D}$ as the starting point for training dataset generation. The idea is to start with the verified program, and remove all combination of specifications and ghost statements to get the unverified programs and the corresponding verification error. Hence, if the program has m lines of specifications, we get 2^m unverified programs. The algorithm is thus Algorithm 3.

Note that some partial programs P'_u obtained on line 6 might still verify (e.g. a combination of post-condition and Fold omitted). We exclude such examples from the training dataset.

Moreover, the number of specs in P_v range anywhere from 3 to 15. So as to not overrepresent a particular example, we limit the number of specs to be considered for removal to 10. This means a maximum of 1024 examples per method.

Finally, we apply the prompt template shown in Listing 3.6 to every example in $\mathcal{D}^{\text{exp}}$. At inference time, the model is prompted with the prompt template applied to (P_u, e) until `### Verified program:`.

Introducing $\mathcal{D}^{\text{exp}}_{\text{ext}}$

All examples in $\mathcal{D}^{\text{exp}}$ go from an unverified program with fewer specs (annotations, used interchangeably) to a verified program. We observed

Algorithm 3 Training dataset generation

Input: Root dataset $\mathcal{D}$
Output: Training dataset $\mathcal{D}^{\text{exp}}$

$\mathcal{D}^{\text{exp}} \leftarrow \emptyset$
for all $P_v \in \mathcal{D}$ **do**
  $specs \leftarrow \text{get\_specs}(P_v)$
  for all $subset \in \text{powerset}(specs)$ **do**
    $P'_u \leftarrow \text{remove\_specs}(P_v, subset)$
    $e' \leftarrow \text{verify}(P'_u)$
    if e' is not None **then**
      continue
    end if
    $\mathcal{D}^{\text{exp}} \leftarrow \mathcal{D}^{\text{exp}} \cup \{(P'_u, e', P_v)\}$
  end for
end for
return $\mathcal{D}^{\text{exp}}$

Listing 3.6: Prompt template for fine-tuning.

```
{System prompt 1}
### Unverified program:
{P_u}
### Verification error:
{e}
### Verified program:
{P_v}
```

during evaluation that the model never really 'deletes' an unnecessary spec. Hence, we also want to have examples where the verified program has fewer specs than the unverified program. To this end, we manually augmented some examples (i.e. verified programs) in the root dataset $\mathcal{D}$ with spurious annotations. That is, we inserted `Fold`, `Unfold`, `Invariant`, etc. statements at random program points. The examples chosen for this augmentation were ones that have fewer specs to begin with.

Let this extended root dataset be $\mathcal{D}_{\text{ext}}$. We then use $\mathcal{D}_{\text{ext}}$ as the input to the algorithm 3. We call the resulting training dataset $\mathcal{D}^{\text{exp}}_{\text{ext}}$.

Parameter efficient fine-tuning

For fine-tuning, we use the widely adopted LoRA approach [4] to efficiently utilize GPU resources. We also use 8-bit quantization to load the model so that the model and optimizer state fits in the GPU memory. We set

up the training using the HuggingFace Transformers library [13]. We use the AdamW optimizer with a learning rate of 2e-4 and a batch size of 32 (simulated with 8 gradient accumulation steps). We ran the training on Amazon SageMaker with a ml.g5.2xlarge instance which has 8 vCPUs, 32 GiB memory, and 1 NVIDIA A10G GPU with 24 GiB memory.

3.4.2 Experiments

With the aforementioned setup, we fine-tuned two models: the first on $\mathcal{D}^{\text{exp}}$ for 4 epochs and the second on $\mathcal{D}^{\text{exp}}_{\text{ext}}$ for 3 epochs and evaluated their performance. The training and evaluation split is described in Appendix A.2.

Chapter 4

Evaluation

In this chapter, we detail the experiments we conducted to evaluate our approach and the insights we gained from them. First, we start with prompting and show the influence of varying the system prompt. Next, we show the performance of our fine-tuned model and compare it with the pre-trained model zero-shot and few-shot.

4.1 Prompting

4.1.1 Few-shot prompting

The few-shot examples consist of all other methods for the predicate, except the one being verified. The results shown are $verify@(k = 3, n = 3)$ as percentages (i.e. the fraction of methods that verify for the predicate over total number of methods for the predicate).

Table 4.1: Performance of few-shot prompting on different system prompts.

	GPT-4	
	sys_1	sys_2
list	94.4	**100.0**
tree	90.9	**100.0**
lseg	76.2	**80.1**
	84.0	**90.0**

The detailed system prompt `sys_2` always outperforms `sys_1`. Specifically, for list and tree, the model is able to verify an additional 'hard' method (`reverse_iter` resp. `insert`).

Moreover for lseg, the model is able to verify an additional challenging method (`remove_last`) with the detailed system prompt:

Listing 4.1: Method `remove_last` for the `lseg` predicate.

```
1  def remove_last(first: Optional[Node], last: Node) ->
       Optional[Node]:
2      """Remove the last node from the list and returns
           the new last"""
3      Requires(lseg(first, last))
4      Ensures(lseg(first, Result()))
5      ...
6      Unfold(lseg(first, last))
7      if first.next is last:
8          Fold(lseg(first, first))
9          return first
10     ...
```

With `sys_1`, the model either infers the incorrect post-condition Ensures(lseg(Result(), first)), or does not infer the Fold(lseg(first, first)) on line 8 required for the post-condition to hold in the base case. This shows that even in the presence of few-shot examples, the model benefits from the detailed explanation in the system prompt.

4.1.2 Incremental verification

We only show results for the most capable model: GPT-4 (gpt-4-1106). The tree predicate was ran on the latest gpt4-turbo-preview, which performs worse than gpt-4-1106, also few-shot. Due to time and resource constraints, the lseg predicate was omitted. Results are percentage of successful verifications after running Algorithm 2 with U initialized to all examples for the predicate at $k = 4, n = 5$.

Table 4.2: Performance of incremental verification on different system prompts. Averaged across 3 randomized runs.

	sys_1	sys_2
list	24.0	**93.3**
tree	36.3	**87.8**
	28.7	**90.6**

The effect of system prompt is more pronounced in incremental verification. The average performance over 3 random orderings of U is shown for reasons explained in detail in subsection 3.3.2.

With `sys_1` for the list predicate, only one ordering of U results in any successful verifications. The other orderings resulted in 0 successful verifications (all false positives with `@ContractOnly` annotation).

4.2 Our fine-tuned model

We contrast the zero-shot performance of our fine-tuned models with the zero-shot and few-shot performance of the pre-trained model. For the pre-trained model, we report two metrics: (1) on all examples (N=50) and (2) on evaluation set used for our fine-tuned models (N=17).

For the fine-tuned model, we consider two versions, one trained on $\mathcal{D}^{\text{exp}}$, and the other trained on $\mathcal{D}^{\text{exp}}_{\text{ext}}$ and report zero-shot performance on the evaluation set. Both fine-tuned models perform close to 100% on the training set.

All results are percentages $verify@(k = 4, n = 3)$.

Table 4.3: Performance of our fine-tuned models compared to the pre-trained model.

	zero-shot	few-shot		$\mathcal{D}^{\text{exp}}$	$\mathcal{D}^{\text{exp}}_{\text{ext}}$
	all	all	eval		
list	0.0	22.2	16.7	66.7	**66.7**
tree	0.0	72.7	66.7	33.3	**100.0**
lseg	0.0	44.4	50.0	25.0	**62.5**
	0.0	50.0	41.2	41.2	**70.5**

An explanation of the results follows. When prompted zero-shot, the pre-trained model simply reproduces the unverified program for all attempts k. [1] Needless to say, increasing the error depth n has no effect, as the model keeps cycling between the same unverified program and error message.

Few-shot, the pre-trained model does perform better. We give a breakdown of the difficulty of methods that pre-trained model is able to tackle. For list and tree, the model is only able to verify some easy methods.

For lseg, impressively the model is able to verify all iterative methods of medium difficulty (`contains_iter`, `count_iter`, `index_of_iter`). All these methods walk the list segment and maintain the `lseg` invariant for the prefix and the suffix of the list segment. Recall the loop invariant from `count_iter` method presented in Listing 3.1:

Listing 4.2: Loop invariant common to `count_iter`, `contains_iter`, `index_of_iter` methods.

```
while ptr is not None:
    Invariant(lseg(head, ptr))
    Invariant(lseg(ptr, None))
```

[1] Both zero-shot and few-shot, the pre-trained model does not respect the prompt format and generates extraneous output until *max_new_tokens=320* is reached—hence we have implemented logic to extract the program snippet of interest. On the other hand, the fine-tuned model correctly produces an EOS token when it believes the verified program is complete.

Moreover they are quite similar in structure, which explains the pre-trained model's success in the few-shot setting.

Our best fine-tuned model outperforms the pre-trained model in all scenarios (by 29.2% on the evaluation set). Specifically for the list predicate, we attribute the better performance to the fact that the fine-tuned model has seen similar examples, albeit for a *different* predicate in the training data. The pre-trained model fails to infer the correct pre- or post-condition for `count` and `insert` methods for the list predicate. The fine-tuned model, on the other hand, is able to generalize from similar examples seen over different predicates in the training data (i.e. `tree::count` and `lseg::insert`). Next, we show a more challenging case where the fine-tuned model seems to generalize from a different predicate.

Case study: `lseg::reverse_iter` Both the pre-trained and fine-tuned models fail to verify `reverse_iter` for the `lseg` predicate. Nevertheless, it is interesting to see how their outputs differ.

We saw that the pre-trained model is able to verify some iterative methods, and attributed this success to having similar invariants in the few-shot example. However, `revers_iter` is unique in that it requires maintaining the invariant in *opposite* directions: the currently reversed prefix `lseg(prev, None)` and the rest of the list `lseg(ptr, None)`:

Listing 4.3: The correct invariant for `lseg::reverse_iter` method.

```
while ptr is not None:
    Invariant(lseg(ptr, None))
    Invariant(lseg(prev, None))
```

The pre-trained model still incorrectly produces the invariant shown in Listing 4.2. The fine-tuned model, on the other hand, produces the correct invariant. This is perhaps due to the `list::reverse_iter` method being in the training data, which maintains a similar invariant for the `is_list` predicate:

Listing 4.4: Loop invariant for `list::reverse_iter` method.

```
while ptr is not None:
    Invariant(Implies(ptr is not None, is_list(ptr)))
    Invariant(Implies(prev is not None, is_list(prev)))
```

Pitfall: `list::drop_iter` In the previous paragraph, we saw an advantage of the fine-tuned model generalizing from similar examples for a different predicate in the training data. However, the model also makes false generalizations as described next.

Both the pre-trained and fine-tuned models fail to verify `drop_iter` for the list predicate. The pre-trained model does not generate the loop invariant at all, but infers all other specifications correctly. The fine-tuned model generates a `Fold(is_list(head))` before the loop, or an extra `join` in the loop:

Listing 4.5: Code produced by the fine-tuned model for `list::drop_iter`

```
1 Fold(is_list(head))
2 while ptr is not None:
3     Invariant(is_list(head))
4     Invariant(is_list(ptr))
5     ...
6     join(head, tmp, ptr)
```

This verification strategy is common for iterative methods in for the `lseg` predicate (e.g. `count_iter` in Listing 3.1), but the model incorrectly applies it here. Note that the list predicate does not contain the `join` method at all. We attribute this behavior to data imbalance: The (root) training data for `list` contains only 1 iterative method, compared 6 for the `lseg` predicate. This number is further amplified by Algorithm 3.

$\mathcal{D}^{\text{exp}}$ vs. $\mathcal{D}^{\text{exp}}_{\text{ext}}$ Recall that the rationale behind adding spurious spec statements to generate $\mathcal{D}^{\text{exp}}_{\text{ext}}$ was that the model might learn to 'delete' specs. However, after fine-tuning we find that the model does not consistently do so. The model fine-tuned on $\mathcal{D}^{\text{exp}}$ always inserts an extra Fold shown on line 3 for `lseg::index_of_iter` and does not respond to the error message 'Fold might fail ...'.

Listing 4.6: Code produced by the $\mathcal{D}^{\text{exp}}$-fine-tuned model for `lseg::index_of_iter`

```
1 ...
2 if Unfolding(lseg(ptr, last), ptr.val) == val:
3     Fold(lseg(ptr, last))
4     join(first, ptr, last)
5     return index
6 ...
```

The model fine-tuned on $\mathcal{D}^{\text{exp}}_{\text{ext}}$ inserts this extra fold initially (error depth $n = 1$), but promptly removes it at $n = 2$.

However, for `merge_sort` for both list and lseg predicates, both the fine-tuned models insert an extra Unfold before the call to `split` and do not respond to the resulting error message 'Pre-condition of split might not hold ...'.

Listing 4.7: Code produced by the fine-tuned models for `merge_sort`

```
1 def merge_sort(head: Optional[Node]) -> Optional[Node]:
2 ...
```

```
    mid = count(head) // 2
    Unfold(lseg(head, None))
    rest = split(head, mid)
...
```

Discussion: token limit The pre-trained CodeLlama model has a token limit of 16k tokens. Already with the `lseg` predicate (N=23), we start approaching the limit in the few-shot setting (22 examples constituting around 9k tokens). Although increasing the token limit is an active area of research, with the latest GPT models supporting 128k tokens, the token requirement (and hence the cost per inference) scales linearly with the number of examples in the few-shot prompt. As the fine-tuned model performs zero-shot inference, it has a constant dependence on the token requirement, consuming only around 1k tokens per inference. We highlight this as an additional advantage of the fine-tuned model.

Chapter 5

Conclusion

We conclude by summarizing the key contributions and results of this thesis and discuss directions for future work.

This thesis tackles the problem of using LLMs to automatically infer memory safety specifications for Python programs. First, we created a dataset of verified programs that serves as the basis of our work and a benchmark for evaluation.

We developed the strategy of prompting with errors and instantiated it for GPT-4. We showed that it performs quite well (i.e. 90% on our benchmark) in the few-shot setting. To reflect the real-world setting, we developed an algorithm for incremental verification, highlighting the role of system prompt to get better performance. This algorithm could be used to verify entire standard library implementations from scratch. Our method is implemented as a generic framework and can be easily extended to different models.

We fine-tuned an open-source pre-trained LLM (CodeLlama-7B) for the verification task, proposing a method to generate a much larger training dataset using our root dataset. We showed that fine-tuning improves the performance of the pre-trained model (by 29.2% on our benchmark). We explained the results qualitatively, highlighting the generalization capabilities of our fine-tuning approach as a strength and data imbalance in the training data as a weakness.

5.1 Future work

5.1.1 Properties beyond memory safety

With verified programs for memory safety, our dataset lays the groundwork for verifying more complex properties e.g. functional correctness. For exam-

ple, methods in the tree predicate can be verified for the BST property. The dataset can be extended with new data structures and predicates.

Both our few-shot prompting and incremental verification approach can be applied with minimal modifications on this new dataset.

We believe the qualitative insights gained by restricting to the class of memory safety specifications could inform the extension of both our approaches to more complex properties.

5.1.2 Automating equivalence checks

As discussed in the problem statement (Section 3.2), constraints 1 and 3 require some form of equivalence checking. We propose automating these aspects of our approach so that it can scale to larger datasets (i.e. automate the evaluation of the benchmark) and improve usability for the end user.

- We saw that the model output may contain slight modifications of the original program. Currently, the user has to manually check that the inferred program is equivalent to the unverified program. We propose defining a set of admissible transformations on programs that can be automatically checked for equivalence (e.g. using AST analysis). Model outputs that fail this check can then be discarded as false positives, reducing the noise for the end-user and automating this aspect in the evaluation of our benchmark.
- As the dataset grows and includes more complex properties, manually checking the equivalence of pre- and post-conditions with ground truth specification becomes infeasible. This check could be discharged to a theorem prover (e.g. Z3) to further reduce false positives.
- In absence of ground truth specifications, we could add a sanity check to ensure that the pre-condition does not contain a contradiction. We propose adding `Ensures(False)` post-condition to the inferred specifications and checking that the resulting not program does *not* verify.

5.1.3 Reinforcement learning

Fine-tuning LLMs with reinforcement learning has been shown to be effective in aligning the model to human feedback [7]. The idea applies even when the reward function is not based on human feedback, but a scalar reward from an oracle. Hence, a reward function parametrized by the Nagini verification error would need to be constructed.

Note that a supervised fine-tuned model is a prerequisite for this approach, in order to ensure the model produces outputs that are 'in-distribution'. Our thesis has already achieved this first step.

Appendix A

Dataset

The (root) dataset consists of 50 examples of verified programs across 3 predicates: list, tree, and lseg. Moreover, we have classified the methods into 3 difficulty levels: easy, medium, and hard. The classification is rather informal, but roughly proportional to number of spec statements required to verify the method. Iterative methods and methods with dependencies are considered more challenging. `lseg(a,b)` predicate is considered more challenging than `list(a)` as it's parametrized by two arguments, both of which must be inferred. Additionally, in many cases the model needs to infer specs of the form `Fold(lseg(ptr, ptr))` or `Fold(lseg(None, None))` as the base case of the proof.

As convention, all iterative methods have an `_iter` suffix. All other methods are recursive, except `prepend` and `remove_first` in list and lseg and `val_head` in tree.

Table A.1: Distribution of methods by difficulty.

	easy	medium	hard
list	5	10	3
tree	8	2	1
lseg	5	11	5
	18	23	9

A.1 Root dataset

A.1.1 List

The list predicate is defined as:

```
@Predicate
def list(head: Node) -> bool:
    return (
```

```
        and Acc(head.val)
        and Acc(head.next)
        and Implies(head.next is not None, list(head.
            next))
    )
```

The list methods are shown in Table A.2.

Table A.2: List methods: $N = 18$, $N_{\text{iter}} = 2$, $N_{\text{rec}} = 14$

Method	Depends on	Difficulty
prepend		easy
append		medium
remove_first		easy
remove_last		medium
join_lists		easy
contains		easy
insert	prepend	medium
remove		medium
index_of		medium
drop		medium
drop_iter		hard
reverse_iter		hard
insert_sorted		medium
insertion_sort	insert_sorted	medium
count		easy
split		hard
merge		medium
merge_sort	count, split, merge	medium

A.1.2 Tree

The tree predicate is defined as:

```
@Predicate
def tree(n: TreeNode) -> bool:
    return (
        Acc(n.key)
        and Acc(n.left)
        and Acc(n.right)
        and Implies(n.left is not None, tree(n.left))
        and Implies(n.right is not None, tree(n.right))
    )
```

The tree methods are shown in Table A.3. The depends on column is omitted as no methods have dependencies.

Table A.3: Tree methods: $N = 11$, $N_{\text{iter}} = 0$, $N_{\text{rec}} = 10$

Method	Difficulty
val_head	easy
height	easy
count	easy
sum	easy
insert	hard
contains	easy
inorder	easy
min	easy
mirror	easy
subtree	medium
min_depth	medium

A.1.3 List segment

The lseg predicate is defined as:

```
@Predicate
def lseg(first: Optional[Node], last: Optional[Node])
    -> bool:
    return Implies(
        first is not last,
        Acc(first.val)
        and Acc(first.next)
        and lseg(first.next, last)
    )
```

The lseg methods are shown in Table A.4.

Table A.4: List segment methods: $N = 21$, $N_{\text{iter}} = 9$, $N_{\text{rec}} = 10$

Method	Depends on	Difficulty
join		medium
prepend		easy
remove_first		easy
remove_last		medium
contains		easy
contains_iter	join	medium
insert	prepend	easy
insert_iter	prepend, join	hard
append		medium
append_iter	join	hard
index_of_iter	join	medium
reverse_iter		hard
insert_sorted		medium
insertion_sort	insert_sorted	medium
insertion_sort_iter	insert_sorted	hard
merge		medium
count		easy
count_iter	join	medium
split		medium
split_iter	join	hard
merge_sort	count, split, merge	medium

A.2 Training dataset

Examples held out for fine-tuning (N=17):

```
list: insert, remove_last, drop_iter, count, split, merge_sort (N=6)
tree: insert, height, min_depth (N=3)
lseg: remove_last, count_iter, index_of, reverse, count, split,
      merge, merge_sort (N=8)
```

After removing the hold-out examples from $\mathcal{D}$ and running Algorithm 3, we obtain the training dataset $\mathcal{D}^{\text{exp}}$ with $N = 4611$ examples.

The extended dataset: $\mathcal{D}^{\text{exp}}_{\text{ext}}$ is obtained by running Algorithm 3 on the dataset $\mathcal{D}_{\text{ext}}$ extended with spurious specifications for a total of $N = 11415$ examples.

Appendix B

Prompts

B.1 System prompt

B.1.1 System prompt 1

```
You are an assistant that given a python program, annotates it with appropriate Nagini
annotations so that verification succeeds. Nagini is a static verifier for Python.
Our aim is to given a statically typed Python program, to come up with appropriate
preconditions (e.g. Requires(is_list(head)), Requires(Implies(n is not None, predicate(n)))),
postcondition (e.g. Ensures(is_list(Result()))), loop invariants (Invariant(assertion)),
predicate fold/unfolds (e.g. Fold(is_list(head)) / Unfold(is_list(head))) so that the program
verifies correctly.
Unfolding(e1, e2) evaluates e2 in the context where predicate e1 is temporarily unfolded.

The user will provide Python code and the verification errors.
You must add, remove or change the specifications so that the resulting code verifies correctly.
Return only the code without any explanation or wrapping.

The is_list predicate is defined recursively as:
@Predicate
def is_list(head: Node) -> bool:
    return (
        head is not None
        and Acc(head.val)
        and Acc(head.next)
        and Implies(head.next is not None, is_list(head.next))
    )
```

B.1.2 System prompt 2

System prompt 2 is System prompt 1 plus:

```
Unfolding a predicate exchanges the predicate instance for it's body.
Folding a predicate does the opposite - i.e. permissions specified by in the body, if available
in the current program state, are taken away and the predicate instance becomes available again.
A fold operation failing means the permissions specified in the predicate body are not available
```

```
in the program state. An unfold operation failing means the predicate instance is not available
in the program state. This might mean it is already unfolded and that the permissions in the body
are available in the program state instead. Inside a loop, Invariant() specifies which assertion
holds before the entry of the loop, during each iteration and after the loop exits. This means,
the invariant must be true on entry, that is before the first iteration of the loop.
Then it should be preserved from one iteration to the next. And finally, it must be true after
the loop exits. Permissions from/to outer contexts are not transferred to the loop,
unless specified in the invariant.

An example {unverified program + verification error, verified program} for
an unrelated ‘is_positive‘ predicate is shown below:
@Predicate
def is_positive(c: Cell) -> bool:
    return Acc(c.value) and c.value > 0

{ommited example: increment}

def compare(c1: Cell, c2: Cell) -> bool:
    """Returns True if the value in c1 is greater than value in c2"""
    if c1.value > c2.value:
        return True
    return False
Conditional statement might fail. There might be insufficient permission to access c1.value.
at line 3.7

def compare(c1: Cell, c2: Cell) -> bool:
    """Returns True if the value in c1 is greater than value in c2"""
    Requires(is_positive(c1) and is_positive(c2))
    Ensures(is_positive(c1) and is_positive(c2))
    if Unfolding(is_positive(c1), c1.value) > Unfolding(is_positive(c2), c2.value):
        return True
    return False

Equivalently, we can use Unfold and Fold instead of Unfolding, but need to make sure
to Fold the predicate in both cases so that the post condition is satisfied in both cases:
def compare(c1: Cell, c2: Cell) -> bool:
    """Returns True if the value in c1 is greater than value in c2"""
    Requires(is_positive(c1) and is_positive(c2))
    Ensures(is_positive(c1) and is_positive(c2))
    Unfold(is_positive(c1))
    Unfold(is_positive(c2))
    if c1.value > c2.value:
        Fold(is_positive(c1))
        Fold(is_positive(c2))
        return True
    Fold(is_positive(c1))
    Fold(is_positive(c2))
    return False
```

Bibliography

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[3] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In *International Conference on Computer Aided Verification*, 2018.

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

[5] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Inter-*

pretation (VMCAI), volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[6] OpenAI. Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering, Accessed: 2024-04-01.

[7] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[8] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[9] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. 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.

[10] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), may 2012.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[12] Patrick von Platen. How to generate. https://huggingface.co/blog/how-to-generate, Accessed: 2024-04-01.

[13] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[14] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification, 2024.

[15] 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.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

◉ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

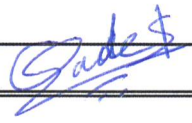
Large Language Models for Verified Programs

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Zade	Omkar

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 2024-04-10	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard