Synthetic Programming Elicitation for Text-to-Code in Very Low-Resource Programming and Formal Languages

Federico Mora[1] **Justin Wong**[1] **Haley Lepe**[2] **Sahil Bhatia**[1] **Karim Elmaaroufi**[1]
George Varghese[3] **Joseph E. González**[1] **Elizabeth Polgreen**[4] **Sanjit A. Seshia**[1]
[1]UC Berkeley [2]Stanford University [3]UCLA [4]University of Edinburgh
{fmora, justin.wong, sahilbhatia, k.e, jegonzal, sseshia}@berkeley.edu
halepe@stanford.edu, varghese@cs.ucla.edu, elizabeth.polgreen@ed.ac.uk

Abstract

Recent advances in large language models (LLMs) for code applications have demonstrated remarkable zero-shot fluency and instruction following on challenging code related tasks ranging from test case generation to self-repair. Unsurprisingly, however, models struggle to compose syntactically valid programs in programming languages unrepresented in pre-training, referred to as very low-resource Programming Languages (VLPLs). VLPLs appear in crucial settings, including domain-specific languages for internal tools, tool-chains for legacy languages, and formal verification frameworks. Inspired by a technique called natural programming elicitation, we propose designing an intermediate language that LLMs "naturally" know how to use and which can be automatically compiled to a target VLPL. When LLMs generate code that lies outside of this intermediate language, we use compiler techniques to repair the code into programs in the intermediate language. Overall, we introduce *synthetic programming elicitation and compilation* (SPEAC), an approach that enables LLMs to generate syntactically valid code even for VLPLs. We empirically evaluate the performance of SPEAC in a case study for the UCLID5 formal verification language and find that, compared to existing retrieval and fine-tuning baselines, SPEAC produces syntactically correct programs more frequently and without sacrificing semantic correctness.

1 Introduction

Large language models (LLMs) have demonstrated an exceptional ability to generate code from natural language prompts for popular programming languages, like Python and Java [13]. Unfortunately, these same language models struggle to generate code for low-resource programming languages, like many domain-specific languages (e.g., CUDA [40]). These challenges are even more pronounced for very low-resource programming languages (VLPLs) (e.g., formal verification languages like UCLID5 [34, 37]). Existing work has attempted to remedy this issue through prompting, constrained decoding, and fine-tuning strategies. Unfortunately, these approaches fail to capture the intricacies of real VLPLs and so success remains limited. To see why, consider the following three exemplars.

First, Wang at el. [42] include context-free grammars in text-to-code prompts to guide LLMs toward syntactically correct answers. This approach works well for simple languages but cannot capture many programming languages that are context-sensitive. Second, Agrawal et al. [1] use static analysis techniques to reject tokens that lead to syntactically incorrect output programs. This technique can go beyond context-free languages but assumes a linear programming process. Unfortunately, it is well known that the root cause of a programming error need not surface as it is written [32], necessitating

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

backtracking and a nonlinear programming process. Third, Cassano et al. [10] translate training data from high resource languages to low resource languages and then use this new data to fine-tune models. This approach is restricted to languages where the LLM is able to translate to the language reliably but unable to generate code from natural language. Further, this approach makes the overly restrictive assumption that the target low-resource language is general purpose: e.g., we cannot translate arbitrary Java programs to CUDA.

In this paper, we propose a text-to-code approach that is fundamentally different (and complementary) to prompting, decoding, and fine-tuning strategies. The first key idea behind our approach comes from natural programming elicitation, a kind of study that helps programming language designers understand how programmers "naturally" approach problems from a given programming domain [29, 11]. Programming language designers use the results of these studies to create languages that are aligned with the expectations of users, leading to less programming friction and more effective developers. We borrow this idea for the setting where LLMs are the "users" of programming languages. Akin to uncovering what human users find "natural" for a given domain, we uncover what LLMs find "natural." Specifically, our first insight is to embrace LLM's tendencies and design an intermediate language that aligns with these LLM expectations.

The second key idea in our approach is that program analyses and repair that are overly aggressive for human users may be suitable for LLM "users." For example, in UCLID5, all variables have to be declared and statically typed: an assignment like `x = 0;` would require a corresponding declaration like `var x: integer;`. But, if an LLM generates code that had an assignment without a declaration, instead of crashing, one could automatically "repair" the program and output the result.

We use these two ideas to define a new text-to-code approach called *synthetic programming elicitation and compilation* (SPEAC, pronounced "speak"). Specifically, for a target VLPL T, we use synthetic programming elicitation to select an intermediate language P (the "parent" language) and define a subset of the language C (the "child" language). The language P should be one that LLMs are good at generating (e.g. Python); the language C should be easy to compile to the target VLPL T. Our approach takes P, C, and a compiler from C to T, and produces a text-to-code pipeline for the VLPL T. This pipeline uses deductive techniques to automatically repair programs generated by LLMs that are in P but not in C. When these deductive techniques are unable to fully repair a program, we insert a "hole" and ask an LLM to finish the repair, repeating as necessary.

We demonstrate the effectiveness of this idea by implementing a prototype, called Eudoxus, that targets UCLID5 [34, 37], a language used for formal modeling and verification of state transition systems. UCLID5 has code examples numbering in the hundreds rather than thousands or millions. Furthermore, UCLID5 programs rely heavily on the notion of a transition system, which is not frequently found in other programming languages. As such, state-of-the-art LLMs are unable to generate any useful UCLID5 code out-of-the-box (see §5.1). In our case study, we use Python as the parent language P and a subset of Python as the child language C, and improve the performance of LLM code generation for UCLID5.

Overall, we make the following contributions: 1) We present SPEAC, a novel method for generating syntactically correct code from LLMs in very low resource programming languages; 2) We implement this method for the UCLID5 verification language; and 3) We demonstrate substantial improvement with SPEAC in syntactic correctness, producing parsable code in UCLID5 84.8% of the time compared to 9.1% by gpt-3.5-turbo fine-tuning and 12.1% by gpt-4-turbo in-context learning.

2 Related Work

LLMs for Code Generation. Modern language models perform exceptionally well on natural language to code generation tasks. For example, proprietary models like GPT-4 [31], the Gemini [41] and Claude 3 Families [1], and open-source models such as Code-Llama [35] and Deepseek-Coder [20] have achieved impressive success on benchmarks such as HumanEval [13], Mostly Basic Python Problems (MBPP) [3], and LiveCodeBench [22]. However, LLMs perform better on popular programming languages that are well represented in training sets (e.g., Python) than less popular programming languages (e.g., Bash) [20, 26, 33]. This effect is even more pronounced for very low resource programming languages, like UCLID5, as shown in §6.

[1] https://www.anthropic.com/news/claude-3-family

Model a state machine describing a traffic light at a pedestrian crosswalk. This is a time triggered machine that assumes it will react once per second. It starts in the red state and counts 60 seconds with the help of the variable count. It then transitions to green, where it will remain until the pure input pedestrian is present. That input is generated by some other subsystem that detects when a pedestrian is present, and should be modelled as nondeterministic. When pedestrian is present, the machine transitions to yellow if it has been green for at least 60 seconds. Otherwise, it transitions to pending, where it stays for the remainder of the 60 second interval. This ensures that once the light goes green, it stays green for at least 60 seconds. At the end of 60 seconds, it will transition to yellow ...

(a)

```
module TrafficLight {                                      1
  var sigG, sigR, sigY: boolean;                           2
  var count, state: integer;                               3
  var pedestrian: boolean;                                 4
  init {                                                   5
    sigG = false; sigY = false; sigR = true;               6
    state = 0; count = 0; pedestrian = false; }            7
  procedure step()                                         8
    modifies sigG; modifies sigY; modifies sigR;           9
    modifies count; modifies state; {                      10
    if (state == 0) {                                      11
      sigG = false; sigY = false; sigR = true;             12
      count = 0;                                           13
      if (count < 60) { count = (count + 1); }             14
      else { state = 1;}}                                  15
    else { if (state == 1) {                               16
      sigR = false; sigY = false; sigG = true;             17
      if (pedestrian) {                                    18
        if (count >= 60) { state = 2; }                    19
        else { state = 3; }}}                              20
...                                                        21
```

(b)

Figure 1: Partial task description from Lee and Seshia [23] (a) and partial output of Eudoxus in UCLID5 (b). We interpret `sigG, sigR, sigY` to represent green, red, and yellow light signals, respectively. The procedure `step` captures the transition relation of the state machine. `state` appears to be a bookkeeping variable that is used to track the cases in the task, and `count` represents a timer.

Training Free Approaches for Low-Resource Programming Languages. In constrained decoding, syntactically incorrect code is avoided by rejecting impossible prefixes, without producing the full code [18, 1, 36, 27]. In the context of autoregressive LLMs that naturally produce code left to right, it remains an open problem how to best include the inductive bias of a grammar. Bhatia et al. [8] use LLMs to rewrite existing code into domain-specific languages and prove that the translation is correct. Misu et al. [28] use retrieval-augmented generation to write Dafny code, another low-resource language for verification [25]. Unlike these works, we focus on very low resourced languages (VLPLs), which have far fewer training examples in the public domain. Our approach is most similar to techniques that allow LLMs to hallucinate but iteratively repair errors [38, 14, 30]. For example, Elmaaroufi et al. [16] use a mixture of prompting and compiler feedback to generate and iteratively repair Scenic code—a probabilistic VLPL that looks like Python [17].

Training LLMs for Low-Resource Programming Languages. Recent work has considered augmenting LLMs with support for low-resource programming languages [10, 12, 20]. Chen et al. [12] show that, on smaller 125M parameter encoder-only models, fine-tuning on adjacent languages improves the monolingual performance coding tasks. Synthetic fine-tuning datasets curated and cleaned by LLMs have shown promise for programming tasks. For example, Cassano et al.[10] targets low-resource programming languages (e.g., Julia), using an LLM to translate code examples from high-resource languages to the low-resource language. This process is promising for cases where the language model already has a baseline knowledge necessary to translate to the low-resource language and the target language is general purpose, which is not always the case for VLPLs.

3 Overview and Running Example

Given a natural language description of a programming task and a target programming language T, the text-to-code problem is to generate a program $t \in T$ that satisfies the task specification. Fig. 1 shows a real input-output pair generated by an instance of our approach targeting the UCLID5 programming language. Specifically, Fig. 1a shows a task extracted from Lee and Seshia [23], and Fig. 1b shows the output corresponding to that task using a prototype implementation of our approach. Fig. 1b passes all compiler checks but has a subtle semantic mistake on line 4.

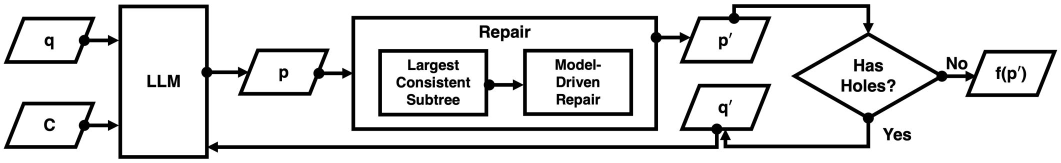

Figure 2: The SPEAC workflow. Users input q, a task in natural language, and C, a description of the intermediate language. The LLM takes these inputs and generates p, a program in P. We use formal techniques to repair p and produce p', a program in C that possibly contains holes. If p' does not contain holes, SPEAC applies f, a compiler from C to the target language, T, and returns the result. Otherwise, SPEAC generates a new prompt, q', and repeats by asking the LLM to fill in the holes.

Fig. 2 shows the workflow that generates models based on task descriptions as shown in Fig. 1b. The workflow is parameterized by an LLM, L (e.g., gpt-3.5-turbo-0125); a target language, T (e.g., UCLID5); a parent language, P (e.g., Python); a child language, $C \subset P$ (e.g., a subset of Python); and a compiler, f, from C to T (e.g., a syntax-directed translation from the subset of Python to UCLID5). Given an input task, we create a prompt q that asks the LLM L to generate code, $p \in C$, which satisfies the task description, q. The second step of the workflow is to repair p. If there is nothing wrong with p, or p can be fixed using formal techniques described in §5.3 and §5.4, then repairing will generate a new, complete program p' and return $f(p')$ (i.e., a program in the target language, like Fig. 1b). Frequently, however, repairing will generate a partial program containing "holes" (denoted "`??`"). For example, Fig. 5b shows the first p generated for the task in Fig. 1a and Fig. 5a shows the corresponding partial program p' that was automatically generated using our formal techniques. Programs with holes cannot be compiled to the target language, so the third step of the workflow is ask the LLM to complete the partial program p', generating a new p. We use the template in Fig. 4b to generate the LLM prompt. Fig. 6a shows the output generated by gpt-3.5-turbo-0125 when asked to repair the partial program in Fig. 5a. This program is still incorrect, but, it is now close enough that we can automatically repair it to a complete program without holes. Fig. 6b shows the relevant portion of that complete program. This final, complete program is directly translated to the output in Fig. 1b. §5 elaborates on each of the workflow components.

To understand the subtle mistake in Fig. 1b one needs some understanding of UCLID5 [34, 37]. UCLID5 is a *verification language* used to model hardware and software systems as "state machine" like transition systems and to automatically check if the transition system does indeed satisfy a formal logic specification. UCLID5 transition systems primarily consist of a state space given by a set of variable declarations (e.g., lines 2-4 in Fig. 1b), an initialization block that represents a set of acceptable start states (e.g., lines 5-7 in Fig. 1b), and a transition relation block that defines how the transition system moves from one state to the next state (e.g., code starting at line 8 in Fig. 1b). The `var` keyword is used to declare variables that are internal to the transition system in question while the `input` keyword is used to declare read-only variables that are outside the control of the transition system in question. Fig. 1b passes all compiler checks but has a subtle semantic mistake on line 4: `var pedestrian: boolean;` should be `input pedestrian: boolean;` because the presence of a pedestrian should not be controlled by the traffic light transition system. When manually assessing correctness in §6, this subtle mistake would prevent us from marking this example as fully correct.

4 Background

In this section we provide the necessary technical background to understand the formal techniques that are used in our approach and are described in §5.3 and §5.4.

4.1 Algebraic Data Types and Abstract Syntax Trees

Algebraic data types (ADTs) are a representation of finite trees that are common in functional programming languages. We provide an informal definition of ADTs and point the interested reader to Barrett et al. [6] for a more formal treatment.

An ADT consists of a set of constructors (node names), selectors (directed edge names), and testers (predicates). Each constructor has a fixed set of selectors associated with it (possible edges out).

```
from FormalVerificationLibrary import Module # GTCODE 1: imports a class from the hypothetical library
m = Module("myModule") # GTCODE 2: creates an instance of an imported class
print(m) # GTCODE 3: uses a dunder method (in this case __str__) of an imported class
```

Figure 3: Hypothetical LLM output with grounded theory codes as comments. Codes are determined and assigned manually.

Each selector has an associated type (each edge can only point to a fixed set of node names). Every constructor is associated with exactly one unique tester: that tester returns true on a given tree *iff* the root of the tree is labeled with the corresponding constructor. Every instance of an ADT (a particular finite tree built from those node and edge names) must be acyclic.

Abstract syntax trees (ASTs) are instances of ADTs—i.e., ASTs are concrete finite trees—that represent programs. Every programming language has a corresponding ADT that represents a superset of all possible programs in that programming language. Some instances of a languages's ADT will not correspond to valid programs, e.g., if they do not additionally type check.

4.2 Satisfiability Modulo Theories and Weighted Maximum Satisfiability

Satisfiability Modulo Theories (SMT) [7] is a class of problems generalizing Boolean satisfiability (SAT) to first-order logics with additional background logical theories. We give an informal presentation of satisfiability modulo theories (SMT) that focuses on only the theory of ADTs and covers weighted maximum satisfiability (MAX-SMT). Further details may be found in a book chapter [7].

Let Γ be a set of ADTs and let V be a set of typed variables (pairs of names and types). For simplicity we assume that variable types are exactly elements of Γ. In reality, variables can also have function types (e.g., $V \doteq \{(z, \texttt{Bool}), (f, \texttt{Bool} \mapsto \texttt{Bool})\}$ would be fine). An *atomic formula* is an equation or the application of a single tester over V (e.g., $z = \texttt{True}$ and $\texttt{is\_And}(z)$ are both atomic formulas). A *theory literal* is an atomic formula or its negation. A *clause* is a set of theory literals. A *conjunctive normal form (CNF) formula*, or *formula* for short, is a set of clauses. For example, if $\Gamma \doteq \{\texttt{Bool}\}$ and $V \doteq \{(z, \texttt{Bool})\}$, then $\{\{z = \texttt{True}\}, \{\texttt{is\_And}(z)\}\}$ is a formula. An *interpretation* is a mapping from variables to elements of their corresponding type. For example, $I(x) \doteq \texttt{True}$ if $x = z$ and $I(x) \doteq \texttt{False}$ otherwise, is an interpretation. Interpretations are extended to atomic formulas in the natural way When an atomic formula ϕ evaluates to true under an interpretation I, we say that I *satisfies* ϕ and write $I \models \phi$. We extend the notion of satisfiability to literals, clauses, and formulas in the natural way and reuse the same notation. The satisfiability modulo theories problem is to determine if, for a given formula ϕ, there exists an interpretation I such that $I \models \phi$. When such an I exists we say that ϕ is *satisfiable* (**sat**). When no such I exists, we say that ϕ is *unsatisfiable* (**unsat**). For example, $\{\{z = \texttt{True}\}, \{\texttt{is\_And}(z)\}\}$ is **unsat**.

The maximum satisfiability problem is, for a given (CNF) formula ϕ, to determine the largest subset of ϕ that is **sat** (solvers aim to satisfy as many clauses as possible). The weighted maximum satisfiability problem (MAX-SMT) is a variation with two differences. First, some clauses can be "hard"—meaning they must be satisfied. Second, every "soft" clause (any clause that is not "hard") has an associated weight. The problem is then to determine subset of ϕ that maximizes the sum of weights while being **sat** and containing all "hard" clauses.

5 Approach

In this section we describe the SPEAC approach. We begin with how to select P and C (§5.1). We then describe the promoting we use to interface with LLMs (§5.2) and two formal techniques at the heart of our automated repair (§5.3 and §5.4).

5.1 Synthetic Programming Elicitation

In this section, we present *synthetic programming elicitation* as a kind of programming study—where LLMs are the subject—that follows three steps. The results of these studies are used to select P and C. We call this process synthetic programming elicitation since it is inspired heavily by natural programming elicitation [29]. To make this section more concrete, for each step of the study, we describe the specific steps we followed for targeting UCLID5.

First Step: Setup. First prepare LLM subjects, select a target language, and collect or construct a set of programming tasks. In our case, our target language is UCLID5, our LLM subjects are gpt-4-turbo-2024-04-09 and gpt-3.5-turbo-0125, and our programming tasks are a set of regression tests written by UCLID5 developers with corresponding natural language descriptions that were written by hand (see §6 for more details on all the benchmarks).

The second part of the study setup is to create multiple kinds of specialized prompts. The first will ask for the output to be in the target language directly. Subsequent prompts will ask for the output to use an imaginary API in a popular host language, like Python or Java. For example, for UCLID5, we generated prompts that look like the following.

1. "Write a UCLID5 model for a system that counts the number of times the temperature exceeds a threshold [. . .]"

2. " FormalVerificationLibrary is a Python library for generating verification models [. . .]. Use the FormalVerificationLibrary to model a system that counts the number of times the temperature exceeds a threshold [. . .]"

Second Step: Execute. Now we execute every LLM subject on every prompt and collect the outputs. Each query should be executed independently.

Third Step: Analyze and Design. Finally, we analyze the collected LLM outputs and select P and C based on the analysis. To do this analysis, we follow a lightweight version of grounded theory [19]. See e.g., Stol et al. [39] for general guidelines on how to use grounded theory in software engineering research. See e.g., Barke et al. [5] for a specific, detailed case-study. Synthetic programming elicitation studies are inspired by and borrow many of the methods from those works. Our studies are categorically different, however, because we are not working with humans. We cannot interact with our participants in the same way, e.g., we cannot conduct semi-structured interviews. The goal of our studies is to design an intermediate language that more closely matches the code that LLMs tend to generate when compared to our desired target language.

The first step of our lightweight grounded theory analysis is to "code" the outputs generated by the LLMs. In grounded theory parlance, "code" is akin to manually labeling parts of the generated outputs. For example, Fig. 3 shows a hypothetical LLM output for our running example along with grounded theory codes as comments. The second step is to group codes into concepts. Concepts are slightly more abstract than codes. For example, a concept that may emerge from Fig. 3 is the group of codes 1, 2, and 3. The corresponding concept could be "using a class from the hypothetical library." In the third step, we group concepts into categories. For example, we may group concepts related to the object oriented programming paradigm as one category. Finally, we select a P and C that are consistent with the final categories we observed across multiple prompts.

In our study, we found that, when asked to write UCLID5 code without any special prompting or training, no LLM was able to produce code that parses (660 attempts: 10 samples per LLM per benchmark, 33 benchmarks, two LLMs). Worse still, the code generated by LLMs was inconsistent, with each LLM giving different outputs that resemble different programming languages at different times. When asked to write Python code that used a non-existent formal verification API, however, the LLM outputs were more consistent. Therefore, we selected Python as our parent language, P.

Specifically, the Python code was more consistent because LLM outputs fell into three broad categories, which we call "from-scratch," "procedural," and "object-oriented," respectively. Programs in the "from-scratch" category used existing APIs (e.g., the Z3 solver API [15]) to re-implement what UCLID5 does, e.g., to manually model transition systems. This was the smallest significant category. Programs in the "procedural" category imported a class from the hypothetical API, created an instance of it, and then called methods from the class to declare variables, assert specifications and so on. Programs in the "object-oriented" category imported a class from the hypothetical API and extended it, including methods that correspond closely to parts of UCLID5 code. For example, these extended classes often included a method that represented a transition relation—a critical component of UCLID5 programs and of transition systems generally.

After analyzing the outputs, we concluded that it would be easiest to translate Python programs from the "object-oriented" category to UCLID5. For example, we observed that methods which represent

```
1  Write [PARENT LANGUAGE] code to complete the
   following task.
2  > [TASK GOES HERE]
3  Reply with your code inside one unique code block
4  [DESCRIBE CHILD LANGUAGE]
5  I can definitely do that! Here is the code:
6  ‘‘‘
```

(a) Template for prompt q in Fig. 2

```
Fix the following [PARENT LANGUAGE] code by        1
replacing every occurrence of ‘??‘ with the correct
code.
[CODE WITH HOLES GOES HERE]                        2
Make sure that your code completes the following   3
task.
[LINES 2–6 OF (a)]                                 4
```

(b) Template for prompt q' in Fig. 2

Figure 4: Partial SPEAC prompt templates for first generation (a) and hole filling (b).

transition relations used a limited set of names, and that methods themselves could be compiled to UCLID5 rather directly. Therefore we defined a subset of Python that matches the "object-oriented" category and used that as our child language, C. Essentially, C is an abstract class that the LLM must extend. For example, the abstract class includes an abstract method called "next" that corresponds to the transition relation. Fig. 6b shows a portion of an example of a Python program in C and Fig. 1b shows a portion of its corresponding UCLID5 program.

5.2 (Minimal) Prompting

After our synthetic programming elicitation study—once P and C have been selected—we use minimal prompting to interface with LLMs. For example, for UCLID5, we use the prompt template in Fig 4a to create initial programs and the prompt template in Fig. 4b to fill in holes. More advanced prompting techniques are complementary and could help here. However, for this work, we used minimal prompting so as to independently evaluate our contribution.

5.3 Largest Consistent Subtree

Given a program p in P, but not in C, our aim is to remove as little as possible from p to bring it into the language C. That is, given the AST for p, we wish to generate the largest subtree of the AST, possibly containing holes, that is not rejected by the static checks of the language C. For example, the code in Fig. 5b contains some errors, including mismatches between variable and constant types (UCLID5 does not automatically cast constants and so the line self .count = 1 is interpreted as assigning the integer literal 1 to the bitvector variable self .count, which is a type error).

We find the largest consistent subtree in three steps. First, we use an error tolerant parser to build an AST, A, for the language P. Second, beginning at the root of A, we recursively descend and prune anything that cannot exist in an AST for the language C, placing holes wherever they are needed. This is an aggressive program slice, similar to Akhundov et al. [2], who give a new AST A'. A' may or may not pass compiler checks, like type checking, or be semantically correct.

The third step finds the minimal number of AST nodes that need to be replaced with holes such that all static checks of the language C pass, using the approach of Pavlinovic et al. [32]. Specifically, for every static check of the language, for every node of the AST, we generate a set of soft clauses. Let S be the the union of all the generated clauses, and let S' be the largest subset of clauses that is **sat** (the output of a MAX-SMT solver). Any clause, c, that appears in S but does not appear in S' represents a bug: if we add c to S', then we will get an **unsat** set of clauses. So, for every such c, we replace the AST node that generated c with a hole.

For example, in UCLID5, the two statements `var x:  bv32;` and `x = 0;` would produce a set of clauses corresponding to the assertions that (a) `x` is a bitvector and that (b) `x` is an integer (since `0` is an integer). These assertions, together, are **unsat**, so MAX-SMT would return a set of clauses that does not include both (a) and (b). One solution would be to remove the bitvector type, producing the partial, but consistent, two statements `var x:  ??;` and `x = 0;`.

We can bias the solver to include one or the other by adjusting the weights on the corresponding clauses. In practice, we assign clauses weights that are proportional to the depth of the corresponding node in the AST. This ensures that edits are closer to the leafs of the AST but it breaks syntactic minimality guarantees. In the case where weights are uniform and every node corresponds to exactly one clause, the MAX-SMT approach will make the minimal edit to the AST.

```
 1 class TrafficLight(Module):
 2     def types(self):
 3         self.state_t = BitVector(2)
 4     def locals(self):
 5         self.state = BitVector(2)
 6         self.count = BitVector(6)
 7         self.pedestrian = Boolean()
 8         self.sigG = Boolean()
 9         self.sigY = Boolean()
10         self.sigR = Boolean()
11     def inputs(self):
12         self.pedestrian = Boolean()
13     def outputs(self):
14         self.sigG = Boolean()
15         self.sigY = Boolean()
16         self.sigR = Boolean()
17     def init(self):
18         self.state = BitVector(2)
19         self.count = BitVector(6)
20         self.pedestrian = Boolean()
21         self.sigG = Boolean()
22         self.sigY = Boolean()
23         self.sigR = Boolean()
```

```
24 def next(self):
25     if self.state == 0:
26         self.sigG = False
27         self.sigY = False
28         self.sigR = True
29         self.count = 0
30         if self.count < 60:
31             self.count += 1
32         else:
33             self.state = 1
34     ...
35     elif self.state == 3:
36         self.sigG = False
37         self.sigY = False
38         self.sigR = False
39         if self.count < 60:
40             self.count += 1
41         else:
42             self.state = 0
```

(a)

```
class TrafficLight(Module):              1
    def locals(self):                    2
        self.state = int                 3
        self.count = int                 4
        self.pedestrian = bool           5
    def outputs(self):                   6
        self.sigG = bool                 7
        self.sigY = bool                 8
        self.sigR = bool                 9
    def init(self):                     10
        self.state = ??                 11
        self.count = ??                 12
        self.pedestrian = ??            13
        self.sigG = ??                  14
        self.sigY = ??                  15
        self.sigR = ??                  16
    def next(self):                     17
        if self.state == 0:             18
            self.sigG = False           19
            self.sigY = False           20
            self.sigR = True            21
            self.count = 0              22
            ...                         23
```

(b)

Figure 5: Partial response for task in Fig. 1a using gpt-3.5-turbo-0125 (a) and partial first repair (b). Line 5 in (a) declares `state` as a local variable of bit-vector type; lines 25, 33, and 42 use `state` as an integer. Line 3 in (b) repairs the type of `state` to be an integer.

5.4 Model-Based Repair

Once we have a satisfiable set of clauses, we can generate a corresponding satisfying model and use the model to repair the partial program. This is where our work most differs from Pavlinovic et al. [32]. For example, the partial program `var x: ??;` from above would correspond to the set of clauses `is_Integer(??)`. This clause would be satisfied by setting `??` to `integer` and so we can repair the partial program to be `var x: integer;`.

Fig. 6 shows one buggy AST (Fig.6a) and the corresponding fix (Fig. 6b). For example, the variable `count` is declared as an integer in the repaired program because it is used as an integer in the buggy program. For repairs that cannot be automatically resolved by the MAX-SMT model, we use the LLM to generate code to replace the AST holes, as shown in Fig. 2.

6 Evaluation

In this section, we implement a prototype of SPEAC for UCLID5, called Eudoxus, and use it to evaluate the performance of SPEAC across three research questions. RQ1: Does Eudoxus perform better than LLM baselines on syntactic correctness (passing compiler checks)? RQ2: Does Eudoxus perform better on semantic correctness? RQ3: Does MAX-SMT cost too much computation time? Eudoxus uses the P and C described in §5 and is implemented in Python.[2] We use tree-sitter to parse and search partial programs, and Z3 [15, 9] to solve MAX-SMT queries. We allow Eudoxus to repeat the repair loop five times. All MAX-SMT queries are solved locally on a 2.3 GHz Quad-Core Intel Core i7 with 32 GB of RAM. All LLM calls are made through the OpenAI Python API.

We compare Eudoxus against three LLM baselines: few-shot, self-repair, and fine-tuning. All using gpt-3.5-turbo-0125 (GPT3t) and gpt-4-turbo-2024-04-09 (GPT4t). **Few-shot with and without Chain-of-Thought.** For the few-shot baseline, we provide examples taken from the UCLID5 GitHub repository to the LLM in context. We tried with one in context example, and then again with three in context examples. We also combine few-shot prompting with chain-of-thought [44]. **Fine-tuning.**

[2] Available at: `https://github.com/FedericoAureliano/eudoxus`

```
1   class TrafficLight(Module):
2       def locals(self):
3           self.state = 0
4           self.count = 0
5           self.pedestrian = False
6       def outputs(self):
7           self.sigG = False
8           self.sigY = False
9           self.sigR = False
10      def init(self):
11          self.state = 0
12          self.count = 0
13          self.pedestrian = False
14          self.sigG = False
15          self.sigY = False
16          self.sigR = True
17      ...
```

(a)

```
class TrafficLight(Module):                 1
    def locals(self):                       2
        self.count = int                    3
        self.pedestrian = bool              4
        self.sigG = bool                    5
        self.sigR = bool                    6
        self.sigY = bool                    7
        self.state = int                    8
                                            9
    def init(self):                         10
        self.state = 0                      11
        self.count = 0                      12
        self.pedestrian = False             13
        self.sigG = False                   14
        self.sigY = False                   15
        self.sigR = True                    16
    ...                                     17
```

(b)

Figure 6: Partial response for Fig. 5a using gpt-3.5-turbo-0125 (a) and partial second repair (b). The `outputs` method should declare variables and their types but on lines 6-9 of (a) it instead assigns variables to values (i.e., `False` instead of `bool`). The repair in (b) declares these variables with the appropriate types on lines 5-7.

As with many VLPL, there are very limited number of natural language to UCLID5 examples in the public domain, and certainly not enough for fine-tuning. We do, however, have access to 317 regression tests from the open-source code base. For the purposes of fine-tuning, we use self-instruct [43] to first ask the model to summarize the UCLID5 regression tests in natural language and then provide this as the natural language description during fine-tuning. We fine-tune GPT3t for 284 steps with learning-rate multiplier 0.16. **Self-repair.** We gave the LLMs compiler feedback in a self-repair loop starting from a zero-shot prompt. We use five iterations to be consistent with Eudoxus.

We use three sets of UCLID5 benchmarks. The first set is a curated set of 33 regression tests with handwritten natural language descriptions. These tests are designed to cover a broad range of UCLID5 features and were used for the synthetic programming elicitation study in § 5.1. The second set is a set of 317 regression tests without descriptions taken directly from the UCLID5 GitHub repository. These tests were used for fine-tuning our baseline model, as described above. This set is representative of the quantity and quality of data that a user may have available for a VLPL. The third and final set is a set of 33 exercises and examples taken from three textbooks [4, 24, 21]. These benchmarks cover formal modeling of concurrent systems, linear-time properties, model checking and systems with discrete dynamics. We used this final set for the end-to-end evaluation below.

6.1 Results

We run two variations of Eudoxus (one using GPT3t, one using GPT4t) and 11 variations of the baselines on all 33 textbook benchmarks. We report the fraction of outputs that pass all compiler checks ("parse," for short) and semantic correctness over all 11 approaches in Table 1. Semantic correctness is manually assessed by one author using a combination of manual reasoning and handwritten specifications for all programs that compile. Correctness is rated on a scale from from $1 - 5$, where 1 is entirely wrong, and 4 indicates that the model is correct with only a few minor errors (e.g., the example output described in §3 and Fig. 5). Intuitively, any score of ≥ 4 indicates that we believe the output would be useful to text-to-code users.

RQ1: Syntactic Correctness. Eudoxus outperforms all baselines in terms of compiler checks (see "Parse Rate" in Table 1), passing all compiler checks on 78% of benchmarks. There are four benchmarks on which Eudoxus hits the iteration limit and fails to produce a result, and three benchmarks with small syntax errors due to bugs in our Eudoxus implementation (e.g., failing to extract code snippets from LLM outputs or printing UCLID5 code incorrectly). In contrast, we find that GPT3t and GPT4t rarely produce UCLID5 models that even parse. The best results for GPT3t

		Semantic Score				
	Parse Rate	1	2	3	4	5
Eudoxus (GPT4t)	24/33	1/24	1/24	3/24	8/24	11/24
Eudoxus (GPT3t)	28/33	3/28	6/28	5/28	5/28	9/28
Fine-tuned GPT3t	2/33	0	2/2	0	0	0
One-shot with COT (GPT4t)	1/33	0	0	0	1/1	0
One-shot with COT (GPT3t)	0	-	-	-	-	-
One-shot (GPT4t)	0	-	-	-	-	-
One-shot (GPT3t)	0	-	-	-	-	-
Three-shot with COT (GPT4t)	3/33	0	0	0	2/3	1/3
Three-shot with COT (GPT3t)	1/33	0	0	0	0	1/1
Three-shot (GPT4t)	4/33	0	0	0	3/4	1/4
Three-shot (GPT3t)	2/33	0	0	1/2	0	1/2
Self-repair (GPT4t)	0	-	-	-	-	-
Self-repair (GPT3t)	0	-	-	-	-	-

Table 1: Eudoxus compared to baselines. We report the semantic score over all correctly parsed models. 1 is completely wrong; 5 is fully correct. Eudoxus is limited to five LLM calls per benchmark, and four benchmarks hit this limit.

come from fine-tuning, but it is only able to produce two programs that parse. The best results for GPT4t come from three-shot prompting, but it is only able to produce four programs that parse. Given this data, we answer RQ1 in the affirmative: Eudoxus perform better than standard LLM baselines on syntactic correctness. Even more interesting, the two versions of Eudoxus generated programs that passed all compiler checks for 30/33 unique problems; the 11 versions of the baselines together generated programs that passed all compiler checks for only 6/33 unique problems.

RQ2: Semantic Correctness. Every unique problem that the baselines perform well on—four unique problems where some baseline scored four or higher—was also solved by Eudoxus. These are likely the easiest problems in the test set. On the other hand, $33/52$ programs (24 unique problems) that pass compiler checks produced by Eudoxus scored four or higher. This data suggests that our approach does not negatively impact semantic performance, but it is difficult to draw conclusions since so few of the programs generated by the baselines pass compiler checks.

RQ3: MAX-SMT Performance. In terms of execution time, Eudoxus with GPT3t took an average of 0.9 seconds (SD 0.6) in repair steps and 7.2 seconds (SD 4.6) in LLM calls. Eudoxus with GPT4t took an average of 1.8 seconds (SD 2.0) in repair steps and 35.1 seconds (SD 24.7) in LLM calls. We conclude that, in terms of execution time, LLM calls are more expensive than our static repairs.

6.2 Threats to Validity

While our results are promising, there are three limitations to our evaluation. First, we only evaluated SPEAC on one VLPL, UCLID5. It remains to be seen if our results can be replicated across different VLPLs, especially those in different domains. Second, we only evaluated SPEAC with two LLMs. It is possible that our work will not generalize across different kinds of LLMs. Third, only one author evaluated the semantic correctness of programs generated by Eudoxus. While the main takeaway of our work is that we are able to generate syntactically correct programs where baselines cannot, it is possible that we have over or under-estimated semantic correctness.

7 Conclusion

We have presented a synthetic program elicitation and compilation method (SPEAC) that supports natural language to code generation for very low resource programming languages (VLPLs). The two key ideas behind SPEAC are (1) to design an interface that is "natural" for LLM "users" and (2) to use deductive techniques, which could be deemed too aggressive for human users, to automatically repair LLM outputs when possible. We implemented a prototype of SPEAC called Eudoxus that targets the UCLID5 VLPL and evaluated it on a set of 33 benchmarks from textbooks in the same domain as UCLID5. Eudoxus performs significantly better than standard LLM baselines on syntactic correctness without sacrificing semantic correctness.

Acknowledgments and Disclosure of Funding

We would like to thank Adwait Godbole, Ameesh Shah, Max Willsey, and the anonymous reviewers for their insightful feedback. We would like to thank Justin Lubin for pointing us to natural programming elicitation. Part of this work was done during the Transfer-to-Excellence Summer Research Program at UC Berkeley in 2023 and part during UC Berkeley's CS 294-260: Declarative Program Analysis and Optimization in 2024. This work was supported in part by a Qualcomm Innovation Fellowship, a Royal Academy of Engineering Research Fellowship, DARPA Contract FA8750-23-C-0080 (ANSR), C3DTI, an Amazon Research Award, NSF grant 2303564, by Nissan and Toyota under the iCyPhy center, and by Intel under the Scalable Assurance program.

References

[1] Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram Rajamani. Monitor-guided decoding of code LMs with static analysis of repository context. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=qPUbKxKvXq`.

[2] Murad Akhundov, Federico Mora, Nick Feng, Vincent Hui, and Marsha Chechik. Verification by gambling on program slices. In *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings 19*, pages 266–282. Springer, 2021.

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

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[5] Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi: 10.1145/3586030. URL `https://doi.org/10.1145/3586030`.

[6] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[7] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 33, pages 1267–1329. IOS Press, second edition, 2021.

[8] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Verified code transpilation with llms, 2024. URL `https://arxiv.org/abs/2406.03003`.

[9] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz-an optimizing smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, pages 194–199. Springer, 2015.

[10] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Carolyn Jane Anderson, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code llms. *arXiv preprint arXiv:2308.09895*, 2023.

[11] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. PL and HCI: better together. *Commun. ACM*, 64(8):98–106, jul 2021. ISSN 0001-0782. doi: 10.1145/3469279. URL `https://doi.org/10.1145/3469279`.

[12] Fuxiang Chen, Fatemeh H Fard, David Lo, and Timofey Bryksin. On the transferability of pre-trained language models for low-resource programming languages. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 401–412, 2022.

[13] 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. *arXiv preprint arXiv:2107.03374*, 2021.

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

[15] 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.

[16] Karim Elmaaroufi, Devan Shanker, Ana Cismaru, Marcell Vazquez-Chanlatte, Alberto L. Sangiovanni-Vincentelli, Matei Zaharia, and Sanjit A. Seshia. ScenicNL: generating probabilistic scenario programs from natural language. In *Conference on Language Models (COLM)*, 2024.

[17] Daniel J Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: A language for scenario specification and data generation. *Machine Learning*, 112(10):3805–3849, 2023.

[18] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.674. URL `https://aclanthology.org/2023.emnlp-main.674`.

[19] Barney Glaser and Anselm Strauss. *Discovery of grounded theory: Strategies for qualitative research.* Routledge, 2017.

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

[21] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.).* Cambridge University Press, 2004.

[22] 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 arXiv:2403.07974*, 2024.

[23] Edward A. Lee and Sanjit A. Seshia. An introductory textbook on cyber-physical systems. In *WESE*, page 1. ACM, 2010.

[24] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach.* MIT Press, second edition edition, 2016. URL `http://leeseshia.org`.

[25] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[26] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, et al. Starcoder: may the source be with you!, 2023. URL `https://arxiv.org/abs/2305.06161`.

[27] Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. Constrained decoding for code language models via efficient left and right quotienting of context-sensitive grammars. *arXiv preprint arXiv:2402.17988*, 2024.

[28] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards ai-assisted synthesis of verified dafny methods. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3643763. URL `https://doi.org/10.1145/3643763`.

[29] Brad A. Myers, John F. Pane, and Amy J. Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, sep 2004. ISSN 0001-0782. doi: 10.1145/1015864.1015888. URL `https://doi.org/10.1145/1015864.1015888`.

[30] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2023.

[31] 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, , et al. Gpt-4 technical report, 2024.

[32] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. *SIGPLAN Not.*, 49(10):525–542, oct 2014. ISSN 0362-1340. doi: 10.1145/2714064.2660230. URL `https://doi.org/10.1145/2714064.2660230`.

[33] Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization, 2024. URL `https://arxiv.org/abs/2402.16694`.

[34] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. UCLID5: multi-modal formal modeling, verification, and synthesis. In *CAV (1)*, volume 13371 of *Lecture Notes in Computer Science*, pages 538–551. Springer, 2022.

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

[36] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*, 2021.

[37] Sanjit A. Seshia and Pramod Subramanyan. UCLID5: integrating modeling, verification, synthesis and learning. In *MEMOCODE*, pages 1–10. IEEE, 2018.

[38] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[39] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 120–131, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884833. URL `https://doi.org/10.1145/2884781.2884833`.

[40] Artur Tarassow. The potential of llms for coding with low-resource and domain-specific programming languages. *arXiv preprint arXiv:2307.13018*, 2023.

[41] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, et al. Gemini: A family of highly capable multimodal models, 2024. URL `https://arxiv.org/abs/2312.11805`.

[42] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[43] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pages

13484–13508, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.754. URL `https://aclanthology.org/2023.acl-long.754`.

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9781713871088.

A NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes] ,

   Justification: the claims are concretely supported by results in the evaluation

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: There is a limitation paragraph in the evaluation section

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA] .

   Justification: no theoretical results

   Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The SPEAC approach is described in sufficient detail to reproduce it. The UCLID5 language and verification tool is open source and publicly available, along with all the regression tests for fine-tuning. The textbooks and code are publicly available.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

Justification: In supplementary material.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer:[Yes]

   Justification: Experiments use openAI LLMs and report the model names and temperature settings used and training details for the baselines.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [NA]

   Justification: We do not make claims about statistical significance. We show data for a case study.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Hardware listed in Sec. 6.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification:

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The paper is about getting code to pass compiler checks.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We are not releasing any new models. The data we are using is not new, and is taken from publicly available textbooks and open source code repositories.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: UCLID5 is open source and licensed under 3-clause BSD, and cited accordingly. Examples taken from textbooks are cited accordingly.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: All code and documentation is publically available on the GitHub repository.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.