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

Federico Mora, Adwait Godbole
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
{fmora, adwait}@berkeley.edu

Abstract

Low-resource programming languages (LPLs) include domain-specific languages, less-used programming languages, old legacy languages, and languages used internal to tools and tool-chains. In this paper, we consider the text-to-code problem for the low end of LPLs, a class we term *very* low-resource programming languages (VLPLs). Specifically, we propose *synthetic programming elicitation and compilation* (SPEAK), an approach that lets users of VLPLs keep using their favorite languages and still get good text-to-code support from large language models (LLMs). We implement SPEAK by using an intermediate language (IL) to bridge the gap between what LLMs are good at generating and what users want to generate (code in a target VLPL). This IL must satisfy two conditions. First, LLMs must be good at generating code in the IL (e.g., Python). Second, there must exist a *well-defined* subset of this language that can easily be compiled to the target VLPL (e.g., Python regular expression strings can be directly translated to POSIX regular expressions). Given such an IL, SPEAK follows a four step pipeline for text-to-code generation. First, it asks an LLM to generate a program in the IL. Second, it uses the definition of the IL subset to automatically generate a *partial program*. Third, it completes the partial program with LLM calls, iterating as necessary. Finally, it compiles the complete IL program to the the target VLPL. We instantiate SPEAK in a case study for a verification language, and compare the performance of our generator to existing baselines. We find that SPEAK substantially improves the performance of text-to-code tools on both VLPLs.

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 [Chen et al., 2021]. Unfortunately, these same language models struggle to generate code for low resource programming languages, like many domain specific languages [Tarassow, 2023]. These challenges are even more pronounced for very low resource programming languages (VLPLs). Existing work has attempted to remedy this issue by focusing on prompting, decoding, and fine-tuning strategies. We present a thorough review of related work in Sec. 5 but first we highlight three exemplary examples, one from each category, that help explain the context of our work.

First, Wang et al. [2024] 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 most non-trivial programming languages, which are context-sensitive. Second, Agrawal et al. [2023] use program analysis techniques to avoid the generation of tokens that must lead to syntactically incorrect output programs. This technique can go beyond context-free languages but

Preprint. Under review.

assumes a linear programming process. Unfortunately, it is well known that many errors cannot be fixed by adding code, necessitating backtracking and a nonlinear programming process (see Sec. 3.1 for more details and an example). Third, Cassano et al. [2023] translate training data from high resource languages to low resource languages and then use this new data to fine-tune models. The problem with this approach is that it assumes that the target low-resource language is general purpose: e.g., we cannot translate arbitrary Java programs to Posix regular expressions.

In this paper, we propose a text-to-code approach that is fundamentally different to prompting, decoding, and fine-tuning strategies. The first key idea behind our approach is inspired by decades of work at the intersection of human-computer interaction and programming languages research [Chasins et al., 2021], especially the notion of *natural programming elicitation* [Myers et al., 2004]. Natural programming elicitation seeks to understand how programmers "naturally" approach problems from a programming domain. With this understanding in hand, programming language designers can create a language that is 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. Instead of uncovering what human users find "natural" for a given domain, we uncover what LLMs find "natural" and then select an intermediate language that meets these expectations.

Specifically, for a target VLPL language T (e.g., Posix regular expressions), followers of our approach 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 low resource language (e.g., Python regular expressions can be directly translated to Posix regular expressions). Our approach takes P, C, and a compiler from C to T (C should be selected so that this is trivial), and produces text-to-code pipeline that outperforms the state-of-the-art for the language T.

The second key idea behind our approach is that program analysis techniques that are overly aggressive for human users may be suitable for LLM users. For example, in Java, users cannot use an integer as the condition in an if-statement: `int x = 0; if (x) {...}` is not allowed. This is a reasonable design decision for human users. But, if an LLM wrote this code, instead of crashing, one could automatically "repair" the program: `int x = 0; if (x == 0) {...}`. Unlike human users, LLMs are not offended or confused when their programs are automatically modified.

We use these two ideas to define a new text-to-code approach called *synthetic programming elicitation and compilation* (SPEAK). Users first define the parent language, P, the child language, C, and a compiler from C to the target language, T. Given these artifacts, we use existing program analysis techniques to "break" programs in P and then "repair" the "broken" programs, with the help of an LLM, into programs in C. We especially rely on existing work on *finding minimum error sources* [Pavlinovic et al., 2014].

The rest of the paper is organized as follows. In Sec. 2 we give the necessary background on programming languages to understand our approach. In Sec. 3, we describe our pipeline in detail. In Sec. 4 we perform a synthetic programming elicitation study and describe a corresponding pipeline implementations for UCLID5 [Polgreen et al., 2022]. UCLID5 is a modelling language based on the paradigm of transition systems that is used for the formal verification of hardware-software systems. In Sec. 4.4 we empirically evaluate our approach. Finally, we conclude the paper with a discussion of related work in Sec. 5.

2 Background

In this section we provide the necessary technical background to understand our approach.

2.1 Algebraic Data Types

Algebraic data types (ADTs) are a representation of finite trees that is common in functional programming languages. We provide an informal definition of ADTs and point the interested reader to Barrett et al. [2010] for a more formal treatment.

An ADT consists of a set of constructors (node names), selectors (directed edge names), and testers (predicates about finite trees). Each constructor has a fixed set of selectors associated with it (nodes

with a particular name have a fixed set of outgoing edges). Each selector has an associated type (each edge can only point to a fixed set of node names). Unlike constructors and selectors, testers are not part of the finite trees that ADTs represent. Instead, testers are predicates that are used to ask questions about the finite trees. Specifically, 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.

For example, consider an ADT called `Bool` that represents simple Boolean logic expressions. `Bool` has four constructors: `Not`, `And`, `True`, and `False`. The constructor `Not` is associated with the selector `arg`, which in turn is associated with the type `Bool`; The constructor `And` is associated with the selectors `arg1` and `arg2`, which in turn are both associated with the type `Bool`; and the constructors `True` and `False` are associated with no selectors. An instance of `Bool` is `Not(arg=And(arg1=True, arg2=False))`. This is a finite tree whose root node is `Not` and whose two leaves are `True` and `False`.

2.2 Satisfiability Modulo Theories and Weighted Maximum Satisfiability

We give an informal presentation of satisfiability modulo theories (SMT) that focuses on only the theory of ADTs and is in line with standard presentations of weighted maxmimum satisfiability (MAX-SAT). For a complete and formal presentation on SMT, see Barrett et al. [2010].

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,

$$I(x) \doteq \begin{cases} \texttt{True} & x = z \\ \texttt{False} & \text{otherwise} \end{cases}$$

is an interpretation. Interpretations are extended to atomic formulas in the natural way (plug-in the values for the variables and evaluate using the standard semantics of Boolean logic, equality, and testers). 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: a clause is satisfied if any of its elements is satisfied; a formula is satisfied if all of its elements are satisfied. 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 satisfiable (solvers aim to satisfy as many clauses as possible). The weighted maximum satisfiability problem (MAX-SAT) 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 satisfiable and containing all "hard" clauses.

2.3 Programming Languages, Verification, and the Text-to-Code Problem

We will not attempt to characterize all programming languages. Instead we will provide two abstract definitions—for executable and verification programming languages—that will help explain the scope of our work and our approach. To begin, programs are binary strings $\{0, 1\}^*$ and, for a program p, we say that p' is *a subprogram of* p iff p' can be obtained by deleting characters of p.

In our formulation, an executable programming language P is given by a parser ρ, an ADT definition A, a constraint generator Φ, and an interpreter (or executor) e. Parsers are functions from binary strings to finite trees or an error state: $\rho : \{0, 1\}^* \mapsto \{A, \bot\}$. We say that a binary string p *passes the syntactic checks* of P iff $\rho(p) \in A$. We call the non-error producing outputs of the parser (the

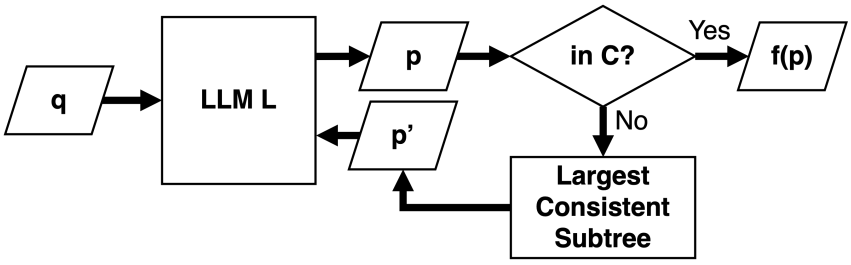

Figure 1: Overview of the SPEAK workflow. The user inputs q, a natural language prompt. The output is $f(p)$, a program in the target language T.

instances of A) *abstract syntax trees*. The constraint generator Φ is a function that takes an abstract syntax tree and produces a formula. For simplicity, we say that $\Phi(\perp)$ returns $\{\{\}\}$ (i.e., false). We say that a binary string p *belongs to* P (written $p \in P$) iff $\Phi(\rho(p))$ is **sat**. Finally, the interpreter e takes a $p \in P$ and a stream of environment inputs as needed, and returns a *trace*. The concrete definition of a trace is not important in our context, the only property we will use is that every trace is an element of a set (which we will denote Tr).

Definition 1 (Correctness) *A* specification S *is a set of traces ($S \subseteq$ Tr). We say that a binary string $p \in P$ satisfies a specification S, written $p \models S$, iff every trace generated by e on p is an element of S. When S captures the human-intended behaviour of p and $p \models S$, we say that p* is correct*.*

Verification programming languages are like executable programming languages but instead of having an interpreter they have a checker. Checkers take a binary string $p \in P$ along with a specification S and (attempt to) determine if $p \models S$. These languages make determining correctness easier.

UCLID5 Polgreen et al. [2022], Seshia and Subramanyan [2018] is one example of a verification language. Specifically, UCLID5 is a language and tool suite designed for the verification of concurrent systems, which can be represented as modular transition systems against global or local specifications. It uses model checking to generate logical queries that check whether or not a system satisfies a specification.

Definition 2 (Text-to-Code) *Given a natural language description of a programming task and a target programming language P, the text-to-code problem is to generate a correct program $p \in P$.*

3 Approach

Fig. 1 shows an overview of the SPEAK workflow. The workflow is parameterized by an LLM, L; a target language, T; a parent language, P; a child language, C; and a compiler, f, from C to T. We assume that L can generate and edit trees from P.

Given an input task q, the first step of the workflow is to ask L to generate code, $p \in C$, that achieves q. If L succeeds, then SPEAK returns $f(p)$. More often than not, however, p will be in P but not C. In that case, the second step of the workflow is to find the *largest consistent subprogram* of p in C, called p'. This subtree will be a partial program, so the third step of the workflow is to ask L to complete the program p', assigning the resulting program to p. Steps two and three repeat until p is in C, at which point SPEAK returns $f(p)$.

In the remainder of this section, we describe the largest consistent subprogram and how to find it. We then describe an optimization of step three in Fig. 1, called *micro-prompting*. The idea behind micro-prompting is to use compiler information to generate small prompts that help guide the LLM towards complete programs more quickly. Finally, we finish this section with a description of *synthetic programming elicitation* and how to use it to select languages P and C given a target, T.

3.1 Largest Consistent Subprogram

Fig. 2 (code before edit) shows an example code snippet in OCaml that parses but fails to type check. The issue in this code is that the function "duplicate" defined on line 1 expects a list of integers, but

```
1 @@ -1,3 +1,4 @@
2 -let duplicate (ls : int list) = List.append ls ls;;
3 +(* Fill this hole with a type expression that unifies ``int'' and ``string'' *)
4 +let duplicate (ls : ?? list) = List.append ls ls;;
5  assert (duplicate [1; 2] = [1; 2; 1; 2]);;
6  assert (duplicate ["a"; "b"; "c"] = ["a"; "b"; "c"; "a"; "b"; "c"]);;
```

Figure 2: Example OCaml code that fails to type check (lines 2, 5, and 6). The error does not appear until line 6, but the minimal error is the use of "int" on line 2. Line 4 shows the edit that needs to be made to generate the largest consistent subprogram ("??" is a hole). Line 3 shows a potential micro prompt for helping the LLM to fill the hole. A correct solution would be to insert the type parameter "'a" in the hole.

the call on line 3 provides a list of strings. Existing techniques, like those in the vein of monitor guided decoding Agrawal et al. [2023], should disallow the generation of line 3 because it creates a program that violates static checks. Unfortunately, the error is actually manifest much earlier in the code (at line 1). By only allowing the generation of prefixes that can lead to valid programs, these prefix-guided techniques force the LMs away from correct solutions. This is akin to only allowing developers to fix their programs by appending code.

To remedy this, we propose the notion of *largest consistent subprogram*. The idea is to take an AST, generate a largest sub-AST that is not rejected by the static checks of the language (there could be multiple largest subtress of the same size), and then make a second LM call to create a new, full AST from the generated sub-AST. Fig. 2 (code after edit) shows a largest consistent subprogram corresponding to the code before the edit. The only difference between the two versions is that "int" has been replaced with "??". We call the position in the AST indicated by "??" a hole.

Finding the largest consistent subprogram is a two step process. The first step is to find the largest finite tree that can be represented in the child language. The second step is to edit that finite tree. Specifically, let ρ_C be the parser for the child language, let A_C be the ADT for the ASTs of the child language, let Φ_C be the constraint generator for the child language, and let p be the input program. The first step is to find the largest subprogram p' of p such that $\rho_C(p') \in A_C$. The most naive approach is to try every subprogram of p. This will terminate and is correct but is inefficient, requiring $\mathcal{O}(2^n)$ calls to ρ_C. In our implementation, briefly described in Sec. 4.3, we use a recursive descent search to find an approximate p' much more quickly.

The second step is to take the AST $\rho_C(p')$ and find the largest subtree that passes all static checks of the language. To do this, we construct a new constraint generator for the child language, Φ'_C such that for every $a \in A_C$ it is the case that $\Phi_C(a)$ is **sat** iff $\Phi'_C(a)$ is **sat**. Furthermore, Φ'_C must generate a MAX-SAT problem—designate clauses as hard or soft and generate weights for the soft clauses—and associate parts of p' to each clause. This MAX-SAT problem must be created such that a soft clause being set to false in the solution implies that $p' \notin C$ but that there exists a substitution to the parts of p' associated with the soft clause that could create a new program p'' that does belong to C. Creating this query is language dependent.

3.2 Micro Prompting

In this section, we describe how to generate hole-specific prompts using compiler information, called *micro prompts*, to guide the LM in filling AST holes. Specifically, after generating the largest consistent subprogram, we use the child language's static checks to determine what could go in the hole. Implementing micro prompts amounts to inserting automatically generated error messages immediately before the locations identified by the largest consistent subprogram identification step.

For example, line 3 of Fig. 2 shows a micro prompt in an OCaml program. Simple Hindley-Milner [Damas, 1984] type inference can determine that the input argument to the "duplicate" function defined on line 2 must unify the types "int" and "string", due to the uses of the function on lines 3 and 4. The micro prompt provides this information to the LM as a comment. We evaluate the impact of micro prompting in Sec. 4.4.

We have not implemented this yet, but just like we associate a program position with each clause of the MAX-SAT query, we can just as easily associate a "reason" with the inclusion of each clause. When a clause is falsified in the MAX-SAT solution, we can use this "reason" to generate the micro prompts.

3.3 Synthetic Programming Elicitation

To keep this report somewhat short, this section has been moved to Appendix A.

4 Case Study: UCLID5

In this section we conduct a case study where UCLID5 is the target language. The case study consists of a synthetic programming elicitation study, where we attempt to identify suitable parent (P) and child (C) languages, and an implementation description, where we describe the compiler from C to T and our largest consistent subprogram search.

UCLID5 is an extremely low resource language, with 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 general purpose programming languages. As such, state-of-the-art LLMs are unable to generate code any useful UCLID5 code out-of-the-box (see Sec. B).

4.1 UCLID5 benchmarks

Our UCLID5 dataset consists of a set of regression tests and a set of problems and examples taken from verification textbooks.

Each regression test consists of a natural language description of the test and the corresponding UCLID5 code. We obtain tests from three different sources: 1. We analyze recent GitHub issues reported by UCLID5 users, extract tests from the reported issues, and manually add labels. 2. We examine recently merged pull requests implementing new UCLID5 features and extract regression tests from these pull requests. 3. Two UCLID5 developers hand-wrote tests and labels.

We extract verification problems from three textbooks: Baier and Katoen [2008] (21 benchmarks), Lee and Seshia [2010] (10 benchmarks), and Huth and Ryan [2004] (three benchmarks). From Baier and Katoen [2008] we extract descriptions of each uniquely described system in examples in chapter two, as well as descriptions of systems from exercises in chapter two and chapter three. These chapters cover modelling concurrent systems, and linear-time properties. From Lee and Seshia [2010] we extract descriptions of each uniquely described system in examples in chapter three, which covers systems with discrete dynamics. From Huth and Ryan [2004] we extract descriptions of systems from examples used in chapter three, which covers verification by model checking.

Each example consists of a brief natural language description of the system, copied directly from the textbook. If the description refers to a figure in the textbook, we modify the description to include the relevant information from that figure. We also add an instruction stating that the LLM should model the system described. If the example included a property that should be checked or proven, we ask the LLM to write such a property.

The number of examples we can extract is limited by the fact that the textbooks tend to reuse the same system as examples to illustrate multiple different concepts, in which case we only extract the system description into one singular benchmark.

4.2 UCLID5 Synthetic Programming Elicitation Study (Selecting P and C)

To keep this report somewhat short, this section has been moved to Appendix B.

The import conclusion is that, after analyzing the outputs, we concluded that it would be easiest to translate Python programs from the "object-oriented" category to UCLID5. Therefore we defined a subset of Python that matches the "object-orineted" category and used that as our child language, C. Fig. 3 shows an example of a Python program in C (Fig. 3a) and its corresponding UCLID5 program

```python
class TemperatureEventCounter(Module):
  def types(self):
    self.Temp = Integer()
  def locals(self):
    self.count = Integer()
    self.prev_temp = self.Temp
  def inputs(self):
    self.threshold = self.Temp
    self.current_temp = self.Temp
  def init(self):
    self.count = 0
    self.prev_temp = 0
  def next(self):
    if self.current_temp > self.threshold and
    ↪  self.prev_temp <= self.threshold:
        self.count = self.count + 1
    self.prev_temp = self.current_temp
```

(a) Trimmed Python code generated by gpt4.

```
module TemperatureEventCounter {

  type Temp = integer;

  var count : integer;
  var prev_temp : Temp;

  input threshold : Temp;
  input current_temp : Temp;
  init {
    count = 0;
    prev_temp = 0;}
  next {
    if (current_temp > threshold && prev_temp <=
    ↪  threshold) {
      count' = count + 1;}
    prev_temp' = current_temp;}}
```

(b) Corresponding, aligned UCLID5 code.

Figure 3: SPEAK responses for the task "Model an event counter that is used in a weather station to count the number of times that a temperature rises above some threshold."

(Fig. 3b)for a benchmark from Lee and Seshia [2010]. The Python code was generated by gpt4 and the UCLID5 code was generated by our final implementation, which we describe next, in Sec. 4.3.

4.3 SPEAK UCLID5 Implementation

We implement a prototype of our approach for UCLID5 as a Python project and attach the code as a zip file. The project depends heavily on tree-sitter and Z3. We use tree-sitter to parse Python code and then use tree-sitter queries in a recursive descent manner to generate an AST. We then walk this AST to generate a MAX-SAT problem and use Z3 to solve this problem. We use the model generated by Z3 to identify the parts of the AST that need to be modified and use a set of heuristic rules to try to repair the program locations based on the model and the original code at that location.

The included zip file has a number of examples (in the "examples" folder). Each example has three files associated with it: one that contains "input," one Python file the contains "output," and one UCLID5 file that contains "output." Running the tool on all input files should produce all the output files. The most interesting examples are the ones that also contain the word "inference." These examples show off the MAX-SAT encoding best.

The included zip also has the best description of the MAX-SAT encoding. Specifically, the function `encode` on line 183 of src/eudoxus/checker/type.py is essentially a big match statement, where each clause handles a part of the AST and the comments for each clause describe the corresponding hard and soft constraints.

4.4 Evaluation

We have not completed a full evaluation yet. However, we intend to empirically evaluate our two prototype implementations across the following research questions.

- RQ1 How do our prototype implementations perform compared to the state-of-the-art in terms of pass@k?
- RQ2 Are LLMs able to consistently generate programs in our parent languages (P)?
- RQ3 How many iterations does it take before SPEAK can successfully generate a program in our child language (C) from a program in our parent language (P)?
- RQ4 Does micro-prompting help reduce the number of iterations?

5 Related Work

LLMs for Low-resource Programming Languages. Recent work has started considering training models for low-resource programming languages [Cassano et al., 2023, Chen et al., 2022]. Chen et al. [2022] shows on smaller 125M parameter encoder-only models fine-tuning on adjacent languages improves the monolingual performance coding tasks. However, it remains an open question whether autoregressive language models share this benefit and whether this fine-tuning trend scales to larger (7B to 70B) language models.

Synthetic fine-tuning datasets curated and cleaned by LLMs have shown promise for programming tasks [Cassano et al., 2023]. Cassano et al. [2023] in particular tackle low-resource programming languages, using the 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 where a good source of semantically similar code exists. In formal settings, there may not be obvious sources of high-resource examples that are semantically relevant for specific targeted domains.

6 Conclusions and Future Work

There is still a lot left to do and we plan to submit this work in mid May. After that, however, we want to explore two uses of term rewriting. First, we may use a rewriting engine to implement the simple compiler from the child language to the target language. Right now we are writing a custom compiler, but it could be nice to have this step be declarative. Second, when our definition of the child language is too strict, the pipeline can suffer. So the fourth idea is to use a rewriting engine to expand the scope of what qualifies as a program in the child language through rewrites on the parent language.

References

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`.

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

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.

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.

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`.

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.

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.

Luis Damas. Type assignment in programming languages. *KB thesis scanning project 2015*, 1984.

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

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

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

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`.

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`.

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.

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

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

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.

```python
# GTCODE 1: imports a class from hypothetical library
from RegularExpressionLibrary import Regex
# GTCODE 2: reuses host language syntax
# GTCODE 3: creates an instance of an imported class
r = Regex("1.*1")
# GTCODE 4: uses a dunder method (in this case __str__) of an imported class
print(r)
```

Figure 4: Hypothetical LM output with grounded theory codes as comments.

A Synthetic Programming Elicitation

Synthetic programming elicitation draws heavily on natural programming elicitation. But, LMs are not like humans and so LM studies are not like human studies. In this section, we present a three step recipe for conducting LM programming studies—studies where LMs are the subject. The results of these studies can be used to design a new programming language tailored for LM use.

First Step: Setup To setup a synthetic programming elicitation study you need to prepare LM subjects, select a target language, and collect or construct a set of programming tasks. The target language should be a low resource language and the set of tasks should be natural language descriptions of programs that are suitable for this target language but not specific to the target language. For example, if your target language is Posix regular expressions, than a task could be "Define a language that accepts all strings that start and end with the number 1."

Our ultimate goal is to create a domain-specific language (DSL) that acts as an intermediary between LM outputs and the target language we picked. DSLs come in two general flavors: standalone and embedded. Standalone DSLs are independent languages with their own syntax and semantics. Embedded DSLs can be thought of as extensions to a host language. The simplest form of an embedded DSL is a library, like Pandas, Numpy, or PyTorch. Depending on the target language, it can be easier to create a standalone or embedded DSL as the new intermediary.

The final part of the study setup is to create two different specialized prompts per task. The first specialized prompt will ask for the output to be in the target language (standalone DSL). The second specialized prompt will ask for the output to use an API in a popular host language, like Python (embedded DSL). For example, our regular expression task above could result in two different prompts

1. "Write a Posix regular expression to complete the following task. Define a language that accepts all strings that start and end with the number 1."
2. "RegularExpressionLibrary is a Python library for creating regular expressions. Use the RegularExpressionLibrary to complete the following task. Define a language that accepts all strings that start and end with the number 1."

Second Step: Execute The second step of the synthetic programming elicitation study is the most straightforward: execute every LM subject on every prompt and collect the outputs. Each query should be executed independently. Note that some of the prompts will ask the LM to use a library that does not exist. After the study is concluded, programming language designers may decide to implement this library.

Third Step: Analyze and Design The final step is to analyze the collected LM outputs and then design a new language based on the analysis. To perform this analysis, we follow a lightweight version of grounded theory [Glaser and Strauss, 2017].

The first step of our grounded theory analysis is to "code" the outputs generated by the LMs. In grounded theory parlance, "code" is akin to labelling parts of the generated outputs. For example, Fig. 4 shows a hypothetical LM output for our regualr expression 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. 4 is the group of codes 1, 3, and 4. 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 propose a new DSL design that is consistent with the final categories we observed across multiple prompts. For example, a DSL design for regular expressions that would be consistent with the observations of Fig. 4 could be a single Python class that holds a string. An implementation of this DSL would consist of a function that takes this single string and compiles it down to our target language, Posix regular expressions. This DSL design can be refined by observing more examples or repeating the grounded theory analysis.

B UCLID5 Synthetic Programming Elicitation Study (Selecting P and C)

To select the parent (P) and child (C) languages for UCLID5 we followed the synthetic programming elicitation procedure described in Sec. 3.3. Specifically, we used the set of regression tests described in Sec. 4.1 to query four different LLMs (gemini-0.7, gpt-3.5-turbo-0125, gpt-4-0.7, gpt-4-0125-preview-0.7) with two different kinds of prompts. The first kind of prompt asked for UCLID5 code that fit the regression test description. The second kind of prompt asked for Python code that uses a formal verification API (which did not exist) and fits the regression test description. The following is an example prompt.

Synthetic Programming Elicitation Prompt

```
You are an expert user of FormalVerificationLibrary, a Python API for the formal
modeling and verification of transition systems.  Please write Python code using the
FormalVerificationLibrary API that fits the description below.  The module implements
a transition system with two variables, a and b.  The variables are both initialized
as 1, and the next block increments the variables according to the Fibonacci sequence.
An inline assertion in the next block checks that a is always less than or equal to b.
There is no global property or invariant.  The assertions are checked using bounded model
checking for 3 steps.
```

We found that, when asked to write UCLID5 code, no LLM was able to produce code that parses (1320 attempts: 10 samples per LLM per benchmark, 33 benchmarks, four 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 [**?**]) 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.

After analyzing the outputs, we concluded that it would be easiest to translate Python programs from the "object-oriented" category to UCLID5. Therefore we defined a subset of Python that matches the "object-orineted" category and used that as our child language, C. Fig. 3 shows an example of a Python program in C (Fig. 3a) and its corresponding UCLID5 program (Fig. 3b)for a benchmark from Lee and Seshia [2010]. The Python code was generated by gpt4 and the UCLID5 code was generated by our final implementation, which we describe next, in Sec. 4.3.