Automatic Program Synthesis with Correctness Guarantees

Sankha Narayan Guria

1 Introduction

Computing has emerged as an essential foundation of modern society. Even as researchers, our productivity has increased by orders of magnitude due to the ability to simulate experiments and run computational analysis at speeds unimaginable even half a century earlier. This innovation fundamentally relies on our ability to program computers *efficiently* and *correctly*. Modern programming languages have vastly improved the state-of-the-art in programming. For example, it is possible for a person to write a program in Python (a modern day higher-level programming language) in a fraction of the time it took for the same program to be written in Assembly language (a lower level machine language) five decades ago. That is the power of programming languages—it increases our efficiency of solving problems by enabling us to write correct code by giving us higher levels of abstractions.

Despite all the advances made in programming technology, we are still suffering from the *software crisis* [14]: an ever increasing reliance of our society on computing resulting in the growing power and complexity of these systems. Our tools for writing software have not evolved to handle this rising complexity, as can be evidenced in the regular security vulnerabilities and bugs encountered in software. If trained professionals practicing software engineering fail to achieve this goal, how would people without software expertise that use computing fare? Events such as the crash of the NASA Mars Climate Orbiter due to a software bug [3] in 1999 or wrong conclusions from data analysis in social sciences research even as recently as February 2024 [10] demonstrate that this problem still remains unsolved.

The proposed project fills this gap by designing novel *program synthesis* tools that allow non-expert programmers to provide a high level description of a problem, and then automatically generates a correct program, thus eliminating bugs and computing accessible to a broader audience.

2 Background and Methods

Program synthesis, or the capability to automatically write a program from its specification has been called the *holy grail of Computer Science* [6] since the inception of AI in the 1950s. Two broad lines of research has emerged as promising approaches to realizing this goal.

Deep learning has seen significant success in program synthesis in the last decade. Due to rise of open-source code hosting platforms such as GitHub, the availability of training data has exploded. This has enabled models to be trained on a vast corpus of data. Approaches like DeepCoder [2] learn common patterns used by programmers to generate similar code. Recently, Large Language Models (LLMs) have emerged as a powerful class of models for code generation from natural language prompts. Foundation models such as GPT-4 [15] (basis of commercial tools like ChatGPT, Microsoft Copilot) or Meta's Llama family of open-source models [4, 18] have become competitive at tasks of automatically writing programs. These models shine at generating longer programs than were previously possible, while still matching a human programmer because it was trained on such data. However, these approaches suffer from two major flaws. First, these models tend to hallucinate, i.e., make up programs that are not possible. For example a model may output code that may lead to type errors, or even non-existent library code being called. Second, if a program is generated from a natural language prompt, there can be no mathematical proof that a program has the intended behavior, because natural language are informal and ambiguous by definition.

In contrast, a contemporary approach grounded in formal methods and logic has seen a lot of success in program synthesis as well. Here, the tools have rigorous mathematical definition of semantics of the programming language, enable reasoning over programs in a given language. This approach has seen some commercial success in Microsoft Excel as the FlashFill feature [5], which takes some example of data transformation as input-output examples and generalizes it to missing values in the spreadsheet by inferring a program under the hood. In general, this family of techniques accept some input/output examples [1], first-order logic formula [19], or even type signatures [8] of the functions to be synthesized. Because automated reasoning based on mathematical logic is at the core of these techniques these methods guarantee correctness of synthesized program. Hence, formal methods based synthesis has been a popular tool in the high-assurance, security, and sensitive software communities because correctness is of utmost importance. However, the key limitation of this approach has been the overhead in formally specifying program behavior in logic before such tools can be used. Moreover, as these tools use algorithmic insights to systematically search through all possible programs they do not scale to larger programs.

While these two approaches have very complementary strengths and weaknesses, we do not know how to combine machine learning-based methods with formal methods. Building a bridge between these two worlds will allow us to scale automatic program synthesis to larger programs while still ensuring correctness, opening up programming access to a broader audience. This project builds on the PI's expertise and prior success in formal methods based synthesis to propose novel combination of LLM based program generation with formal methods based program analysis and synthesis and coordinate these with a feedback loop.

3 Research Plan

The key focus of the proposal is to build program synthesis tools that allow domain experts to encode their expertise and letting the computer do the generation of programs that realizes the intent *correctly*. The novelty of the proposed approach lies in the reduction of manual effort needed to synthesize programs by combining strength of the discussed complementary approaches. Consider a common task in academic research as well as in industry: data wrangling, i.e., reshaping of data for downstream analysis. Figure 1 shows an illustrative example, given two tables (Table 1a and 1b) you have to compute a new table (Table 1c) based on the given data. One might reach for languages like Python to write a program that does this task at hand. However, to be productive they have to learn Python, a data frame (i.e., tables in our example) processing library like Pandas [17], and use it like an expert for the task at hand. These people are domain experts in their own areas but not necessarily in data analysis, often resulting in disastrous errors in downstream tasks. The alternative would be to use a synthesis tool to automatically produce the following program that is correct for all input data frames for this task:

```
arg0.merge(arg1, on=['id']).query("valueA != valueB")
```

A synthesis tool based on LLMs is likely to get the strings such as `"valueA != valueB"` or arguments such as `on =['id']` wrong. A formal methods based tool will require massive human effort to specify mathematically the precise behavior of Pandas functions like `merge` or `query` before a user can start using the synthesis tool.

Program generation with LLMs. We propose to use models like GPT-4[16] and open-source models such as CodeLlama [18] to generate the candidate program. We plan to pass the natural language description of the programming task along with the concrete input-output examples (Figure 1). As these models tend to hallucinate, we plan to take special precautions to minimize that by using prompt engineering techniques [20]. Crucially, this approach overcomes limitations of the formal methods approach by learning from publicly available corpus of programs to learn about commonly used libraries functions (like Pandas in our example) without any manual burden to mathematically specify library behavior.

	id	valueA
0	255	1141
1	91	1130
⋮	⋮	⋮
8	225	638
9	257	616

(a) First data frame (`arg0`)

	id	valueB
0	255	1231
1	91	1170
⋮	⋮	⋮
12	211	575
13	25	530

(b) Second data frame (`arg1`)

	id	valueA	valueB
0	255	1141	1231
1	91	1130	1170
2	347	830	870
5	159	715	734
8	225	638	644

(c) Output data frame (`out`)

Fig. 1. Example of a data wrangling task: given the data frame in (1a) and (1b) and the expected output (1c) the synthesizer should produce a program that transforms the input to the output.

Program analysis with formal reasoning. A program generated from LLM will not have any formal correctness guarantees. We propose to design program analyses that will check generated candidate programs using simple mathematical properties derived from the given input-output examples. For our running example, our system will automatically sample a few properties like $valueA \neq valueB$ and $numRows(out) \leq numRows(arg0)$ among others. The first property states in the output data frame, no value in $valueA$ column is same as a value in $valueB$ column. The second states that number of rows in the output is same or less than that in the input data frame `arg0`. These are examples of a few properties can be inferred from the given user-provided input and outputs. Based on the PI's prior experience, we will design program analyses that will automatically check any generated program from the LLM *satisfies these properties and the input-output examples*. If it does, we have a provably correct solution, else it localizes the part of the program that fails to satisfy the inferred properties. We propose to pass these localized problem points of the program to the LLM with the logical reason for incorrectness computed by the program analysis. This forms a *feedback loop* to the LLM which generates a new program, to be analyzed again. Crucially, this step ensures our synthesizer produces a correct program or helps the model converge towards the correct solution otherwise.

Evaluation. We will curate a suite of benchmark program synthesis tasks that are similar to the ones that can be handled by existing synthesis tools [7, 8]. We plan to use this as a baseline for our evaluation, i.e., our project should match or exceed the success rate for the baseline. Further, we propose to evaluate if our novel approach of combining LLMs with formal methods based program analysis actually exceeds the capability of prior work, we will curate challenge problems not handled by prior work. We will evaluate our project by running it on this set of challenge problems.

4 Feasibility and Future Prospects

PI Guria is an expert in programming languages focused on program synthesis, and has published multiple papers related to ensuring program correctness [12, 13], synthesizing programs [8, 9]. His recent work on Absynthe [7], a framework for program synthesis will be the foundation for the proposed project. The PI has experience in transitioning research to practice: his work on a type system for the Ruby programming language [11] has been transitioned to the industry and used at major companies like Stripe, GitHub, and Shopify. The PI has designed and teaches a graduate-level class in Program Synthesis that covers latest research in formal methods and machine learning based synthesis techniques. The PI aims to publish the results of the proposed work in a paper at top-tier programming languages or machines learning research venue such as POPL, PLDI, and NeurIPS. Additionally, the work proposed in this project falls in the core areas of NSF's CISE Directorate and funded by the Software and Hardware Foundations (SHF) program. The project proposed is a first step towards the larger goal of laying down the foundations of trustworthy programming tools. Ultimately, this project will act as preliminary results for an eventual NSF CAREER award.

References

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 319–336. DOI: `10.1007/978-3-662-54577-5\_18`. URL: `https://doi.org/10.1007/978-3-662-54577-5%5C_18`.

[2] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: `https://openreview.net/forum?id=ByldLrqlx`.

[3] Mishap Investigation Board. *Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999*. 1999. URL: `https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf`.

[4] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: `2407.21783 [cs.AI]`. URL: `https://arxiv.org/abs/2407.21783`.

[5] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 317–330. DOI: `10.1145/1926385.1926423`. URL: `https://doi.org/10.1145/1926385.1926423`.

[6] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 2325-1107. DOI: `10.1561/2500000010`.

[7] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. "Absynthe: Abstract Interpretation-Guided Synthesis". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1584–1607. DOI: `10.1145/3591285`. URL: `https://doi.org/10.1145/3591285`.

[8] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. "RbSyn: type- and effect-guided program synthesis". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 344–358. DOI: `10.1145/3453483.3454048`. URL: `https://doi.org/10.1145/3453483.3454048`.

[9] Sankha Narayan Guria, Niki Vazou, Marco Guarnieri, and James Parker. "Anosy: Approximated Knowledge Synthesis with Refinement Types for Declassification". In: *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. 2022. doi: `10.1145/3519939.3523725`.

[10] Vincent Holst, Andres Algaba, Floriano Tori, Sylvia Wenmackers, and Vincent Ginis. *Dataset Artefacts are the Hidden Drivers of the Declining Disruptiveness in Science*. 2024. arXiv: `2402.14583 [cs.DL]`.

[11] Jeffrey S. Foster and Brianna M. Ren and T. Stephen Strickland and Alexander T. Yu and Milod Kazerounian and Sankha Narayan Guria. *RDL: Types, type checking, and contracts for Ruby*. Version 2.2.0. June 9, 2019. url: `https://github.com/tupl-tufts/rdl`.

[12] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. "Type-level computations for Ruby libraries". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 966–979. doi: `10.1145/3314221.3314630`. url: `https://doi.org/10.1145/3314221.3314630`.

[13] Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. "Transparent Object Proxies in JavaScript". In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 2015. doi: `10.4230/LIPIcs.ECOOP.2015.149`.

[14] Peter Naur and Brian Randell. "Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th October 1968". In: (1969). url: `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`.

[15] OpenAI et al. *GPT-4 Technical Report*. 2023. arXiv: `2303.08774 [cs.CL]`.

[16] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: `2303.08774 [cs.CL]`. url: `https://arxiv.org/abs/2303.08774`.

[17] Jeff Reback et al. *pandas-dev/pandas: Pandas 1.4.4*. Version v1.4.4. Aug. 2022. doi: `10.5281/zenodo.7037953`. url: `https://doi.org/10.5281/zenodo.7037953`.

[18] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: `2308.12950 [cs.CL]`.

[19] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. "Combinatorial sketching for finite programs". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. Ed. by John Paul Shen and Margaret Martonosi. ACM, 2006, pp. 404–415. doi: `10.1145/1168857.1168907`. url: `https://doi.org/10.1145/1168857.1168907`.

[20] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo et al. 2022. url: `http://papers.nips.cc/paper%5C_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html`.