Prompting-Based UCLID5 Code Generation Using an Intermediate Language

Haley Lepe[1 2], Federico Mora[3], and Sanjit A. Seshia[3]

[1]Department of Computer Science, MiraCosta College
[2] Transfer-to-Excellence Research Experience for Undergraduates, University of California, Berkeley
[3]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
Email: haleylepe@gmail.com

***Abstract* — UCLID5 is a Domain-Specific Language (DSL) for the formal modeling, specification, and verification of systems. Programming in UCLID5, as in any language, can be time-consuming and tedious. Recent advances in Large Language Model (LLM)-based program synthesis have helped to alleviate some of the manual burden of programming and may also be applicable in the UCLID5 domain. LLMs have been successful in translating natural language prompts into working code, but so far these advances have been mostly limited to popular, general-purpose, programming languages, like Python. Success with DSLs, and especially those for formal reasoning like UCLID5, has so far been limited, with LLMs struggling to even produce syntactically correct code.**

Our proposal, called UPYGEN, asks LLMs to write Python code that uses a UCLID5 API—a Python library that generates UCLID5 code. The LLM will generate Python code, which it is good at doing, and we execute the Python code to get UCLID5 code. We evaluate and compare our approach to existing techniques by generating models of distributed protocols (algorithms that are frequently modeled in tools like UCLID5) from natural language descriptions found in prior work and find that UPYGEN outperformed existing prompting techniques by 15% for full UCLID5 module creation.

Keywords: Domain-Specific Language, Large Language Model, Prompting Techniques

I. Introduction

Automatic Code generation has vastly improved due to recent advancements within Large Language Models (LLM). Code generation has previously been approached by Program Synthesis and Deep learning models, but LLM advancement with GPT-3 in particular has been addressing the limitations of these existing techniques, such as scalability and frequent semantic errors [3]. Although LLMs are changing the field of code generation, LLMs can still face difficulties producing code. Optimization of LLM code generation has used many tactics, some of these being fine-tuning and prompting techniques. Fine-tuning is a method that further trains a Large Language Model in order to complete a specific Task. However, this technique is not a viable method for a DSL like UCLID5, as large datasets and plenty of time are necessary in order to fine-tune a model. Prompting techniques are a way to optimize the capabilities of LLMs, as asking the right question can help you get the right answer, and does not require the time and data necessary for fine-tuning. This paper will focus on natural language prompts and their production of code, and we will build an approach that mimics a [3], a Text-to-Code (T2C) experience.

Although techniques such as Chain-of-Thought have been shown to improve accuracy [1], preliminary results show these existing prompting techniques only improve the chance for a syntactically correct UCLID5 module by a mere 15%, as seen from Sections 3-C. This is further supported by related work [2], which suggests that these techniques will be more applicable to problems that are meant to produce code for popular languages and will be less effective for Domain Specific Languages (DSL) such as UCLID5. Considering this, we plan to create a pipeline strategy (see e.g., [5] for an example pipeline strategy), that uses the strength of LLMs Python generation along with prompting techniques to use Python as an intermediate step in the production of UCLID5.

II. Methods

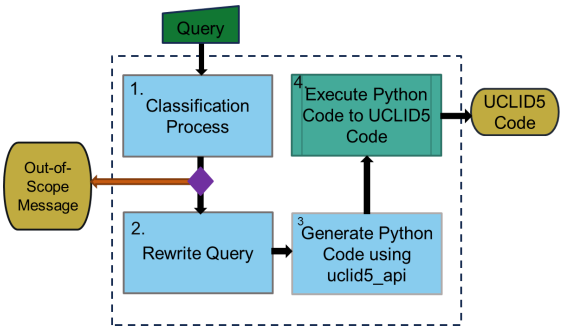


Figure 1 - UPYGEN

A. What is UPYGEN?

It is important first to understand our desired input and output of UPYGEN. The desired user input references UCLID5 and a system to model within UCLID5's applications. (e.g. "Represent a model of a traffic light in UCLID5"). The desired output of our approach is syntactically correct UCLID code. Building on this, our approach, UPYGEN, implements an internal process that breaks down our approach into 3 main steps: the query, the LLM/API internal process, and UCLID5 Output. We will discuss in-depth each part of the internal process.

Figure 2: Classifier Examination

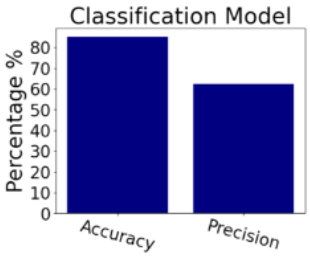


Figure 3: Syntax Analysis

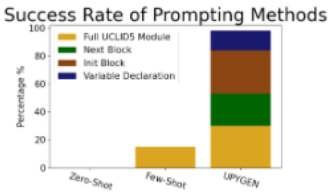
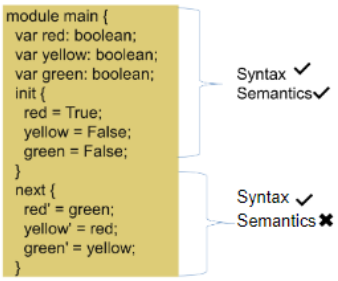


Figure 4: UPYGEN Case Study



A. *UPYGEN Internal Process Overview*

The internal process is depicted within the large dotted box as seen in Figure 1. This process is not visible to the user and overall is used to ensure the user only receives and sees the requested UCLID5 code. After the LLM retrieves a query, it initiates the 4-step internal process of UPYGEN, which uses prompting and multiple LLM calls.

A. Few-shot Internal Process Prompting

The Classification process, as shown in Phase 1, determines the suitability of the original prompt for UCLID5 purposes. We prompted the LLM using a few-shot method [4] that allowed the LLM to choose whether UCLID5 code generation can be executed from the prompt given. If the query is invalid, the program stops and outputs an out-of-scope message. If valid, the process moves on to Phase 2. In Phase 2, the model utilizes few-shot prompting again to guide the LLM in rewriting the original query and adding two crucial parts to the original query. These are 'user of the uclid5_api' along with 'generate Python code'. Although this is added, it keeps the integrity of the original user's query. Ex: The original query "Represent a Traffic Light in UCLID5" would be rewritten to say "Use the uclid5_api to generate Python Code that represents a model of a traffic light"

B. UCLID5 API and LLM Process

Once this Phase 1 & 2 are complete, the model deploys the rewritten prompt to instruct the LLM to generate Python code using the API in Phase 3. The UCLID5 API will then be used to execute UCLID5 code from the generated Python code in Phase 4. This outputs a UCLID5 module. Considering errors from the LLM, we ensure that some parts of the UCLID5 code will be outputted. This can be between the 3 main parts of UCLID5 code, which are the variable declaration section, init block to initialize values to these variables, and the next to actually change these variables. Basically, if the LLM gets stuck at any part of the UCLID5 module creation, it will output all the syntactically correct code up until this point.

III. Testing & Results

UPYGEN serves as a promising approach to produce UCLID5 modules, yet we have to evaluate whether this approach meets its intended purpose. We decided to question UPYGEN on the following themes:

1. How well does UPYGEN classify incoming queries?
2. How reliably does UPYGEN generate syntactically correct code compared to previously defined prompting methods?
3. Does the created UCLID5 code semantically make sense?

We created a sample set of 80 queries. In this sample set

- 20 queries were valid
- 60 queries were invalid (20 adversarial)

We used this sample set to test the three model questions.

A. How Well Does UPYGEN's Classifier Work?

Figure 2 depicts UPYGENS's ability to correctly classify a Sample Set of Queries we created. We used a confusion matrix [6], which is a common method to evaluate classifiers, and from the sample set, all 20 valid queries(TPs) were classified as valid by the model. However, it had more false positives (FPS) than expected, meaning it accepted more positive values than expected. This reduced the precision rate of UPYGEN's classifier. Although UPYGEN classifier could use some more improvement, currently, this doesn't cause significant harm because it's preferable to accommodate any query rather than completely restrict the user.

B. Comparative Analysis of Generated Modules Syntax

Figure 3 conducts a comparative analysis of prompting methods compared to UPYGEN. As seen, Few-Shot was able to produce UCLID5 modules when given an example of a syntactically correct module, but only with a 15% chance. Our model surpasses Few-Shot, by doubling the chance of full UCLID5 module generation to 30% and allowing our model to always output a syntactically correct part of a UCLID5 module, that few-shot prompting was not capable of.

C. Case Study

Using a generated syntactically correct UCLID5 module from the Syntax Analysis, we evaluated the semantics of this code. Figure 4 shows the result of an entire process of UPYGEN that uses the same example of the representation of a traffic light. This UCLID5 module syntactically and semantically makes sense for the original problem. However, this module shows limitations in the next block. Instead of transitioning from red to green to yellow, the model shows red to green to yellow. Although this was an example of a single generated UCLID5 Module, this pattern of the syntax was correct but the semantics of the module needing improvement continued.

IV. Conclusion

Overall, UPYGEN is a novel approach to assist LLMs creation of UCLID5 modules. Although previous prompting techniques showed promise in creating UCLID5 code, it was not reliable with only 15% of success. UPYGEN leverages LLMs ability in generating reliable Python code and uses this as an intermediate step to create a UCLID5 module, while also using prompting in order to optimize the production of full UCLID5 modules. UPYGEN has outperformed other prompting techniques with their initial success producing UCLID5 code. Although our results indicate the semantics of our generated UCLID5 modules could be improved, the syntax of this code can assist individuals who plan to use UCLID5 within their project, as having a foundation base can significantly help individuals who may feel overwhelmed creating code for a DSL such as UCLID5. Upon the completion of these results, we plan on increasing the generation of full UCLID5 modules. We plan to do this by creating a Python script that allows the LLM to have multiple chances in producing a UCLID5 block by creating a loop for each UCLID5 block. Although UPYGEN is for UCLID5, our approach can also be used for DSLs with a Python API.

Acknowledgments

This work was funded by the Hopper-Dean Foundation and the Transfer-to-Excellence Summer Research Program at the University of California, Berkeley. I thank my mentor Federico Mora Rocha and Professor Sanjit A. Seshia for their support and guidance this summer. I would like to also thank the TTE REU staff, Marcia, Marvin, Muhammad, Amanda, and the entire TTE cohort for their help throughout this research experience.

References

[1]
J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." arXiv, Jan. 10, 2023. Accessed: Jul. 07, 2023. [Online]. Available: http://arxiv.org/abs/2201.11903

[2]
B. Wang, Z. Wang, X. Wang, Y. Cao, R. A. Saurous, and Y. Kim, "Grammar Prompting for Domain-Specific Language Generation with Large Language Models." arXiv, May 31, 2023. Accessed: Jul. 07, 2023. [Online]. Available: http://arxiv.org/abs/2305.19234

[3]
C. Liu *et al.*, "Improving ChatGPT Prompt for Code Generation." arXiv, May 15, 2023. Accessed: Jul. 07, 2023. [Online]. Available: http://arxiv.org/abs/2305.08360

[4]
T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large Language Models are Zero-Shot Reasoners." arXiv, Jan. 29, 2023. Accessed: Jul. 07, 2023. [Online]. Available: http://arxiv.org/abs/2205.11916

[5]
S. Jiang, Y. Wang, and Y. Wang, "SelfEvolve: A Code Evolution Framework via Large Language Models." arXiv, Jun. 05, 2023. Accessed: Jul. 07, 2023. [Online]. Available: http://arxiv.org/abs/2306.02907

[6]
Navin JR, Maria "Performance Analysis of Text Classification Algorithms using Confusion Matrix" https://www.academia.edu/34526999/IJETR042741_pdf (accessed Aug. 11, 2023).

[7]
J. White *et al.*, "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT." arXiv, Feb. 21, 2023. Accessed: Jul. 07, 2023. [Online]. Available: http://arxiv.org/abs/2302.11382