REGAL: Refactoring Programs to Discover Generalizable Abstractions

Elias Stengel-Eskin* **Archiki Prasad*** **Mohit Bansal**
UNC Chapel Hill
{esteng, archiki, mbansal}@cs.unc.edu

Abstract

While large language models (LLMs) are increasingly being used for program synthesis, they lack the global view needed to develop useful abstractions; they generally predict programs one at a time, often repeating the same functionality. Generating redundant code from scratch is both inefficient and error-prone. To address this, we propose **Re**factoring for **G**eneralizable **A**bstraction **L**earning (REGAL), a gradient-free method for learning a library of *reusable* functions via code *refactorization*, i.e., restructuring code without changing its execution output. REGAL learns from a small set of existing programs, iteratively verifying and refining its abstractions via execution. We find that the shared function libraries discovered by REGAL make programs *easier to predict* across diverse domains. On three datasets (LOGO graphics generation, Date reasoning, and TextCraft, a Minecraft-based text-game), both open-source and proprietary LLMs improve in accuracy when predicting programs with REGAL functions. For CodeLlama-13B, REGAL results in absolute accuracy increases of 11.5% on LOGO, 26.1% on date understanding, and 8.1% on TextCraft, outperforming GPT-3.5 in two of three domains. Our analysis reveals REGAL's abstractions encapsulate frequently-used subroutines as well as environment dynamics.[1]

1 Introduction

An increasing range of tasks can be tackled by using a large language model (LLM) to generate an executable program for a given query; this paradigm has been applied in computer vision (Surís et al., 2023; Gupta et al., 2018; Cho et al., 2023), robotics (Ahn et al., 2022; Singh et al., 2023), tool use (Schick et al., 2023; Lu et al., 2023; Qin et al., 2023), and complex reasoning (Lyu et al., 2023). In all these cases, the overall program generation framework is the same: an individual query is given (along with an instructive prompt) to an LLM, which produces a program that, when executed, yields the desired result. Crucially, each program is generated *independently* (as shown in Fig. 1), with no reference to other queries or programs, and is composed of *primitive* operations, i.e., the domain language's built-in operations. This approach has two major and related limitations:

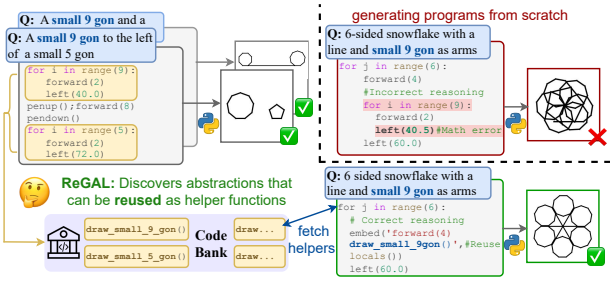

Figure 1: REGAL trains by refactoring primitive-only programs into abstractions that are verified and stored. This has two benefits: **Reusability**: Rewriting the same code multiple times leads to errors; **Abstraction**: REGAL makes prediction easier by allowing matching between the query and the abstractions.

1) Lack of Reusability: Each program is designed as a one-off script to solve a given example but is not reused by other examples. This increases redundancy and can result in unnecessary errors: for two examples requiring a shared subroutine, the model might correctly generate the subroutine in one example and make a mistake in the other. For instance, in Fig. 1 (top) although the "primitive-only" model had previously generated nonagons, it draws a polygon with an incorrect angle. REGAL's `draw_small_9gon()` function, on the other hand, executes correctly.

2) Lack of Abstraction: Shared abstractions can improve accuracy by making skills more accessible to the model. When generating from primitives alone, the model must interpret the query and generate the correct mapping from the query to

*Equal Contribution
[1]Our full paper and code are publicly available.

multiple primitives, requiring more reasoning. The model's overall task becomes *easier* when it uses interpretable abstractions, as it is choosing a function name from a library instead of reasoning from scratch. In Fig. 1 (bottom) a model augmented with abstractions can match the sub-query *"a small 9 gon"* to `draw_small_9gon()`; with this part of the task simplified, the model reasons correctly about the remaining code, while the primitive-only model fails to correctly embed the shape in a loop.

Both limitations can be traced to a *lack of global context* as the model sees each example separately, so it lacks a mechanism for developing reusable abstractions. This differs greatly from how humans write code: generally, developers might start solving individual tasks with one-off solutions, but quickly begin to develop a library of shared abstractions and code snippets for related problems, thereby reducing redundancy in their code, promoting efficiency and readability (McConnell, 2004; Downey, 2012). Furthermore, functions can be *verified*: once we have tested a function, we can rely on it in the future – something that is harder to do for ever-changing one-off code snippets. Such abstraction and verification is only sensible if the code synthesis process takes place over the course of multiple examples. In other words, if presented with a single, one-off task, there is no reason not to write a one-off script.

While abstraction offers numerous benefits, it comes with the risk of over-fitting, where a function tailored to a specific example loses its generalizability. For instance, in Fig. 1, a function like `draw_9gon_snowflake()` may perfectly match one example but fails to generalize. Conversely, `draw_small_9gon()` is a more versatile function applicable in various contexts. The ability to produce novel programs using primitive operations needs to be balanced with the benefits of encoding subroutines into reusable abstractions (O'Donnell, 2015). A similar balance between flexibility and efficiency appears in a variety of domains, including language (O'Donnell, 2015; Yang, 2016), biology (Futuyma and Moreno, 1988), manufacturing (Flynn and Jacobs, 1987), and programming (Ellis et al., 2021). To strike this balance in LLM-based program synthesis, we propose **Re**factoring for **G**eneralizable **A**bstraction **L**earning (REGAL). REGAL refines abstractions iteratively by refactoring programs as well as verifying, correcting, and pruning abstractions such that overly specific or incorrect programs are improved upon or removed. REGAL relies on two key elements: a small set of programs using primitive operations (i.e., *primitive programs*) and an execution environment (e.g., Python). Importantly, we show REGAL can learn from LLM-generated programs without requiring any human annotations.

REGAL follows a familiar train-test paradigm: during REGAL's modular training phase, it iteratively refactors a small set of *(query, program)* examples to produce a library of useful abstractions. REGAL uses an LLM to write helper functions for a batch of examples, which are verified against the expected result; successful helper functions are added to the library and the refactored program serves as an example of the function's usage. REGAL can take success feedback into account to correct and debug errors, and it periodically edits the helper functions to make them more generalizable or – if they cannot be made more generic – prunes functions that are overly specific. Note that the training is gradient-free, relying on a frozen LLM to refactor programs. In the testing phase, an LLM agent is tasked with predicting programs for test queries. The agent has access to REGAL's library of helper functions and demonstrations of how to use them.

We demonstrate the broad applicability of REGAL by testing it on three diverse datasets: LOGO (Ellis et al., 2021; Wong et al., 2021), a program induction task; a date reasoning task (Srivastava et al., 2022) known to challenge LLMs (Suzgun et al., 2022); and TextCraft (Prasad et al., 2023), a text-based game for crafting Minecraft objects. Across these tasks, REGAL significantly improves the accuracy of the predicted programs from various LLMs – especially open-source LLMs – over a baseline that predicts primitive programs (i.e., programs without REGAL's abstractions). For instance, CodeLlama-13B's (Roziere et al., 2023) accuracy increases by 11.5%, 26.1%, and 8.1% on LOGO, Date, and TextCraft respectively, surpassing larger models like GPT-3.5 (Ouyang et al., 2022). Moreover, we show that REGAL's abstractions are reusable across examples, encapsulate key domain functionalities, and we include an error analysis further highlighting the features that make REGAL effective. Finally, we show that REGAL can improve over baseline primitive programs with minimal examples, yielding major improvements even with a 50% reduced training set.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. 2022. Do as i can and not as i say: Grounding language in robotic affordances. In *arXiv preprint arXiv:2204.01691*.

Jaemin Cho, Abhay Zala, and Mohit Bansal. 2023. Visual programming for text-to-image generation and evaluation. *Thirty-seventh Conference on Neural Information Processing Systems (NeurIPS)*.

Allen Downey. 2012. *Think python*. " O'Reilly Media, Inc.".

Kevin Ellis, Lio Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 835–850.

Barbara B Flynn and F Robert Jacobs. 1987. Applications and implementation: an experimental comparison of cellular (group technology) layout with process layout. *Decision Sciences*, 18(4):562–581.

Douglas J Futuyma and Gabriel Moreno. 1988. The evolution of ecological specialization. *Annual review of Ecology and Systematics*, 19(1):207–233.

Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. 2018. Semantic parsing for task oriented dialog using hierarchical representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2787–2792, Brussels, Belgium. Association for Computational Linguistics.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 305–329, Nusa Dua, Bali. Association for Computational Linguistics.

Steve McConnell. 2004. *Code complete*. Pearson Education.

Timothy J O'Donnell. 2015. *Productivity and reuse in language: A theory of linguistic computation and storage*. MIT Press.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. 2023. Adapt: As-needed decomposition and planning with language models. *arXiv preprint arXiv:2311.05772*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*.

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

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2023. ProgPrompt: Generating situated robot task plans using Large Language Models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual inference via Python execution for reasoning. *arXiv preprint arXiv:2303.08128*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny

Zhou, et al. 2022. Challenging Big-Bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Lio Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. 2021. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pages 11193–11204. PMLR.

Charles Yang. 2016. *The price of linguistic productivity: How children learn to break the rules of language*. MIT press.