Semantic Mutation via LLMs: A Hybrid Approach to Evolutionary Program Synthesis

Woletemaryam Liyew
woleteml@gm.gist.ac.kr
Gwangju Institute of Science and Technology
Gwangju, South Korea

Dojun Oh
ohdojun34@gm.gist.ac.kr
Gwangju Institute of Science and Technology
Gwangju, South Korea

Seokki Lee
sklee1103@gm.gist.ac.kr
Gwangju Institute of Science and Technology
Gwangju, South Korea

Sejin Kim
sejinkim@gist.ac.kr
Gwangju Institute of Science and Technology
Gwangju, South Korea

Sundong Kim
sundong@gist.ac.kr
Gwangju Institute of Science and Technology
Gwangju, South Korea

Abstract

Program synthesis over combinatorial Domain-Specific Language (DSL) spaces is challenging because the search space is vast and small program edits rarely produce meaningful semantic changes. Genetic Programming (GP) offers an interpretable framework for this kind of task, but its mutation operators make purely structural changes without understanding task semantics, causing it to frequently stall in local optima. To mitigate this, we propose a hybrid framework that integrates Large Language Models (LLMs) into GP in two complementary roles: as a semantic mutation operator that proposes non-local program rewrites during evolution, and as a post-evolution repair that iteratively refines single failed programs after search terminates. All LLM outputs are treated as untrusted and filtered through grammar constraints and typed parsing before entering the population. Evaluated on the Abstraction and Reasoning Corpus (ARC-AGI-1), our framework solves 85 out of 400 tasks, raising pixel accuracy from 54% to 74%.

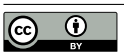
CCS Concepts

• **Theory of computation** → **Program synthesis**; • **Computing methodologies** → *Machine learning*.

Keywords

Genetic programming, program synthesis, large language models, semantic mutation, hybridization

ACM Reference Format:
Woletemaryam Liyew, Dojun Oh, Seokki Lee, Sejin Kim, and Sundong Kim. 2026. Semantic Mutation via LLMs: A Hybrid Approach to Evolutionary Program Synthesis. In *Genetic and Evolutionary Computation Conference (GECCO Companion '26), July 13–17, 2026, San Jose, Costa Rica.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3795101.3805323



GECCO Companion '26, San Jose, Costa Rica

ACM ISBN 979-8-4007-2488-6/2026/07
https://doi.org/10.1145/3795101.3805323

1 INTRODUCTION

The task of program synthesis is to automatically generate a program that fulfills a given specification, often represented by input-output examples [11]. A robust program synthesizer must compose programs from reusable sub-components and generalize beyond training examples combining learned subprograms in new ways to synthesize longer or conceptually novel programs that have the ability to reason over multi-step transformations [14]. While neural-based program synthesis approaches have made progress in this area, these approaches have important limitations: (a) they are computationally expensive and hard to train, (b) a model has to be trained for each task (program) separately, (c) it is hard to interpret or verify the correctness of the learnt mapping [13].

Given these limitations, alternative approaches based on evolutionary search have received renewed interest in program synthesis. In particular, Genetic Programming (GP) has remained a robust and interpretable framework for learning-based program synthesis, due to its ability to evolve structured programs over Domain-Specific Languages (DSLs) using selection, crossover, and mutation [15]. GP directly manipulates symbolic program trees, enabling compositional reuse and open-ended exploration of the program space [6]. These characteristics make GP especially appealing for synthesis tasks that require multi-step reasoning, hierarchical abstraction, and few-shot generalization, aligning closely with the capabilities expected of a strong synthesizer [10]. The Abstraction and Reasoning Corpus (ARC-AGI) [3] is a strong testbed for DSL-based program synthesis each task presents a few input-output grid pairs requiring a latent transformation rule that cannot be solved by pattern matching alone.

However, GP frequently stalls in local optima because its mutation operators make purely structural changes without any understanding of task semantics [12]. Recent work integrates LLMs into evolutionary processes as variation operators, performing crossover and mutation guided by semantic understanding such LLM-induced variations have demonstrated effectiveness across diverse domain including symbolic regression, code generation [8].

In this paper, we propose a hybrid framework that augments GP with two complementary LLM-based components: a semantic mutation operator that proposes non-local program rewrites informed

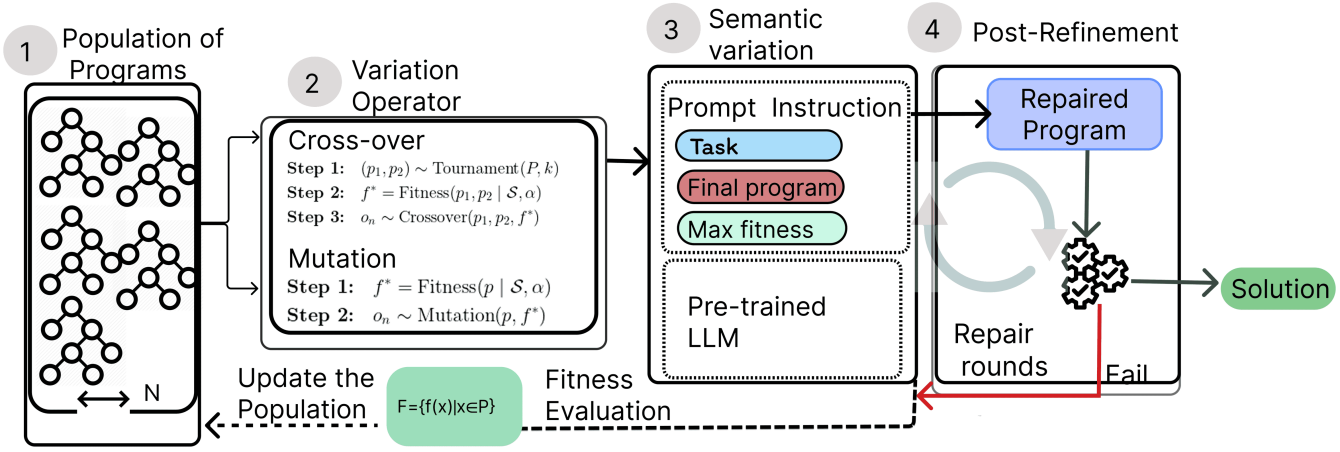


Figure 1: Full Framework: Overview of the proposed LLM-augmented evolutionary program synthesis framework.

by task semantics to widen the reachable program space during evolution, and a post-evolution repair phase that iteratively refines single failed programs to resolve localized errors that structural mutation alone cannot fix.

2 Method

We formulate program synthesis on ARC-AGI as a search over a typed DSL adopted from [16]. Our base synthesizer is Strongly Typed Genetic Programming (STGP) [9], implemented using the DEAP evolutionary computation library [4]. Each candidate program $p \in \mathcal{P}$ is represented as a typed expression tree $T = [n_0, n_1, \ldots, n_k]$, where each internal node n_i corresponds to a DSL primitive of the form $r : (\tau_1, \ldots, \tau_m) \to \tau_{\text{out}}$, and leaf nodes are terminals drawn from constants, booleans, or the input grid. All types τ are drawn from a finite set including `Grid`, `Colour`, and `List[Grid]` [5]. The GP system maintains a population $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$, where each individual is evaluated on a training set $\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{N}$ using a pixel-wise fitness function:

$$f(p) = \sum_{i=1}^{N} \text{sim}(p(x_i), y_i)$$

and the goal of evolution is to find $p^\star = \arg\max_{p \in \mathcal{P}} f(p)$.

However, we observed that STGP frequently converges to local optima in the large program space of ARC tasks, its mutation operators make purely structural changes without any understanding of task semantics, so small local edits rarely produce the coordinated program restructuring needed to escape fitness plateaus. To mitigate this, we propose two complementary LLM-based components (Full framework is shown in Figure 1). The first is a semantic mutation operator that augments GP during evolution by proposing non-local program rewrites informed by task semantics, widening the set of reachable programs per generation (Section 2.1). The second is a post-evolution repair phase that targets near-correct programs left unsolved by GP, iteratively refining a single failed program by patching localized errors after evolution terminates (Section 2.2).

2.1 LLM-Based Semantic Mutation

Standard GP mutation operators like subtree replacement and node-level perturbation make only local changes to tree structure [9]. In typed DSL search over ARC, this locality is a hard constraint: operators cannot coordinate changes across distant tree nodes, making it difficult to escape fitness plateaus that require restructuring large sub-expressions simultaneously [12]. We address this by treating an LLM as an additional mutation operator that proposes structurally novel candidates informed by task semantics.

At each generation, when the population-best fitness score does not improve for g_{plateau} consecutive generations, the LLM mutation is triggered and the plateau counter resets after any fitness improvement. The LLM is then queried with: (i) all training input–output pairs, (ii) a truncated DSL reference listing available primitives and type signatures, and (iii) the current elite programs and their fitness scores. The LLM is instructed to output K complete DSL expressions per elite individual, with no additional text (all values including g_{plateau} and $K(= n_{cand})$ are reported in Table 2). All outputs are treated as untrusted and undergo strict filtering before entering the population: candidates are parsed and type-checked against the DSL grammar, malformed or type-inconsistent expressions are rejected, and duplicates of previously seen programs are discarded. Surviving candidates are evaluated under the same fitness function as GP-generated individuals and compete directly through tournament selection.

2.2 Post-Evolution Program Refinement

While LLM-based semantic mutation widens the reachable program space during search, it operates at the population level and is ill-suited for exploiting a single near-correct program. Specifically, when GP terminates with a best program $p^\star$ that partially matches outputs but fails on a small localized error, the evolutionary loop has no mechanism to focus repair on that specific program. The near-correct solution is abandoned rather than fixed. To address this, inspired by [7], we introduced a dedicated repair phase after GP terminates. If $p^\star$ does not achieve perfect fitness after G generations, `GPT-4o-mini` iteratively patches $p^\star$ for up to R rounds. At each

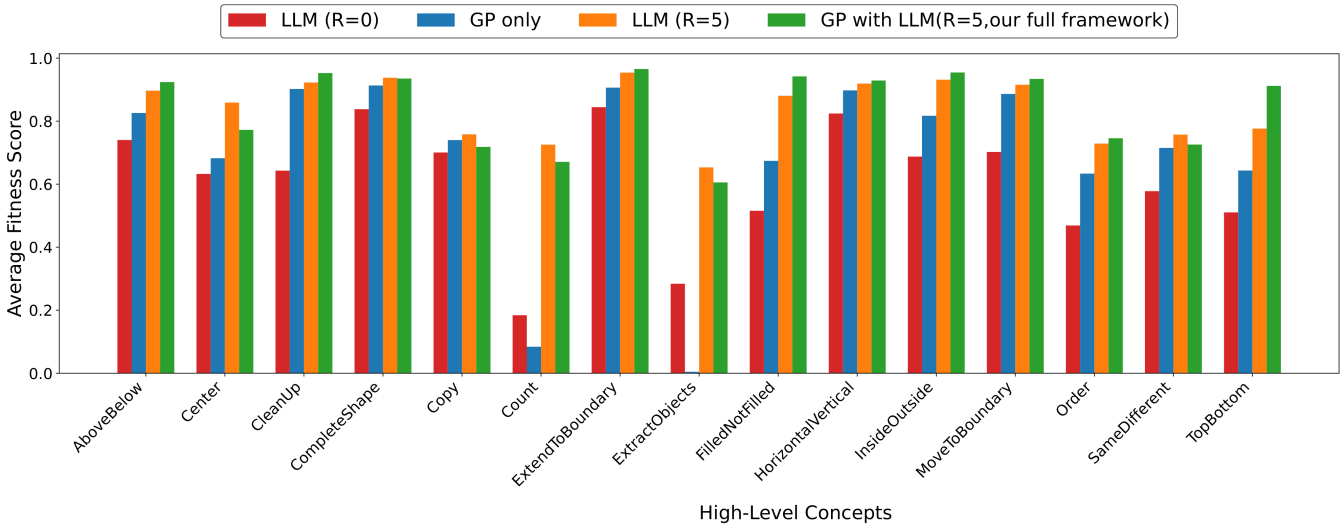


Figure 2: Performance comparison of LLM-only without repair round ($R = 0$), GP-only, LLM-only with five repair rounds ($R = 5$), and GP with LLM code edit with five repair rounds ($R = 5$, our full framework) on the ConceptARC over the high-level concept categorization.

round r, a prompt π_r is constructed from: (i) all training input–output pairs; (ii) the DSL python code of the current best program $p^\star$; and (iii) the current and maximum fitness scores. The LLM returns a revised program $p'_r \sim \text{LLM}(\pi_r)$, which is executed and evaluated. The revised program is accepted only if it improves over the current best:

$$p^\star \leftarrow p'_r \quad \text{iff} \quad F(p'_r) > F(p^\star).$$

The loop terminates on perfect fitness or exhaustion of R rounds. All experiments use $R = 5$; the effect of repair is analyzed in Figure 2.

Table 1: Performance comparison between the proposed evolutionary process with semantic mutation and existing systems.

System	Solved Tasks
ARGA [17]	49 / 400
Dreamcoder [1]	23 / 400
Dreamcoder-DC [2]	70 / 400
Proposed Framework(This work)	85 / 400

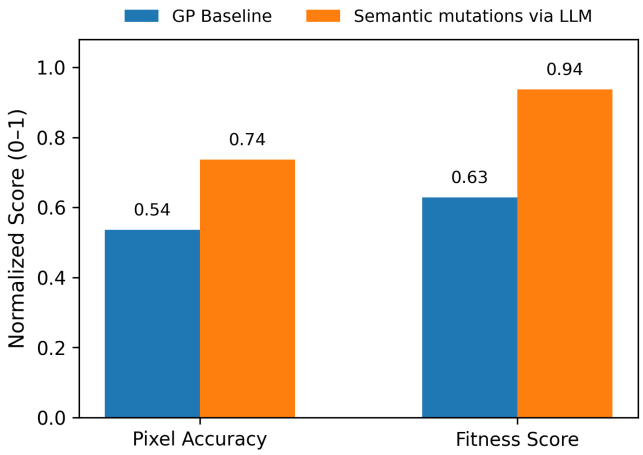


Figure 3: Comparison of aggregate performance on ARC training tasks. Semantic mutation yields approximately 37% and 49% relative gains in pixel accuracy and normalized fitness, respectively, over the baseline.

3 Experiments and Results

We evaluate on two benchmarks: the ARC-AGI-1 Training Set (400 tasks) for task-level comparisons, and ConceptARC (160 tasks across 16 reasoning concepts) for concept-level analysis. We report pixel accuracy, normalized fitness, and task solve rate as evaluation metrics. Each run evolves a population of 250 individuals over 100 generations, with GPT-4o-mini serving as both the semantic mutation and post-evolution refinement operator (full hyperparameters in Table 2). As shown in Table 1, our framework solves 85 out of 400 tasks, outperforming ARGA (49 tasks), DreamCoder (23 tasks), and its improved variant (70 tasks). Beyond task counts, augmenting GP with semantic mutation raises pixel accuracy from 54% to 74% and normalized fitness from 63% to 94% (Figure 3). The gain in normalized fitness reflects the mutation operator's role in correcting global properties like grid dimensions, object placement, and color consistency that typed subtree mutation alone cannot reach.

Post-evolution LLM refinement yields further gains when increasing from $R = 0$ to $R = 5$ refinement rounds (see Section 2.2), especially in categories where GP fails on near-correct programs with localized errors, such as *ExtractObjects*, *Count*, and *MoveToBoundary* (Figure 2). Categories like *CleanUp* and *CompleteShape*

Table 2: Hyperparameter Configuration

Parameter	Value
Population Size (N_{pop})	250
Maximum Generations	100
Elite Count (k_{elite})	2
Mutation Probability (p_{mut})	0.7
Tournament Size	3
LLM Candidates per Elite (n_{cand})	15
LLM Temperature (τ)	0.75
LLM model	`GPT-4o-mini`
LLM Max Tokens	600
Post-refinement Iterations (R)	5
Plateau Generations (g_{plateau})	4

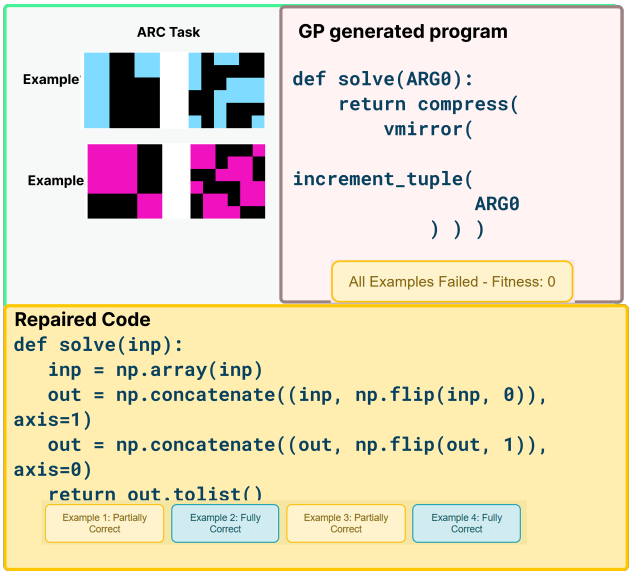


Figure 4: Case study: Comparison of predicted outputs between the repaired code and the GP-synthesized program

show smaller gains, suggesting they either require larger structural changes beyond local repair or are already well-handled by GP alone.

Case study: Figure 4 illustrates both failure modes on a symmetry task requiring horizontal and vertical reflection. The GP-synthesized program (top right) produces zero fitness across all training pairs despite being syntactically valid and correctly typed, but unable to recover through subtree mutation alone. The repaired program (bottom) shows the result after LLM-based post-evolution refinement by conditioned on the program and its fitness gap, achieves partial correctness on Examples 1 and 3 and full correctness on Examples 2 and 4, without modifying the primitive set or retraining the model. This demonstrates that post-evolution repair can resolve localized semantic errors in programs that GP leaves unsolved after search terminates.

4 Discussion and Conclusion

The results confirm that GP and LLM-based components address distinct failure modes. LLM-based mutation improves global program structure during search, reaching candidates that typed subtree mutation cannot, while post-evolution repair resolves localized errors in programs that GP leaves nearly correct but unsolved.

However, our framework has the following limitations: (a) the LLM mutation is triggered by a fixed plateau threshold, (b) many LLM-generated programs are discarded during type and grammar filtering, leading to inefficiency; and (c) post-evolution refinement operates on a single program and struggles to correct errors requiring larger structural changes. Future work can address these limitations by improving LLM integration efficiency and extending the framework to broader program synthesis domains beyond ARC.

To support reproducibility the project website is available here and the source code here.

Acknowledgments

This work was supported by NRF (RS-2024-00451162; 50%) and GIST (KH0870; 50%). Computing resource were supported by GIST SCENT-AI and KISTI.

References

[1] Simon Alford. 2021. *A neurosymbolic approach to abstraction and reasoning.* Ph. D. Dissertation. Massachusetts Institute of Technology.
[2] Mikel Bober-Irizar and Soumya Banerjee. 2024. Neural networks for abstraction and reasoning: Towards broad generalization in machines. *arXiv preprint arXiv:2402.03507* (2024).
[3] François Chollet. 2019. On the Measure of Intelligence. *arXiv preprint arXiv:1911.01547* (2019).
[4] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.
[5] Michael Hodel. 2024. arc-dsl: A Domain-Specific Language for ARC. https://github.com/michaelhodel/arc-dsl. Accessed: 2025-07-25.
[6] John R. Koza. 1994. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA.
[7] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL] https://arxiv.org/abs/2303.17651
[8] Elliot Meyerson, Mark Nelson, Herbie Bradely, Adam Gaier, Arash Moradi, Amy K Hoover, and Joel Lehman. 2024. Language Model Crossover: Variation through Few-Shot Prompting. *ACM Transactions on Evolutionary Learning* 4, 4 (2024).
[9] David J. Montana. 1995. Strongly Typed Genetic Programming. *Evolutionary Computation* 3, 2 (1995), 199–230.
[10] Ion Muslea. 1997. SINERGY: A linear planner based on genetic programming. In *European Conference on Planning*. Springer, 312–324.
[11] Michael O'Neill and Lee Spector. 2020. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines* 21, 1 (2020), 251–262.
[12] Edward Pantridge and Thomas Helmuth. 2023. Solving novel program synthesis problems with genetic programming using parametric polymorphism. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1175–1183.
[13] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).
[14] Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. 2023. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883* (2023).
[15] Dominik Sobania, Dirk Schweim, and Franz Rothlauf. 2021. Recent developments in program synthesis with evolutionary algorithms. *arXiv preprint arXiv:2108.12227* (2021).
[16] Johan S. Wind. 2020. *DSL solution to the ARC challenge.* https://github.com/top-quarks/ARC-solution
[17] Yudong Xu, Elias B Khalil, and Scott Sanner. 2023. Graphs, constraints, and search for the abstraction and reasoning corpus. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 4115–4122.