POSTER

Program Trace Optimization for Language Model Search: the Abstraction and Reasoning Corpus case

ALBERTO MORAGLIO, University of Exeter, Exeter, Devon, U.K.

ALBERTO PAOLO TONDA, National Research Institute for Agriculture, Food and Environment, Paris, Ile-de-France, France

Published: 14 July 2025

Citation in BibTeX format

GECCO '25 Companion: Genetic and Evolutionary Computation Conference Companion
July 14 - 18, 2025
Malaga, Spain

Conference Sponsors:
SIGEVO

https://doi.org/10.1145/3712255.3726554
ISBN: 9798400714641

Program Trace Optimization for Language Model Search: the Abstraction and Reasoning Corpus case

Alberto Moraglio
a.moraglio@exeter.ac.uk
University of Exeter
Exeter, UK

Alberto Tonda
alberto.tonda@inrae.fr
INRAE
Paris, France

Abstract

Program Trace Optimization (PTO) is a framework that provides automatic representation design for arbitrary problem structures by separating problem specification from search algorithm application. Problems in PTO are specified through two components: a generator that creates candidate solutions and a fitness function that evaluates them. A key strength of PTO is its ability to work with unrestricted programs as generators. We explore PTO's application to sophisticated generators based on language models, using the Abstraction and Reasoning Corpus (ARC) as our case study. Our results demonstrate how PTO can effectively search the space of programs generated by language models trained on domain-specific languages, with context-aware operators that automatically preserve the model's statistical patterns during evolutionary search. This work shows how PTO can seamlessly incorporate advanced generative methods without requiring modification of the underlying search algorithms.

CCS Concepts

• **Computing methodologies** → **Randomized search**.

Keywords

Program synthesis, Genetic programming, Representation, Language model, Working principles of evolutionary computing

ACM Reference Format:
Alberto Moraglio and Alberto Tonda. 2025. Program Trace Optimization for Language Model Search: the Abstraction and Reasoning Corpus case. In *Genetic and Evolutionary Computation Conference (GECCO '25 Companion), July 14–18, 2025, Malaga, Spain.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3712255.3726554

1 Introduction

Program Trace Optimization [8] (PTO) is a framework that makes optimization problems more accessible by addressing a key challenge in evolutionary computation: the need for expertise in tailoring search algorithms to specific problems. PTO achieves this through two innovations: (1) it separates problem specification from search algorithms using a universal trace representation, and (2) it automatically captures problem structure from user-provided solution generators, enabling domain-specific search operators. PTO currently supports various search algorithms including Genetic Algorithm, Hill Climber, and Random Search [8, 9].

A powerful feature of PTO is that generators can be unrestricted programs. While previous work has explored using constructive heuristics as generators [7], language models (LMs) offer exciting new possibilities. LMs capture patterns in sequences, with neural models now able to learn complex dependencies in both natural language and programming code [6]. Applications range from code completion [4] to incorporating programming patterns in genetic programming [10].

The Abstraction and Reasoning Corpus (ARC) [1] provides an ideal test case for our approach. ARC consists of visual reasoning tasks where systems infer transformation rules from input-output grid patterns. While humans achieve 97-98% success rates, traditional AI systems struggle with ARC's requirement for genuine abstract reasoning. Recent approaches using domain-specific languages (DSLs) [5] and neural-guided program synthesis [2] have shown promise but face challenges with the vast search space.

Our main contribution is demonstrating how language models can be effectively integrated with PTO, using ARC as a case study. We train a language model on human-written DSL programs that solve ARC tasks and use this model as a generator within PTO. Critically, PTO's search operators inherit knowledge from the language model, biasing search toward human-like solutions while maintaining evolutionary search properties. This preliminary study shows how PTO can seamlessly incorporate domain knowledge from language models through generators without modifying the underlying evolutionary algorithms, creating a bridge between traditional evolutionary computation and modern machine learning approaches to program synthesis.

2 Background on Program Trace Optimization

Program Trace Optimization (PTO) [7, 8] is a framework designed to separate problem specification from search algorithm application, enabling any search algorithm to be applied to any problem without customization (see Figure 1). PTO consists of three main components:

(1) **Problem Specification**: Users define the optimization problem by providing a random solution generator and a fitness function.
(2) **Universal Trace Representation**: PTO automatically represents solutions using a standardized linear structure—the program trace—by recording the sequence of random calls and their corresponding values during generator execution.

GECCO '25 Companion, July 14–18, 2025, Malaga, Spain

ACM ISBN 979-8-4007-1464-1/2025/07
https://doi.org/10.1145/3712255.3726554

(3) **Search Algorithms**: Generic solvers operate on this universal representation without problem-specific customization.

A key innovation in PTO is trace annotation, which augments each entry in the trace with contextual information about its execution state. This context-awareness enables meaningful search operations on the linear traces. Mutation in PTO simply resamples entries in the trace, while crossover aligns traces based on their context labels before recombination. The context labels preserve structural relationships between solution components during search, ensuring that genetic operations respect the underlying solution structure.

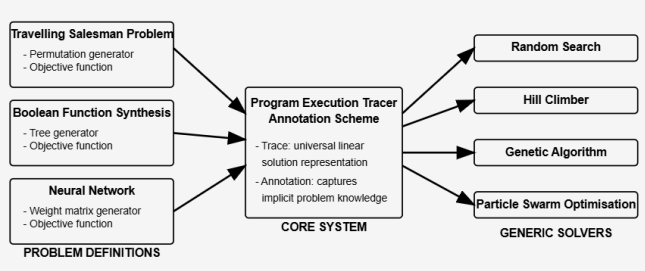

Figure 1: PTO framework with its three main components: problem specification (generator and fitness function), program execution tracer that creates the universal trace representation, and generic search algorithms.

The power of PTO lies in its ability to induce meaningful phenotypic search operators without explicitly programming them. For example, when the generator produces tree-structured solutions like Boolean expressions (Figure 2), operations on context-annotated traces (Table 1) automatically translate to operations that respect the tree structure, such as homologous crossover that exchanges structurally corresponding subtrees (Figure 3).

```
def gen():
  match rnd.choice([var,uop,biop]):
    case 'var': return rnd.choice([x1,x2,x3])
    case 'uop': return [not, gen()]
    case 'biop': return [gen(),
                         rnd.choice([and,or]), gen()]
def fitness(x):
# error on training set
```

Figure 2: Example of specification of a simple problem whose solution domain – specified by the generator – consists of Boolean expressions.

PTO derives its strength from generators, which can be unrestricted programs ranging from simple random samplers to sophisticated language models. This versatility allows generators to incorporate domain knowledge while maintaining the separation between problem specification and search algorithms.

Table 1: Trace of generating the expression ((x1 and x2) or x3): the sequence number of the random choice made by the generator (Seq), the context of where in the recursion tree the random choice was made (Name), the specific random choice made (Call), and its returned value (Result).

Seq	Name	Call	Result
0	root/rnd_type	choice(var,uop,biop)	biop
1	root/gen_b/rnd_type	choice(var,uop,biop)	biop
2	root/gen_b/gen_b/rnd_type	choice(var,uop,biop)	var
3	root/gen_b/gen_b/rnd_var	choice(x1,x2,x3)	x1
4	root/gen_b/rnd_biop	choice(and,or)	and
5	root/gen_b/gen_c/rnd_type	choice(var,uop,biop)	var
6	root/gen_b/gen_c/rnd_var	choice(x1,x2,x3)	x2
7	root/rnd_biop	choice(and,or)	or
8	root/gen_c/rnd_type	choice(var,uop,biop)	var
9	root/gen_c/rnd_var	choice(x1,x2,x3)	x3

biop	biop	var	x1	and	biop	var	x1	or	var	x2	and	var	x3
biop	biop	var	x1	and	var	x2	or	var	x3				
biop	biop	var	x1	and	var	var	x1	or	var	x2	and	var	x3
biop	biop	var	x1	and	var	x1	and	uop	var	x2	and	var	x3
biop	biop	var	x1	and	biop	var	x1	or	var	x2		and	var	x3
biop	biop	var	x1	and	var						x2	or	var	x3
biop	biop	var	x1	and	var	var	x1	or	var	x2	x2	and	var	x3
biop	biop	var	x1	and	var	var	x1	or	var	x2	x2	and	var	x3

Figure 3: One-point crossover (at position 6) on traces aligned by position (top) vs context (bottom) for Boolean expressions. In each table: rows 1-2 show parent traces for expressions ((x1 and (x1 or x2)) and x3) and ((x1 and x2) or x3); row 3 shows crossover result combining prefix from row 2 with suffix from row 1; row 4 shows trace execution with necessary repairs (red) and unreachable entries (gray). Position-based alignment produces ((x1 and x1) and not(x2)), requiring disruptive repairs, while name-based alignment produces ((x1 and x2) and x3), preserving meaningful expression structure.

3 Abstraction and Reasoning Corpus

The Abstraction and Reasoning Corpus (ARC) [1] is a benchmark for measuring abstract reasoning abilities in AI systems. Each ARC task consists of a few demonstration pairs showing input-output examples of colored grid patterns, followed by a test pair where the system must infer the correct transformation rule and apply it to a new input.

What makes ARC particularly challenging is that each task follows a different underlying logic, requiring genuine abstract reasoning rather than pattern matching. While humans solve these tasks with 97-98% accuracy, state-of-the-art AI systems reach only 55.5% accuracy [2]. ARC assumes only "core knowledge" as prior information—concepts that humans acquire in early childhood, such as objectness, basic arithmetic, and elementary geometry [11].

A prominent approach to solving ARC tasks is through program synthesis using domain-specific languages (DSLs). Hodel's DSL [5] is particularly effective, providing a functional, type-based language with approximately 160 primitive operations organized into three categories: transformation functions (rotation, mirroring), property functions (shape, size, color), and utility functions (filtering, composition).

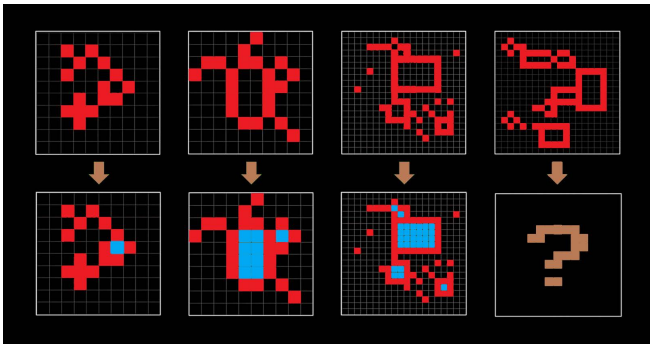

Figure 4: Example ARC task: the system must infer that black regions enclosed by black borders should be filled with blue.

Figure 5 shows a program solving the task in Figure 4. The program first identifies black objects not touching the grid border, then fills these enclosed regions with blue. This demonstrates how complex transformations can be expressed through composition of simple primitives—making the DSL well-suited for our language model approach.

```
def verify_00d62c1b(I: Grid) -> Grid:
    x0 = objects(I, T, F, F)
    x1 = mostcolor(I)
    x2 = colorfilter(x0, x1)
    x3 = rbind(bordering, I)
    x4 = compose(flip, x3)
    x5 = mfilter(x2, x4)
    x6 = fill(I, FOUR, x5)
    return x6
```

Figure 5: DSL program solving the ARC task in Figure 4.

4 Language Model Generator

To apply PTO to the ARC challenge, we need to specify a generator and a fitness function. Our key innovation is using a language model as the generator, trained on human-written DSL programs that solve ARC tasks.

4.1 Model Training and Representation

We begin by transforming the variable-assignment style programs into purely functional nested expressions that can be naturally represented as expression trees (Figure 6).

```
def verify_00d62c1b(I):
  return fill(I,FOUR,
    mfilter(
      colorfilter(objects(I,T,F,F),mostcolor(I)),
      compose(flip,rbind(bordering,I))))
```

Figure 6: Functional form of the program from Figure 5.

Our model is inspired by probabilistic context-free grammars and represents the probability distribution of program trees using production rules of the form `Parent -> C1 ... Cn [p]`, indicating that a node labeled `Parent` expands to children nodes C_1 through C_n with probability p. For training, we decompose each program into its component expression tree, extract production rules, and estimate probabilities by counting rule frequencies across the corpus.

4.2 Program Generator and Fitness in PTO

To generate programs, we sample from this model starting from the root node and recursively expanding nodes according to the learned probabilities (Figure 7).

```
def generate_rnd_expression(model, max_depth=10):
  def expand_node(node, current_depth = 0):
    if terminal(node) or current_depth >= max_depth:
      return node
    children_lists, probs = model[node]
    children = rnd.choices(children_lists, probs)
    exp_children = [expand_node(child,
                    current_depth + 1)
                    for child in children]
    return f"{node}({','.join(exp_children)})"
  return expand_node(root)
```

Figure 7: The PTO generator based on the language model.

For fitness evaluation, we measure pixel-wise differences between the outputs produced by candidate programs and the expected outputs in the demonstration pairs, with lower values indicating better solutions.

4.3 Model-Guided Search

A key insight of our approach is that PTO's context-aware trace operators naturally preserve the language model's knowledge throughout the search process. When PTO applies mutation to a trace, it resamples parts of the trace using the generator, which effectively regrows subtrees according to the language model's probabilities. Similarly, when PTO performs crossover between traces, it aligns them based on context, effectively exchanging structurally corresponding subtrees in the program trees, preserving the model's probabilities.

This creates search operators that are distribution-invariant with respect to the language model—they preserve the statistical patterns learned by the model. Unlike merely seeding an initial population with model-generated programs, PTO's operators continue to leverage the model's knowledge throughout the search, combining it with fitness-based selection to guide the search toward both plausible and correct programs.

5 Experimental Results

We evaluated our approach by comparing three search algorithms using PTO's model-guided operators: Random Search (RS), Hill Climbing (HC), and Genetic Algorithm (GA). Each algorithm was given a budget of 10^6 function evaluations. HC used position-wise mutation, while GA employed a population of 100 individuals with truncation selection, one-point crossover, and position-wise mutation.

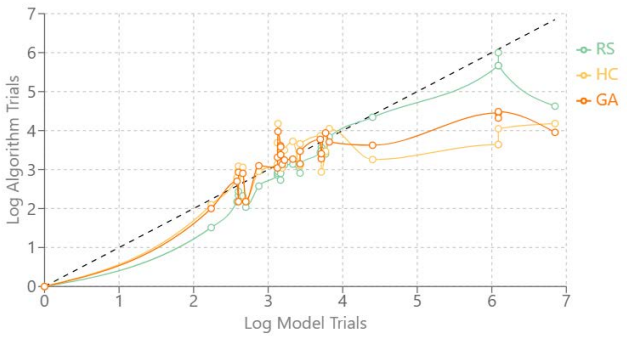

Figure 8: Model predictions vs. actual performance: As program complexity increases (x-axis), HC and GA require fewer trials than RS to find solutions (y-axis).

We selected 28 ARC tasks with varying complexity based on the log probability of their solutions under our language model (ranging from -2.24 to -6.85). Figure 8 compares the predicted difficulty of tasks (model trials) with the actual performance of each algorithm (trials needed to find a solution). For simpler programs, all algorithms perform similarly to the predicted upper bound. However, as program complexity increases, HC and GA significantly outperform RS, demonstrating the value of combining model bias with guided search for complex programs.

Method	Mean (%)	Std (%)	Method	Mean (%)	Std (%)
Runtime Error Frequency			*Perfect Prediction (Test)*		
RS	79.0	0.0	RS	93.8	1.2
HC	48.4	0.6	HC	96.1	0.7
GA	53.0	1.9	GA	97.3	0.5
Do Nothing Frequency			*Perfect Fit (Training)*		
RS	4.5	0.1	RS	95.4	0.9
HC	14.5	2.5	HC	99.9	0.0
GA	15.4	2.0	GA	100.0	0.0

Table 2: Performance metrics across all benchmark tasks.

Table 2 shows the performance metrics across all tasks. HC and GA produced significantly fewer runtime errors than RS (48.4% and 53.0% vs. 79.0%), indicating that guided search leverages the language model's structural knowledge better than pure random sampling. However, HC and GA produced more do-nothing programs (14.5% and 15.4% vs. 4.5%), suggesting they sometimes get trapped in the identity program local optimum due to selection pressure.

Most importantly, all methods demonstrated excellent generalization to test cases, with success rates of 93.8% (RS), 96.1% (HC), and 97.3% (GA). This confirms that the language model helps discover genuinely general solutions rather than merely memorized patterns. For comparison, the only other evolutionary approach for ARC [3] solved a similar number of training tasks using grammatical evolution, but did not report test case performance.

These results demonstrate that PTO effectively leverages the language model's knowledge through context-aware operators, with both HC and GA performing substantially better than RS and achieving similar levels of success.

6 Conclusion

We have demonstrated how Program Trace Optimization can effectively integrate language models as generators to guide the search for solutions to complex reasoning tasks. Our approach leverages PTO's ability to automatically design search operators that inherit the statistical patterns learned by the language model, creating a bridge between evolutionary computation and modern machine learning techniques.

The language model trained on human-written ARC solutions successfully constrained the vast search space of DSL programs, leading to excellent generalization capability across all search algorithms (93.8% to 97.3% test accuracy). While HC and GA outperformed RS in finding solutions with fewer runtime errors, the relatively competitive performance of RS suggests that the naive pixel-based fitness function provides only limited guidance for the search algorithms to fully exploit their adaptive capabilities.

This preliminary study opens several promising directions. For ARC specifically, the approach could be enhanced by incorporating type inference, adapting the model and fitness function to specific tasks, or developing a higher-level DSL. More broadly, PTO's ability to work with advanced generators could be explored with other programming languages or different types of generators such as fuzzers for software evolution.

Our work demonstrates that PTO can seamlessly incorporate domain knowledge from language models without modifying the underlying search algorithms, offering a flexible framework for combining the strengths of evolutionary computation with the pattern-learning capabilities of language models for program synthesis tasks.

References

[1] François Chollet. 2019. On the Measure of Intelligence. *CoRR* abs/1911.01547 (2019).
[2] François Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. 2024. ARC Prize 2024: Technical Report. *CoRR* abs/2412.04604 (2024).
[3] Raphael Fischer, Matthias Jakobs, Sascha Mücke, and Katharina Morik. 2020. Solving Abstract Reasoning Tasks with Grammatical Evolution.. In *LWDA*. 6–10.
[4] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.
[5] Michael Hodel. 2024. Addressing the Abstraction and Reasoning Corpus via Procedural Example Generation. *arXiv preprint arXiv:2404.07353* (2024).
[6] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *CoRR* abs/2406.00515 (2024).
[7] James McDermott and Alberto Moraglio. 2019. Program Trace Optimization with Constructive Heuristics for Combinatorial Problems. In *EvoCOP (Lecture Notes in Computer Science, Vol. 11452)*. Springer, 196–212.
[8] Alberto Moraglio and James McDermott. 2018. Program Trace Optimization. In *PPSN (2) (Lecture Notes in Computer Science, Vol. 11102)*. Springer, 334–346.
[9] Alberto Moraglio and James McDermott. 2025. Geometric Particle Swarm Optimization in Program Trace Optimization. In *EvoApplications@EvoStar (Lecture Notes in Computer Science)*. Springer.
[10] Dominik Sobania and Franz Rothlauf. 2019. Teaching GP to program like a human software developer: using perplexity pressure to guide program synthesis approaches. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1065–1074. doi:10.1145/3321707.3321738
[11] Elizabeth S Spelke and Katherine D Kinzler. 2007. Core knowledge. *Developmental science* 10, 1 (2007), 89–96.