Typed Linear Genetic Programming for Abstraction and Reasoning Corpus

Zhixing Huang [1], Bing Xue [1], *Fellow, IEEE,*, Mengjie Zhang [1], *Fellow, IEEE*
Centre for Data Science and Artificial Intelligence & School of Engineering and Computer Science,
Victoria University of Wellington, Wellington, 6140, New Zealand.
{zhixing.huang, bing.xue, mengjie.zhang}@ecs.vuw.ac.nz

***Abstract*—Abstraction and reasoning are essential abilities in human's intelligence, which help humans understand the world and extend our knowledge to unseen cases. However, abstraction and reasoning are still big challenges for existing artificial intelligence models due to limited training instances and domain-specific concepts. Existing studies usually model the abstraction and reasoning tasks as program synthesis tasks. Genetic programming features in program synthesis. However, genetic programming is hardly applied to abstraction and reasoning tasks. To fill this gap and explore genetic programming's capability for abstraction and reasoning, this paper proposed a typed linear genetic programming method and evaluated its performance on the Abstraction and Reasoning Corpus. The results show that the typed linear genetic programming has a superior performance to many existing methods, implying a good potential to fulfill abstraction and reasoning in artificial intelligence.**

***Index Terms*—Abstraction and Reasoning Corpus, Linear Genetic Programming, Program Synthesis.**

I. Introduction

Human cognition exhibits a pronounced capacity for abstraction and reasoning. We can understand and interpret concepts from only a handful of examples. In contrast, contemporary artificial intelligence (AI) models continue to exhibit significant limitations in these abilities because they regularly require substantially larger volumes of training data and prolonged training procedures before they can acquire even relatively simple concepts.

Chollet [1] introduced the Abstraction and Reasoning Corpus (ARC) to evaluate the abstraction and reasoning capabilities of AI systems. ARC consists of program-by-example tasks in which human players or AI models must infer a transformation that maps the provided inputs to their corresponding outputs. Each task includes only a small number of example input–output pairs. The inputs and outputs are represented as colored two-dimensional grids with various patterns. While these tasks are generally simple and straightforward for humans, most mainstream AI models struggle to solve them effectively. ARC is also launched as a Kaggle competition[1].

Fig.1 shows an example task of ARC. There are four input-output pairs. AI models are required to find a logical mapping (i.e., a program) from inputs to outputs. Then, the logical mapping is further verified by unseen inputs. The logical mapper is successful when the output of test instances perfectly coincides with the ground truth. In Fig.1, humans might easily find a logical mapping: 1) scale the input grid by three times; 2) replace the colored pixels with the input grid.

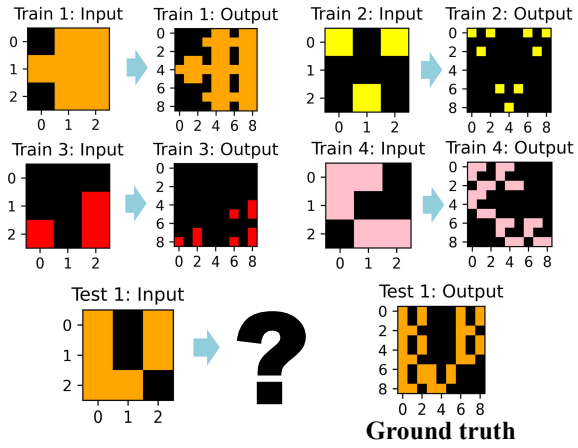

Fig. 1. An example ARC task (ID: 007bbfb7). Our model aims to find a logical mapping, based on the example (train) input-output pairs, and produce the ground truth when facing unseen (test) inputs.

Most existing studies on ARC frame the tasks as a program synthesis problem, i.e., synthesizing logical transformation from inputs to outputs. Searching program spaces is an important step in program synthesis for ARC tasks [2].

Genetic programming (GP) is a population-based metaheuristic method, featured in discrete searching for computer programs [3]. GP has undergone decades of development and has shown a competitive global search ability for symbolic search problems [4], [5]. Particularly, GP has been widely applied in symbolic regression and auto-design of decision rules [6], [7]. However, surprisingly, there are very few works applying GP to solve ARC tasks to the best of our knowledge. There is only one paper, where Fischer et al. [8] used a variant of GP, grammatical evolution [9], to solve ARC tasks. However, this study was simple and only evaluated a small subset of ARC tasks.

The main goal of this paper is to explore the effectiveness of GP in fulfilling abstraction and reasoning ability, taking ARC tasks as examples. To achieve this goal, this paper proposed

[1]https://www.kaggle.com/competitions/arc-prize-2025/overview

a **T**yped **L**inear **G**enetic **P**rogramming (TypeLGP). TypeLGP extends existing LGP techniques to problems with data type constraints by the typed registers and typed mutation. The typed registers and mutation address the multiple data types, i.e., abstract concepts, in ARC tasks. To apply TypeLGP to ARC tasks, we further adapt the domain specific language for ARC and propose a fitness function. The empirical results show a superior performance of TypeLGP to many existing methods, implying a good potential of GP methods in abstraction and reasoning.

II. Preliminary

A. Domain-specific Language

Existing studies adopted a set of grid manipulations to map input grids into outputs. These grid manipulations work with each other based on a domain-specific language (DSL). DSL is an imperative style language that specifies the derivation of grid manipulations.

One of the crucial features of DSL for ARC tasks is object-centric. The object-centric DSL extracts and generates grids based on object patterns. DSL-ARC [10] is one of the representatives of the object-centric DSL. Hodel [10] proposed six design principles for DSL and developed data types and a corresponding DSL based on the principles. These principles encourage DSL to be as specific and detailed as possible to avoid ambiguity and redundance. DSL-ARC defines a relatively comprehensive search space for many following methods, including many LLM-based models [10], [11].

Different DSLs lead to different search spaces [12], [13]. Although there have been multiple DSLs for ARC tasks, there is no clear evidence about their comparisons.

B. Search Methods

Existing studies mainly treat ARC tasks as a kind of search problem over grid manipulations. The search methods iteratively generate new programs and test their performance. Based on the way of generating new programs, existing search methods have three main search strategies: discrete search, neural-guided search, and LLM-based search.

The discrete methods solve ARC tasks through enumerating all the possible programs or enumerating programs based on heuristic rules. For example, Xu et al. [12] and Rocha et al. [14] applied greedy best-first search and iterative logic programming (ILP) method, respectively, to construct programs step-by-step.

Neural-guided search methods make use of neural networks as a model to extract knowledge from training tasks or search history. For example, DreamCoder alternatively executes two stages, "wake" and "sleep", to generate new programs and extract useful patterns from effective programs [15], [16]. Neural-guided search methods apply reinforcement learning (RL) to search ARC programs as well [17].

Large-Language-Model-based search is an emerging method for solving ARC tasks. Taking advantage of the versatile agents in LLM, LLM-based search method has been used to 1) produce Chain-of-Thoughts and hypotheses to guide the program search [18], 2) predict output grids directly based on the given input-output examples [19], and 3) score candidate programs [20].

Despite the three search paradigms, the program sampling methods in the three paradigms are not efficient enough. For example, enumerating all the possible programs cannot handle long programs. The LLM-based search methods often sample very similar programs without a good global search ability. In this paper, we take advantage of GP, which is one of the symbolic search methods that has undergone decades of development.

C. Linear Genetic Programming

Linear genetic programming is a well-known variation of GP. The distinct feature of LGP is its linear representation. Each individual is a sequence of register-based instructions, which are executed sequentially to form a computer program.

LGP offers several advantages. First, its linear representation closely resembles that of many known ground-truth ARC programs [21]. Second, LGP has been shown to yield concise programs with high probability [4], [5]. This is a desirable property for improving generalization ability as each ARC task only provides a small number of input–output examples. Finally, LGP has advantages in leveraging neutrality and building block reusing, which enhances global search capability.

III. Typed Linear Genetic Programming

There are two distinct features of TypeLGP: typed registers and typed genetic operators. The typed registers and genetic operators handle the multiple data types in ARC problems. The rest of the components in TypeLGP remain the same as those in basic LGP.

A. Typed Registers

Each typed register of TypeLGP stores an array of intermediate results from all possible data types. Every register can work with different typed functions directly. By this means, typed registers serve as a kind of adhesive among typed functions, which decouple typed functions from complicated grammar dependence. Many existing genetic operators can be applied to TypeLGP as well, as typed registers free them from the annoying grammar legality. The typed registers have initial values for every possible data type before program execution.

Fig. 2 shows an example TypeLGP program that solves the task 007bbfb7, to illustrate the idea of typed registers. The example program has 6 instructions, 2 ineffective instructions in grey and 4 effective instructions in black. Specifically, the first instruction is ineffective because its destination register `R2` does not contribute to the final output. The fifth instruction is ineffective because the output type of the instruction (i.e., `Objects`) is not the required data type (`Indices`) when `R1` contributes to the last instruction. Each typed register stores an array of results from different data types (e.g., `Boolean` and `IntegerTuple`). For example, `R1` simultaneously keeps

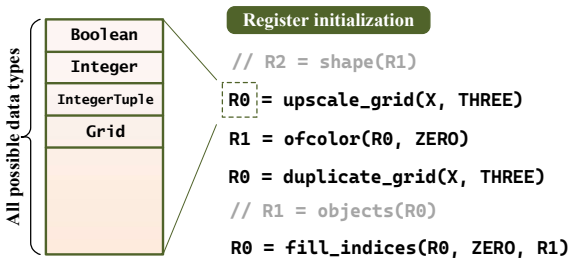

Fig. 2. An example TypeLGP program for solving the ARC task 007bbfb7.

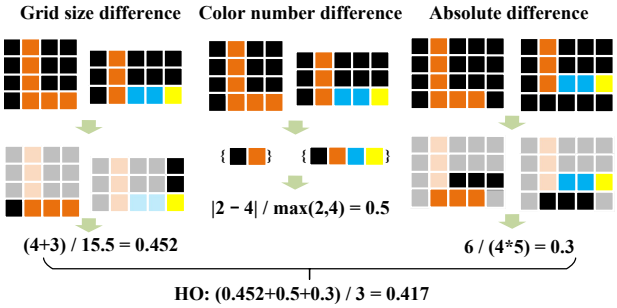

Fig. 3. An example of calculating the hierarchical objective between two example grids.

the result of type `Objects` from the fifth instruction and the result of type `Set[Integer]` from the third instruction.

Based on the design of the typed registers, the example program in Fig. 2 solves task 007bbfb7. It first upscales the input grid `X` three times and stores in `R0`. Then, it extracts the indices with no color (i.e., color of `ZERO`). It duplicates the input grid `X` three times vertically and horizontally, and concatenates them as a new grid. Finally, the program fills the indices on `R0` into the color of `ZERO`.

B. Typed Mutation Operators

The key idea of typed mutation operators is to extend the concept of effective instructions to typed instructions. A TypeLGP instruction is effective only when its output register contributes to a program output with the corresponding output data type, as shown in Fig. 2. We design three kinds of typed mutation, typed effective mutation (TEM), typed neutral mutation (TNM), and typed micro mutation (TMM).

TEM produces new programs by inserting (or removing) a random effective instruction (featured by the term "effective") into a TypeLGP program. In contrast, TNM produces new programs by inserting (or removing) a random ineffective instruction (featured by the term "neutral") into a TypeLGP program. Specifically, in the insert operation, TNM samples an effective instruction from existing competitive individuals, inserts it into the parent program, and turn it into ineffective instructions by changing the register index. TMM produces new programs by replacing a primitive (i.e., registers and functions) in an effective instruction with another primitive with consistent input and output data types, which does not affect the total number of instructions (featured by "micro").

IV. TypeLGP for ARC

There are two key adaptations when applying TypeLGP for solving ARC, DSL design and fitness function design. The DSL defines the primitive set of TypeLGP and specify the constraints of primitives. The fitness function evaluates the goodness of candidate programs.

A. DSL design

The DSL in this work follows the one in [10]. The DSL defines 11 data types, including `Boolean`, `Integer`, `Grid`, `Object`, etc. To specify the data types concretely, the abstract data types, such as `Union` and `Container`, are not allowed. Each data type has a set of possible constants and its own default value. For example, the type `IntegerTuple` has a default value of $(0, 0)$. The default values of the data types are used to initialize registers of TypeLGP.

DSL defines the possible functions among data types. The detailed design of the DSL can be found in the supplement. To simplify the search space and specify function behaviors as detailed as possible, we propose seven principles to adapt the DSL in [10].

1) The output type of all functions must not be `Callable`.
2) The input arguments of all functions can have up to one callable function. The callable input argument is required to put at the first argument.
3) For functions with callable and container (e.g., `Tuple` and `FrozenSet`) input arguments, the input data type of the callable argument must be the type of container element.
4) All functions must not nest input types. For example, if the input is a list of elements, the output type must not be a list of lists of elements.
5) All functions must not accept a concrete data type, but output a type placeholder (i.e., `Any`).
6) A function must be feasible for any possible values for a specific input data type.
7) All functions must not return an empty container. If the output is an empty container, the function should return the original container or the default value.

B. Fitness Function

The fitness function for solving ARC tasks consists of two parts: a hierarchical objective and a counter objective. The fitness value is the average of these two objectives.

1) Hierarchical Objective: Hierarchical objective (HO) measures the quality of an ARC solution over multiple granularities. Specifically, HO measures three sub-objectives: grid size difference, color number difference, and absolute difference between two given grids. These three sub-objectives are normalized to $[0, 1]$. HO returns the average over the sub-objectives.

Fig. 3 shows examples of the three sub-objectives. Grid size difference returns the number of out-of-shape pixels from a target grid, normalized by the average number of pixels of the

two grids. Color number difference returns the average of the normalized differences between the two color numbers in each grid. The absolute difference returns the number of pixels with different colors. Note that the absolute difference pads grids with black pixels if a grid has a different shape from the target.

2) Counter Objective: The counter objective (CO) encourages search methods to focus on the mistake pixels. CO is essentially a weighted absolute difference from the target grid. For every 5 generations, CO summarizes the mistake pixels by the top individuals and assigns a large weight to the pixels with a large number of mistakes. The weights are normalized to ensure CO values to range in $[0, 1]$. The pixels that all top individuals correctly predict to have a zero weight.

V. Experiment Design

A. Dataset

We evaluate TypeLGP on ARC-AGI-1. ARC-AGI-1 contains 1000 tasks, 400 public training tasks, 400 public evaluation tasks, 100 semi-private evaluation tasks, and 100 private evaluation tasks. The semi-private evaluations tasks are exposed to commercial LLM APIs but without public release. The private evaluation tasks are used to evaluate methods by ARC organizers, which are not released to the public either.

This experiment uses the 400 public training tasks and the 400 public evaluation tasks to evaluate the proposed method. The 400 training tasks are easier than the evaluation tasks. Each task has a set of example input-output pairs for abstracting and reasoning the logical transformations, and a set of unseen input-output pairs for verifying the output logical transformations. Each ARC task has input and output grids with up to 30-by-30 pixels.

B. Comparison Methods

We compare TypeLGP with discrete search methods and LLM-based methods. For methods without a specific name, we refer to them by their author's name.

The comparison discrete search methods include Ferré [22], grammatical evolution (GE) [8], and brute-force search with a depth of 3 (BFS-depth3) and with a learned distribution from training ground truth (BFS-MLE) [21]. The GE method stands for a very early trial of applying GP techniques to ARC tasks. Ferré and the two BFS methods both applied object-centric models to extract and generate grids, but with different search methods. These comparison methods provide a diverse perspective for discrete search methods.

The comparison LLM models include Mirchandani [23], OpenAI GPT-4o, and Google Gemini 1.5. Mirchandani applied a pre-trained LLM, text-davinci-003, as a pattern generation machine based on a probabilistic context-free grammar. The two commercial LLM models, GPT-4o and Gemini 1.5, both use the same publicly available prompting approach. The results of GPT-4o and Gemini 1.5 are reported in the technical report by the ARC benchmark organizers [24].

Existing literature did not have a unified standard to fairly compare different types of methods. In the Kaggle ARC competition, it mainly limits the computation time for generating the final output program to be less than 12 hours for both CPU and GPU, even though GPU is usually more powerful than CPU. The Kaggle ARC competition allows competitors to use pre-trained models, whose training time and cost are hardly considered in a fair comparison. In addition to training time, existing studies also limit the training resources by the cost of training (e.g., the cost of prompting to LLM) or total number of sampled programs [11], [22], [25].

In this experiment, we limit the training resources of TypeLGP by the number of fitness evaluations (i.e., the number of sampled programs). We limit the total number of fitness evaluations to 100,000, which is a common maximum number of sampling programs in existing methods [11], [22]. A training process terminates when it either reaches the maximum training time or the maximum number of fitness evaluation. TypeLGP splits the 100,000 fitness evaluation into five independent runs, each run with 20000 fitness evaluations (i.e., evolving 80 generations with 250 individuals). Each run has a unique random seed, from 7 to 11. TypeLGP outputs five programs from each run. TypeLGP succeeds once one of the five solutions perfectly addresses all unseen test cases.

To solely verify the symbolic search ability of GP, we did not introduce those ARC tricks into TypeLGP, such as pre-training on the public training set, data augmentation [25], and ensemble strategies [16].

TABLE I
Proportion (%) of addressed tasks based on unseen inputs

Method	Training set	Evaluation set
Discrete Search Methods		
Ferré	24	5.75
GE	7.68	-
BFS-depth3	**26**	6.5
BFS-MLE	17.5	4.25
TypeLGP(ours)	21.75	**10.25**
LLM-based Methods		
Mirchandan	14	6.75
OpenAI GPT-4o	-	9
Google Gemini 1.5.	-	8

VI. Results

A. Performance of Unseen Task Inputs

Table I shows the score (the proportion of addressed tasks) of the comparison methods on the public training and evaluation sets. A task is addressed when all its unseen input-output pairs are perfectly addressed. First of all, TypeLGP achieves the highest score (i.e., 10.25%) on the evaluation set among the comparison methods. Although TypeLGP solved less training tasks than BFS-depth3 and Ferré, BFS-depth3 requires a huge amount of training time as it enumerates the whole search space within the limited program size, and Ferré likely suffers from overfitting with a performance drop of nearly 20% from the training set to the evaluation set. On the contrary, when BFS-MLE samples programs based on the learned program distribution rather than enumerating all possible programs, BFS-MLE has a performance drop on both the training and evaluation sets, whch is less competitive than TypeLGP.

TABLE II
SUCCESS RATE, PROPORTION OF ADDRESSED TASKS, AND AVERAGE TRAINING TIME BY TYPELGP

Input-output pairs	Training set		Evaluation set	
	Example	Unseen	Example	Unseen
Addressed tasks (%)	25.5	21.75	9.25	10.25
Success Rate (%)	63.92	58.62	44.32	38.42
Avg. Training Time	4.65 hours		7.3 hours	

Second, TypeLGP has a better performance than the LLM-based methods even without the expensive training budget. TypeLGP is better than Mirchandan on both the training and evaluation sets and is better than the two commercial LLM APIs on the evaluation set. Because GPT-4o and Gemini 1.5 used the given ground truth programs in the training set to fine-tune their models, they did not report their performance on the training set. More importantly, using LLM models is much more expensive than TypeLGP. As reported by [24], using LLM models to address ARC tasks consumes up to $10,000 per task, while TypeLGP is an open-sourced Python program running on a single CPU thread.

The results of TypeLGP are encouraging in fulfilling the abstraction and reasoning ability of AI models. TypeLGP competitively constructs the logical transformation based on a given search space and an affordable search budget.

B. Search Performance

To verify the search effectiveness of TypeLGP, this section investigates the search performance of TypeLGP on example input-output pairs, as shown in Table II TypeLGP addressed a task based on the example (or unseen) input-output pairs once it perfectly predicts all the example (or unseen) input-output pairs. The success rate implies how often TypeLGP makes perfect hits among the five random seeds.

First, the proportions of addressed tasks on evaluation set are similar (with a gap less than 4%) between the example and unseen input-output pairs. This implies that TypeLGP likely captures the essential logical transformation hinted by the training data. The output programs by TypeLGP have a good generalization ability as well.

Second, the success rates over the five random seeds on the addressed tasks (each random seed with 20000 fitness evaluations) are about 60% on the training set and about 40% on the evaluation set. This implies that TypeLGP at least addresses the task twice among the five random seeds for every successful task. TypeLGP has a relatively good robustness on different random seeds.

Finally, the average training time of TypeLGP on an AMD EPYC 7702 computing node (single thread) shows that the training budget of TypeLGP is cheap and affordable, compared to the common computation budget (12 hours).

To investigate the search efficiency of TypeLGP, Fig. 4 shows the median generations that TypeLGP addresses the example input-output pairs for those successful tasks. The bars show the number of tasks being addressed at a certain generation. We can see that most of the successful tasks are

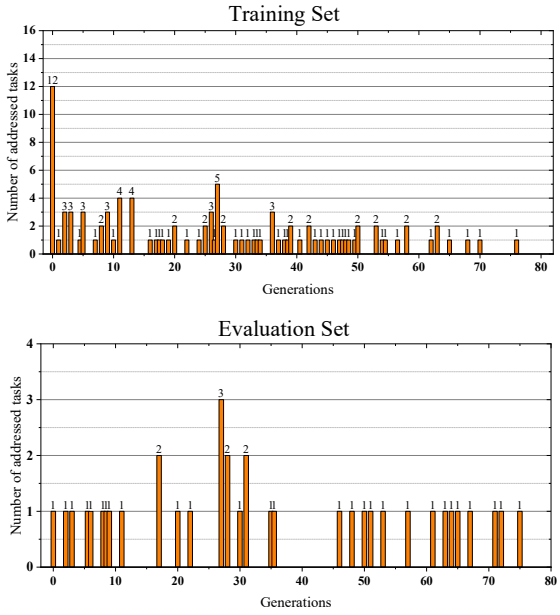

Fig. 4. The number of addressed tasks at each median generation.

```
//Ins 0: (R1= (replace I FIVE R3))
//Ins 1: (R2= (move I R0 TWO_BY_ZERO))
//Ins 2: (R2= (branch  (portrait I) R2  (smallest_obj I)))
//Ins 3: (R1= (uppermost_indices R3))
//Ins 4: (R2= (branch  (portrait I) R3  (most_color_obj I)))
Ins 5: (R0= (assign_right R0  (largest_obj R0)))
//Ins 6: (R0= (replace R1 TWO R1))
Ins 7: (R1= (move I R0 TWO_BY_ZERO))
Ins 8: (R2= (nonbackgrd_color_indices R1))
//Ins 9: (R3= (rot180 R0))
//Ins 10: (R1= (move I R0 TWO_BY_ZERO))
//Ins 11: (R1= (branch  (portrait I) R2  (smallest_obj R0)))
Ins 12: (R0= (toobject_indices R2 R0))
//Ins 13: (R3= (uppermost_indices R3))
//Ins 14: (R0= (identity R2))
//Ins 15: (R1= (move I R0 TWO_BY_ZERO))
Ins 16: (R0= (move I R0 TWO_BY_ZERO))
//Ins 17: (R2= (branch  (portrait I) R0  (most_color_obj I)))
//Ins 18: (R0= (toobject_indices R3 R0))
```

```
def program(I):
    // Initializing registers before program execution
    R0[Object] = assign_right(I, largest_obj(R0[Grid]))
    R1[Grid] = move(I, R0[Object], TWO_BY_ZERO)
    R2[Indices] = nonbackgrd_color_indices(R1[Grid])
    R0[Object] = toobject_indices(R2[Indices], R0[Grid])
    R0[Grid] = move(I, R0[Object], TWO_BY_ZERO)
    return R0[Grid]
```

Fig. 5. The raw output program of TypeLGP (upper) and its simplified version (lower) for task 3618c87e.

addressed by TypeLGP before the 60^{th} generation on both training and evaluation sets. Particularly, TypeLGP addressed quite a few tasks around the first 30 generations, indicating a good search efficiency of TypeLGP.

C. Example Program

To better understand TypeLGP search behaviors, this section investigates a successful program that solved the task 3618c87e, whose main idea is to move the blue pixels to the bottom. Fig. 5 shows the raw output program by TypeLGP and its simplified version. The output program has three main steps: 1) remove the grey pixels from the grid by moving the largest object 2 pixels toward the bottom; 2) locate the

blue pixels by extracting the non-background pixels from the resultant grid of step 1; 3) move the blue pixels in the input grids two pixels toward the bottom. Note that the grid type element of all registers is initialized as the input grid at the beginning.

There are two characteristics of TypeLGP programs that help symbolically search the vast search space. First, TypeLGP programs have neutral parts. The neutral sub-programs are actually the commented instructions in the raw output program, which do not contribute to the final output. Programs with the neutral part would have diverse potential building blocks without affecting programs' behaviors, which made a larger chance for TypeLGP to move out from local optima.

Second, TypeLGP programs reuse intermediate results building blocks. For example, `R0[Grid]` (i.e., the `Grid` data type of the register `R0`) is reused for twice in the first and forth instructions of the simplified program. The building block `move(I, R0[Object], TWO_BY_ZERO)` is reused in the second and fifth instructions. Reusing intermediate results and building blocks helps reduce the search space and express more different functions with shorter programs.

VII. Conclusions and Future Work

This paper explores the potential of GP for solving the challenging ARC tasks by proposing a TypeLGP method. The results imply that the population-based stochastic search of GP methods is a potential method for fulfilling the abstracting and reasoning ability of AI, with a much affordable training budget than many LLM methods. By taking advantage of neutral sub-programs and reusing building blocks, TypeLGP has a good global search ability in the vast DSL-based search space and outputs concise programs.

There are several future improvements for GP. First, it is necessary to design a more effective and efficient DSL for GP methods. The current DSL might be too low-level to make full use of the domain knowledge in training tasks. Second, cooperating GP methods with pre-training, data augmentation, and ensemble techniques are useful tricks that likely improve GP performance for ARC tasks

References

[1] F. Chollet, "On the Measure of Intelligence," 2019, arXiv:1911.01547 [cs]. [Online]. Available: https://arxiv.org/abs/1911.01547

[2] J. Pourcel, C. Colas, and P.-Y. Oudeyer, "Self-Improving Language Models for Evolutionary Program Synthesis:A Case Study on ARC-AGI," in *Proceedings of the 42 nd International Conference on Machine Learning*. PMLR 267, 2025, pp. 49 659–49 688.

[3] J. R. Koza, *Genetic Programming : On the Programming of Computers By Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[4] T. Hu, W. Banzhaf, and G. Ochoa, "How Neutrality Shapes Evolution: Simplicity Bias and Search," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Jul. 2025, pp. 1008–1016.

[5] Z. Huang, Y. Mei, F. Zhang, M. Zhang, and W. Banzhaf, "Bridging fitness with search spaces by fitness supremums: A theoretical study on lgp," 2025, arXiv:2505.21991 [cs]. [Online]. Available: https://arxiv.org/abs/2505.21991

[6] J. Zhong, J. Dong, W.-L. Liu, L. Feng, and J. Zhang, "Multiform Genetic Programming Framework for Symbolic Regression Problems," *IEEE Transactions on Evolutionary Computation*, vol. 29, no. 2, pp. 429–443, 2025.

[7] Z. Huang, "Advanced Linear Genetic Programming and Applications to Dynamic Job Shop Scheduling," Ph.D. dissertation, Victoria University of Wellington, 2025.

[8] R. Fischer, M. Jakobs, S. Mücke, and K. Morik, "Solving Abstract Reasoning Tasks with Grammatical Evolution," 2020. [Online]. Available: https://ceur-ws.org/Vol-2738/LWDA2020_paper_8.pdf

[9] C. Ryan, J. Collins, and M. O. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Proceedings of European Conference on Genetic Programming*, 1998, pp. 83–96.

[10] M. Hodel, "Domain-Specific Language for the Abstraction and Reasoning Corpus," 2023. [Online]. Available: https://github.com/michaelhodel/arc-dsl

[11] N. Butt, B. Manczak, A. Wiggers, C. Rainone, D. W. Zhang, M. Defferrard, and T. Cohen, "CodeIt: Self-Improving Language Models with Prioritized Hindsight Replay," 2024, arXiv:2402.04858 [cs]. [Online]. Available: http://arxiv.org/abs/2402.04858

[12] Y. Xu, E. B. Khalil, and S. Sanner, "Graphs, Constraints, and Search for the Abstraction and Reasoning Corpus," in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, 2023, p. 8.

[13] G. H. B. Costa, M. Freire, and A. L. Oliveira, "The role of positional encodings in the ARC benchmark," 2025, arXiv:2502.00174 [cs]. [Online]. Available: http://arxiv.org/abs/2502.00174

[14] F. M. Rocha, I. Dutra, and V. S. Costa, "Program Synthesis using Inductive Logic Programming for the Abstraction and Reasoning Corpus," 2024, arXiv:2405.06399 [cs]. [Online]. Available: http://arxiv.org/abs/2405.06399

[15] K. Ellis, L. Wong, M. Nye, M. Sablé-Meyer, L. Cary, L. Anaya Pozo, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum, "DreamCoder: growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 381, no. 2251, p. 20220050, 2023.

[16] M. Bober-Irizar and S. Banerjee, "Neural networks for abstraction and reasoning: Towards broad generalization in machines," 2024, arXiv:2402.03507 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2402.03507

[17] H. Lee, S. Kim, S. Lee, S. Hwang, J. Lee, B.-J. Lee, and S. Kim, "ARCLE: The Abstraction And Reasoning Corpus Learning Environment For Reinforcement Learning," 2024, arXiv:2407.20806 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2407.20806

[18] R. Wang, E. Zelikman, G. Poesia, Y. Pu, N. Haber, and N. D. Goodman, "Hypothesis Search: Inductive Reasoning with Language Models," 2024, arXiv:2309.05660 [cs]. [Online]. Available: http://arxiv.org/abs/2309.05660

[19] Y. Xu, W. Li, P. Vaezipoor, S. Sanner, and E. B. Khalil, "LLMs and the Abstraction and Reasoning Corpus: Successes, Failures, and the Importance of Object-based Representations," 2024, arXiv:2305.18354 [cs]. [Online]. Available: http://arxiv.org/abs/2305.18354

[20] K. Singhal and G. Shroff, "ConceptSearch: Towards Efficient Program Search Using LLMs for Abstraction and Reasoning Corpus (ARC)," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39(28), 2025, pp. 29 493–29 494.

[21] J. Ainooson, D. Sanyal, J. P. Michelson, Y. Yang, and M. Kunda, "An Approach for Solving Tasks on the Abstract Reasoning Corpus," 2023, arXiv:2302.09425 [cs]. [Online]. Available: http://arxiv.org/abs/2302.09425

[22] S. Ferré, "Tackling the Abstraction and Reasoning Corpus (ARC) with Object-centric Models and the MDL Principle," 2023, arXiv:2311.00545 [cs]. [Online]. Available: http://arxiv.org/abs/2311.00545

[23] S. Mirchandani, F. Xia, P. Florence, B. Ichter, D. Driess, M. G. Arenas, K. Rao, D. Sadigh, and A. Zeng, "Large Language Models as General Pattern Machines," 2023, arXiv:2307.04721 [cs]. [Online]. Available: http://arxiv.org/abs/2307.04721

[24] F. Chollet, M. Knoop, G. Kamradt, and B. Landers, "ARC Prize 2024: Technical Report," 2025, arXiv:2412.04604 [cs]. [Online]. Available: http://arxiv.org/abs/2412.04604

[25] C. Hocquette and A. Cropper, "Relational Decomposition for Program Synthesis," in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, 2025, pp. 4526–4534.