DECOMPOSING ARC PROGRAMS TO CREATE SIMPLER TASKS

Matthew Simpson & Soumya Banerjee
Department of Computer Science and Technology
University of Cambridge
Cambridge, UK
{mgws3,sb2333}@cam.ac.uk

ABSTRACT

We introduce a program-driven method to augment the ARC (Abstraction and Reasoning Corpus) training set by decomposing ground-truth DSL solutions at split points: locations where a function returns an intermediate grid that fully captures prior computation. From each split point, we synthesize two new tasks: (1) the left subprogram with the intermediate grid as the target, and (2) the right subprogram with that intermediate grid as the input. By recursively applying this procedure and filtering for variable- and dependency-safe splits, our pipeline produces tasks that are grounded in both the original ARC distribution and conceptually simpler than their parents tasks. Applied to the original training set, our method yields 366 new unique tasks; when layered on top of Butt's Codeit mutations (20,000 tasks), it produces 6,011 unique tasks (of which 3,634 are distinct programs). Generated tasks are shorter on average, consistent with lower DSL-program length being a proxy for reduced difficulty. Qualitative case studies show the decomposition often isolates natural "mental steps" in ARC problems, suggesting a route to explicit curricula for solvers. We discuss limitations (method likely conservatively splits and only at explicitly typed grid-returning functions) and outline extensions (empirical evaluation of solver improvements). Our method complements existing task-generation techniques by producing interpretable, stepwise tasks that can help probe and train program- and model-based ARC solvers.

1 INTRODUCTION

The Abstraction and Reasoning Corpus (ARC) Chollet evaluates a system's ability to infer transformations that map input grids to output grids from a small number of examples. A common representation for these transformations is as programs written in a domain-specific language (DSL), where a sequence of predefined operations transforms the input grids to the output grids.

A major challenge of learning-based approaches to ARC is the limited number of available training tasks. Several works have therefore explored methods for generating additional ARC tasks. Existing approaches fall broadly into three categories.

First, **input augmentation** methods generate new input–output grid pairs while preserving the underlying transformation. For example, RE-ARC Hodel (2024) uses procedural generation to create new inputs that remain consistent with the original task transformation. Second, **transformation mutation** approaches retain the original inputs while modifying the transformation itself. CodeIt Butt et al. mutates DSL programs that represent ARC tasks by replacing functions with alternatives that return similar types, while AugARC Bikov et al. (2025) applies predefined transformations such as rotations or permutations. Third, **task synthesis** methods construct entirely new tasks by generating both new inputs and transformations. ConceptARC Moskvichev et al. creates tasks grouped around core ARC concepts, while other work constructs ARC-style tasks either manually Kim et al. or using large language models Li et al..

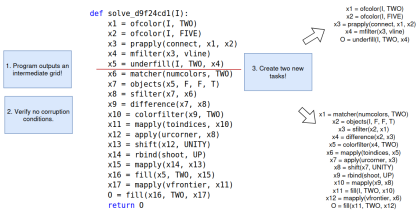

Figure 1: Overview of the task generation procedure. At each intermediate grid (split-point), we create two new tasks: one with the intermediate grid as the output (left program), and one with it as the input (right program) provided it does not violate the conditions we outlined. This procedure can be applied recursively to generate further tasks.

However, these approaches either maintain or increase task complexity, or produce tasks whose relationship to the original ARC distribution is uncertain. In particular, they do not generate simpler tasks derived directly from existing ARC tasks.

2 METHOD

Our method operates on the ground-truth DSL programs for ARC training tasks provided by the Hodel DSL representation Hodel (2024). In this representation, ARC transformations are expressed as programs that apply a sequence of pre-defined operations to transform an input grid into an output grid.

We generate new ARC tasks by exploiting a structural property of many such programs: the presence of intermediate grid outputs. In many cases, the DSL programs produce intermediate grids through functions that return grid objects. These intermediate grids often fully capture the computation performed up to that point, meaning the preceding operations are no longer required to describe the full transformation. We refer to any program location where a function returns a grid as a *split point*.

At each split point we derive two new tasks. The first corresponds to the prefix of the program up to the split point, with the intermediate grid used as the target output. The second uses the intermediate grid as the input while retaining the remainder of the original program to produce the final output. This effectively decomposes the original transformation into two simpler sub-transformations. The procedure can also be applied recursively to newly generated programs to create additional tasks.

Ideally, a new task could be created at every function that returns a grid. However, program dependencies restrict where valid splits can occur. For the left program (the prefix before the split point), all variables must remain used in the resulting program to avoid introducing redundant computations that could dilute the new ground truth programs. For the right program (the suffix after the split point), no variables from the left program can be referenced; otherwise the intermediate grid would not fully capture the preceding computation.

Once these conditions are satisfied, the resulting programs are reformatted to match the Hodel DSL convention. In particular, variables are renamed sequentially (e.g., $x_1, x_2, \ldots$) and the final output variable is standardized as O. The programs are then added to a duplicate-free queue, allowing the procedure to be applied recursively so that additional split points within generated programs can also be used.

A limitation of the current implementation is that it only considers functions that explicitly return grid objects. In the Hodel DSL, return types can also be implicit (e.g., functions with an `any`-type return value). In such cases it is not immediate whether a function produces a grid, which prevents these locations from being used as split points. Extending the method with a type inference mechanism would likely increase the number of tasks that can be generated.

3 RESULTS

We evaluate our method with respect to the following research questions:

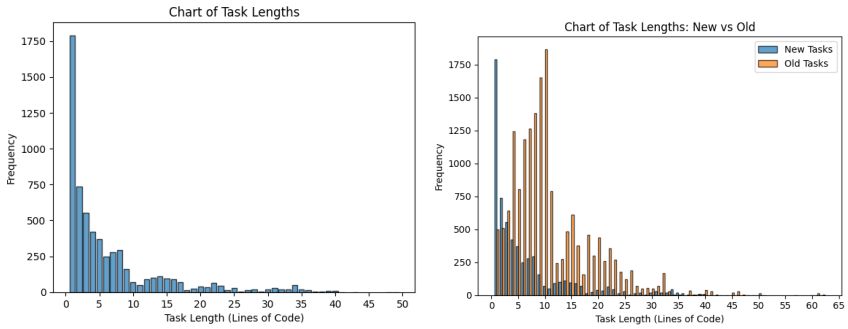

Figure 2: Distribution of task lengths for generated tasks. (Left) Length distribution of the 6011 newly created tasks. (Right) Comparison between new task lengths and those of the original training set.

- **RQ1:** How many new tasks does our method generate, and what are their statistical properties? Additionally, can the method be effectively combined with other task generation approaches?
- **RQ2:** Are the generated tasks qualitatively different from existing ARC tasks?

To answer these questions, we apply our method in two experimental settings. First, we run the method directly on the ARC training set. Second, we apply the method to the tasks produced by the CodeIt mutation procedure Butt et al.. This allows us to evaluate the behaviour of our approach both in isolation and when combined with an existing task generation pipeline.

3.1 Task Properties

3.1.1 Summative Statistics of Tasks Generated

Using only the ARC training set as input, our method generates **366** unique tasks, nearly doubling the size of the original training dataset. When applied to the tasks produced by CodeIt's mutation procedure, which contains approximately 20,000 tasks including the original training set, our method generates **6011** unique tasks. For the remainder of this analysis, we focus on the **6011** generated tasks.

To examine the structural properties of the generated tasks, we analyse the distribution of program lengths, shown in Figure 2. Compared to the original ARC task distribution, the generated tasks are generally shorter, resulting in a noticeably skewed distribution.

A potential concern is that many of these short tasks correspond to identical transformations applied to different inputs. When counting the number of unique programs generated, we find that **3634** of the 6011 tasks contain unique programs. The remaining have the same underlying function yet different inputs, which may still be useful training examples. Particularly considering there exists categories of approaches which aim to achieve just this (category 1 in our introduction).

We now investigate the origin of the tasks created.

3.1.2 Task Origin

Figure 3 shows the distribution of new tasks generated per original task. We observe a long-tailed distribution: many tasks generate no new tasks, followed by tasks generating only a single new task, while a small number produce up to six new tasks. This pattern can be explained by differences in program structure. Short programs often contain no intermediate grid outputs and therefore cannot be split, resulting in zero generated tasks. In contrast, longer programs are more likely to contain multiple intermediate grids, enabling several valid split points and consequently generating more tasks.

To further analyse how tasks are produced by the pipeline, we track the origin of generated programs, summarised in Tables 1 and 2. Of the generated tasks, **3310** correspond to left programs and **2701**

Left Programs	Right Programs
3310	2701

Table 1: Number of tasks generated by program position.

Inter-Split	First Pass
947	5064

Table 2: Number of tasks by generation type.

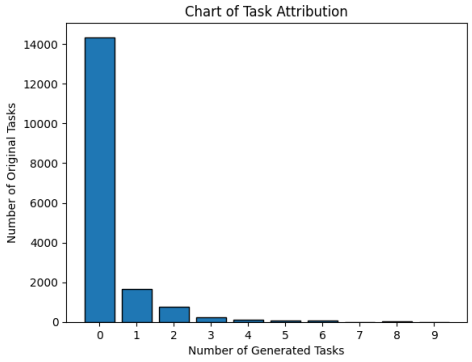

Figure 3: Distribution of task origin and types

correspond to right programs. This imbalance is expected, as the constraints for generating a valid right program are stricter than those for generating a left program.

Finally, recursive application of the splitting procedure produces additional tasks beyond the first pass. In total, **947** tasks are generated through such inter-split applications, while **5064** tasks are produced during the initial pass.

3.2 Case Study: Task Decomposition via Splitting

We provide a qualitative walk-through of representative generated tasks. We begin with the example shown in our visual diagram, task d9f24cd1. The original task, its corresponding program, and the two derived subtasks are illustrated in Figure 4.

In this case, the decomposition aligns closely with two intuitive "mental steps" that a human solver might follow when solving the task. Notably, although both subtasks correspond to conceptually similar transformations, their program lengths differ substantially.

This example illustrates two broader characteristics of ARC tasks. First, the number of implicit reasoning steps within a task is not fixed, and can vary even within a single example. This contributes to task difficulty, particularly when training examples—intended to be simpler than the test sets—still require multiple such steps. Second, these steps are often entangled within a single program, raising the question of how a learning system can effectively disentangle and represent them. Our decomposition procedure provides a mechanism for explicitly isolating these intermediate transformations. Furthermore, the disparity in program lengths between subtasks supports prior findings Li et al. that certain ARC concepts are inherently more difficult to express as functions, even when they appear conceptually simple.

Figure 4 also illustrates a limitation of our method. Here, the decomposition procedure is applied to both an original training task and a mutated variant derived from it. Although the two tasks differ at the level of the full program, their initial steps are highly similar. As a result, the resulting left-program subtasks are themselves highly similar, differing only by a colour change and hence adding little variation to the dataset. This highlights that our method is sensitive to similarities in the early stages of the programs it is applied to.

All generated tasks are made publicly available and can be explored in the accompanying notebook (code available at: https://github.com/MGWSimpson/AlphaARC), enabling further analysis and use in training.

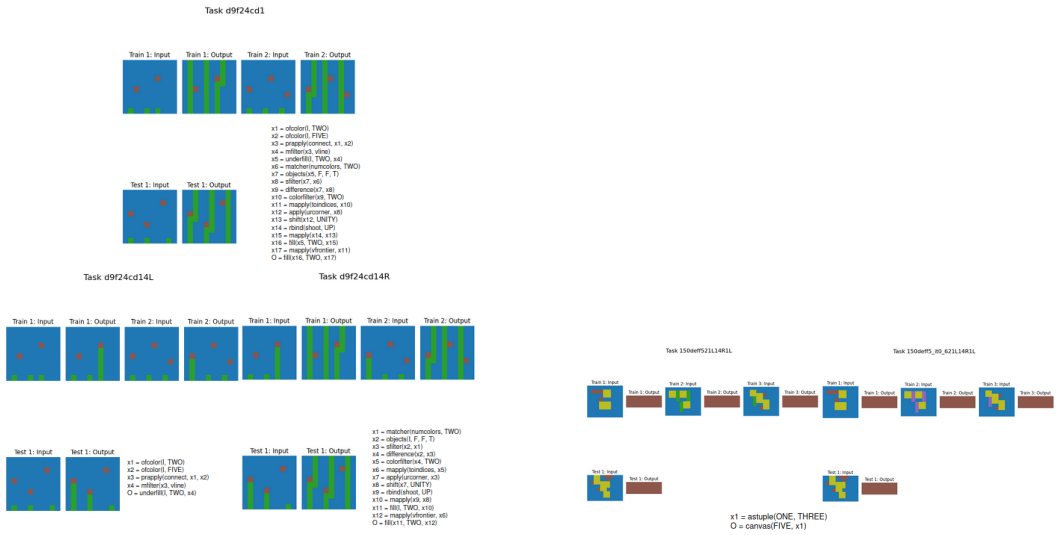

Figure 4: Task decomposition examples. Left: decomposition aligns with intuitive reasoning steps. Right: limitation where similar prefixes yield near-duplicate subtasks.

4 Discussion

In this work, we introduce a program decomposition approach for generating new ARC tasks, yielding 6,011 new ARC tasks when applied to existing datasets. Compared to prior task generation approaches, our method yields fewer tasks overall: AugARC Bikov et al. (2025) generates up to 18 million new tasks (depending on the transformations applied), while CodeIt Butt et al. produces 19,600. However, the generated tasks differ in structure, and we identify three advantages based on quantitative and qualitative analysis.

First, our new tasks explicitly decompose the multi-step reasoning required in ARC tasks into shorter, more interpretable steps, while remaining grounded in the original task. This decomposition may enable future work to identify the cognitive priors needed to solve ARC tasks by isolating individual reasoning steps, that have thus far remained implicit. While similar analysis has been explored in ConceptARC Moskvichev et al., our method operates directly on the original ARC dataset rather than by constructing a proxy dataset.

This may have implications for learning-based solvers. At the time of writing, the most successful DSL-based learning approach is CodeIt Butt et al., a method based on the Expert Iteration algorithm Anthony et al. (2017). Analysis by Gulcehre Gulcehre et al. (2023) suggested the success of methods based on Expert Iteration arises from the formation of an implicit curriculum. Our approach offers a way to construct an explicit curriculum: by learning to solve the simpler decompositions of tasks first, models may acquire the capabilities needed for more complex ones.

Second, DSL-based ARC solvers (e.g., CodeIt Butt et al.; Bober-Irizar & Banerjee) tend to solve shorter programs, as search cost grows exponentially with length. Consequently, many DSL-based solvers succeed on a similar subset of ARC tasks. By generating additional short-program tasks, our method may enable finer-grained differentiation between program synthesis approaches, unlike prior work Butt et al.; Bikov et al. (2025), which maintains or increases program length.

Finally, our approach can be applied to any task with a ground truth program.

We identify three avenues for future work: First, our current method handles only explicitly typed functions, whereas the Hodel DSL also includes flexible typing. Extending the approach with a static type inference mechanism would allow additional valid split points to be considered. Second, our method is likely highly conservative, and extensions could further compress programs to create new tasks; addressing this would likely require drawing on program analysis techniques. Finally, the impact of the generated tasks on ARC solver performance remains to be evaluated.

REFERENCES

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search, 2017. URL https://arxiv.org/abs/1705.08439.

Kiril Bikov, Mikel Bober-Irizar, and Soumya Banerjee. Augarc: Augmented abstraction and reasoning benchmark for large language models. 2025.

Mikel Bober-Irizar and Soumya Banerjee. Neural networks for abstraction and reasoning: Towards broad generalization in machines. URL http://arxiv.org/abs/2402.03507.

Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David W. Zhang, Michaël Defferrard, and Taco Cohen. CodeIt: Self-improving language models with prioritized hindsight replay. URL http://arxiv.org/abs/2402.04858.

François Chollet. On the measure of intelligence. URL http://arxiv.org/abs/1911.01547.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. Reinforced self-training (rest) for language modeling, 2023. URL https://arxiv.org/abs/2308.08998.

Michael Hodel. Addressing the abstraction and reasoning corpus via procedural example generation, 2024. URL https://arxiv.org/abs/2404.07353.

Subin Kim, Prin Phunyaphibarn, Donghyun Ahn, and Sundong Kim. Playgrounds for abstraction and reasoning.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, Wei-Long Zheng, Zenna Tavares, Yewen Pu, and Kevin Ellis. Combining induction and transduction for abstract reasoning. URL http://arxiv.org/abs/2411.02272.

Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. The ConceptARC benchmark: Evaluating understanding and generalization in the ARC domain. URL http://arxiv.org/abs/2305.07141.