Decomposing ARC Programs to Create Simpler Tasks

Matthew Simpson
Department of Computer Science
University of Cambridge
UK

Soumya Banerjee
Department of Computer Science
University of Cambridge
UK
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.

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 and include 3,310 left-programs, 2,701 right-programs, with 947 additional inter-split derivations from repeated application. 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 (current implementation only handles explicitly typed grid-returning functions and is conservative in splitting) and outline extensions (static type inference for the Hodel DSL, more aggressive program compression, and 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. Our code is available from here: https://github.com/MGWSimpson/AlphaARC

1 Introduction

A core challenge of solving ARC tasks with Machine Learning based approaches come from the limited number of training examples. To address this, there exist many works which seek to create new ARC tasks. We identify three categories which existing methods fall into.

The first seeks to generate new input-output grid pairs, whilst retaining the same underlying transformation. Hodel introduced RE-ARC Hodel [2024], which uses procedural generation to create new input grids consistent with the underlying transformation.

The second category of approaches retain the original input, yet change the underlying function. Butt Butt et al. takes this approach through mutating Hodel's DSL solutions by replacing functions which returns a particular data type, with a different function which returns the same type. Bikov Bikov et al. [2025] achieves this by applying pre-defined transformations to the tasks, such as rotation or permutation of order.

Preprint.

The final category does both, generating both new input grids and corresponding transformation functions. Moskvichev et al., created ARC tasks grouped by concepts, based on the core priors Chollet originally outlined. Kim et al. manually created a dataset of simpler tasks restricted to a grid size of 5x5. Li et al. created new tasks by prompting a powerful LLM (GPT4) to augment 100 manually created python program tasks.

In this work we introduce a new task-generation technique that falls into category 2 of our taxonomy: it keeps the original input grids while synthesising a new transformation. Prior methods in this category, notably those of Butt and Bikov mutate the program but require the same (or higher) number of transformations. Whilst Moskichev and Kim hand-craft easier tasks as these are authored from scratch, their connection to the core ARC distribution is uncertain. There exists a gap then to construct simpler tasks by augmenting the original training set. Our method bridges that gap by decomposing existing ARC solutions into smaller sub-programs. As the procedure requires only the ground-truth program of a task, it can be layered atop any program generation pipeline. When combined with Butt's mutation approach, our methods yield **6011** new ARC tasks.

Our code is available from here: `https://github.com/MGWSimpson/AlphaARC`

2 Method

We propose a novel method for generating new tasks in ARC by exploiting a key structural property of many ARC DSL solutions: the presence of intermediate output grid. There exists many cases where the DSL program will produce an intermediate grid through the application of a function which returns a grid type. Often, these intermediate grids fully describe the previous functions which lead to them. The intermediate grid hence contains all the required information up to that point, and the previous lines are in essence redundant. We refer to these points as split points. At a given split point, we can create two new tasks. One is the original program up to the split point, however, with the intermediate grid as the target output. We can double the number of tasks we create with the observation that we can create a new task from the other half of the split point, in this case, setting the input grid to the intermediate grid and retaining the original output grid. Finally, we can recursively apply this procedure in-between split-points to create further tasks.

Idealistically, we wish to create new programs on every function which returns a grid. However, the main obstacle to overcome is program dependencies. When considering a program up to the split-point (which we refer to as the left-program), we wish for every variable present in the sub-program to be utilized. We believe that having this not be the case would lead to diluted programs, which could harm ML based methods which train on the new tasks. The other dependency to consider is the program after the split-point (or the right program) as these programs must not make reference to any variable in the left-program, otherwise the new input-grid does not fully describe the previous lines, and hence cannot be removed.

Once we have handled either of these conditions for creating new programs, we reformat them so they match the Hodel DSL format and adds them to a duplicate-free queue to be reprocessed allowing for "inter" split-points to be used. A visual overview is provided in figure 1.

There is a notable limitation of our implementation we wish to make explicit. The current version only considers functions which explicitly return grid functions, not cases where the type is left implicit (such as any type functions). We initially attempted to create a type-inference mechanism, however the complications it ensued were outwith the scope of a Machine Learning based thesis. This improvement would increase the scope of generatable tasks.

3 Results

We wish to understand the following questions when considering this method:

- **RQ1:** How many new tasks do we generate, and what are the properties of them? Can this method be combined with other task generating methods?
- **RQ2:** Are these tasks qualitatively different from previous tasks?

To answer these questions we deploy our method in two settings: 1) we use our method on purely the training set. 2) We use our method on the generated tasks from the Codeit mutation procedure.

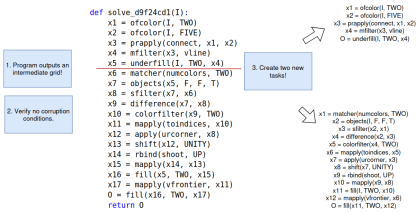

Figure 1: Overview of the task generation procedure. At each intermediate grid (split-point), we create two new tasks: one with the intermediate grid as the output (left program), and one with it as the input (right program). Assuming it does not violate the conditions we outlined. This procedure can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

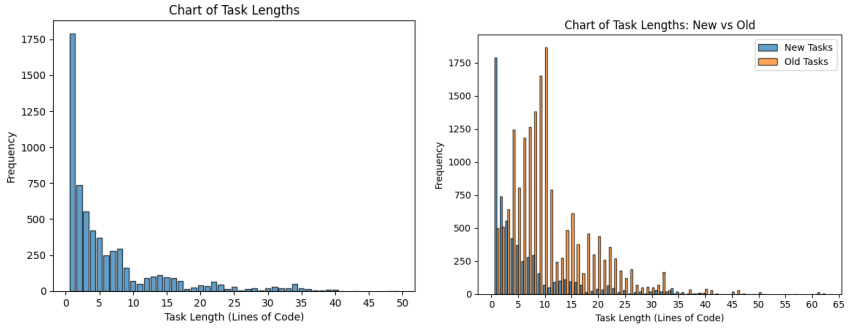

Figure 2: Distribution of task lengths for generated tasks. (Left) Length distribution of the 6011 newly created tasks. (Right) Comparison between new task lengths and those of the original training set.

3.1 Task Properties

3.1.1 Summative Statistics of Tasks Generated

We find that when using our method on just the training set, we generate a total of **366** unique new tasks, nearly doubling the training data. When applied to the tasks generated by Butt (which totals 20,000 tasks including the original training set), our method generates a total **6011** unique tasks. For future analysis we will focus on the **6011** generated tasks for brevity.

Examining the properties of tasks, we plot the lengths of the newly created tasks, visible in figure 2. Comparing this to the original task distribution, the newly generated tasks are generally shorter in length, as seen by the skewed distribution. As in DSL based solvers, task length corresponds with difficulty, we can claim that the generated tasks are generally easier than the original training set, aligning with our goal.

One concern we have is that many of these short tasks will be duplicate tasks, that is, they have the same underlying transformation function but different inputs. When counting the number of unique programs generated, we find that **3634** of the original 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

In figure 3, we plot the distribution of new tasks generated per tasks. We observe a long tail distribution, with many tasks generating 0 new tasks, followed closely by generating just one, however, some generate as many as six. When considering the original task lengths, we attribute this

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.

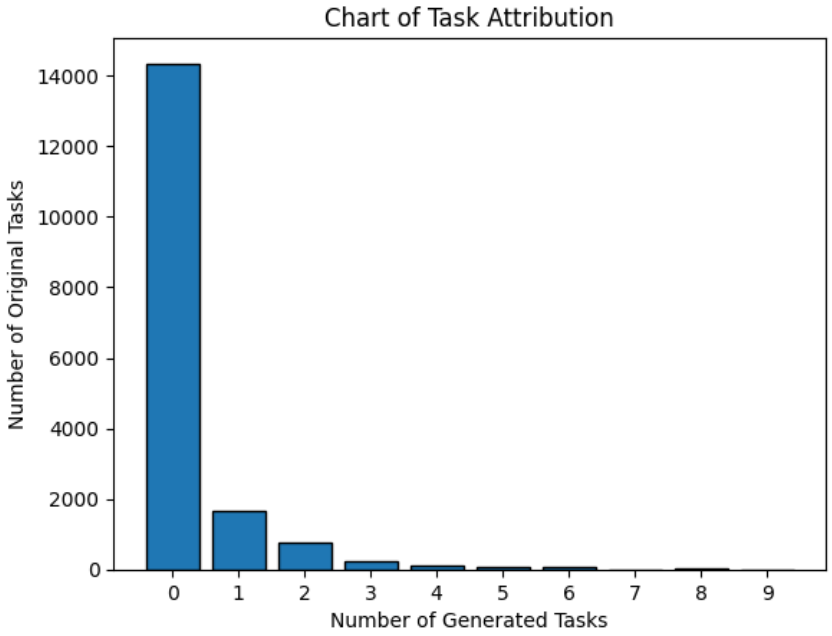

Figure 3: Distribution of task origin and types

to short programs not containing intermediate output grids generating 0 tasks, whereas the longer programs, which are likely to contain many sub programs, generating the greater number of tasks.

We wish to briefly evaluate the need for the different components in our system, with a summary of results presented in tables 1 and 2. Tracking their generation origin point, we find that **3310** tasks are left programs whereas **2701** tasks are right programs. This aligns with expectations, considering the conditions for generating a right program are stricter than generating a left program. Finally, repeated applications was warranted, generating an additional 947 tasks.

3.2 Case Study: Task Decomposition via Splitting

Finally, we wish to provide a qualitative walk-through of some of the newly generated tasks. The first case we will examine is the case outlined in our visual diagram, task d9f24cd1. The task and it's program, and the two tasks it creates, can be seen in figure 4. It can be seen that the decomposition of this specific task, maps cleanly to the two mental "steps" one may undergo when attempting to solve an ARC task. We note that despite being composed of two broad steps, the program lengths differ significantly, despite doing something conceptually similar. The process we applied reveal two interesting characteristics about ARC tasks. First, within any given task, the number of "steps" one must undergo can vary, this adds additional complexity, particularly when the training set (which should be easier), is composed of many of these steps. It is natural to ask how a system may learn to neatly decompose these steps when they are combined in this way. We hope our method is a step towards this. The fact that both steps programs vary greatly, contribute to previous findings by Li et al. that certain ARC concepts are difficult to express in a function.

We also examine a task which share a common transformation. In this case, we picked a worst case scenario, where the change to the input grid is fairly minimal (consisting of a colour change). Of course, this highlights that our method inherits any weakness of methods that is applied upon, leading to some tasks which do not differ significantly. We make all tasks available for viewing

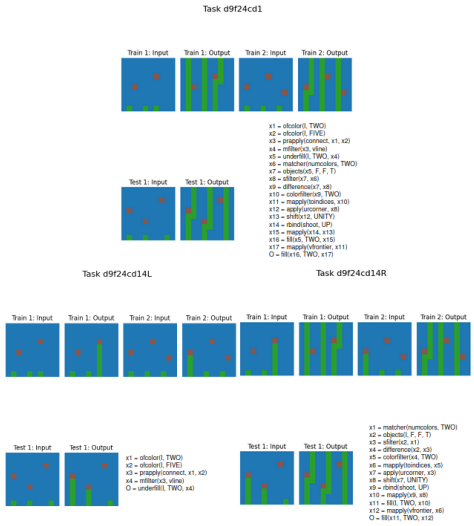

Figure 4: Example of task decomposition. The top task is decomposed into two sub tasks pictured below.

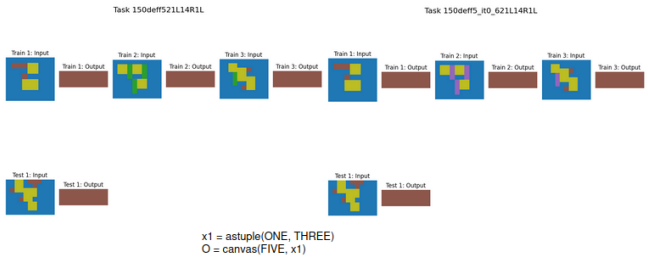

Figure 5: Case study showing a limitation of the task generation method.

within the notebook titled task creation and all tasks are released to the public for additional training and analysis.

4 Discussion

In this work, we propose a novel method for generating ARC tasks, leading to **6011** new ARC tasks. These tasks are shorter in length, and decompose the typical "mental steps" that ARC tasks comprise of. Our hope is that these tasks will allow for better benchmarking of previous approaches, and provide more training data for model-based methods.

Considering other work on task-generation through function transformation, such as that by Bikov et al. [2025] and Butt et al. our method generates fewer overall tasks. Bikov generates between 2000 and up to over 18 million tasks depending on the transformations applied, whereas Butt generates 19600 tasks, with ours only generating 366 tasks (from the original training set). However, through both qualitative analysis and examining task distributions, we believe we have three meaningful advantages over prior approaches.

First, our tasks explicitly decompose the multi-step reasoning often required in ARC tasks, making them shorter and more interpretable, whilst still being grounded in the core task. This decomposition may allow future work to identify the specific cognitive priors needed to solve ARC tasks, which hitherto have been left implicit, by isolating the individual steps involved. While similar analysis has been attempted in ConceptARC Moskvichev et al., our method performs this decomposition directly on the original ARC dataset rather than creating a proxy dataset. This has implications for solver design, particularly models based on Expert Iteration Anthony et al. [2017], such as Butt's Codeit Butt et al. Gulcehre Gulcehre et al. [2023], suggesting that the success of these methods arises from the formation of an implicit curriculum. Our approach offers a way to construct an explicit curriculum: by learning to solve simpler tasks first, models may build the capabilities needed for more complex ones.

Second, our tasks are simpler, which should allow for further stratification between the strengths of solvers, constrasting with Butt and Bikov's approaches, which either maintain task complexity or increase it.

Finally, like prior methods, our approach can be applied to any task with a ground truth program, and as demonstrated, can be used complementary to prior and future techniques.

Our method is simple, and there exist many extensions which would create more tasks. We identify two such cases for future work. The first is that our method handles only explicitly typed functions, however, there exists flexible typing within the Hodel DSL. The creation of a static type interpreter for the Hodel DSL would address this issue. The other is that our method is likely highly conservative, and there exist many ways with which to further compress programs and consequently create new tasks. Addressing this would likely require drawing on the literature on compilers, which is outside the scope of this work.

References

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

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`.

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

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`.

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.

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.

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.