Efficient learning through programmatic representations

Improving transformation search in BEN

Daria Condratov[1]
Supervisor(s): Sebastijan Dumančić[1], **Dekel Zak**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Abstract

The Abstraction and Reasoning Corpus (ARC) challenges systems to recognise patterns from just a few input-output examples which proves to be a setting where most current approaches struggle. Program synthesis systems like BEN tackle this by decomposing the images into objects rather than pixels and then searching for transformation programs that explain the examples. However, its reliance on exhaustive enumeration makes the transformation search computationally expensive and difficult to scale.

This project aims to improve the transformation search and, in doing so, asks whether correspondence-tailored grammar pruning can reduce the search space and improve the efficiency of BEN's conquer step. BEN is reimplemented in Julia using the Herb.jl library. On top of that, a pruning strategy that exploits structural similarities between matched input-output object pairs is added. Both the baseline and the improved version are evaluated on the 400-task ARC training set.

The pruning reduces the average number of candidate programs evaluated by 77%, but it has a seemingly slight negative impact on the tasks that get solved. The number of correctly solved tasks decreases from 31 to 30. These results show that simple structural observations about the matched object pairs substantially reduce the search space. More informed search looks like a promising direction for improving program synthesis systems on ARC.

1 Introduction

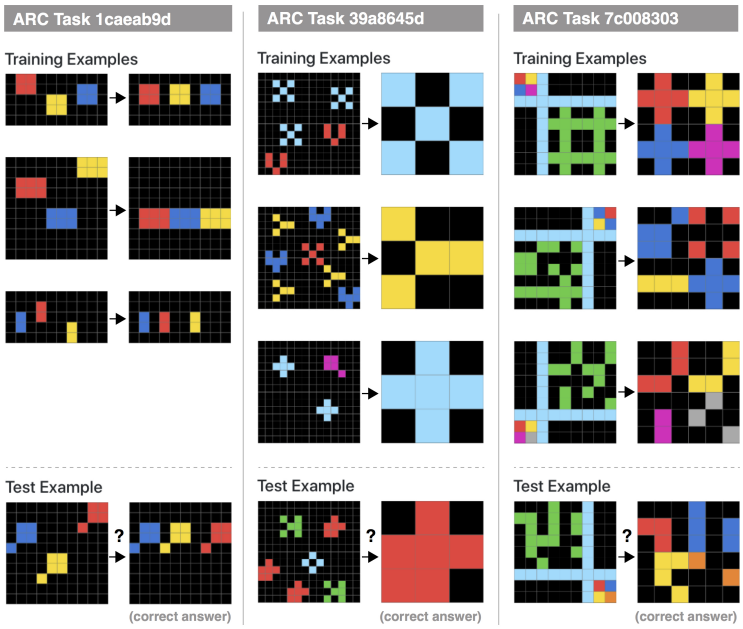

Figure 1: ARC tasks [1]. On the left, you are expected to align the objects at the blue one's level. In the middle, to pick the object that appears the most times. Finally, the task on the right requires you to apply the coloring scheme to the green pattern.

Computers are known to excel at learning from millions of examples, but when shown just a few of those, things tend to get a lot more difficult. The Abstraction and Reasoning

Corpus (ARC) aims to test AI systems at just that: finding patterns in puzzles from just 2–10 input-output image pairs and applying those to a new input [2]. A few examples of such tasks can be seen in Figure 1.

Program synthesis has proven to be a useful approach to solving ARC. Program synthesis represents the problem of automatic generation of programs based on a specification [3]. It is characterized by three components: its specification, the search space and search strategy. The aim is to find a solution that matches this specification from the search space using the search strategy to achieve that. The specification can be a set of input-output examples.

BEN is such a program synthesis system for solving ARC tasks. Its core idea is to mirror how humans reason about these tasks by searching for programs that transform input into output. It addresses the scalability challenge by decomposing each ARC task into smaller sub-problems [4]. BEN has a three-stage "Divide, Align and Conquer" approach. Rather than operating on a pixel-by-pixel level, it divides the input and output images into objects, also known as the divide step. Then, in the align part, it uses analogical reasoning to match corresponding objects across the input and output, prioritizing pairs that share the most structural similarity. These correspondences are then sent to the final conquer step to find their transformation programs and the conditions under which they apply. Each step represents an independent part.

BEN's transformation search, which is found in the conquer step, currently relies on exhaustive enumeration to find transformation programs. Despite the fact that this strategy has proven to be quite efficient [3], it still scales poorly. As the required program depth increases, the search space grows exponentially and quickly becomes infeasible within a time limit.

The research question is "How can we better find the necessary transformation programs?" This project investigates whether pruning the grammar based on the correspondence at hand (the matched input-ouput objects pair) can reduce the number of programs evaluated before exiting the transformation search, improving both BEN's solve rate and its runtime. To this end, BEN is reimplemented in Julia using Herb.jl [5], which is to represent a baseline, and the pruning mechanism add-on is then evaluated on the ARC training set against said baseline.

The main contributions of this work are: (i) a reimplementation of BEN's transformation search in Julia adapted to use the Herb library; (ii) a pruning strategy implementation add-on; and (iii) empirical evidence for whether this optimisation yields any gains in programs evaluated and/or solve rate.

The remainder of this paper is structured as follows. Section 2 provides background on program synthesis, BEN's pipeline and possible directions for improvements. Section 3 details methodology, including the reimplementation and improvements to the transformation search. Section 4 presents the experimental setup and results. Section 5 concludes with directions for future work, and Section 6 addresses responsible research considerations.

2 Background

Program synthesis is the task of automatically constructing a program that satisfies what the user asks for. A synthesizer has three main components: the specification used to express the intent of the user, the space of candidate programs searched, and the search technique employed [3]. Specifications range from logical relations between inputs and outputs to execution traces, natural language, or partial programs [3]. Searching for an arbitrary program that fits a specification is, in the general case, an undecidable problem. That means that there is no algorithm guaranteed to terminate and correctly decide whether a satisfying program exists and produce one when it does. To make the problem manageable, systems restrict the set of programs they are willing to consider in the first place, an idea known as syntactic bias. This is typically done by limiting candidates to a domain-specific language (DSL), a grammar, or a partial program, built from a fixed set of operators and rules for combining them.

Given this restricted space, the search continues through one or a combination of four broad strategies: enumerative, deductive, constraint-based, or statistical. Enumerative search generates all possible candidates and tests each of those against the given specification. Deductive search instead works by recursively decomposing a specification into subspecifications for sub-expressions by exploiting the inverse semantics of operators. Constraint-based approaches translate the whole problem into a set of logical constraints and hand it to an automated solver, which searches for values that satisfy those constraints directly. Finally, statistical techniques search the space probabilistically, often using a learned model to prioritize promising regions of the search space.

The BEN pipeline follows a three-stage "Divide, Align and Conquer" approach [4]. Its steps and core mechanism are also summarized in Figure 2. In the divide step, input and output images are segmented into sets of objects using a learned decomposition function. The latter is chosen from multiple segmentation modes (e.g., based on color and adjacency constraints), ranked by estimated usefulness. The align step then applies analogical matching to link each output object to its most likely corresponding input object, ranking candidate pairs according to their shared relational structure and feature similarity. This enables the next step to explore the most promising correspondences first, but eventually all pairs can be explored if necessary. Finally, the conquer step synthesises a transformation program for each matched candidate pair independently and checks whether the found program generalises across all examples; if not, the search continues down the candidate ranking. All of the successful transformation programs are then passed to a concept learner, which determines the exact logical conditions under which each program applies. During test evaluation, new input objects are segmented and these learned condition-transformation rules are evaluated to construct the final output image. This image should be identical with the output image provided by the task, so that the task is considered solved.

The transformation search must find a program $p'(o_i) = o_j$, which represents the transformation that would turn the input object from the matched pair into its respective output. Candidate programs are created from a grammar, containing primitives such as rotate, mirror, color, scale, move, and replace. The initial implementation uses a standard exhaustive search, generating all candidates up to a fixed depth d and testing each in order of increasing depth. In order to speed up the search, the original BEN deduces the arguments to be

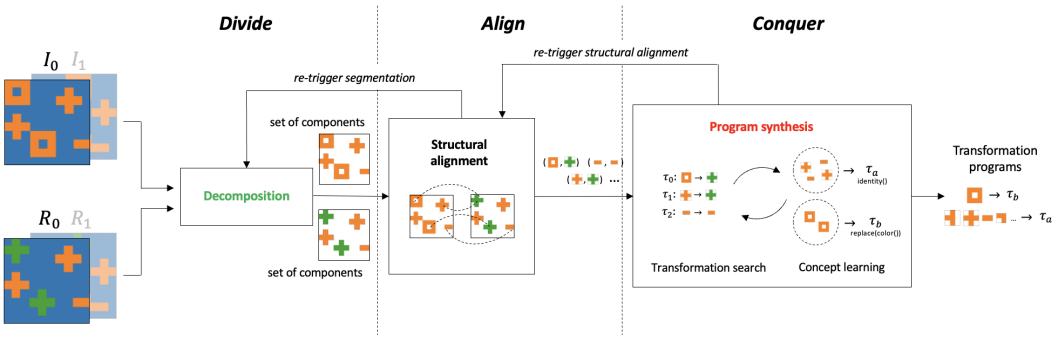

Figure 2: BEN's pipeline [4]. It highlights the divide, align and conquer steps and how they interact with each other.

applied to some of the primitives based on the matched output.

The grammar's search space has a significant influence on the outcome of the task, i.e. whether it gets solved or not. Adding 25% non-sense primitives to the initial BEN's grammar causes a 22% drop in tasks solved within a 30-second budget, though performance recovers under a larger time budget [4]. These results indicate that the primitives included in the grammar can have an impact on the efficiency of BEN. This sensitivity is not surprising: exhaustive enumeration over a DSL is a fundamental challenge in program synthesis, as the search space grows exponentially with program depth [3].

An already investigated possible response is to use constraints to prune the program space before enumeration begins. This other system, BART, demonstrates this concretely: it eliminates up to 99% of the program space and reduces the enumeration time by two to three orders of magnitude [6]. This happens by checking constraints on program syntax without executing programs. A key insight from BART is that many useful constraints follow directly from the domain and are straightforward to derive. The pruning strategy proposed in this work follows the same principle, but derives constraints from a more specific source: the structural properties of each matched input-output object pair. Rather than general syntactic rules, each correspondence determines which primitives are inapplicable for that specific transformation, allowing the grammar to be tailored per sub-problem before search begins.

An alternative direction for improving search efficiency is to guide enumeration using learned ranking functions rather than pruning the space outright. Program synthesis systems have increasingly incorporated statistical guidance to improve search efficiency [3]. One common strategy is to learn, from an offline training corpus, the likelihood of each grammar production rule given cues from the input-output examples [3]. These probabilities then bias the order in which programs are explored during enumeration. DeepCoder applies this idea by training a neural network on a corpus of synthesis problems to predict which primitives are likely to appear in the solution. Then, it uses these predictions to bias the order in which programs are explored and achieves order-of-magnitude speedups over unguided enumeration [7]. HYSYNTH takes a similar approach in the ARC setting specifically: it samples programs from a large language model and fits a context-free surrogate

model to those samples, which then guides synthesis search [8]. Both systems demonstrate that incorporating prior knowledge about likely program structure can substantially reduce the portion of the search space that needs to be explored in practice.

3 Methodology

BEN shows that decomposing ARC tasks into small synthesis sub-problems enables solving them with a small number of primitives. However, the transformation search makes no systematic use of the information available from matched input-output components beyond parameter deduction (when extending the grammar dynamically). This project follows a structured experimental methodology. First, BEN is reimplemented in Julia using Herb.jl, replicating the original search behaviour as closely as possible to establish a reliable baseline. This baseline serves as the reference point against which all subsequent changes are measured.

Once the baseline is established, the implementation is analysed to identify bottlenecks and opportunities for improvement. This includes examining how the search strategy, grammar design, and heuristic guidance affect the number of programs evaluated, the solve rate, and the runtime. Based on this analysis, the improvement is designed and implemented, motivated by a specific weakness observed in the baseline.

3.1 Baseline Reimplementation of BEN

The transformation search was reimplemented in Julia using the Herb library. The reimplementation mirrors BEN's original Python implementation as closely as possible, preserving the stopping criterion, search strategy, primitive set, etc.

The transformation grammar is defined as a context-sensitive grammar (CSG) using Herb's *@csgrammar* macro. Unlike a context-free grammar, where production rules apply regardless of surrounding context, a CSG allows rules to depend on neighbouring symbols (context). This is a distinction that enables Herb's constraint features. These "constrain the syntax of the programs explored," so they can be used here to prune the search space [9]. The grammar encodes all 11 primitives from BEN: color, rotate, mirror, scale, inner, complement, denoise, border, cut, move and shape. Color changes just the object's color to the provided one, rotate changes its orientation by rotating it a fixed number of degrees, and mirror reflects it across a chosen axis. Scale enlarges or shrinks the object according to the provided parameter, while move shifts its position within the grid. Inner fills enclosed empty regions inside an object, border creates an outline around it, and denoise removes isolated pixels and fills small gaps to smooth the shape. Cut crops the object by reducing its dimensions, complement swaps object and background regions, and shape replaces the object's geometry with a predefined pattern while keeping its original color. Rotate, mirror, scale, cut and border each have a parameter that is limited to a discrete set of values.

A key design choice inherited from the original BEN is that the grammar is extended dynamically per correspondence. Before searching, correspondence-specific constants are added to the grammar to create the parameter values set for the move, shape and color primitives. These represent the difference between the positions (coordinates) of the input

and output objects within the grid, and the shape and colors of the output object. This is equivalent to initial BEN's mechanism: rather than searching over all possible parameter values, the grammar is pruned to the values deduced from the matched pair.

One significant adaptation is that the search needs to now happen over *Grids*. A grid is a two-dimensional matrix whose cells contain integer values (0-9) representing colors, or 0 for the background. For the transformation search, the grid contains only one object but also the background. The rest of the grid's objects that constituted the image before the divide step are faded into the background. However, the original BEN program was designed to perform the transformation search with isolated objects, so without the background. This difference in approaches made it difficult to imitate the scale or cut primitives' intent when reimplementing the algorithm. After noticing the need to change grid sizes on a few tasks, we decided to add 2 new primitives: resize and crop. Resize now will turn the grid to have the provided height and width, deduced from the output object, and center the old grid in the middle. The grammar extends dynamically with the arguments for resize as well (width and height of the new grid). Crop reduces the grid to the object's bounding box.

The search process proceeds using Herb's *BFSIterator*. This was chosen to imitate the original BEN's search strategy, which is a breadth-first exhaustive enumeration. This enumerates programs in order of increasing depth up to a fixed depth d. For each candidate program, the expression is applied to the input object's grid. Its result is then compared against the output object's grid. If the two are identical, the program is considered useful and saved. An additional *Unique* constraint (part of the Herb library) ensures that each primitive is used at most once in each program. This also mirrors the original BEN's approach.

Once a transformation program is found for a given correspondence, the algorithm tests it across all correspondence pairs to assess how broadly it applies. The best-covering transformations per output object are tracked. The search stops early if all output objects are covered by transformations that each generalize to at least two training examples. This aims to avoid overfitting the solution to specific cases. Otherwise, the search goes through each of all of the correspondences and transmits all transformation programs that apply to at least one correspondence to the concept learner step.

3.2 Improvements

The primary issue found in the reimplementation's transformation search is that it is based on exhaustive search. It makes use of 13 primitives that result in a huge search space as the depth increases. The similarities between the matched objects in the correspondences are not exploited, although easily determined. Such a first resemblance that was easy to spot and sparked this idea was: If the 2 objects have the same color from the start, why still include the primitive in the grammar when that makes for about 1/30 of the search space? This was further generalized to the move primitive when they have the same position, and then to most of the other primitives.

Another possible approach would be to guide enumeration using learned ranking functions, as in DeepCoder or HYSYNTH [7, 8]. However, such methods are not suitable here.

They assume the search can exit early once a promising program is found, but BEN's conquer step evaluates correspondences exhaustively. The concept learner downstream expects all viable transformations, not just the first one found. Reordering the search space would therefore yield no benefit without also redesigning the concept learner, which is beyond the scope of this work.

Therefore, this project suggests using the similarities between the input and output objects of each correspondence in pruning their respective grammar. Once the grammar has been extended dynamically for a given correspondence, the primitives are checked against the correspondence's input and output objects. The primitives that could never be used for the given correspondence and only inflate the search space are removed. Herb supports this kind of removal through its *Forbidden* constraint. Every grammar rule is internally represented as an expression, so a rule such as *Grid = move(Grid, Row, Column)* is stored as the call expression for the move primitive.

To remove a primitive, we collect the indices of every rule that matches that primitive's name, which in our case is just one per primitive. We accumulate these indices across all primitives flagged as inapplicable for the correspondence. For each index, we look up the rule's arity and build a pattern: a *RuleNode* wrapping that index around one *VarNode* structure per argument slot (both structures are part of the Herb library). This is done such that the pattern matches any combination of arguments without requiring them to be equal to one another. Registering this pattern as a *Forbidden* constraint blocks the iterator from ever instantiating that rule, at any position in a program and for any argument values. So, any primitive marked inapplicable is guaranteed to be absent from every program the iterator produces and without disturbing the indices of the rules that remain.

To avoid excessive pruning, this improved version forbids 11/13 primitives, where guaranteed inapplicability was found. This restriction follows directly from the completeness requirement set out above: the concept learner depends on the transformation search returning every viable transformation. Forbidding a primitive that could still apply with certain parameters, or in certain combinations of primitives, would potentially remove a correct explanation. In such a case, the extra search space is worth it. So, each primitive is forbidden only if none of the values it could take would make the input more similar to the output. For the denoise and border primitives, no such check has been implemented.

Color is pruned if the objects already have the same color attributes. The move primitive is excluded if they are placed in the same spots inside the grid according to their x and y coordinates, stored inside the object. For shape, the two objects' pixel layouts are compared after normalizing both to a common origin. This normalization is performed by translating every pixel coordinate so that the minimum row and minimum column become (0,0), allowing shapes to be compared independently of their absolute position in the grid. For both scale and cut, we compare both bounding boxes' widths and heights. The dimensions of the bounding box are computed from the occupied pixels of the object as *(max_row - min_row + 1, max_col - min_col + 1)*. If the widths and heights are both the same, then scale will not be applicable, and if they are both larger or equal, then cut would not be a good fit either.

Rotation is pruned when rotating the output object by 90 degrees produces exactly the same grid. Such an object is rotationally symmetric, meaning that applying any of the

available rotation parameters (90, 180, or 270 degrees) would leave it unchanged. Therefore, this could not be one of the transformations applied to get it as a result. Mirror is similar in the sense that we check for symmetry again. This is determined by checking whether mirroring the output object along each axis produces exactly the same grid. We check all 3 possible axes (horizontal, vertical, and diagonal) to correctly assess the symmetry of the object, in which case mirror becomes forbidden. For inner, the primitive is applied to the input object and the result is compared to the original. If no change occurs, the primitive would not have an influence and is therefore forbidden. Complement is pruned when the output object's pixel count differs from the number of background cells in the input grid. Concretely, if *output_pixels != total_grid_cells - input_pixels*, the primitive is forbidden. Finally, both resize and crop are pruned if the 2 grids have the same dimensions, as they imply changing the grid's sizes.

4 Experiments

The objective of this research project is to determine how to better find the necessary transformations in BEN's transformation search. Specifically, the project investigates whether pruning the grammar at correspondence level can reduce the search space and/or improve solve rate.

Therefore, the research question is divided into the following sub-questions:

Q1. How much does the pruning reduce the search space for different depths?
Q2. Which pruning rules contribute most?
Q3. Is the reduction uniform across tasks?
Q4. How does pruning affect overall system performance?

4.1 Setup and Evaluation

To make repeated measurements computationally feasible, Q1-Q3 were conducted on a randomly chosen subset of 28 tasks from the ARC training set that can be found in the experiments repository. These tasks are roughly equally divided between tasks that get solved and tasks that do not (they fail to give the right solution), to get a good overview of the impact of the improvement. These purposefully do not include timeout tasks. The tasks that timeout during transformation search do not return the evaluated number of candidate programs. If a task times out in one setting and not in the other, the alternative would probably have a very large number of programs evaluated. The average calculations would then not be accurate if one measurement includes a huge number and the other just excludes it. The final evaluation (Q4) was performed on the complete 400-task training set.

The reimplemented version of BEN is evaluated under two configurations: the baseline reimplementation and the improved version with grammar pruning. All runs use a fixed per-task timeout for transformation search of 80 seconds and 140 seconds overall. All the exact configurations for the experiments can be found in commit 94756da of the experiments repository. The experiments were run on an HP ZBook Power 15.6 G9 (Intel Core i7-12700H, 2.30 GHz, 16 GB RAM, Windows 11 64-bit), single-threaded. Julia version 1.12.6 and Herb version 1.0.0 are used throughout.

4.2 Results

The baseline provided a transformation search average of 3.927s for depth 3 and 26.035s for depth 4 (considering only the tasks that did not lead to a transformation search timeout). This highly suggests that when a set of transformation programs is to be found among the ones generated with the grammar, the program will do that way below the set time limit. Therefore, the main expectation is that through the improvements, the search will evaluate fewer programs, but that may not necessarily lead to more tasks solved.

Answering Q1. Figure 3 shows how grammar pruning reduces the number of transformation programs evaluated per task at every depth tested. The magnitude of that reduction grows with depth: 50.6% at depth 2, 72.8% at depth 3, and 84.5% at depth 4. This trend is consistent with the combinatorial growth of the search space. As depth increases, a larger fraction of the grammar's productions become irrelevant to the given task, so pruning has proportionally more programs to remove. That means that the pruning value is not fixed and the technique becomes more useful as the system is pushed towards more complex transformations.

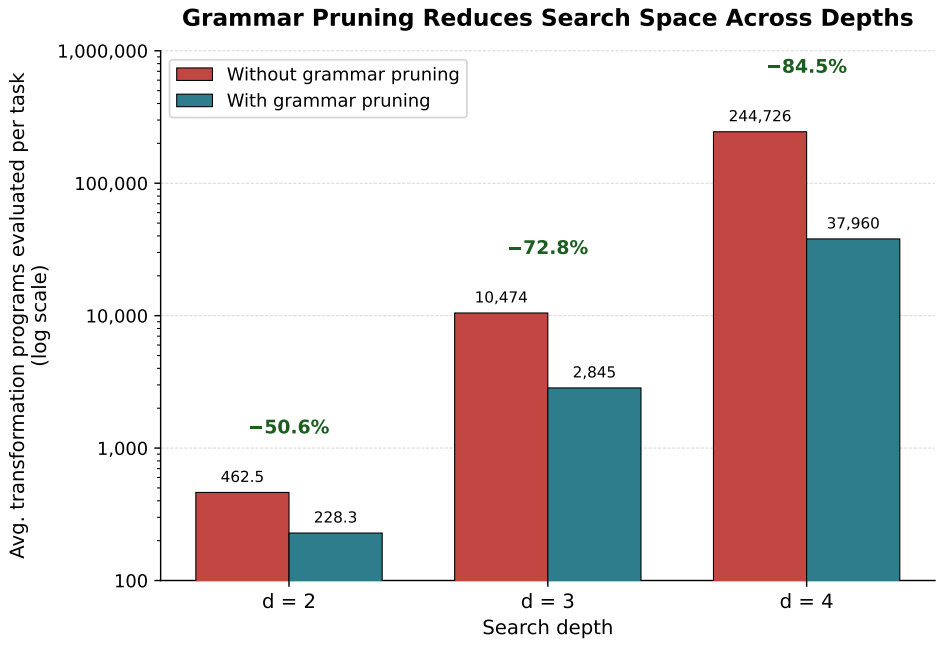

Figure 3: Average transformation programs evaluated per task with and without grammar pruning, across search depths d = 2, 3, and 4 (log scale). Percentages indicate the relative reduction in search space at each depth.

Answering Q2. Out of the 28 evaluated tasks, there were 7 that appeared to be most relevant to answer this question. Some of the tasks shared the same number of transformation program candidates for the baseline, yet very different amounts for the pruned version. Therefore, we noticed that grouping tasks based on identical baseline candidates

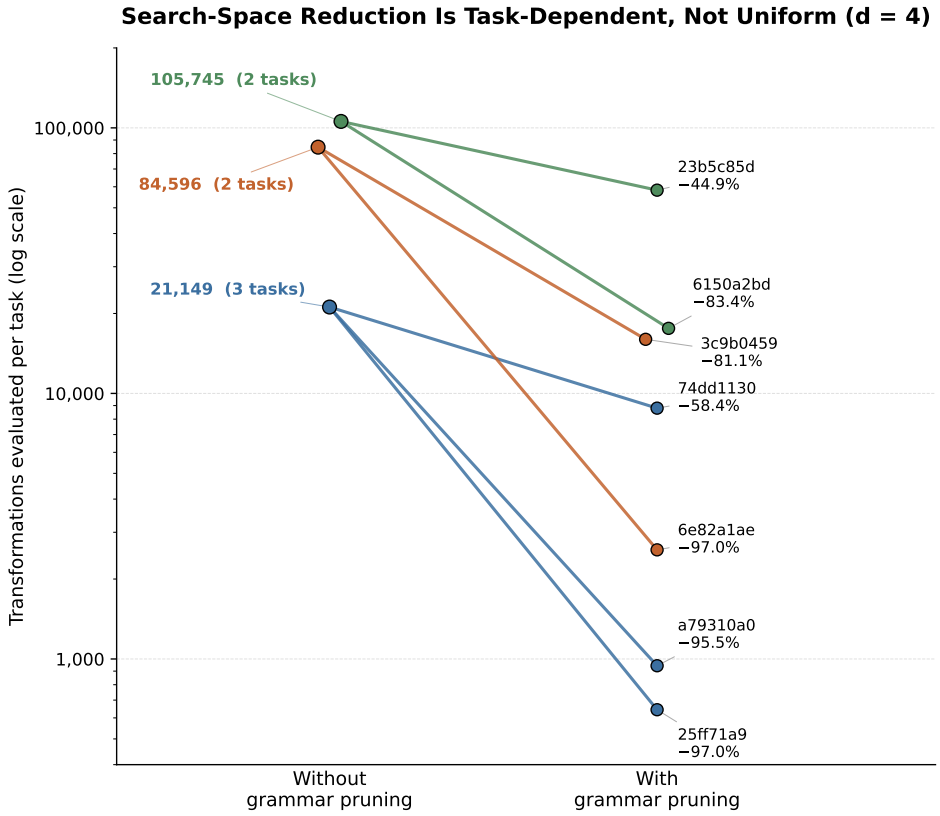

Figure 4: Search-space reduction for seven representative tasks at d = 4, grouped by identical unpruned baseline. Lines connect each task's pre-pruning count (left) to its post-pruning count (right).

reveals that the pruning effectiveness is task-dependent rather than a fixed multiplier, as depicted in Figure 4. Among tasks that share the same 105,745-program baseline, one is reduced by only 44.9% while another is reduced by 83.4%; a similar spread appears in the 21,149-program group where the pruning percetange varies between 58.4% and 97.0% across 3 different task. The fact that this is not an isolated event, but one that was observed in 3 different instances, indicates that the gain from pruning depends on how well a task's correct transformation aligns with pruning rules, rather than on depth or the baseline's programs size alone. Therefore, the pruning gain is very much task-dependent.

Answering Q3. This was evaluated by individually applying each of the pruning rules. In isolation, most of them contribute modestly: nine of the eleven rules tested reduce the search space by less than 12% individually, as can be observed in Figure 5. Two rules dominate: cut alone accounts for a 53.4% reduction and scale for 29.1%. This result makes sense as those 2 primitives have the largest discrete sets of values. To interpret the mirror and rotate results, despite each primitive having 3 possible parameter values, the output object's symmetry is not the only case when those primitives would not be applicable, rather

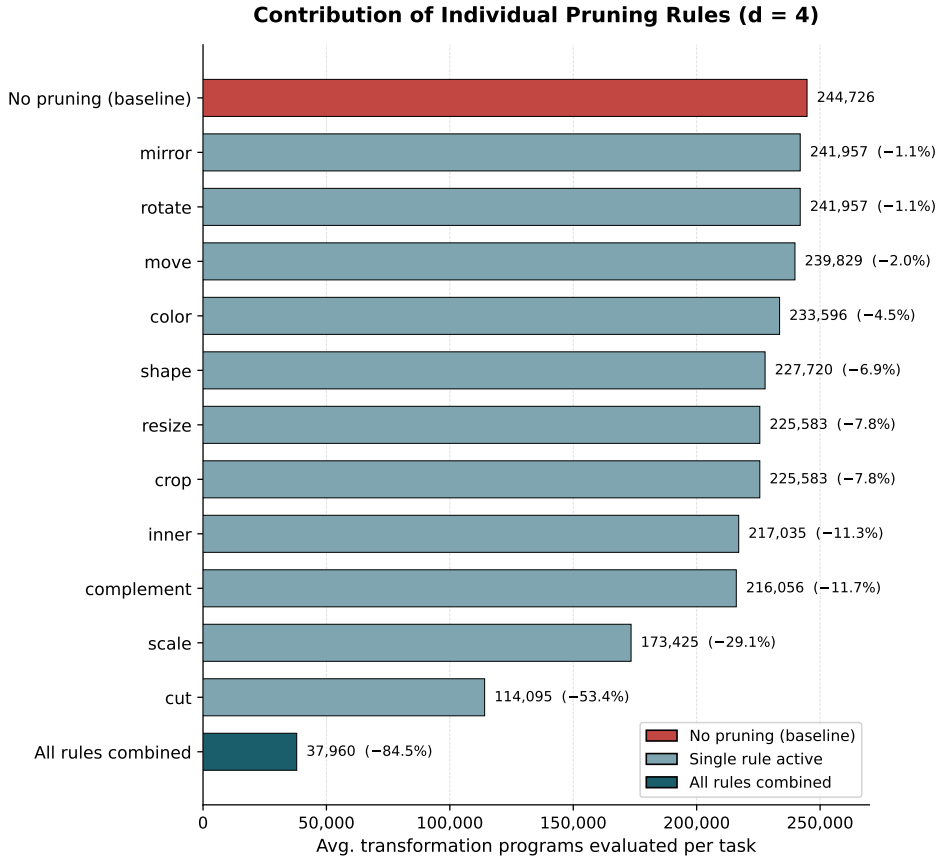

Figure 5: Average transformation programs evaluated per task at d = 4 when a single pruning rule is active in isolation, compared to the unpruned baseline and to all rules combined. Percentages denote reduction relative to baseline.

it represents a pretty specific case. Therefore, the expectations from those rules were not very high. Furthermore, the move, shape, color and resize primitives had their arguments extended dynamically. This means that their impact on the search space was pretty limited already. Which is consistent with the results. The crop, inner and complement primitives need no arguments, which explains their small reduction again. Even, we could say that their rules had a big pruning influence when we take into account the fact that they each make for just around 1/30 of the search space. However, the fully combined system achieves an 84.5% reduction, which is far less than the sum of the individual percentages would suggest (over 130% if treated as additive). This gap is explained by the fact that the rules have overlapping coverage (multiple pruned primitives often eliminate the same programs). This suggests that achieving high pruning performance depends more on having a diverse set of complementary rules than on maximizing the strength of any single rule.

Answering Q4. Overall performance is measured along three axes: (i) the number of candidate programs evaluated before a correct transformation is found, averaged over non-

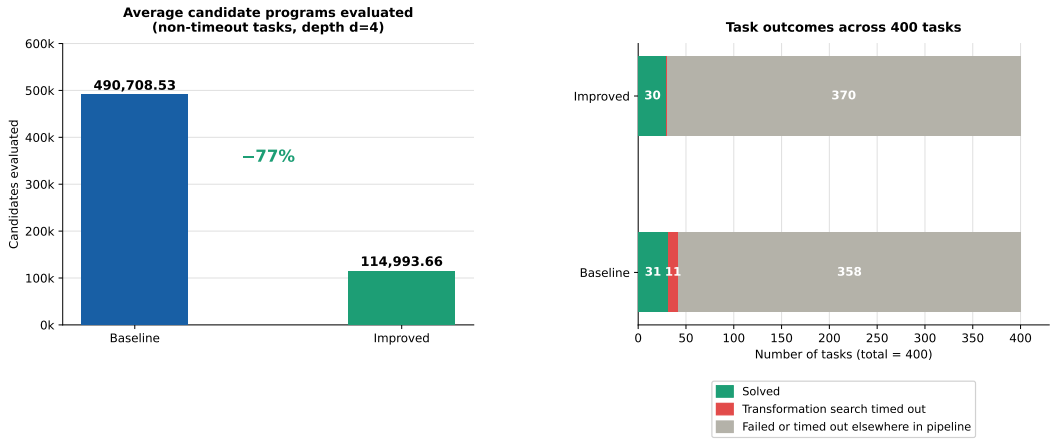

Figure 6: Comparison between baseline and improved version performance at depth 4; (left) compares the average number of programs evaluated for tasks that did not get any sort of timeout; (right) compares the task outcomes defined as solved, timeout during transformation search, and failure or timeout elsewhere in the pipeline.

timeout tasks; (ii) the overall solve rate, defined as the number of tasks for which a valid solution is found within the time limit; and (iii) the number of tasks that hit the timeout. Together these metrics capture both the efficiency of the search and its effectiveness.

Figure 6 summarises the performance of the baseline and improved configurations at depth 4 over the whole training set (400 tasks). The improved configuration evaluates substantially fewer candidate programs on average, reducing the mean candidate count from 490,708 to 114,993 - a reduction of approximately 77%. However, the number of tasks solved changes from 31 to 30. There are also 11 timeouts observed for baseline, while 0 for the improved. These numbers suggest that the grammar pruning successfully reduces the size of the search space explored per task. This efficiency gain translates into managing to go through the whole pipeline for tasks that previously timed out. As mentioned above, including the tasks that would time out in one setting and not in the other would make the calculations inaccurate. Therefore, the averages were calculated exclusively based on tasks that would not time out in any of the 2 versions of the reimplementation. Tasks that time out are probably unsolvable with the given segmentation mode, grammar depth and/or the primitives at hand. These tasks are, however, included in the comparison between the outcomes.

The results indicate that grammar pruning based on similarities between matched input-output pairs is an effective strategy for reducing the search space. The 77% reduction in average candidates evaluated is substantial, and directly reflects the core idea: if a primitive cannot possibly produce any meaningful change given the correspondence at hand (such as applying a color transformation when the objects already share the same color), it should not be part of the search at all. By marking such primitives as forbidden before search begins, a large portion of the program space is eliminated without any risk of discarding correct solutions.

However, the change in solve rate, while small in absolute terms (one fewer task), is inconsistent with this interpretation. The baseline and improved versions do not solve the same set of tasks: the baseline solves 3 tasks that the improved version does not (0d3d703e, d037b0a7, d10ecb37), while the improved version solves 2 tasks that the baseline does not (be94b721, 97999447). After investigating the 3 failing tasks for the improved version, we could not come to a reason as to why this is happening. It appears that the right transformations are still being sent, yet not picked at the concept learner phase. The extra solved tasks are ones that would previously fail with the alternative transformation chosen by the downstream concept learner. This shows that this optimization can not only make the search easier and faster, but also reduces the likelihood that the concept learner may find an alternative (wrong) solution. At the same time, pruning, while effective at reducing search effort, may in some cases affect the rules that the concept learner finds.

Moreover, at depth d = 4, the search space is large enough that the 80-second timeout is a binding constraint for many tasks. Reducing the number of candidates evaluated per task means that some tasks which previously timed out can now be solved within the budget. Whether or not those would actually prove to be solvable with the current primitives is yet to be determined. This suggests that the benefit of pruning would become even more pronounced at greater depths, where the search space grows exponentially.

These findings directly address the research question: more informed search strategies, i.e. correspondence-aware grammar pruning, can reduce the number of programs evaluated and improve BEN's conquer step. The result aligns with the expectation that exploiting available structural information about matched object pairs would yield efficiency gains, though the regression in solve rate should be investigated further. This also reflects an inherent limitation: the bottleneck for many tasks may lie not in the search strategy but in the grammar itself, which is fixed to 13 primitives. Tasks requiring transformations outside this set remain unsolvable regardless of how efficient the search is.

Compared to the baseline, the improved system is better in terms of candidate programs evaluated and timeouts, which lends confidence to the approach. However, the newly failing tasks currently represent a big downside. Moreover, the overall solve rate of 7.75% on the full training set remains low, highlighting that the transformation search step alone is not the only limiting factor. The divide step upstream also heavily influence which tasks are solvable, and errors introduced there cannot be recovered by a better search. On top of that, even if the transformation search does find the right transformations among the ones passed to the concept learner, the task will not be solved if they are not selected.

5 Conclusions and Future Work

A central challenge in BEN is finding the correct transformation program efficiently. As program depth increases, exhaustive search over all candidate transformations quickly becomes infeasible, making the transformation search a critical bottleneck. This work addressed the question: how can we better find the necessary transformation programs? To answer this, BEN was reimplemented in Julia as a reproducible baseline, and a correspondence-based grammar pruning strategy was designed and evaluated on the full 400-task ARC training

set.

The results provide a clear answer: by exploiting structural similarities between matched input-output object pairs to eliminate inapplicable primitives before search begins, the search can be made substantially more efficient. At depth d = 4, the improved system reduces the average number of candidate programs evaluated by approximately 77%, but it seems to have an impact on the solution picked by the concept learner. However, pruning the search space still appears like a promising direction for better transformation search in BEN.

It is worth noting that when switching the programming language and adapting the algorithm to use Herb, the transformation search aims to reproduce the strategy as closely as possible. However, this adaption nevertheless has its flaws as the primitives were created to work directly on objects and not on grids that contain both the object and the background. As the time allocated to this project was limited, more time spent in the future investigating the effects this transition had could translate into solving more tasks.

Several directions remain open for future work. Further analysis into why would tasks fail when using the improvement is the first step. Then, the pruning strategy could be extended with more pruning rules to yield further reductions. Learned heuristic ranking functions could complement pruning if the concept learner would also test the promising transformations straight away. The overall solve rate of 7.75% also highlights that the conquer step is not the only bottleneck. Improvements to both the divide and align steps upstream and the concept learner would likely have a large impact on the end-to-end performance. Finally, evaluating the approach at greater depths and tighter time budgets would help quantify how the benefits scale with search complexity, where gains are expected to be most pronounced.

6 Responsible Research

6.1 Ethical considerations

This research operates entirely on synthetic, publicly available data and the ARC benchmark contains no personal information, sensitive data, or human subjects. The work carries no direct social risks in its current form.

6.2 Reproducibility

The experimental setup is fully deterministic as no randomness is introduced at any stage of the transformation search, and all experiment configurations are available in the project's experiments repository at https://github.com/Herb-AI. This ensures that readers can reproduce the results by running the same code under the same conditions. Both the reimplementation and experiments instructions are structured to be readable and well-documented, with comments highlighting each improvement. This should make it straightforward to isolate and verify individual contributions.

While the experimental procedure itself is deterministic, full reproducibility also depends on the software and hardware environment. The implementation was developed using a specific version of Julia and Herb.jl, and future changes to either dependency may affect its behaviour, performance, or compatibility. However, recording dependency versions and

making the source code available should ensure that results remain reproducible over time.

Hardware differences can also influence experimental outcomes, particularly when it comes to runtime measurements. During development, noticeable differences were observed between machines used by team members. For example, experiments executed on a MacBook generally achieved lower runtimes and experienced fewer crashes than those run on other systems. Some different results were also observed when running the same code on different laptops. We are not sure why this happened. Reproducing the experiments on different hardware may lead to different execution times and even results despite identical algorithmic behaviour.

6.3 Research integrity and interpretation of results

One threat to research integrity can be benchmark overfitting. Throughout development, the ARC training set was repeatedly used to evaluate modifications and guide design decisions. While this is a standard research practice, repeated exposure to the same benchmark can unintentionally lead to improvements that are specific to the characteristics of the training tasks rather than reflecting generic improved capabilities.

Researcher choices also influence the reported outcomes. Decisions regarding which primitives to include in the grammar, on which settings to evaluate, and which metrics to use can all affect conclusions. To mitigate this risk, unsuccessful approaches and negative results were documented alongside successful modifications whenever possible. Moreover, multiple metrics and parameters were used to assess performance.

6.4 The use of Generative AI

During the development of this thesis, generative AI tools (Claude, ChatGPT, Gemini and GitHub Copilot) were used in a supporting role across three stages of the project: research and brainstorming, implementation, and writing. In the research and brainstorming phase, AI tools were used as guidance through unfamiliar concepts and to explore possible approaches. During implementation, AI tools were used to assist in coding and debugging: explaining error messages and stack traces, suggesting fixes for specific bugs, clarifying unfamiliar library, API documentation, and programming language syntax. In writing the report, AI tools were used to improve the quality of the phrasing, grammar, and structure of the drafted text.

References

[1] Algorithmics, Delft University of Technology. Efficient learning through programmatic representations. `https://projectforum.tudelft.nl/course_editions/130/generic_projects/6760`, 2025. Research Project 2025/2026 Q4. Accessed: 2026-06-20.

[2] Francois Chollet. On the measure of intelligence. arXiv preprint arXiv:1911.01547, 2019.

[3] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 2017.

[4] Jonas Witt, Sebastijan Dumancic, Tias Guns, and Claus-Christian Carbon. A divide, align and conquer strategy for program synthesis. *Journal of Artificial Intelligence Research*, 82:1961–1997, 2025.

[5] Herb.jl Contributors. Herb.jl: A julia framework for program synthesis. `https://herb-ai.github.io/`, 2024. Accessed: 2026-04-23.

[6] Tilman Hinnerichs, Bart Swinkels, Jaap de Jong, Reuben Gardos Reid, Tudor Magirescu, Neil Yorke-Smith, and Sebastijan Dumancic. Modelling program spaces in program synthesis with constraints. In *Technical Communications of the 40th International Conference on Logic Programming (ICLP)*, 2024.

[7] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*, 2017.

[8] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. HYSYNTH: Context-free LLM approximation for guiding program synthesis. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[9] Tilman Hinnerichs, Reuben Gardos Reid, Jaap de Jong, Bart Swinkels, Pamela Wochner, Nicolae Filat, Tudor Magurescu, Issa Hanou, and Sebastijan Dumancic. Herb.jl: A unifying program synthesis library, 2025.