ArChEs Partial Paper

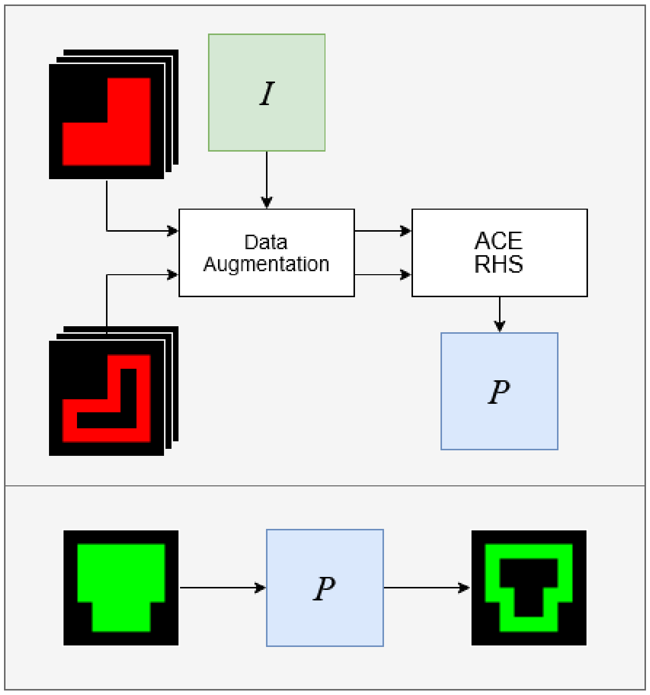
CS 560

Sean Flannery, Ryan Luu, Harjas Monga

March 2021

1 Introduction

${}^{A}_{R}{}^{C}_{H}{}^{E}_{S}$ (pronounced "arches") stands for Abstraction on Constrained Examples, by Ryan, Harjas, and Sean. It is designed to synthesize programs that model reproducible image transformations using only examples (and optionally, knowledge of certain properties of the transformation). The figure below shows how input examples are augmented using invariants I, similarly to SKETCHAX[2]. ${}^{A}_{R}{}^{C}_{H}{}^{E}_{S}$ searches our DSL for a sequence of instructions that match the transformation from the input examples and produces an optimally general program P according to an Occam's razor-based heuristic.



2 Specification

2.1 Description of Problem

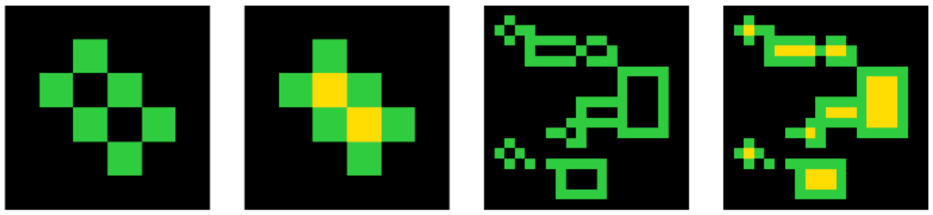
Given a set of image pairs $S_I = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$, synthesize a program that models the function F, which satisfies $F(X_i) = Y_i$ for all $i \in 1, \ldots, n$. The program should also be able to generalize to images not found in S_I.

2.2 Sources of Input/Output Examples

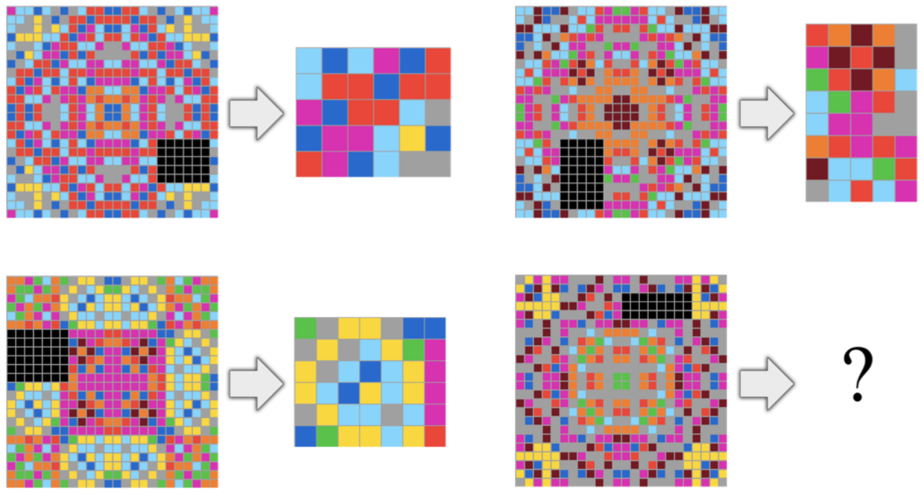
The tasks (sets of image pairs) for this project are inspired by (or directly taken from) those used in the Abstraction & Reasoning Kaggle Competition (ARC). This competition was created by François Chollet in order to better measure the "intelligence" of AI systems[3]. This work critiques the broader trend within the AI/ML community to compare human and machine intelligence on especially-specific tasks like Chess, Go, or Atari. The authors propose a series of improved AI benchmarks, which focus more on the ability of a system to perform generic abstraction and solving of visual tasks.

A top-performing learning-based architecture performed very well on ARC tasks with images up to 10×10 pixels[6]. This subset of the ARC tasks may have a lower level of abstraction as a whole, which is why we will use it to evaluate our approach. This makes the task more of a program synthesis problem, rather than an artificial general intelligence problem.

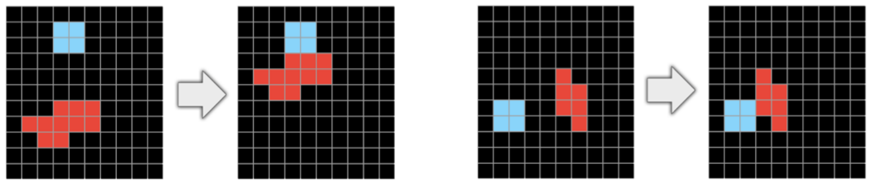
2.2.1 Sample Task: Flood-Fill



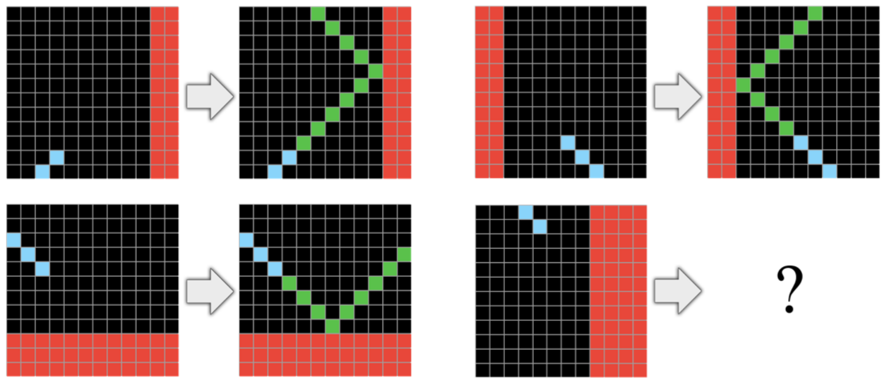
2.2.2 Sample Task: Pattern Completion



2.2.3 Sample Task: Collision Simulation



2.2.4 Sample Task: Trajectory Prediction



3 Approach

3.1 Semantic Properties

An important property that will help guide the data augmentation is the type of transformation being done. If the program we are trying to synthesize is focused on a structural component of the image, then we can mark the color as "unimportant" an property. We can then permute the color property to help augment the data set. On other hand, if we are attempting to synthesize a program that deals with color manipulation we could mark that the structure is the "unimportant" property. In this case we could apply transformations (rotations, flips, etc...) to the images to augment the data set.

3.2 Data Augmentation

For many of these types of tasks, we can augment the provided examples by applying color maps, translations, rotations, reflections along y, x, and diagonal axes. We will also explore increasing the number of examples beyond this automatically for user-provided task instances.

3.3 Search Space Reduction

Many of these tasks can have their search spaces substantially reduced by considering that

- most tasks only modify input boxes of a single color (typically black)
- for some tasks we can view contiguous bits of the same color on input as whole objects/shapes
- in creating our DSL, we'll avoid functions with several arguments to prevent search space explosion

Before applying such space-saving reductions, we can consider smaller images for input-output pairs to validate the correctness of our approach.

4 Ideal Deliverables

4.1 Data Augmentation Pipeline

We shall create a process to generate examples using input examples and known invariants.

4.2 Domain Specific Language

We shall create a high-level DSL for specifying programs that solve these tasks.

4.3 Integration with Microsoft PROSE

We shall integrate our semantics of the operators/transformations of our DSL within Microsoft's PROSE tool[4].

4.4 Synthesized Solutions from Nontrivial Tasks in the ARC

We shall successfully synthesize programs in our DSL using PROSE for non-trivial task examples from the Abstraction & Reasoning Challenge Corpus.

4.5 Stretch Goal: "Create-Your-Own Task" Tool

This tool would enable us to have students/users design their own task and provide limited examples of desired input/output.

4.6 Stretch Goal: User-Interactivity

This would consist of us prompting a user with questions like "would this transformation make sense?" for inputs they've provided.

4.7 Stretch Goal: Live Demonstration for Novel Task

This would mean anonymously asking a person in the class to prepare an example for our system to solve *LIVE*.

5 DSL

5.1 Context Free Grammar

```
program ::= single

single ::=  filterColor(single, color) |
            recolor(single, color)     |
            composeGrowing(multi)      |
            compress(single)           |
            orthogonal(single, axis)   |
            pickMax(multi, id)         |
            image

multi ::=   cut(single)
color ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
axis  ::= Y_AXIS | X_AXIS | XY_AXIS | YX_AXIS | ROT_90 | ROT_180 | ROT_270
id    ::= NONZERO | SIZE | NUM_COLORS | X_CORD | Y_CORD | NONZERO_POST_COMPRESS
```

5.2 Operator Semantics

1. **cut(image) → [image]**: Cut an image into at least two disjoint pieces where the smallest piece is as big as possible.
2. **filterColor(image, color) → image**: Set all pixels, except pixels at *color*, to 0.
3. **recolor(image, color) → image**: Set all nonzero pixels to *color*.
4. **composeGrowing([image]) → image**: Stack images sorted by number of nonzero pixels (fewest on top).
5. **compress(image) → image**: Extracts minimal sub-image containing all non-zero pixels.
6. **orthogonal(image, axis) → image**: Perform a reflection or rotation.
7. **pickMax([image], prop) → image**: Picks the image with maximum property:
   (a) nonzero pixels (+/-)
   (b) size ($w \times h$) (+/-)
   (c) number of colors
   (d) x-coordinate (+/-)
   (e) y-coordinate (+/-)
   (f) nonzero pixels after compress (+/-)

5.3 Reasoning

These set of operations were chosen after considering a kaggle competition focused around using PBE (Programming By Example) to solve image transformation problems [1]. We found that this set of operations is expressive enough to allow us to solve a sufficiently large domain of image transformations problems while being minimal enough to have a manageable search space. For example, the task in Figure 4 can be solved by the program

$$\textbf{compress}(\textbf{pickMax}(\textbf{cut}(image), \text{NONZERO}))$$

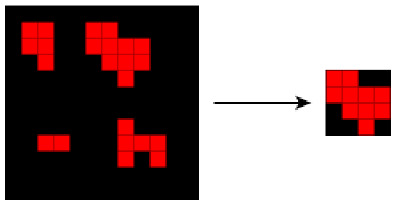


Figure 1: Example task which is solvable by the proposed DSL.

$$\textbf{composeGrowing}([image, \textbf{orthogonal}(image, ROT_180)])$$

OR

$$\textbf{composeGrowing}([image, \textbf{orthogonal}(image, X\_AXIS)])$$

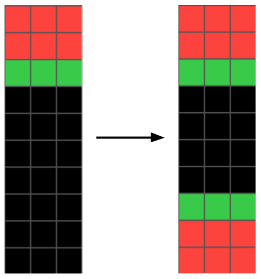


Figure 2: Example reflection task solvable by our DSL.

6 MANTIS Integration

The MANTIS method's main contribution is the augmentation of input-output examples through the application of "relational perturbation properties" [2]. Of relevance to us is determining how we might apply such augmentations to our image-based input-output pairs in order to generate more constraints for our program synthesizer. As a baseline we may manually label our tasks based on what invariants we think would hold (rotation, reflection, color permutation, etc.). This would align with the MANTIS method that relies on users to label the invariants we (as a sort of pseudo-user) determine would apply.

As a stretch goal, we could include the Partial MAX-SMT solver the authors included to automatically detect which sort of transformations ought to apply. In their approach output solution of a given (Partial) MAX-SMT solver will provide all the Relational properties that actually worked for all input-output examples. The consistent relational perturbation properties would be consistent with the given examples, and used to augment our dataset for our synthesizer.

We *have not* implemented this part of our pipeline yet, but would likely tack it on as a sort of pre-processing step when feeding examples into our program synthesizer.

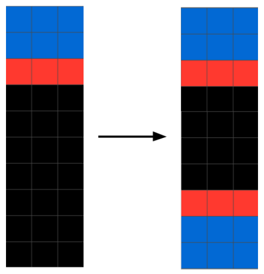


Figure 3: Augmented reflection task example with color permutation.

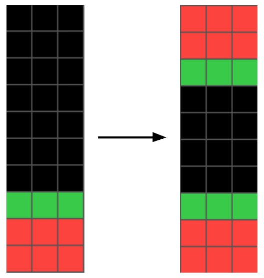


Figure 4: Augmented reflection task example with rotational/reflection permutation.

7 PROSE Integration

7.1 What is PROSE?

PROSE is a PBE framework developed by Microsoft's research group. It is the core technology that powers one of the most successful commercial program synthesis programs, FlashFill for Excel.

This project will utilize PROSE as the synthesis engine for several reasons. PROSE allows the developer to be mostly focused on the specifics of their synthesis problem, rather than the problem in general. PROSE framework will automatically use it's own deductive search algorithm, which combines a bottom-up grammar enumeration with a top-down enumeration[5]. This search technique will allow us to utilize more complex grammars if desired. Additionally, PROSE's search algorithm has builtin APIs that will allow us to rank resulting programs, guide the search via heuristics, and help break the synthesis problem down to smaller synthesis problems[5]. Lastly, PROSE has builtin in support for user interaction.

7.2 Translating Problem to PROSE

In order to translate the problem space to PROSE we need to specify 3 key items. Firstly, we need to specify a DSL for PROSE to use to synthesize programs. The context free grammar and operator semantics for this is as seen above. Secondly, we need to provide witness functions to PROSE. Witness functions operate on each production rule and provides domain specific information to PROSE so that it can attempt to break the synthesis problem down to sub-problems. Lastly, we need to provide heuristics to PROSE so that we guide the search towards programs that are more likely to be correct and generalize well.

7.2.1 Heuristics

PROSE allows us to attach heuristic functions to each of our grammar's production rules to help speed up search and find programs that are more likely to be correct and generalize well. For our operations, the heuristics will be mostly dependent on the color, prop, or axis parameters.

7.2.2 Witness Functions

References

[1] Abstraction and reasoning challenge.

[2] Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. Augmented example-based synthesis using relational perturbation properties. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[3] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[4] Sumit Gulwani and Prateek Jain. Programming by examples: Pl meets ml. In *Asian Symposium on Programming Languages and Systems*, pages 3–20. Springer, 2017.

[5] Sumit Gulwani and Prateek Jain. *Programming by Examples: PL meets ML*. IOS Press, February 2019.

[6] Victor Kolev, Bogdan Georgiev, and Svetlin Penkov. Neural abstract reasoner. 2020.