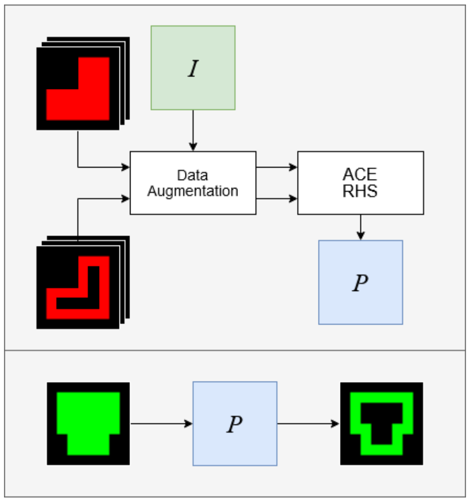
ArChEs

CS 560: Reasoning About Programs
Purdue University
Spring 2021

Ryan Luu, Harjas Monga, Sean Flannery
Equal Author Contributions

Purdue University, Department of Computer Science



1 Introduction

$\overset{A}{R}\,\overset{C}{H}\,\overset{E}{S}$ (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, optionally augmented by knowledge of certain properties of the transformation. We target image transformations that solve tasks used to measure the general intelligence of an agent given very few examples.

The figure above shows how input examples are augmented using invariants I, similarly to SKETCHAX[2]. $\overset{A}{R}\,\overset{C}{H}\,\overset{E}{S}$ 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 Related Work

Our work builds upon much of the existing scholarship related to Programming By Examples (PBE), but this application to general image-based transformation tasks is largely unprecedented.

2.1 Task Sources

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 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 more 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 abstraction and solving of visual tasks.

Soon after the competition, a top-performing learning-based architecture provided a corpus of ARC tasks with images up to 10×10 pixels [5]. This subset of the ARC tasks potentially required a lower level of abstraction and program depth, thus we selected it for evaluating our approach.

2.2 Inspirations for our Approach

The top solution's DSL from the kaggle competition [1] contained over 100 operations, and the author had refined them over several months to achieve the results they did. Given our limited time and resources, along with a desire to make a general proof of concept, rather than compete in a worldwide competition, we created our own pared-down set of operations. We found these to be expressive enough to address nontrivial problems within a smaller search space.

2.3 Our Contribution Relative to Other Work

Our contributions are threefold:

1. Encoding our DSL into the PROSE Framework [4] (to our knowledge, none of the top participants in the competition attempted this)
2. Augmenting our task examples with relational perturbation properties ala MANTIS [2]
3. Making the program search space tractable through novel abstraction on images and their pixel values

3 Illustrative Examples

$$\textbf{Compress}(\textbf{PickMax}(\textbf{Cut}(image), \text{NONZERO}))$$

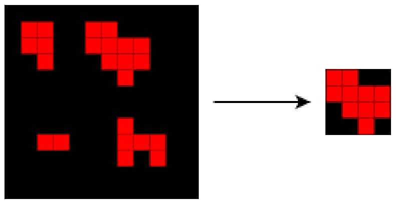


Fig. 1: Example selection task solvable by our DSL.

$$\textbf{Compose}([image, \textbf{Orthogonal}(image, \text{X\_AXIS})])$$

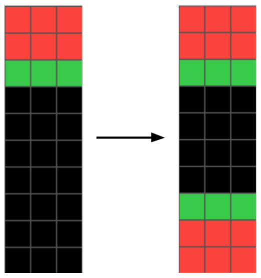


Fig. 2: Example reflection task solvable by our DSL.

4 Background

4.1 PROSE

PROSE is a PBE framework developed by Microsoft Research. 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 creation of an efficient search algorithm. PROSE will automatically use it's own deductive search algorithm, which combines a bottom-up grammar enumeration with a top-down enumeration[4]. This search technique will allow us to make more complex grammars if desired. Additionally, PROSE's search algorithm has built-in APIs that allows us to rank resulting programs, guide the search via domain specific knowledge, and decompose the synthesis problem into subproblems[4].

4.2 MANTIS

The MANTIS method's main contribution is the augmentation of input-output examples through the application of "relational perturbation properties" [2]. We augment our image-based input-output pairs in this fashion to generate more constraints for our program synthesizer.

5 Problem Definition

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.

6 Solution

6.1 Context Free Grammar

```
program ::= single
single  ::=  input_image                |
            FilterColor(single, color) |
            Recolor(single, color)     |
            Orthogonal(single, axis)   |
            *Compose(single, single)   |
            **PickMax(multi, prop)     |
            **Compress(single)         |
            **ComposeGrowing(multi)
color ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
axis  ::= Y_AXIS | X_AXIS | ROT_90
**multi ::=   cut(single)
**prop  ::= NONZERO | SIZE | NUM_COLORS
            | X_COORD | Y_COORD | COMPRESSED_NONZERO
```

```
* (partially implemented)
** (future work)
```

6.2 Operator Semantics

1. **FilterColor(image, color)** $\rightarrow$ **image**: Set all pixels, except pixels at *color*, to 0.
2. **Recolor(image, color)** $\rightarrow$ **image**: Set all nonzero pixels to *color*.
3. **Orthogonal(image, axis)** $\rightarrow$ **image**: Perform a reflection or rotation.
4. **Compose(image, image)** $\rightarrow$ **image**: Stack 2 images (first argument on top of second), where 0 is treated as "transparent."
5. **Cut(image)** $\rightarrow$ **[image]**: Cut an image into at least two disjoint pieces where the smallest piece is as big as possible.
6. **PickMax([image], prop)** $\rightarrow$ **image**: Picks the image with maximum property: nonzero pixels, size, number of colors, x-coordinate, y-coordinate, nonzero pixels after compress.
7. **Compress(image)** $\rightarrow$ **image**: Extracts minimal sub-image containing all non-zero pixels.
8. **ComposeGrowing([image])** $\rightarrow$ **image**: Stack images sorted by number of nonzero pixels (fewest on top).

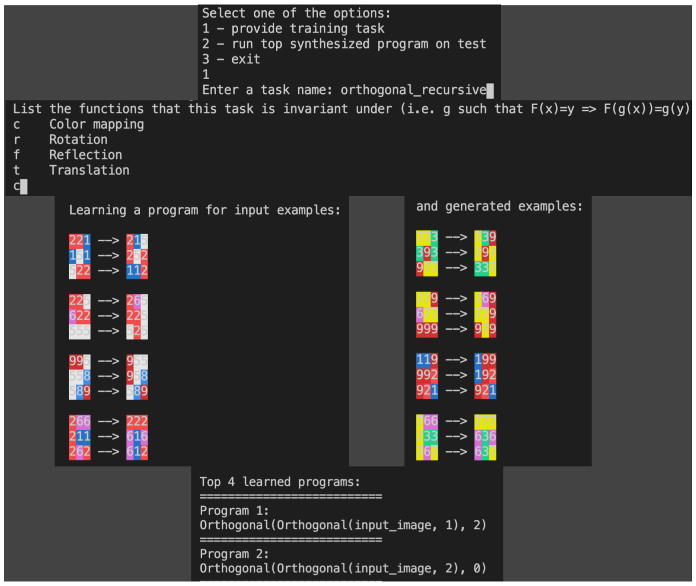
6.3 Data Augmentation - Mantis Integration

For many of these types of tasks, we can augment the provided examples by applying color maps and rigid transformations to the given examples. Inspired by MANTIS, we enable users to manually label our tasks based on what invariants they think would hold.

There are eight invariants of a program P that the user can specify. The first class of invariants involves functions f such that $P(x) = y \implies P(f(x)) = f(y)$. The second class involves functions f such that $P(x) = y \implies P(f(x)) = y$. The set of functions f in both classes are color mapping, rotation, reflection, and translation.

6.4 Search Space Optimization

We first verified the integrity of our DSL implementations by doing a naive enumeration of all potential programs up to a fixed depth, and then checking if (1) a program could be synthesized and (2) whether or not a synthesized program would then be successful on test output.



```
Select one of the options:
1 - provide training task
2 - run top synthesized program on test
3 - exit
1
Enter a task name: orthogonal_recursive
List the functions that this task is invariant under (i.e. g such that F(x)=y => F(g(x))=g(y)
c    Color mapping
r    Rotation
f    Reflection
t    Translation
c
```

```
Learning a program for input examples:        and generated examples:

221 --> 215                                   773 --> 739
151 --> 252                                   393 --> 797
522 --> 112                                   977 --> 337

225 --> 265                                   779 --> 769
622 --> 225                                   677 --> 779
555 --> 525                                   999 --> 979

995 --> 955                                   119 --> 199
558 --> 958                                   992 --> 192
589 --> 589                                   921 --> 921

266 --> 222                                   766 --> 777
211 --> 616                                   733 --> 636
262 --> 612                                   767 --> 637
```

```
Top 4 learned programs:
=========================
Program 1:
Orthogonal(Orthogonal(input_image, 1), 2)
=========================
Program 2:
Orthogonal(Orthogonal(input_image, 2), 0)
```

Fig. 3: Example rotation + reflection task with augmented examples based on color permutation.

Even with just 23 potential function applications, we faced pretty severe consequences for our computational complexity as we increased depth (23^d possible programs at depth d). Depth 3 took a matter of minutes, 4 took an hour, and depth 5 would have taken an entire day.

This naive enumerator was a useful reference for what orders of magnitude we were facing at each depth, and was used to refine our DSL and implementations.

PROSE allows us to decimate the search space of naive enumeration using witness functions. Witness functions operate on each parameter of every production rule in our DSL. Witness functions convert the specification of an operator into specifications on the operator's parameters. The generic task of the witness functions is to compute the set of arguments that, when supplied to a specific operator, would result in the expected output. PROSE's search works by continually applying these witness functions until eventually we arrive at a preimage that is equivalent to the input image. The training output image is passed through a witness function, which returns a set of preimages. If any of those preimages match our input, then we have found a correct program, if not we continue searching by passing all preimages through witness functions. This process continues until a program is found or PROSE hits the specified maximum recursion depth.

Take a simple filter program as an example:

```
            3 1 4 0                      0 1 0 0
input:      8 1 3 4          output:     0 1 0 0
            1 9 9 9                      1 0 0 0
```

The output above would eventually be passed to our **FilterColor** witness functions. For the color it is trivial to see that the color parameter for FilterColor would have to be a 1 as that is the only non-zero pixel in the output. The preimage computation is a bit more complex and results in the following set of preimages.

```
                0 1 0 0         3 1 4 0         9 1 9 9
preimages:  {   0 1 0 0   ...   1 1 1 1   ...   9 1 9 9  }
                1 0 0 0         1 9 9 9         1 9 9 9
```

As you can see, the input is in our set of preimages for **FilterColor**. So PROSE can terminate here because it has found a program that will transform the input into the output. A handwritten ranking function is used to determine the desirability of this program compared to all other correct programs.

In order to encode this very large set of possible images, we designed a compact specification, which we called an Abstract Image. An Abstract Image is a 2D array of bit vectors, each specifying the set of colors the pixel can take on. This representation is powerful enough that we will always be able to represent the set of preimages using 1 Abstract Image. So the example above would become

```
            1111111101 0000000010 1111111101 1111111101
preimage:   1111111101 0000000010 1111111101 1111111111
            0000000010 1111111101 1111111101 1111111101
```

This Abstract Image covers the space of preimages for our color filter example. An important use case of Abstract Images is determining if an image is in the set that it represents. This is critical for PROSE to know if the input image matches the specification generated by a transformation. We use the function below (in pseudocode) to compare an Abstract Image to a normal image.

```
bool CorrectOnProvided(state, candidate_image):
    abstract_space = AbstractImages[state]
    for i in pixel_indices:
        if candidate[i] is not in abstract_space[i]:
            return False
    return True
```

7 Evaluation

With our DSL limited to only FilterColor, Recolor, and Orthogonal operations and a maximum depth of 5, we were able to successfully solve 8 tasks from our limited ARC training set of 179 total tasks [5]. all of the successfully-synthesized programs involved no more than 2 function calls, and only ever used the "Orthogonal" function. Thus, we still need to add more DSL features to achieve less trivial solutions.

We verified these solutions using both a naive enumerative search (up to depth 4) and our PROSE-based solution, $^{A}_{R}{}^{C}_{H}{}^{E}_{S}$ (up to depth 5). While our depth-4 enumerative search took nearly an hour to complete on the 4 cores of a 2015 MacBook Pro, $^{A}_{R}{}^{C}_{H}{}^{E}_{S}$, with depth 5, took only 23 minutes on the same machine. This is thanks to the smart pruning we do within our PROSE witness functions.

If we extrapolate, our naive enumerative search for depth 5 will have taken an entire day, which makes our PROSE solution is on the order of 20 times faster.

8 Conclusion

The main path to continue work on $^{A}_{R}{}^{C}_{H}{}^{E}_{S}$ is to add functionality to the DSL to make it more expressive. The **Compose**(image, image) operator itself would add tremendous expressiveness. See Section 6.1 for more possible DSL improvements. Although PROSE greatly improves upon our enumerative search's runtime, it comes at a development cost, as witness functions, ranking scores, and new specifications are all necessary for adding functionality, all of which become more important and more complex as the DSL grows.

Our novel approach, $^{A}_{R}{}^{C}_{H}{}^{E}_{S}$, allowed us to constrain a huge search space using our abstract image specification, and augment our limited examples using MANTIS-like invariants. We believe this work will push PROSE out of the domain of text based synthesis, and into the realm of fast, efficient, frameworks that solve general tasks in AI.

Bibliography

[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.* IOS Press, February 2019.

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