Internship report:
Program synthesis for ARC

Louis Jalouzot

ENS Lyon

June-July 2021

During my first year as a computer science student at ENS Lyon, I carried out an 8-week internship in machine learning. The goal was to use program synthesis to attack François Chollet's ARC challenge. I was supervised by Nathanaël Fijalkow[1] and Charles Grellois[2] and we were assisted by Théo Matricon[3].

Contents

1 Introduction **3**

1.1 AGI . . . 3

1.2 The ARC dataset . . . 3

1.3 Outline . . . 4

2 The program synthesis approach **5**

2.1 Simply typed lambda calculus . . . 5

2.2 DreamCoder . . . 6

2.3 Improved program exploration: DeepSynth . . . 6

3 Toward ARC **8**

3.1 Restriction to objects manipulation problems . . . 8

3.2 High-level vs. Domain Specific . . . 9

3.3 Difficulties . . . 10

[1] CNRS, LaBRI
[2] CNRS, LIS
[3] LaBRI

4 Problem generation **11**

4.1 Input grids generation 11

4.2 Difficulty I program generation 12

4.3 Program generation using mutation 12

5 Neural network predictions **16**

5.1 Embedding 16

5.2 Neural network structure 16

5.3 Target/predictions 17

6 Conclusions **18**

6.1 Unsatisfying predictions 18

6.2 The more NNs the merrier 18

6.3 Summary 19

A Examples of ARC tasks **21**

B DreamCoder **22**

C My DSL for ARC **23**

D Difficulties **24**

D.1 Examples 24

D.2 General tree shapes 25

E Problem generation examples **26**

E.1 Random grid generation 26

E.2 Correlated grid generation 27

E.3 Problem generation 28

F Task embedding outline **29**

1 Introduction

1.1 AGI

Artificial intelligence is a thriving field of study. It has given birth to new methods and algorithms able to outperform humans for the first time in several intelligent tasks (such as chess or go). However, most of the time those algorithms are only good at the task they were designed for. Humans are more versatile. The subject of creating an artificial intelligence able to understand and learn any task a human brain can is called **Artificial General Intelligence** (AGI).

A more modest subject is **meta-learning**. A meta-learning model or algorithm can be applied to multiple similar tasks and is said to learn to learn. Its performance on each task improves with experience and tasks. The goal of this internship was to design a meta-learning algorithm for the tasks of the ARC dataset.

1.2 The ARC dataset

François Chollet, a Google researcher, published a paper titled *On the Measure of Intelligence* [1] where he proposed a general definition of intelligence and a way of measuring it. He also highlighted how far we are from reaching AGI and proposed a dataset named ARC which is a perfect example of a meta-learning problem. However, today's best algorithms designed for ARC are not even close to partially solve it.

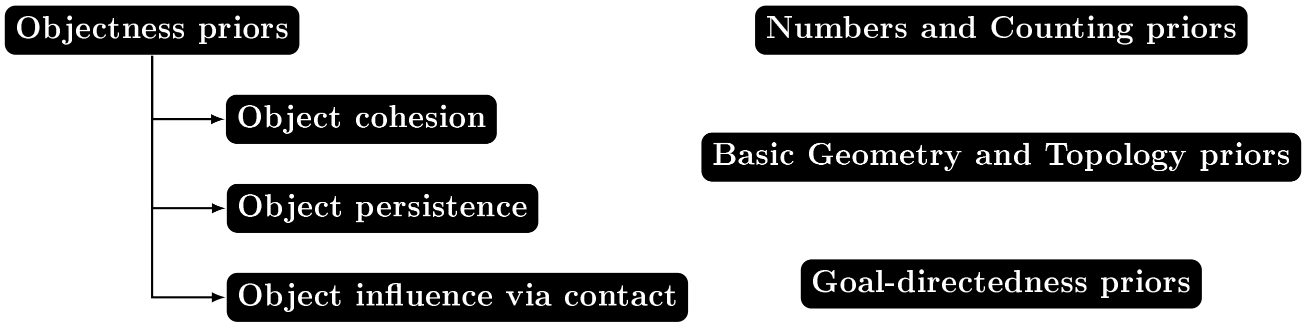
General outline

ARC, which is short for **Abstraction and Reasoning Corpus**, is a set of 1000 similar tasks (or problems) created by hand by François Chollet. They are divided in 3 subsets, a training set of 400 tasks, a public evaluation set of 400 tasks and a private evaluation set of 200 tasks. The latter is not available and is for François Chollet to test if algorithms proposed by the community are really general and use meta-learning (for instance to check if they do not only feature hard-coded solutions for each public task).

Priors

Despite being impossible to solve with a machine for the moment, this dataset can easily be solved by any human without preparation. It was the goal of François Chollet and this is why it is so relevant for machine learning.
When looking at a new task, even though they know nothing about it, humans do not start from scratch. They have notions (or priors) that are innate or that they learned before. For instance, humans do not need to learn again what is an object or what is a straight line before using it. This is the reason why

the author wants the difficulty of ARC to reside in finding how to use those priors to solve tasks and not in discovering them. Therefore a solution to ARC can and should use priors.
The types of priors François Chollet estimated to be useful and sufficient for solving ARC are:


Objectness priors
Object cohesion
Object persistence
Object influence via contact
Numbers and Counting priors
Basic Geometry and Topology priors
Goal-directedness priors

(see [1] for further details).
For this internship we chose priors to be hard-coded black-box functions.

In practice
ARC tasks are composed of a few input/output examples and, most of the time, one test input. An input or an output is a grid whose width and height are at most 30 and each pixel can take 10 different colors.
The input/output examples are for understanding what is the task, establishing a link between an input and its output so as to know what to do with the input test grid. ARC tasks are unambiguous.
For a few example tasks, see Appendix A.

1.3 Outline

We attacked the ARC challenge using program synthesis led by predictions from a neural network. I will introduce this method in Section 2[4]. Section 3 will be focused on how to use programs for solving certain ARC tasks. The designing of a neural network will be addressed in Section 5 but a neural network needs data for its training so we will first see how to generate labeled/solved ARC tasks in Section 4.

[4] after this point my work is more specific to ARC

2 The program synthesis approach

As stressed by François Chollet, ARC can be better understood as a program synthesis benchmark. For a task we would like to generate programs that act on grids and that match the input/output examples and then apply them to the input test grid. We should also create a **Domain-Specific Language** (DSL) which is essentially the hard-coded priors, which we will now call **primitives**, and some other functions that might be useful for creating interesting programs.

2.1 Simply typed lambda calculus

We chose that a program would be a **simply typed lambda term** from the following **context free grammar** (CFG):

$$p \quad := \quad c \quad | \quad \lambda\, x : \; p' \quad | \quad p_1\, p_2 \quad | \quad x$$

Where the constants c would be the (typed) primitives of a DSL.
The basic types I used are integers (int), booleans (bool), objects (obj) and colors (color).

De Bruijn indices

De Bruijn index is a useful way of describing the bound variables in a lambda term without naming them. The variable i in a lambda term would then refer to the i-th closest lambda abstraction it is in. For instance the lambda term $(\lambda\, x :\; \lambda\, y :\; x\; y)$ can simply be written $(\lambda :\; \lambda :\; 1\; 0)$.
We will write De Bruijn indices either var0, var1 etc. or Variable(0), Variable(1), etc.

I will often write lambda terms as trees, which is more intuitive. See this example.

PCFG

Important objects that will be at the center of our program synthesis are **Probabilistic Context Free Grammars** (PCFG). They are CFGs with probabilities on their derivation rules and the probabilities of the rules for a certain non-terminal sum up to 1. Then it is possible to sample a PCFG with the `SamplePCFG` method:

```
Algorithm 1: SamplePCFG
  input : P a PCFG
  output: t a term of the (P)CFG

  t ← S (start symbol)
  while t has a non-terminal symbol do
      A ← left-first non-terminal of t
      sample a rule to apply to A according to the probabilities from P
  return t
```

Now let us see 2 projects, DreamCoder and DeepSynth, on program synthesis which ideas will be useful for our purpose.

2.2 DreamCoder

DreamCoder [2] is a complete meta-learning algorithm. It uses program synthesis led by neural predictions. This algorithm alternates wake and sleep phases. See Appendix B for an outline of DreamCoder.

Wake phase

During the wake phase, DreamCoder takes the tasks it is given, gives them to a neural network which returns predictions for creating a PCFG from a DSL/CFG. Then it samples this PCFG and evaluates the resulting programs on the task to see if it fits the input/output examples.
They take the assumption that a shorter program is more likely than a longer one for solving a task.

Sleep phase

The sleep phase features an abstraction (or compression) phase and a dreaming phase.
During the **abstraction**, DreamCoder goes through the correct programs it has found when awake and searches for recurring patterns. Then it adds them to its DSL/library, so that future programs using this pattern (which are likely to be useful since widely used) are much shorter so easier to find.
When **dreaming**, the algorithm both replays the tasks it has solved during wake phase and creates (and tries to solve) new tasks for training its neural network.

2.3 Improved program exploration: DeepSynth

DreamCoder is computationally intensive so improving any part of it could make it easier to use and maybe lead to better results. DeepSynth [3][5] is being developed by Nathanaël Fijalkow and his team and proposes two new algorithms for sampling PCFGs. One of them is more efficient than the A^* algorithm used in DreamCoder.

Heap Search

The first method is Heap Search and is an enumeration method.
Like A^*, Heap Search enumerates programs optimally, i.e. in non-increasing order of probability[6]. However, thanks to a heap-like data structure, Heap Search is (unlike A^*) bottom-up which allows to evaluate programs at the same time as they are generated and to avoid reevaluating a subprogram that might appear multiple times in a program or in several programs.

[5] no paper has been published yet

[6] the probability of a program within a PCFG is the product of the probabilities of the derivation rules to use to generate this program from the start symbol

SQRT Sampling

The other one, SQRT Sampling, is probabilistic and consists of replacing the distribution $\mathcal{D}$ on the programs given by a PCFG by another distribution $\sqrt{\mathcal{D}}$ called the SQRT distribution of $\mathcal{D}$. This distribution is defined by

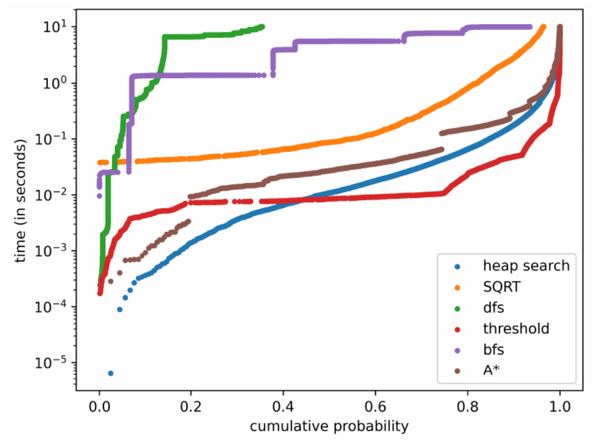
$$\sqrt{\mathcal{D}}(p) = \frac{\sqrt{\mathcal{D}(p)}}{\sum_q \sqrt{\mathcal{D}(q)}}$$

If $\sum_p \sqrt{\mathcal{D}(p)} < +\infty$, sampling according to the SQRT distribution is loss[7] optimal among all sampling algorithms. Otherwise all sampling algorithms have infinite loss.

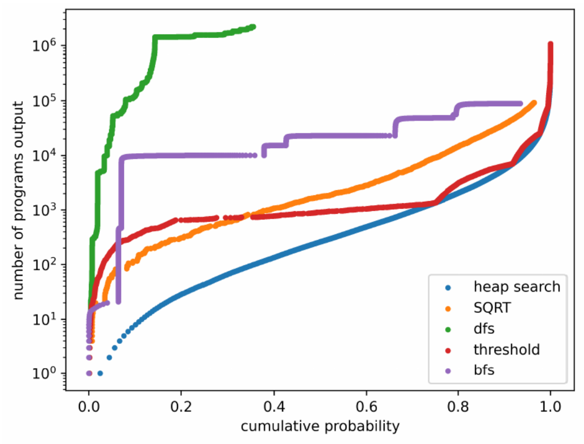
Performance

To compare the performance of Heap Search, SQRT Sampling and other algorithms, Nathanaël Fijalkow and his team sampled a batch of programs from a given PCFG then asked the algorithms to search for them. On Figure 1 part (a) there is the evolution of the cumulative probability (i.e. the sum of the probabilities of the programs from the batch that the algorithms have already found) over time. Part (b) displays the evolution of the number of programs that the algorithms have explored (for finding the programs from the batch) over the cumulative probability. Note that A* has no plot on (b) since it would be the same as Heap Search, both optimally exploring programs, i.e. in non-increasing order of probabilities. As shown on (a), Heap Search is faster than any other algorithm except Threshold after 0.4 cumulative probability. However, as shown on (b), at 0.4 cumulative probability Threshold has already explored 10 times more (useless) programs than HeapSearch. Which would make Threshold much slower if evaluation was needed.

Figure 1: Performance of Heap Search and SQRT Sampling (Figure from [3])



(a) Cumulative probability against time in log-scale.



(b) Cumulative probability against number of programs output in log-scale.

[7] for the loss $\mathcal{L}(\mathcal{D}', \mathcal{D}) = \mathcal{E}_{p \rightsquigarrow \mathcal{D}}\left[\frac{1}{\mathcal{D}'(p)}\right]$, $1/\mathcal{D}'(p)$ being the expectation of the number of tries for $\mathcal{D}'$ to draw p, i.e. the expectation of success for the geometric distribution with parameter $\mathcal{D}'(p)$

3 Toward ARC

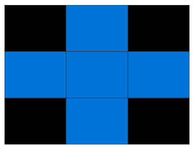
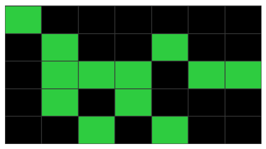
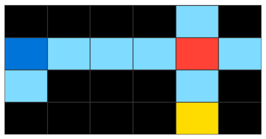
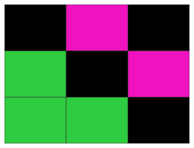
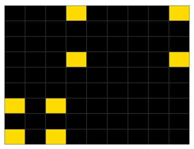
In this section I will explain how to use programs to solve a certain type of ARC tasks.

3.1 Restriction to objects manipulation problems

Objects

Given the complexity of solving the ARC dataset we chose to focus on tasks involving objects and their manipulations, i.e. tasks using only priors about objects. On Appendix A the two first examples are *objects*-problems (the ones of our interest), the third one is an *area*-problem and the last one is some kind of *goal-directed*-problem.

Then we had to decide how to find the objects on a given grid. I regarded objects as lists of their pixels and considered 5 types of cohesion:

Cohesion type	**Example**	**Cohesion type**	**Example**
Contact and color		Contact by point and color	
Contact		Contact by point	
Color			

We also made the assumption that the background is black (black pixels could not belong to an object). This assumption induces biases on the whole process since some ARC objects-problems use backgrounds of other colors.

The idea then was to take for granted object recognition and focus on programs that would take in the list of objects seen on the input grid and produce another list of objects for an output grid: programs of type `List obj` $\rightarrow$ `List obj`.

For choosing the output grid size and the cohesion type see Section 6.2.

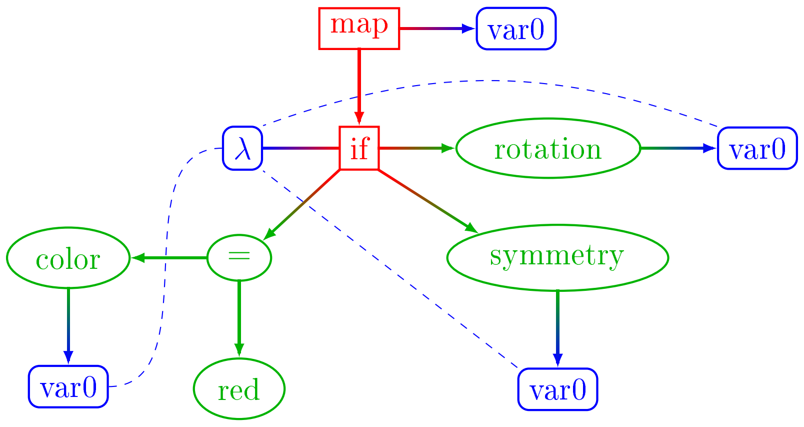
DSL

After that came the moment of designing the DSL so that programs written with its primitives could have the requested type and could handle objects in an interesting way. On Appendix C are the primitives I chose and implemented in Python for ARC along with their type.

A first issue

While testing if I could write solution programs for actual tasks of ARC using my DSL, I encountered the necessity of using lambda abstractions so that the programs would not be ridiculously large. For instance if I wanted to apply a symmetry or a rotation to all objects according to their color, I needed to have a copy of them within several arguments of the same primitive.

Tree of a program[8] for which the issue arise :



This is an issue because in practice we restrain ourselves to only generate programs without lambda abstractions for feasibility.

This *argument duplication* issue, illustrated with dashed blue links, will be solved in the next section.

3.2 High-level vs. Domain Specific

We propose that the primitives of a DSL in general (not only for ARC) could be divided into two subgroups:

- The **domain-specific** or **green** primitives which are really specific to the domain. For instance for ARC those primitives are the ones, such as symmetries or rotations, that manipulate objects.
- The **high-level** or **red** primitives are more utilitarian primitives which appear in multiple DSL for unrelated domains and often have a polymorphic type. The `map` and `if` primitives are two examples.

[8] for simplicity, we write programs with an unbound variable (like the var0 of map) which refers to the list of objects on which we will apply the program (if p is the program we generated and l is the list of objects, we consider $(\lambda :\ p)\ l$)

Deciding whether a primitive is a **domain-specific** or a **high-level** one is not always obvious. See Appendix C for the classification of the primitives of my DSL.
Dividing primitives into those subgroups is useful because high-level ones are hard to predict in practice. Indeed when using a neural network trained for predicting the presence or not of primitives in the solution program of a task, the predictions are often bad for the high-level primitives but not for the domain-specific ones. Actually, asking the network to predict those primitives can even mess up the predictions about the others. Unlike domain-specific primitives, high-level primitives do not *leave a trace* on the grids on which a program containing them is applied.

3.3 Difficulties

Then I defined the **difficulty** of a program as being the maximum number of times a path from the root to a leaf of the program written as a tree changes from red to green primitives. See Appendix D.1 for examples of a program of difficulty I and a program of difficulty II and Appendix D.2 for the general tree shapes of difficulty I and difficulty II programs.

This definition is convenient in the way that it allows to restrain and split the generation. First we can search for a potential solution of difficulty I then of difficulty II etc. Furthermore when generating a program of difficulty I, we only generate d subprograms[9] of green primitives, which we will call *green blocks*, of arbitrary types $\tau_1, \ldots, \tau_d$ and one subprogram of red primitives (*red block*) of type

$$\tau_1 \to \ldots \to \tau_d \to \texttt{List obj} \to \texttt{List obj}$$

then apply it to the green blocks to get a program of correct type `List obj` → `List obj`. Blocks can be generated from specific CFGs.

This generation solves the *argument duplication* issue. For instance the example program can be generated by applying the red block[10]:

$$\lambda : \ \lambda : \ \lambda : \text{map } (\lambda : \text{if var1 var2 var3}) \text{ var3}$$

of type

$$\text{bool} \to \text{obj} \to \text{obj} \to \text{List obj} \to \text{List obj}$$

to the green blocks:

$$\begin{aligned} &(\text{color var0}) = \text{red} && : \text{bool} \\ &\text{symmetry var0} && : \text{obj} \\ &\text{rotation var0} && : \text{obj} \end{aligned}$$

A top-down generation makes it possible to generate blocks that access variables from blocks above like here.

[9] We took $d = 2$ which seemed to be enough to solve a variety of tasks
[10] in practice we would obtain this block using 2 red blocks with an empty green block between them

4 Problem generation

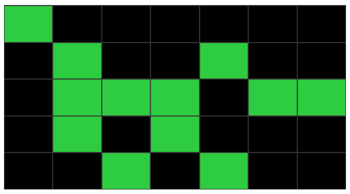
To get PCFGs we need to weight CFGs. To do that we need probabilities for the derivation rules, which we will get using a neural network and supervised learning.
But for training neural networks it takes labeled data, i.e. in our case tasks and their solution program. We cannot directly use the training set of ARC since we do not have the solutions.
I decided to generate example input grids on one hand and programs on the other hand. Then I would evaluate programs on input grids to get output grids, forming a task for which I already have the solution program.

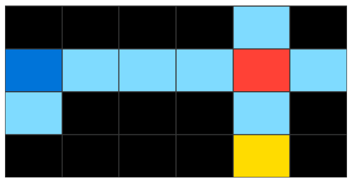
4.1 Input grids generation

Fully random generation
First of all I collected around 4000 objects from 150 tasks of the actual ARC training set[11] and made sure that they were sufficiently dense and not too big so that I would not collect backgrounds or crosses delimiting areas. Then I collected statistics about the dataset like:

- How many objects are of size 1? Of size 2? ...
- How many input grids have 8 objects? ...
- What is the proportion of problems with constant input grid size?
- How many problems have 4 examples? ...

With those data I would be able to form a problem by sampling its characteristics according to their distributions in the training set so that it would resemble actual problems, and then to randomly put objects on the grids. See Appendix E.1 for examples of problems generated using this technique (there are no output grids since they would be obtained later by evaluating a program).
However those grids are not satisfying since picking objects at random leads us to put on the same grids objects that are unrelated like those two:





[11] To do that I had to go through the training set to select object-tasks and label them with a cohesion type so that I would not collect crazy objects

Correlated generation
To counter this problem I just reorganized my objects. I stored an object a along with a list of all the other objects that can be seen in a problem where a appears too[12].
Then to generate a grid, I would pick an object at random and then stick to using other objects that appear in the *list of neighbors* of this object for the rest of the problem. See Appendix E.2 for examples of problems generated using this method.

4.2 Difficulty I program generation

To generate programs to apply to those fresh grids I could not simply sample the CFG induced by the green and red primitives since there are way too many irrelevant programs i.e. programs that are constant, incorrect, that have useless green primitives in them (which we cannot allow for the sake of training a neural network) etc. like those two:

`map (symmetry_x symmetry_x) var0` (identity)
`fill 0 0 (0 - 1) 0 pink` (tries to create a pink rectangle in (0, 0) but of size -1×0)

One approach was to sample programs with difficulty I and with tiny red and green blocks so that the likelihood of them having useless green primitives was low. Furthermore I could easily check the sanity of tiny red blocks[13].
See Appendix E.3 for examples of problems generated using this program generation and the correlated grid generation.
Later on, while discarding more and more useless red blocks, I figured out that three of them would be enough to illustrate the green primitives of my DSL on generated problems:

```
map var0 var1
filter var0 var1
singleton (max var0 var1)
```

4.3 Program generation using mutation

The idea here was to use solution programs of actual ARC problems I wrote using my DSL and slightly modify them to obtain other interesting programs. The hope was that the structure of a solution program is likely to be relevant and might be passed on to its mutations.
My first mutation algorithm would only change one by one the primitives appearing in the program with another primitive of the same type and deterministically go through all the mutations of the

[12] Not recursively to keep relatively close objects
[13] For instance discarding a red block involving an "if 0 = 0"

initial program for this mutation process. With this technique I was sure not to change the structure of the program but the number of mutations would grow exponentially with the size of the initial program. The problem is that we cannot afford to go through all of the mutations and because the process is deterministic we can only explore a small portion of mutations[14] which are therefore close to one another for the mutation process and essentially identical. For instance those two programs are 4000 mutations apart yet almost the same:

```
λ: (λ: cons(translate(x_high var0 +1, y_low var0 +1, var1), singleton var1))
                                            (max y_low var0) (max -y_low var0)

λ: (λ: cons(relative_translate(y_low var0 +1,  y_low var0 +1), var1), singleton var1)
                                            (max y_low var0) (max -y_low var0)
```

Enhanced programs

The next method I implemented was based on manipulating what I will call *enhanced programs*. They are programs along with a type τ and the types of the variables (which are free within the program) in an environment in which the program has type τ. Intuitively an environment of types is (type(var0), type(var1), ...). Below is an example and as we can see, there is a `None` type for one of the variables since the program would have the same type no matter the type of this variable.

Program	`map var0 var2`
Environment of types	`((obj -> int), None, List obj)`
Type	`List int`

Enhanced programs harvesting

Since my idea was to generate programs by assembling interesting enhanced programs, I firstly collected them from the solutions programs I mentioned earlier. To do that I used a unifying algorithm and kept the small subprograms with an environment of types and their type in this environment (enhanced programs). Then, to get rid of biases in some of them (for instance we do not want `CHANGE_COLOR BLUE` to use just the color blue), I applied my first naive mutation algorithm to them and kept all the mutations which were in reasonable amount for small subprograms.

End-type and in-types

Here I will define the end-type and the in-types of a type which will be useful later.
Intuitively, if a type τ is the type of a function, the **end-type** of τ is the type of that function once it is given all the arguments it can receive.

[14] compared to the total number of mutations of the program

On the other hand, the **in-types** are the types of the arguments to give to that function so that it reaches its end-type.
If τ is not the type of a function, its end-type is τ and it has no in-types.
See those examples:

Type	**End-type**	**In-types**
int $\to$ obj	obj	int
(bool $\to$ bool) $\to$ (obj $\to$ obj) $\to$ int	int	(bool $\to$ bool) and (obj $\to$ obj)
List obj	List obj	$\emptyset$

Enhanced programs assembling
I stored those small enhanced programs according to their end-types so that I could easily pick one of them of desired end-type.
Here is the pseudocode for the generation algorithm which I would initially call with
$\tau =$ `List obj` and `e = [List obj]` to get a useful program:

Algorithm 2: Recursive method `GenerateEnhanced`

```
input  : $\tau$ a type, e an environment of types
output: a program p of type $\tau$ in the environment e

if $\tau$ = None then return Failure
else if $\tau = \tau_1 \to \tau_2$ then
    p ← GenerateEnhanced($\tau_2$, [$\tau_1$]+e)
    return ($\lambda$ : p)
else if $\tau$ is polymorphic then
    if $\tau$ is not instantiated then instantiate $\tau$ with an arbitrary type
    $\tau$ ← instantiation of $\tau$
p ← random enhanced program with end-type $\tau$
$e_p$ ← environment of types of p
$e_1$, $e_2$ ← CompareEnv(e, $e_p$)
forall in-type $\tau_{arg}$ of $\tau$ do
    arg ← GenerateEnhanced($\tau_{arg}$, $e_2$)
    p ← p (arg)
forall type $\tau_{env}$ in $e_1$ do
    arg ← GenerateEnhanced($\tau_{env}$, $e_1$)
    p ← ($\lambda$ : p)(arg)
return p
```

The `CompareEnv` method is used to check how much the two environments it is given can *overlap*. It outputs 2 environments:

- one of types that cannot come from the environment `e` in which `p` is being developed
- the other which is essentially `e` but within which certain `None` types may have been instantiated for `p` to have the correct type

See a few examples below for a better understanding:

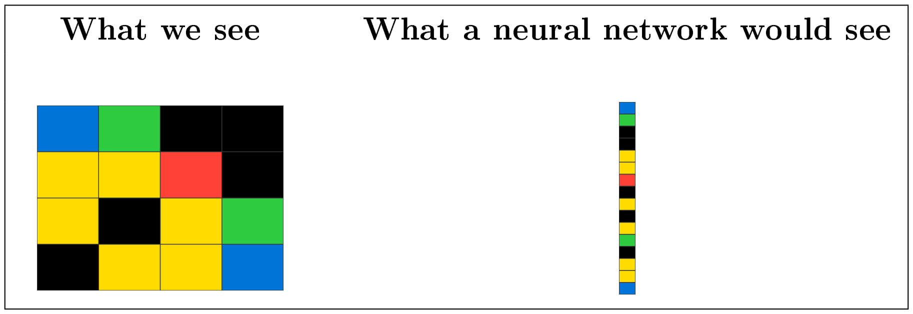
e	**e_p**	**Overlap**	**CompareEnv(e, e_p)**
	(None, int, obj)	None int obj	(None) & (int, obj, color)
(None, obj, color)		None obj color	
	(None, int)	None int	() & (int, int, color)
(int, int, color)		int int color	
	(int, obj)	int obj	(int, obj) & (bool, bool)
(bool, bool)		bool bool	

Combining
With grid and program generation in my pocket I just had to evaluate the programs on the grids to get entire problems and run a bunch of sanity checks to get rid of degenerated problems such as problems where the input and output grids are the same, when the program is constant on the input grids, when objects overlap after evaluation etc.

5 Neural network predictions

5.1 Embedding

Neural networks usually handle tensors of dimension 1 and of fixed size so we had to transform our labeled data into this kind of tensors through a process that is called **embedding**. The problem is that we would lose a dimension compared to the grids/images we had so far and we needed to encode objects well enough so that a neural network could benefit from our object recognition.



Duplicating grids
One idea to sort of tell the neural network about the existence of diverse objects was to duplicate each grid and on each copy make only one object appear. However the tensors have to be of fixed size so we had to make the assumption that a grid cannot have more than 10 objects[15] and that a problem has at most 6 input/output examples, ignoring the other ones, once again for feasibility.
With this embedding then we would take a problem, duplicate each of its grids into 10 grids of maximum size with only one object on each and filling with blank grids when a grid has less than 10 objects or when a problem has less than 6 examples.
At the end we get:

$$\text{max number of examples} \times \text{max number of objects} \times 2 = 120$$

grids for one problem no matter its size[16].
That embedding results in $120 \times 30 \times 30 = 108000$ entry parameters and still a multi-dimensional tensor but of fixed size.
See Appendix F for an outline of this embedding.

5.2 Neural network structure

This embedding would allow us to use an intermediate neural network, learning along with the main one, that can handle images/2-dimensional tensors before flattening our data into a 1-dimensional

[15] unfortunately in practice grids from ARC can have much more than 10 objects
[16] For encoding colors we tried either to encode them within the floats of the grid describing an object or on a specific grid (the latter making it 132 grids in total for a problem)

tensor. This network would play the role of a grid-features extractor. Here is one of the structures we tried written in PyTorch code:

```
nn.Sequential(
 nn.Conv2d(120, 60, 3), nn.ReLU(),
 nn.Conv2d(60, 30, 3),
 nn.MaxPool2d(2), nn.ReLU(),
 nn.Flatten()
)
```

This network, structured to work on images, could take in our multiple grids and squeeze them into a 1-dimensional tensor of size 5070 that we could reasonably feed to a classic linear network like:

```
nn.Sequential(
 nn.Linear(5070, 2048), nn.ReLU(),
 nn.Linear(2048, 512), nn.ReLU(),
 nn.Linear(512, output dimension),
)
```

5.3 Target/predictions

There are still important choices to make before being done with our neural network and it is about what it should be aiming for. Here we have to choose how many parameters the network should output (the `output dimension`) and make them correspond to the target, i.e. the label, to calculate a distance, a loss, which is intuitively how good or bad the network is doing.

The initial idea of DeepSynth [3] is to make the network output the probabilities of the derivation rules of a template CFG. Then, to weight the CFG with them to turn it into a PCFG and finally to compute the probability of the target/the solution program within this PCFG, probability that should be as close to 1 as possible.

However I did not use this technique because it implied predicting the red primitives along with the green ones which we do not want as stressed in Section 3.2. Also this technique requires a unique template CFG which I could not provide if I wanted to use the program generation of difficulty I or II that uses multiple CFGs (one for each type request of red and green blocks). I went for a simpler solution and decided only to try predicting the probability of a green primitive to appear within the solution program. Then I would simply ask the neural network to output as many parameters as there are green primitives and take the loss as being the mean square error of the parameters from the target tensor where there is a 1 if the primitive appears and a 0 otherwise (an embedded target).

6 Conclusions

6.1 Unsatisfying predictions

The predictions given by my neural network were not satisfying. It seems that it hardly learned anything, the predictions it gave were mostly constant. For instance it did not even learn colors because on tasks for which all the objects on the output grids were red, the use of the primitive red is obvious but the network did not even predict a higher probability for red than usual on other tasks. However the network seemed to have learned better the occurrences of `translate` and `relative_translate`, even distinguishing between them.
The reason for that is probably that my generation of data is not good enough, it is biased/not general enough and there are too many garbage tasks.
The network performed unexpectedly poorly on colors probably because of the embedding. It might be having troubles reading colors within float numbers and would maybe prefer one-hot tensors.
Improvements are needed on the task generation, on the embedding and on the structure of the neural network.

6.2 The more NNs the merrier

The whole process I proposed until here is not complete for solving object-oriented ARC tasks. I cannot feed my neural network with raw input/output grids from a task and the generated programs output lists of objects which have to be put on a grid.
Mainly 4 things are missing for completing the process. On a given task I need to find:

- the **cohesion type** of the objects on the grids, i.e. how to read the grids so as to create the lists of objects
- the **background colors of the task grids** (for reading them)
- the **size of the output grid** which can be more tricky than just taking the same size as all the other output grids or the size of the test input grid (see this example)
- the **background color of the output grid** (on which will be put the objects from the programs)

My idea was to design and train one neural network for each of those things, but it is yet to be done. The 2 first networks would take in the raw input/output grids from the task. The 2 others can either work on raw grids or on lists of objects, since they would be available at that time, or even on both.

6.3 Summary

The program synthesis background

The starting point of this internship was the general method of program synthesis led by neural predictions which is used by DreamCoder [2]. I already had implementations of types, programs, PCFGs and HeapSearch.

My contributions

I had to adapt this method to ARC. Chronologically, I first implemented the primitives of my DSL. Then, we divided the primitives into high-level and domain-specific primitives and I defined the difficulty of a program which led to generating programs with it. After that I focused on generating grids, first fully randomly, then in a more correlated fashion. I also tried my idea of generating programs with mutations which proved unsatisfying.
Finally, we designed a neural network and an embedding of tasks for training it.

References

[1] François Chollet. *On the Measure of Intelligence*, 2019.

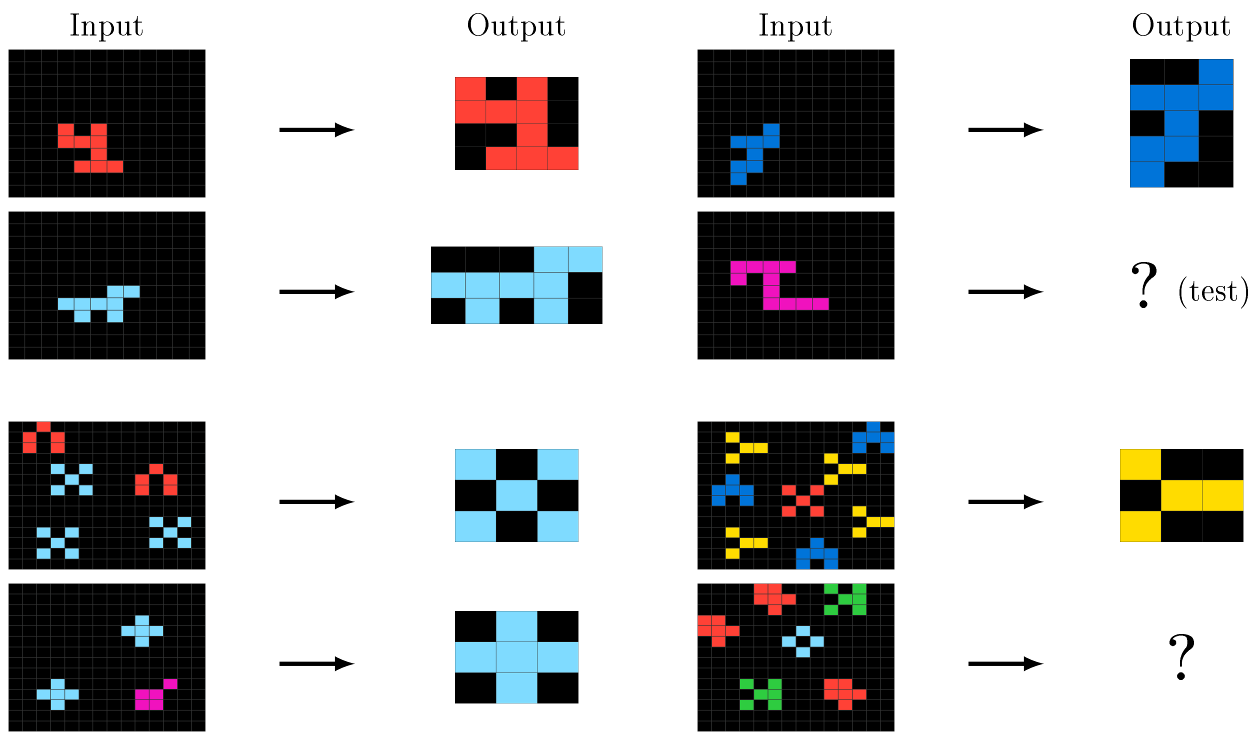
[2] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. *DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning*, 2020.

[3] Nathanaël Fijalkow. *DeepSynth, a general-purpose program synthesiser.* https://github.com/nathanael-fijalkow/DeepSynth, 2021.

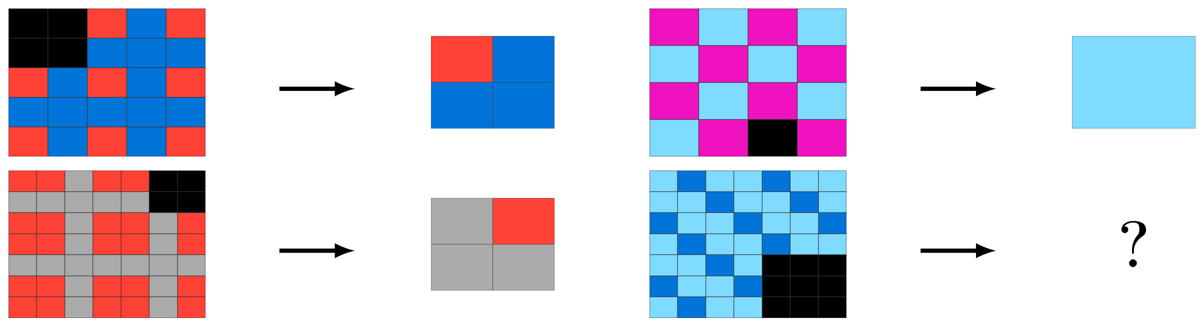
A Examples of ARC tasks

A few examples of ARC tasks with 3 input/output examples and 1 input test

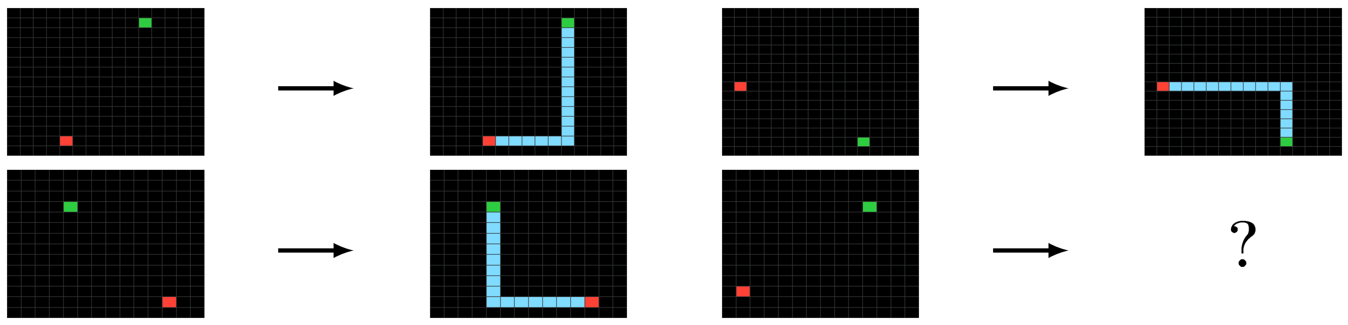
Objects (problems 1cf80156 and 39a8645d)



Areas (problem f9012d9b)

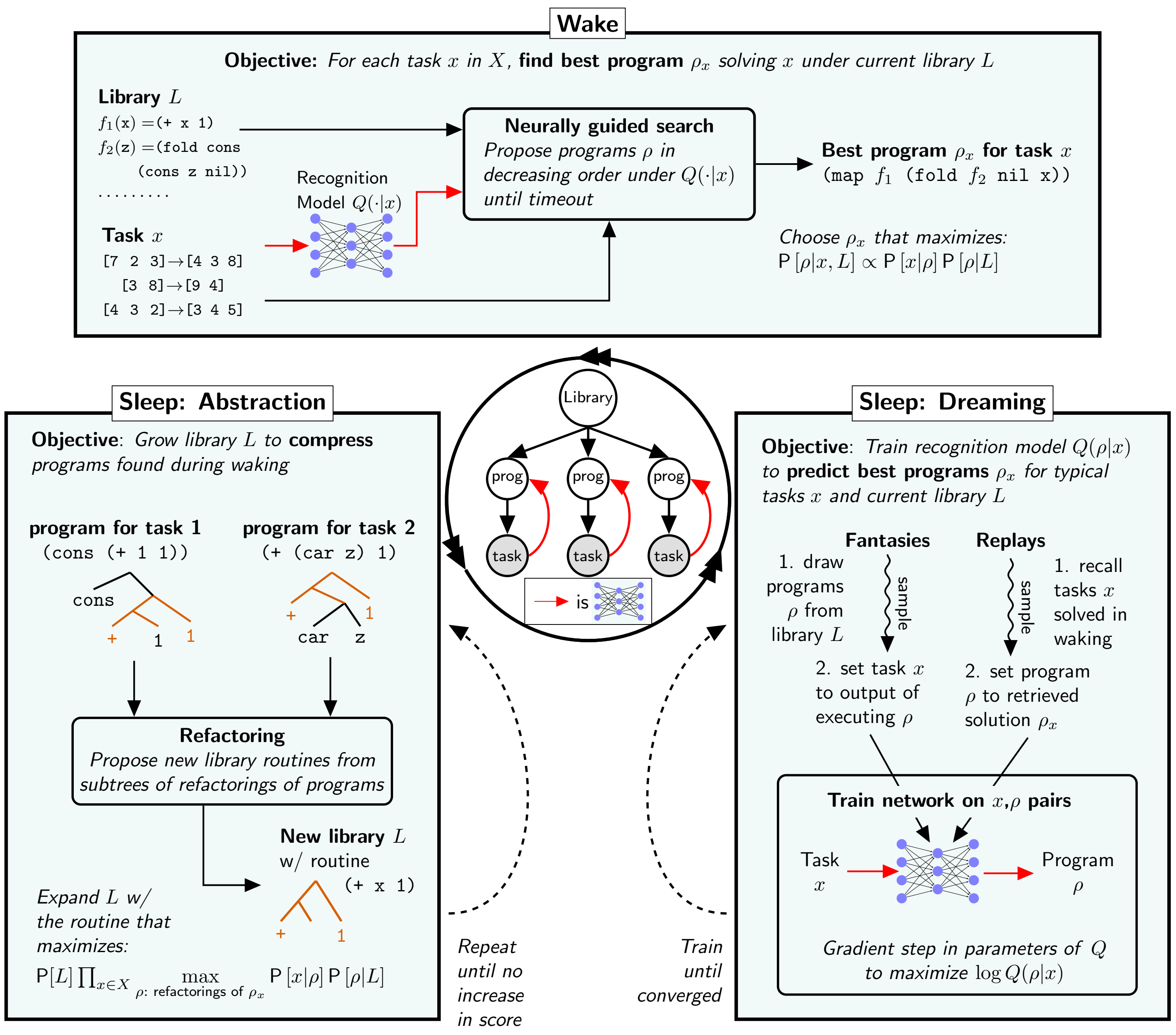


Goal-directed (problem a2fd1cf0)



B DreamCoder

Structure of DreamCoder (Figure from [2])


Wake
Objective: For each task x in X, find best program ρ_x solving x under current library L
Library L
f_1(x) =(+ x 1)
f_2(z) =(fold cons (cons z nil))
Recognition Model $Q(\cdot|x)$
Task x
[7 2 3]→[4 3 8]
[3 8]→[9 4]
[4 3 2]→[3 4 5]
Neurally guided search
Propose programs ρ in decreasing order under $Q(\cdot|x)$ until timeout
Best program ρ_x for task x
(map f_1 (fold f_2 nil x))
Choose ρ_x that maximizes:
$P[\rho|x,L] \propto P[x|\rho] P[\rho|L]$
Library
prog
task
is
Sleep: Abstraction
Objective: Grow library L to compress programs found during waking
program for task 1
(cons (+ 1 1))
program for task 2
(+ (car z) 1)
cons
+
1
car
z
Refactoring
Propose new library routines from subtrees of refactorings of programs
New library L w/ routine (+ x 1)
Expand L w/ the routine that maximizes:
$P[L] \prod_{x \in X} \max_{\rho:\text{ refactorings of } \rho_x} P[x|\rho] P[\rho|L]$
Repeat until no increase in score
Train until converged
Sleep: Dreaming
Objective: Train recognition model $Q(\rho|x)$ to predict best programs ρ_x for typical tasks x and current library L
Fantasies
1. draw programs ρ from library L
sample
2. set task x to output of executing ρ
Replays
1. recall tasks x solved in waking
sample
2. set program ρ to retrieved solution ρ_x
Train network on x,ρ pairs
Task x
Program ρ
Gradient step in parameters of Q to maximize $\log Q(\rho|x)$

C My DSL for ARC

The 16 high-level and 23 domain-specific primitives I chose and implemented to solve the object-tasks of ARC.
t0 and t1 are polymorphic types.

High-level

map
(t0 → t1) → List t0 → List t1
filter
(t0 → bool) → List t0 → List t0
max
(t0 → int) → List t0 → t0
if
bool → t0 → t0 → t0
singleton
t0 → List t0
cons
t0 → List t0 → List t0
car
List t0 → t0
gt?
int → int → bool
eq?
t0 → t0 → bool
or
bool → bool → bool
not
bool → bool
0/1
int
+/-
int → int → int
length
List t0 → int

Domain-specific

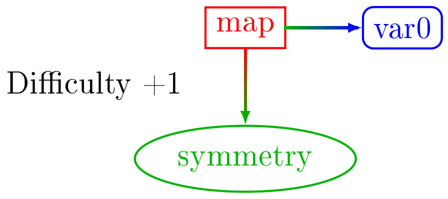
BLACK/BLUE/RED/. . .
color
translate/relative_translate
int → int → obj → obj
symmetry_x/symmetry_y
obj → obj
rotation90
obj → obj
fill
int → int → int → int → color → obj
change_color
color → obj → obj
color
obj → color
x_low/x_high/y_low/y_high
obj → int
size
obj → int

D Difficulties

D.1 Examples

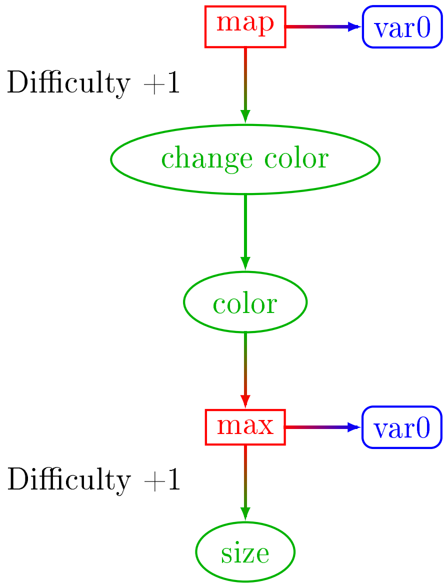
The first program is of difficulty I and the second one of difficulty II.
Difficulty is defined as the maximum number of red-to-green arrows on a path from the root to a leaf.

Difficulty I:



Applies a symmetry to all objects

Difficulty II:

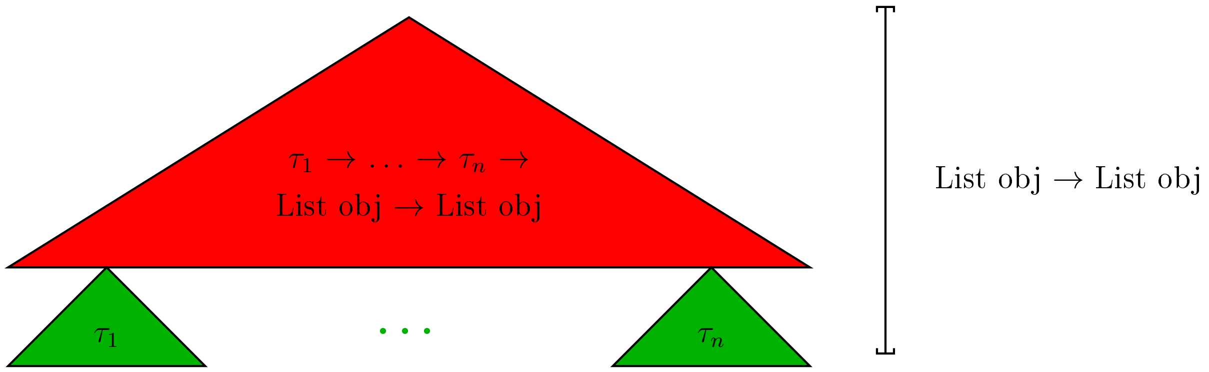


Applies the color of the biggest object to all the others

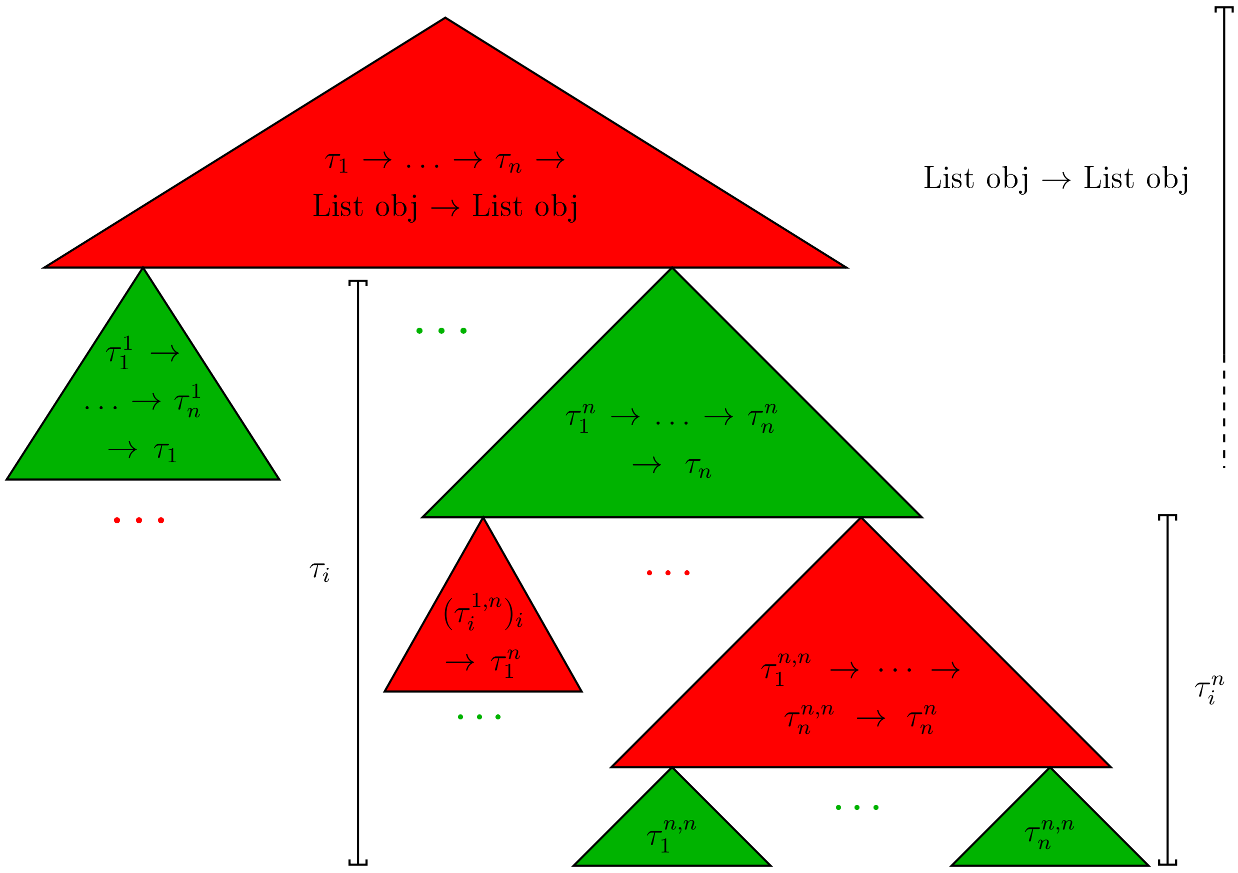
D.2 General tree shapes

General tree shapes of difficulty I (first) and difficulty II (second) programs with block types.

Difficulty I



Difficulty II



$\tau_1 \to \ldots \to \tau_n \to$
List obj → List obj
List obj → List obj
$\tau_1^1 \to \ldots \to \tau_n^1 \to \tau_1$
τ_i
$\tau_1^n \to \ldots \to \tau_n^n \to \tau_n$
$(\tau_i^{1,n})_i \to \tau_1^n$
$\tau_1^{n,n} \to \ldots \to \tau_n^{n,n} \to \tau_n^n$
τ_i^n
$\tau_1^{n,n}$
$\tau_n^{n,n}$

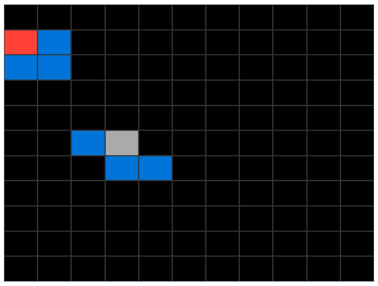
E Problem generation examples

E.1 Random grid generation

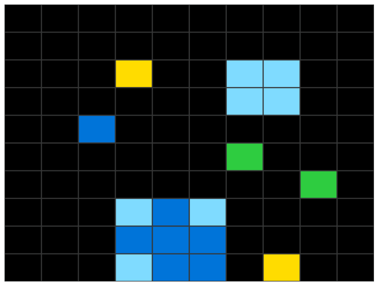
Examples of problems generated with the random grid generator. The goal is to evaluate a program on those input grids to get the corresponding output grids (the last one being the test pair).

First problem

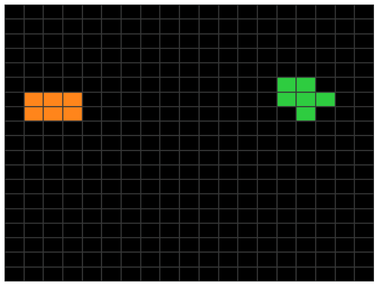
Example 1 input:



Example 2 input:

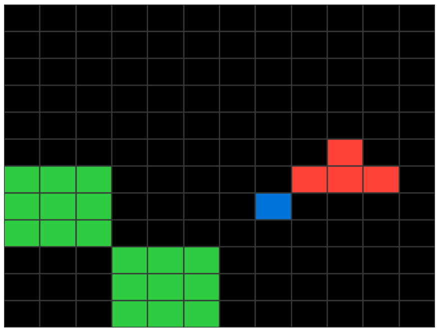


Test input:

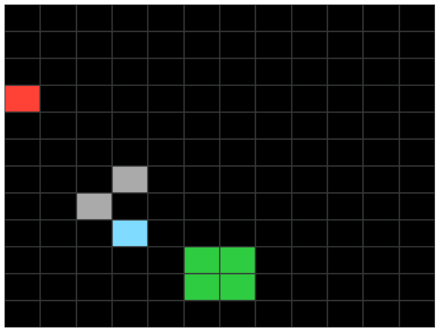


Second problem

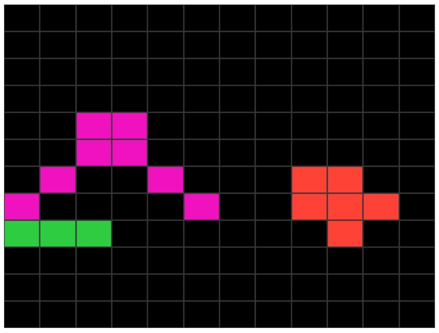
Input 1



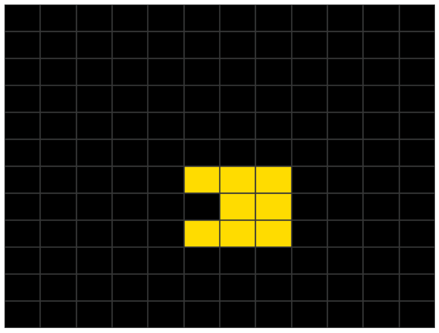
Input 2



Input 3



Test

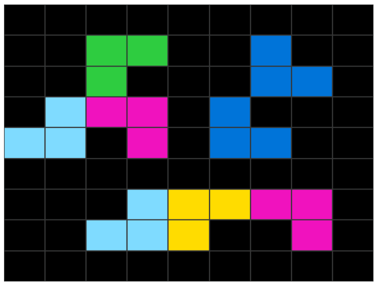


E.2 Correlated grid generation

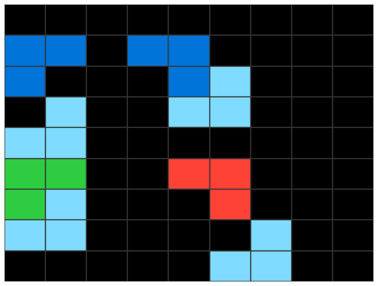
Examples of problems generated with the correlated grid generator.

First problem

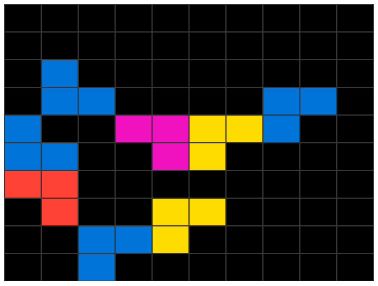
Example 1 input:



Example 2 input:

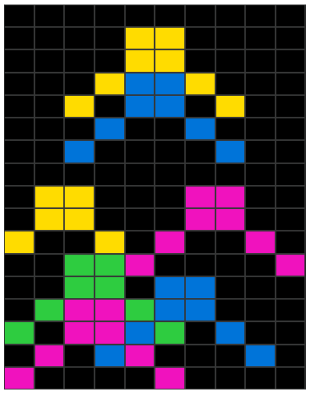


Test input:

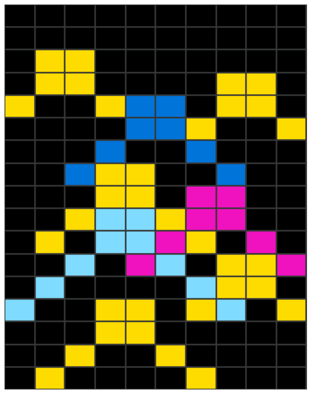


Second problem

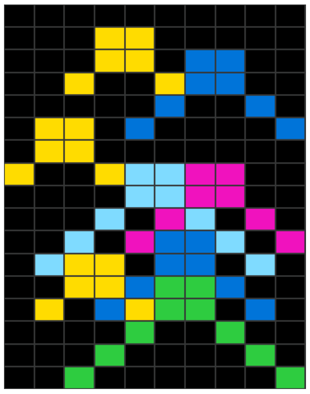
Input 1



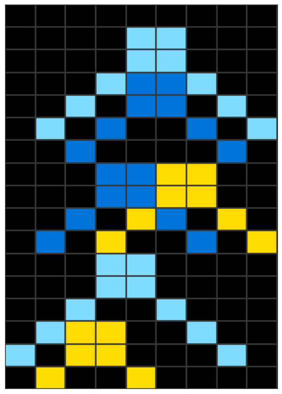
Input 2



Input 3

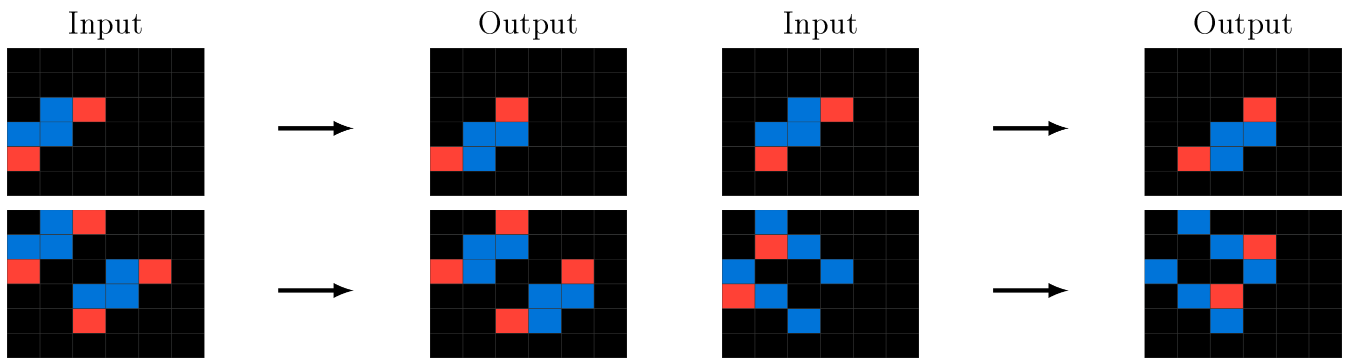


Test

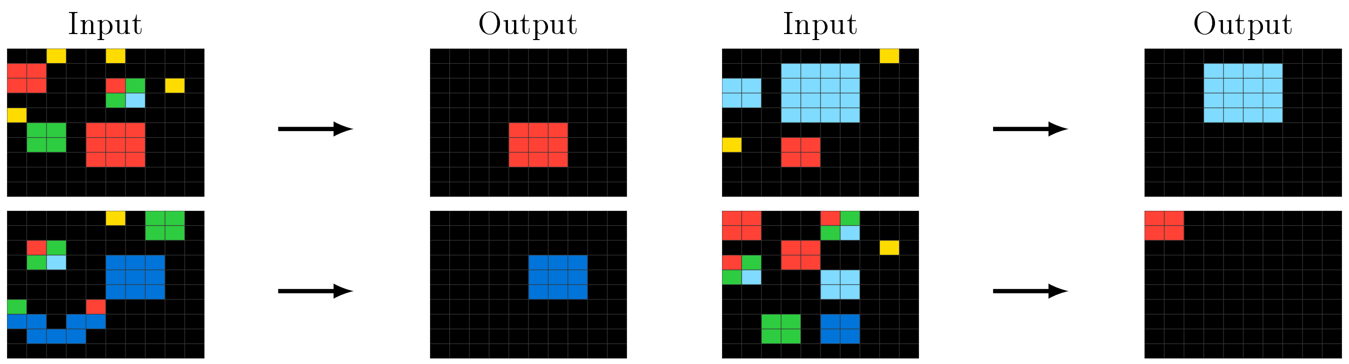


E.3 Problem generation

Examples of problems generated using correlated grid generation and difficulty I program generation.

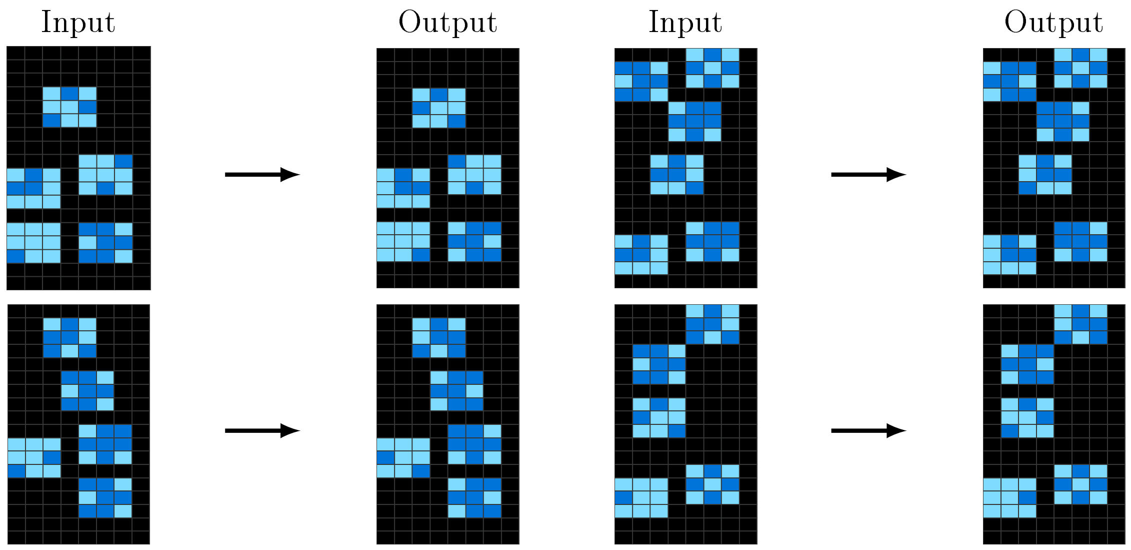


Solution: (λ: map var0 var1) (λ: rotation (rotation var0))



(ambiguous tasks)

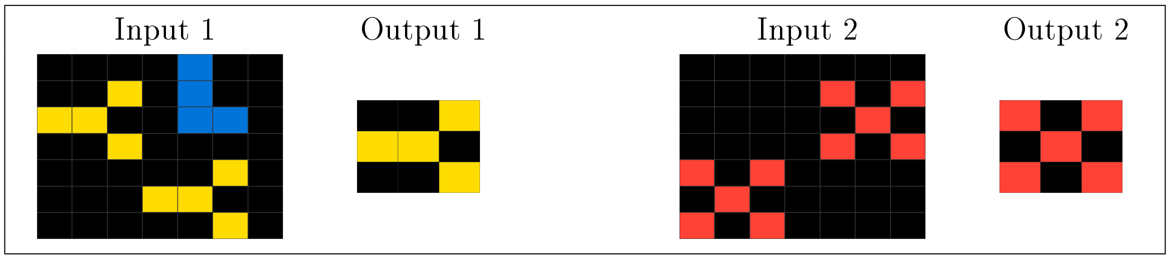
Solution: (λ: singleton (max var0 var1)) (λ: size var0)

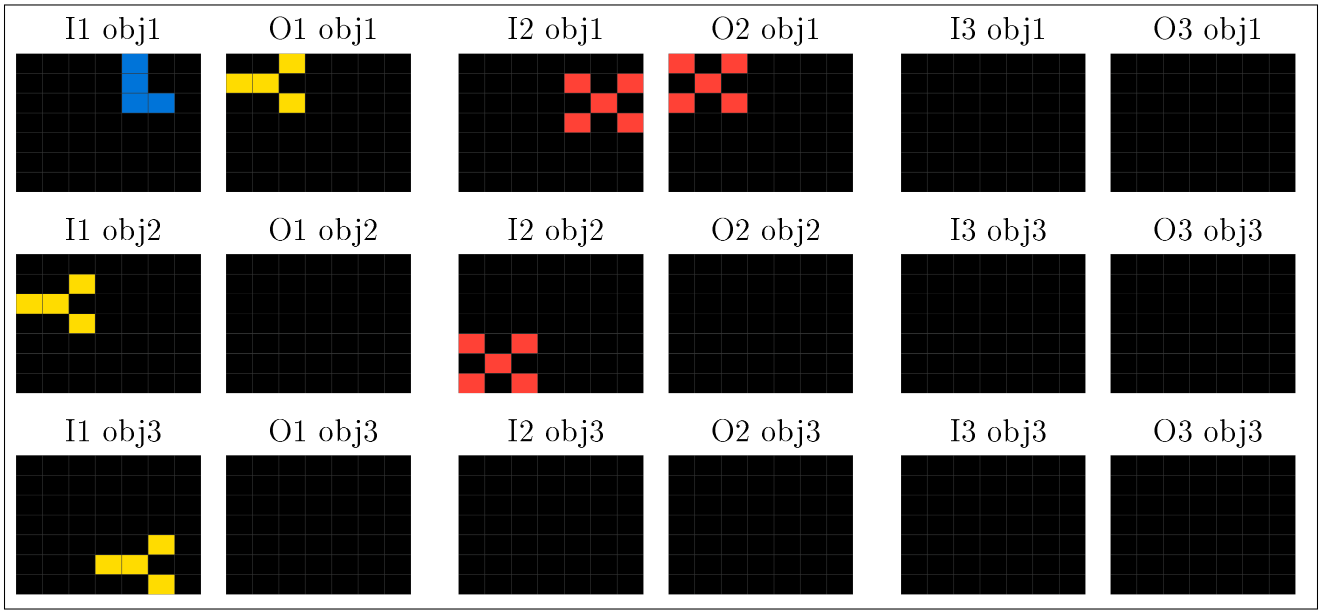


Solution: (λ: map var0 var1) (λ: symmetry_y var0)

F Task embedding outline

Outline of the embedding of a simple example task. For the sake of clarity, I suppose here that a grid has maximum size 7×7 and has at most 3 objects and that a task has at most 3 input/output examples.





```
Sequential(
 Conv2d(120, 60, 3),
 ReLU(),
 Conv2d(60, 30, 3),
 MaxPool2d(2),
 ReLU(),
 Flatten()
)
```

```
Sequential(
 Linear(5070, 2048), ReLU(),
 Linear(2048, 512), ReLU(),
 Linear(512, output dimension),
)
```

Figure 2: Example of a task for which finding the size of the output grid is not trivial