Tackling the Abstraction and Reasoning Corpus (ARC) with Object-centric Models and the MDL Principle

Sébastien Ferré

▶ **To cite this version:**

Sébastien Ferré. Tackling the Abstraction and Reasoning Corpus (ARC) with Object-centric Models and the MDL Principle. IDA 2024 - Symposium on Intelligent Data Analysis, Apr 2024, Stockholm, Sweden. pp.1-12. hal-04602995

HAL Id: hal-04602995
https://inria.hal.science/hal-04602995

Submitted on 6 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Tackling the Abstraction and Reasoning Corpus (ARC) with Object-centric Models and the MDL Principle

Sébastien Ferré[0000−0002−6302−2333]⋆

Univ Rennes, CNRS, Inria, IRISA
Campus de Beaulieu, 35042 Rennes, France
Email: ferre@irisa.fr

Abstract. The Abstraction and Reasoning Corpus (ARC) is a challenging benchmark, introduced to foster AI research towards human-like intelligence. It is a collection of unique tasks about generating colored grids, specified by a few examples only. In contrast to the transformation-based programs of existing work, we introduce object-centric models that are in line with the natural programs produced by humans. Our models can not only perform predictions, but also provide joint descriptions for input/output pairs. The Minimum Description Length (MDL) principle is used to efficiently search the large model space. A diverse range of tasks are solved, and the learned models are similar to natural programs.

1 Introduction

Artificial Intelligence (AI) has made impressive progress in the past decade at specific tasks, sometimes achieving super-human performance: e.g., image recognition [11], board games [15]. However, AI still misses the generality and flexibility of human intelligence to adapt to novel tasks with little training. To foster AI research beyond narrow generalization [8], F. Chollet [4] introduced a measure of intelligence that values *skill-acquisition efficiency* over *skill performance*, i.e. what matters is the amount of prior knowledge and experience that an agent needs to reach a reasonably good level at a range of tasks (e.g., board games), not its absolute performance at any specific task (e.g., chess). Chollet also introduced the Abstraction and Reasoning Corpus (ARC)[1] benchmark in the form of a psychometric test to measure and compare the intelligence of humans and machines. ARC is a collection of tasks that consist in learning how to transform an input colored grid into an output colored grid, given very few examples (3.3 on average). Figure 1 shows two ARC tasks (with the expected test output grid missing). ARC is a very challenging benchmark. While humans can solve more than 80% of the tasks [10], the winner of a Kaggle contest[2] could only solve 20%

⋆ This research is supported by Labex Cominlabs (ANR-10-LABX-07-01).
[1] Data and testing interface at https://github.com/fchollet/ARC
[2] https://www.kaggle.com/c/abstraction-and-reasoning-challenge

of the tasks (with a lot of hard-coded primitives and brute-force search). At the ARCathon'22 contest[3] the winner solved 6% of the tasks, and we ranked 4th by solving 2% of them. The published approaches [7,3,17,2], and also the Kaggle winner, tackle the ARC challenge as a *program synthesis* problem [12], where a program is a composition of primitive transformations, and learning is done by searching the large program space. In contrast, psychological studies [10,1] have shown that, when asked to verbalize instructions on how to solve a task, participants produce object-centric instructions, called *natural programs*. They typically first describe what to expect in the input grid, and then how to generate the output grid based on the elements found in the input grid.

We make two contributions to the ARC problem: (1) *object-centric models* that enable to both parse and generate data (here grids) in terms of object patterns and computations on those objects; and (2) an efficient search of object-centric models based on the *Minimum Description Length (MDL) principle* [14]. A model for an ARC task combines two *grid models*, one for the input grid, and another for the output grid. This closely matches the structure of natural programs. Compared to the transformation-based programs that can only predict an output grid from an input grid, our models can also provide a joint description for a pair of grids. The MDL principle comes from information theory, and says that *"the model that best describes the data is the model that compress them the more"* [14,9]. It has for instance been applied to pattern mining [16]. The MDL principle is used at two levels: (a) to choose the best parses of a grid according to a grid model, and (b) to efficiently search the large model space by incrementally building more and more accurate models. The two contributions support each other because existing search strategies could not handle the large number of elementary components of our grid models, and because the transformation-based programs are not suitable to the incremental evaluation required by MDL-based search. We report promising results based on grid models that are still far from covering all knowledge priors assumed by ARC. Correct models are found for 96/400 varied training tasks with a 60s time budget. Many of those share similarity with the natural programs produced by humans.

Section 2 discusses related work. Section 3 defines our object-centric models, and Section 4 explains how to learn them with the MDL principle. Section 5 reports on experimental results, comparing to existing approaches. Further details and illustrations can be found in the companion paper on arXiv [6].

2 Related Work

The ARC benchmark is recent and not many approaches have been published so far. All those we know define a DSL (Domain-Specific Language) of programs – based on function composition – that transform an input grid into an output grid, and search for a program that is correct on the training examples [7,3,17,2]. The differences mostly lie in the primitive functions (prior knowledge) and in

[3] https://lab42.global/past-challenges/arcathon-2022/

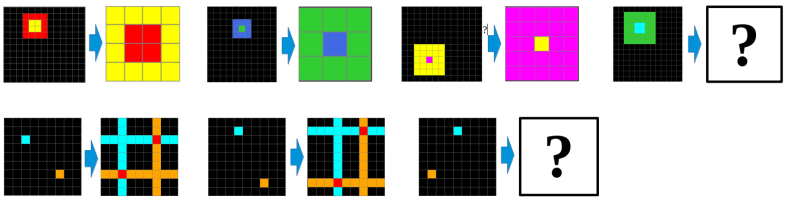

Fig. 1. Training tasks `b94a9452` (top) and `23581191` (bottom).

the search strategy. To guide the search in the huge program space, those approaches use either grammatical evolution [7], neural networks [3], search tree pruning with hashing and Tabu list [17], or stochastic search trained on solved tasks [2]. A difficulty is that the output grids are generally only used to score a candidate program so that the search is kind of blind. Alford [3] improves this with a neural-guided bi-directional search that grows the program in both directions, from input and output. Xu [17] compares the in-progress generated grid to the expected grid but this assumes that output grids are comparable to input grids, which is not true for all tasks. Function-based DSL approaches have a scaling issue because the search space increases exponentially with the number of primitive functions. For this reason, search depth is often bounded by 3 or 4. Ainooson [2] alleviates this difficulty by defining high-level functions that embody specialized search strategies.

Johnson *et al.* [10] report on a psychological study of ARC. It reveals that humans use object-centric mental representations to solve ARC tasks. This is in contrast with existing solutions that are based on grid transformations. Interestingly, the tasks that are found the most difficult by humans are those based on logics (e.g., an exclusive-or between grids) and symmetries (e.g., rotation), precisely those most easily solved by transformation-based approaches. The study exhibits two challenges: (1) the need for a large set of primitives, especially about geometry; (2) the difficulty to identify objects, which can be only visible in part due to overlap or occlusion. A valuable resource is LARC, for Language-annotated ARC [1], collected by crowd-sourcing. It provides for most training tasks one or several *natural programs*.

Beyond the ARC benchmark, a number of work has been done in the domain of *program synthesis*, which is also known as program induction or programming by examples (PbE) [12]. PbE is used in the FlashFill feature of Microsoft Excel 2013 to learn complex string processing formulas from a few examples [13]. Dreamcoder [5] alternates a *wake* phase that uses a neurally guided search to solve tasks, and a *sleep* phase that extends a library of abstractions to compress programs found during wake.

3 Object-centric Models

We introduce *object-centric grid models* as a DSL mixing *patterns* and functions, in contrast to functions only. Unlike function-based programs that are evaluated

Table 1. Patterns by type

type	patterns
Grid	**Layers**(size: *Vector*, color: *Color*, layers: *Layer*[]), **Tiling**(grid: *Grid*, size: *Vector*)
Layer	**Layer**(pos: *Vector*, object: *Object*)
Object	**Colored**(shape: *Shape*, color: *Color*)
Shape	**Point**, **Rectangle**(size: *Vector*, mask: *Mask*)
Mask	**Bitmap**(bitmap: *Bitmap*), **Full**, **Border**, **EvenCheckboard**, ...
Vector	**Vec**(i: *Int*, j: *Int*)

like expressions, our grid models are used to *parse* a grid, i.e. to understand its contents according to the model, and also to *generate* a grid, using the model as a template. Moreover, parsing and generation can be non-deterministic. A *task model* comprises two grid models that enable to predict an output grid or to describe a pair of grids.

3.1 Mixing Patterns and Functions

The purpose of a grid model is to distinguish between invariant and variant elements across the grids of a task. The syntax of grid models is defined by the following EBNF grammar, where M stands for *model* and E for *expression*.

$$M ::= ? \mid pattern(arg_1 : M_1, \ldots, arg_k : M_k) \mid E$$
$$E ::= \ value \mid !path \mid func(E_1, \ldots, E_k)$$

This definition is actually generic and independent of ARC tasks. What is domain-specific is the choice of patterns, values, and functions. A model M can be one of: an *unknown* ? for a totally unconstrained model; a *pattern* for specifying some constraint; or an *expression* E for a completely constrained model, resulting from a computation. For example, the model **Rectangle**($size$: ?, $mask$: **Full**) specifies a full rectangular object of any size. The model uses two patterns: **Rectangle** with two arguments, and **Full** with no argument. This example shows that models can be nested, and that models are not only about grids but also about any kind of grid components exhibited by patterns, e.g. 2D vectors for sizes. Expressions E are defined in the usual way, as a combination of values, functions, and variables. Variables have the form $!path$, and are used in the output grid model in order to retrieve information from the input grid. A path $path = arg_1.arg_2...$ is a chain of pattern arguments that identifies a component of a grid model by navigating from the root of the model, through the nested patterns. Function arguments are not reachable because functions are not invertible, hence they do not need a name. We call *description* a fully specified and computed grid model, it is a nesting of patterns and values only: e.g., **Rectangle**($size$: **Vec**$(2, 2)$, $mask$: **Full**).

We now define the concrete grid models that we have used in our experiments by defining the values, patterns, and functions. Four types of values are used:

$M^i =$ **Layers**(?, black, [**Layer**(?, **Colored**(**Rectangle**(?, **Full**), ?)),
Layer(?, **Colored**(**Rectangle**(?, **Full**), ?))])
$M^o =$ **Layers**(!lay[1].object.shape.size, !lay[0].object.color, [
Layer(!lay[0].pos - !lay[1].pos, *coloring*(!lay[0].object, !lay[1].object.color))])

Fig. 2. A correct model for task `b94a9452` (omitting argument names).

integers, colors, bitmaps (i.e., Boolean matrices), and grids (i.e., color matrices). Table 1 lists the patterns. Each pattern has a result type, and typed arguments. The argument types constrain which values/patterns/functions can be used in arguments. The names of arguments are used to reference the components of a grid model or grid description. Our grid models describe a grid as either a stack of layers on top of a background having some size and color, or as the tiling of a grid up to covering a grid of given size. A layer is an object at some position. An object is so far limited to a one-color shape, where a shape is either a point or some mask-specified shape fitting into a rectangle of some size. A mask is either specified by a bitmap or by one of a few common shapes such as a full rectangle or a rectangular border. Positions and sizes are 2D integer vectors. The 30 available functions (not detailed here for lack of space) essentially cover arithmetic operations on integers and on vectors, where vectors represent positions, sizes, and moves; and geometric notions such as measures (e.g., area), translations, symmetries, scaling, and periodic patterns (e.g., tiling).

A *task model* $M = (M^i, M^o)$ is made of an input grid model M^i and an output grid model M^o. Figure 2 shows a correct model for task `b94a9452`.

3.2 Parsing and Generating Grids with a Grid Model

We introduce two operations that must be defined for any grid model M: the *parsing* of a grid g into a description π and the *generation* of a grid description π, and thus of a grid g. These operations are analogous to the parsing and generation of sentences from a grammar, where syntactic trees correspond to our descriptions π. In both operations, the expressions present in the model M are first evaluated and reduced to their value, using a description as the evaluation context, called *environment* and written ε. Concretely, each variable is a path in ε and is replaced by the sub-description at the end of this path. The functions are then evaluated. The result is a reduced model M' consisting of unknowns, patterns, and values only.

Parsing. The parsing of a grid g consists in replacing the unknowns of the reduced model M' by descriptions corresponding to the content of the grid. Part of this content may be left undescribed, which can be seen as *noise* from the point of view of the model. This noise is taken into account in Section 4.1 when defining description lengths. For example, the parsing of the first input grid in Figure 1 (top) by the input grid model M^i in Figure 2 results in the following description: $\pi^i =$ **Layers**(**Vec**(12,13), black, [**Layer**(**Vec**(2,4), **Colored**(**Rectangle**(**Vec**(2, 2), **Full**), yellow), **Layer**(**Vec**(1,3), **Colored**(**Rectangle**(**Vec**(4,4), **Full**), red)]).

Generation. The generation of a grid consists in replacing the remaining unknowns in the reduced model M' by random descriptions of the right type, in order to obtain a grid description, which can then be converted into a concrete grid. For example, the output model M^o of Figure 2, applied with environment $\varepsilon = \pi^i$ (the above input grid description), generates the following description $\pi^o =$ **Layers**(**Vec**(4,4), yellow, [**Layer**(**Vec**(1,1), **Colored**(**Rectangle**(**Vec**(2,2), **Full**), red)]). This description conforms to the expected output grid.

An important point is that these two operations are *multi-valued* and *ranked* to reflect their non-determinism, i.e. they return an ordered list of descriptions. Indeed, there are often several ways of parsing or generating a grid according to a model, and some are preferable to others. For example, parsing a grid that contains several objects when the model specifies a single object.

3.3 Predict and Describe Grids with Task Models

We demonstrate the versatility of task models by showing that they can be used in two different modes: to *predict* the output grid from the input grid, and to *describe* a pair of grids jointly. We use below the notation $\rho, \pi \in \mathit{parse}(M, \varepsilon, g)$ to say that π is the ρ-th parsing of the grid g according to the model M with environment ε; and the notation $\rho, \pi, g \in \mathit{generate}(M, \varepsilon)$ to say that π is the ρ-th description generated by the model M with environment ε, and that g is the concrete grid described by π. The rank ρ is motivated by the fact that parsing and generation are multi-valued and ranked.

The *predict* mode is used after a model has been learned, in the evaluation phase with test cases. It consists in first parsing the input grid with the input model and the *nil* environment in order to get an input description π^i, and then to generate the output grid by using the ouput model and the input grid description as the environment.

$$\mathit{predict}(M, g^i) = \{(\rho^i, \rho^o, g^o) \mid \rho^i, \pi^i \in \mathit{parse}(M^i, \varepsilon^i = \mathit{nil}, g^i), \\ \rho^o, \pi^o, g^o \in \mathit{generate}(M^o, \varepsilon^o = \pi^i)\}$$

The *describe* mode is used in the learning phase of the model (see Section 4). It allows to obtain a description of a pair of grids. It consists in the parsing of the input grid and the output grid. Let us note that the parsing of the output grid depends on the result of the parsing of the input grid, hence the term "joint description".

$$\mathit{describe}(M, g^i, g^o) = \{(\rho^i, \rho^o, \pi^i, \pi^o) \mid \rho^i, \pi^i \in \mathit{parse}(M^i, \varepsilon^i = \mathit{nil}, g^i), \\ \rho^o, \pi^o \in \mathit{parse}(M^o, \varepsilon^o = \pi^i, g^o)\}$$

In the two modes, the *nil* environment is used with the input model because the input grid comes first, without any prior information. Note also that both modes inherit the multi-valued property of parsing and generation. These two modes highlight an essential difference between our object-centric models and the function-based programs of existing approaches. The latter are designed for prediction (computation of the output as a function of the input), they do not provide a description of the grids.

4 MDL-based Model Learning

MDL-based learning works by searching for the model that compresses the data the more. The data to be compressed is here the set of training examples. We have to define two things: (1) the description lengths of models and examples, and (2) the search space of models and the learning strategy.

4.1 Description Lengths

A common approach in MDL is to define the overall description length (DL) as the sum of two parts (*two-parts MDL*): the model M, and the data D encoded according to the model [9].

$$L(M, D) = L(M) + L(D \,|\, M)$$

The model is here a task model $M = (M^i, M^o)$ composed of two grid models, and the data is the set of training examples, i.e. pairs of grids (g^i, g^o). To compensate for the small number of examples, and to allow for sufficiently complex models, we use a *rehearsal factor* $\alpha \geq 1$, like if each example were seen α times.

$$L(M) = L(M^i) + L(M^o) \qquad L(D \,|\, M) = \alpha \sum_{(g^i, g^o)} L(g^i, g^o \,|\, M)$$

The DL of an example is based on the most compressive joint description.

$$\begin{aligned} L(g^i, g^o \,|\, M) = \; & min_{\rho^i, \rho^o, \pi^i, \pi^o \in describe(M, g^i, g^o)} \\ & [\; L(\rho^i, \pi^i, g^i \,|\, M^i, \varepsilon^i = nil) + L(\rho^o, \pi^o, g^o \,|\, M^o, \varepsilon^o = \pi^i) \;] \end{aligned}$$

Terms of the form $L(\rho, \pi, g \,|\, M, \varepsilon)$ denote the DL of a grid g encoded according to a grid model M and an environment ε, via the ρ-th description π resulting from the parsing. We can decompose these terms by using π as an intermediate representation of the grid.

$$L(\rho, \pi, g \,|\, M, \varepsilon) = L(\rho) + L(\pi \,|\, M, \varepsilon) + L(g \,|\, \pi)$$

The term $L(\rho)$ encodes the choice of the parsed description beyond rank 1, penalizing higher ranks. The term $L(\pi \,|\, M, \varepsilon)$ measures the amount of information that must be added to the model and the environment to encode the description, typically the values of the unknowns. The term $L(g \,|\, \pi)$ measures the differences, if any, between the original grid and the grid produced by the description. A correct model is obtained when, for $\rho^i = 1$ and $\rho^o = 1$, $L(\rho^o, \pi^o, g^o \,|\, M^o, \varepsilon^o = \pi^i) = 0$ for all examples, i.e. when using the first description for each grid, there is nothing left to code for the output grids, and therefore the output grids can be perfectly predicted from the input grids.

Three elementary domain-specific DLs therefore have to be defined: $L(M)$, $L(\pi \,|\, M, \varepsilon)$, $L(g \,|\, \pi)$. We sketch those definitions for the grid models defined in

Section 3. We recall that decription lengths are generally derived from probability distributions with the equation $L(x) = -\log P(x)$, corresponding to an optimal coding [9]. Defining $L(M)$ amounts to encode a syntax tree with unknowns, patterns, values, paths, and functions as nodes. Thanks to types, only a subset of those are actually possible at each node: e.g. type *Layer* has only one pattern. We use uniform distributions across possible nodes, and universal encoding for non-bounded ints. Defining $L(\pi \mid M, \varepsilon)$ amounts to encode the description components that are unknowns in the model. As descriptions form a subset of models, the above definitions for $L(M)$ can be reused, only adjusting the probability distributions to exclude unknowns, paths and functions. Defining $L(g \mid \pi)$ amounts to encode which cells in grid g are wrongly specified by description π. We also have to encode the number of differing cells.

4.2 Search Space and Strategy

The search space for models is characterized by: (1) an initial model, and (2) a *refinement* operator that returns a list of refined models $M_1 \dots M_n$ given a model M. The refinement operator has access to the joint descriptions, so it can be guided by them. Similarly to previous MDL-based approaches [16], we adopt a greedy search strategy based on the description length of models. At each step, starting with the initial model, the refinement that reduces the more $L(M, D)$ is selected. The search stops when no model refinement reduces it. The search necessarily terminates because the DL must decrease at each step but the found model may not be a solution to the task. To compensate for the fact that the input and output grids may have very different sizes, we actually use a *normalized description length* $\hat{L}$ that gives the same weight to the input and output components of the global DL, relative to the initial model.

Our initial model uses the unknown grid `?` for both input and output, i.e. $M_{init} = (?, ?)$. The available refinements are the following:

- the insertion of a new layer in the list of layers – one of **Layer**(?,**Col.**(**Point**,?)), **Layer**(?,**Col.**(**Rectangle**(?,?),?)), **Layer**(?,!*object*), and !*layer* – where !*object* (resp. !*layer*) is a reference to an input object (resp. an input layer);
- the replacement of an unknown at path p by a pattern $P = \mathit{pattern}(?, \dots, ?)$ when for each example, there is a parsed description π s.t. $\pi.p$ matches P;
- the replacement of a model component at path p by an expression e when for each example, there is a description π s.t. $\pi.p = e$.

4.3 Pruning Phase

The learned model sometimes lacks generality, and fails on test examples. This is because the goal of MDL-based learning as defined above is to find the most compressive task model on pairs of grids. This is relevant for the description mode but, in the prediction mode, the input grid model is used as a pattern to match the input grid, and it should be as general as possible provided that it captures the correct information for generating the output grid. For example, if

all input grids in training examples have height 10, then the model will fail on a test example where the input grid has height 12, even if that height does not matter at all for generating the output.

We therefore add a *pruning phase* as a post-processing of the learned model. The principle is to start from this learned model, and to repeatedly apply *inverse refinements* while this does not break correct predictions. Inverse refinements can remove a layer or replace a pattern/value by an unknown.

5 Evaluation

We evaluated our approach on the 800 public ARC tasks, and we also took part in the ARCathon 2022 challenge (as team MADIL). In ARC, a prediction is successful only if the predicted output grid is *strictly equal* to the expected grid for *all* test examples, there is no partial success. However, three trials are allowed for each test example to compensate for potential ambiguities in the training examples. To ensure a good balance of the computational time between parsing and learning, we set some limits that remained stable across our experiments. The number of descriptions produced by the parsing of a grid is limited to 64 and only the 3 most compressive are retained for the computation of refinements. At each step, at most 100,000 expressions are considered and only the 20 most promising refinements, according to a DL estimate, are evaluated. The rehearsal rate α is set to 10. The tasks are processed independently one of each other. The results are given for a learning time per task limited to 60s plus 10s for the pruning phase. We used one run per task set as there is no randomness involved. Our experiments were run with a single-thread implementation[4] on Fedora 32, Intel Core i7x12 with 16GB memory. The learning and prediction logs and the screenshots of the solved training tasks are available at `https://www.irisa.fr/LIS/ferre/pub/ida2024/`.

Task sets and baselines. We consider four task sets for which results have been reported: the 400 training and 400 evaluation public tasks, the 100 secret tasks of Kaggle'20, and the 100 secret tasks of ARCathon'22. We presume that those secret tasks are taken from the 200 secret ARC tasks. As baselines, we consider published methods that report results on the considered task sets [7,3,17,2]. We also include the winners of the two challenges for reference. Unfortunately, the reported results are scarce, and the papers do not provide their code.

Success rates. On the training tasks, for which we have the more results to compare with, our method solves 24% tasks, almost on par with the best method, by Ainooson *et al* (26%). Both methods also solved a similar number of evaluation tasks (5.75% vs 6.50%), and both solved 2% tasks at ARCathon'22, and ranked 4th ex-aequo. Comparing the different task sets, the evaluation tasks appear to be significantly more difficult than the training tasks, and the secret tasks of ARCathon seem even more difficult as the winner could only solve 6% tasks. Icecuber managed to correctly solve an amazing 20.6% tasks at Kaggle'20,

[4] Open source available at https://github.com/sebferre/ARC-MDL

Table 2. Number and percentage of solved tasks and average learning time for solved tasks, for different methods on different task sets

task set	method	solved tasks		runtime
ARC training	Fischer *et al*, 2020	31	7.68%	
(400 tasks)	Alford *et al*, 2021	22	5.50%	
	Xu *et al*, 2022	57	14.25%	
	Ainooson *et al*, 2023	104	26.00%	178.7s
	OURS	96	24.00%	4.6s
ARC evaluation	Ainooson *et al*, 2023	26	6.50%	
(400 tasks)	OURS	23	5.75%	11.4s
Kaggle'20	Icecuber (winner)	20	20.6%	
(100 tasks)	Fischer *et al*	3	3.0%	
ARCathon'22	pablo (winner)	6	6%	
(100 tasks)	Ainooson *et al*	2	2%	
	OURS (4th ex-aequo)	2	2%	

but at the cost of the hand-coding of 142 primitives, 10k lines of code, and brute-force search (millions of computed grids per task).

The ARC evaluation protocol allows for three predictions per test example. However, the first prediction of our method is actually correct in 90 of the 96 solved training tasks. This shows that our learned models are accurate in their understanding of the tasks. To better evaluate the generalization capability of learned models, we also measured the generalization rate as the proportion of models that are correct on training examples that are also correct on test examples: 92% (94/102) on training tasks, and 72% (23/32) on evaluation tasks. This again suggests that the evaluation tasks feature a higher generalization difficulty. Without the pruning phase, this rate decreases to 89% (91/102) on training tasks. This shows that the pruning phase is useful, although description-oriented model learning is already good at generalization. Reasons for failures to generalize are: e.g., the test example has several objects while all training examples have a single object; the training examples have a misleading invariant.

Efficiency and model complexity. Intelligence is the efficiency at acquiring new skills, according to Chollet. Although ARC enforces data efficiency by having only a few training examples per task, and unique tasks, it does not enforce efficiency in the amount of priors, nor in the computation resources. It is therefore useful to assess the latter. We already mentioned Icecuber's method that relies on a large number of primitives, and intense computations. The method of Ainooson *et al*, which has comparable performance to ours, uses 52 primitives and about 700s on average per solved task. In comparison, our method uses 30 primitives and 4.6s per solved training task (21.7s over all training tasks). Those short runtimes are made possible by our greedy strategy. Doubling the learning timeout at 120s does not lead to solving more tasks, so 60s just seems to be enough to find a solution if there is one.

Another way to evaluate efficiency is to look at the complexity of learned models, typically the number of primitives composing the model in program

synthesis approaches. A good proxy for this complexity is the depth of search that was reached in the allocated time. In our case, it is equal to the number of refinements applied to the initial empty model. Few methods provide this information: Icecuber limits depth to 4, and Ainooson's best results are achieved with a brute-force search with maximum depth 3. Methods based on DreamCoder [3] have similar limits but can learn more complex programs by discovering and defining new operations as common compositions of primitives, and reusing them from one task to another. Our method can dive much deeper in less computation time, thanks to its greedy strategy. The number of refinement steps achieved in a timeout of 60s on the training tasks ranges from 4 to 57, with an average of 19 steps. This demonstrates the effectiveness of the MDL criteria to guide the search towards correct models. This claim is reinforced by the fact that a beam search (width=3) did not lead to solving more tasks.

Learned models. The learned models for solved tasks are very diverse despite the simplicity of our models. They express various transformations: e.g., moving an object, extending lines, putting one object behind another, order objects from largest to smallest, remove noise, etc. Note that none of these transformations is a primitive in our models, they are learned in terms of objects, basic arithmetics, simple geometry, and the MDL principle.

We compared our learned models to the natural programs of LARC [1]. Remarkably, many of our models involve the same objects and similar operations than the natural programs. For example, the natural program for task `b94a9452` is: *"[The input has] a square shape with a small square centered inside the large square on a black background. The two squares are of different colors. Make an output grid that is the same size as the large square. The size and position of the small inner square should be the same as in the input grid. The colors of the two squares are exchanged."* For other tasks, our models miss some notions used by natural programs but manage to compensate them: e.g., topological relations such as "next to" or "on top" are compensated by the three attempts; the majority color is compensated by the MDL principle selecting the largest object. However, in most cases, the same objects are identified.

These observations demonstrate that our object-centric models align well with the natural programs produced by humans, unlike approaches based on the composition of grid transformations. An example of a program learned by [7] on the task `23b5c85d` is `strip_black; split_colors; sort_Area; top; crop`, which is a sequence of grid-to-grid transformations, without explicit mention of objects.

6 Conclusion

We have presented a novel and general approach to efficiently learn skills at tasks that consist in generating structured outputs as a function of structured inputs. Our approach is based on descriptive task models that combine object-centric patterns and computations, and on the MDL principle for guiding the search for models. We have detailed an application to ARC tasks on colored grids. We have shown promising results, especially in terms of efficiency, model complexity, and

model naturalness. Going further on ARC will require a substantial design effort as our current models cover so far a small subset of the knowledge priors that are required by ARC tasks (e.g., missing goal-directedness).

References

1. Acquaviva, S., Pu, Y., Kryven, M., Sechopoulos, T., Wong, C., Ecanow, G., Nye, M., Tessler, M., Tenenbaum, J.: Communicating natural programs to humans and machines. Advances in Neural Information Processing Systems **35**, 3731–3743 (2022)
2. Ainooson, J., Sanyal, D., Michelson, J.P., Yang, Y., Kunda, M.: An approach for solving tasks on the abstract reasoning corpus. arXiv preprint arXiv:2302.09425 (2023)
3. Alford, S., Gandhi, A., Rangamani, A., Banburski, A., Wang, T., Dandekar, S., Chin, J., Poggio, T.A., Chin, S.P.: Neural-guided, bidirectional program search for abstraction and reasoning. CoRR **abs/2110.11536** (2021), https://arxiv.org/abs/2110.11536
4. Chollet, F.: A definition of intelligence for the real world. Journal of Artificial General Intelligence **11**(2), 27–30 (2020)
5. Ellis, K., et al.: Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In: ACM Int. Conf. Programming Language Design and Implementation. pp. 835–850 (2021)
6. Ferré, S.: Tackling the abstraction and reasoning corpus (ARC) with object-centric models and the MDL principle. arXiv preprint arXiv:2311.00545 (2023)
7. Fischer, R., Jakobs, M., Mücke, S., Morik, K.: Solving Abstract Reasoning Tasks with Grammatical Evolution. In: LWDA. pp. 6–10. CEUR-WS 2738 (2020)
8. Goertzel, B.: Artificial general intelligence: concept, state of the art, and future prospects. Journal of Artificial General Intelligence **5**(1), 1 (2014)
9. Grünwald, P., Roos, T.: Minimum description length revisited. International journal of mathematics for industry **11**(01) (2019)
10. Johnson, A., Vong, W.K., Lake, B., Gureckis, T.: Fast and flexible: Human program induction in abstract reasoning tasks. arXiv preprint arXiv:2103.05823 (2021)
11. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems **25**, 1097–1105 (2012)
12. Lieberman, H.: Your Wish is My Command. The Morgan Kaufmann series in interactive technologies, Morgan Kaufmann / Elsevier (2001). https://doi.org/10.1016/b978-1-55860-688-3.x5000-3
13. Menon, A., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.: A machine learning framework for programming by example. In: Int. Conf. Machine Learning. pp. 187–195. PMLR (2013)
14. Rissanen, J.: Modeling by shortest data description. Automatica **14**(5), 465–471 (1978)
15. Silver, D., Huang, A., Maddison, C.J., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
16. Vreeken, J., Van Leeuwen, M., Siebes, A.: Krimp: mining itemsets that compress. Data Mining and Knowledge Discovery **23**(1), 169–214 (2011)
17. Xu, Y., Khalil, E.B., Sanner, S.: Graphs, constraints, and search for the abstraction and reasoning corpus. arXiv preprint arXiv:2210.09880 (2022)