Communicating Natural Programs to Humans and Machines

[1]Sam Acquaviva [2]Yewen Pu [1]Marta Kryven [1]Theodoros Sechopoulos [1]MH Tessler [1]Joshua Tenenbaum
.

Abstract— **The Abstraction and Reasoning Corpus (ARC) is a collection of tasks where one must induce a program from a set of given input-output examples and apply it to a new input. This task is challenging for state-of-the-art program synthesis and machine learning techniques, however humans can easily solve most tasks. We hypothesize that humans use intuitive program induction, interpreting ARC tasks by constructing "natural programs" in language. In this work we experimentally study "natural programs" by formulating a two-player game: A participant solves and then communicates the underlying program to another participant using natural language; the second participant must solve the task on a new input using the description alone. We find at least 361 out of 400 tasks can be successfully solved from a natural language description communicated in this way, demonstrating that natural language is sufficient in transmitting these natural programs. We compared the collected natural language programs to computer programs constructed using two separate state-of-the-art program synthesis approaches. We find that that while the length of the natural programs predicts the length of computer programs, natural programs leverage concepts that are distinct from computer programs, such as using metaphors such as "magnet" and an emphasis on communicating specifications in addition to executable procedures. We also study whether it is possible improve the performance of the aforementioned program-synthesis algorithms by leveraging additional natural language annotations.**

I. INTRODUCTION

A long-term goal of Artificial Intelligence (AI) is to build agents that can flexibly solve new problems. Although current AI systems achieve super-human proficiency at certain specific tasks[11], unlike humans, such systems still lack the ability to generalize and readily adapt to novel situations. The ARC dataset is a set of challenges that aim to test algorithms' abilities to express the core elements of human intelligence, such as abstract reasoning, generalizing from few examples, and understanding patterns[1]. The ARC dataset does not require natural scene understanding, but draws on human ability to make abstract analogies, such as analogies used in standardized intelligence tests (e.g. Raven's matrices[2]).

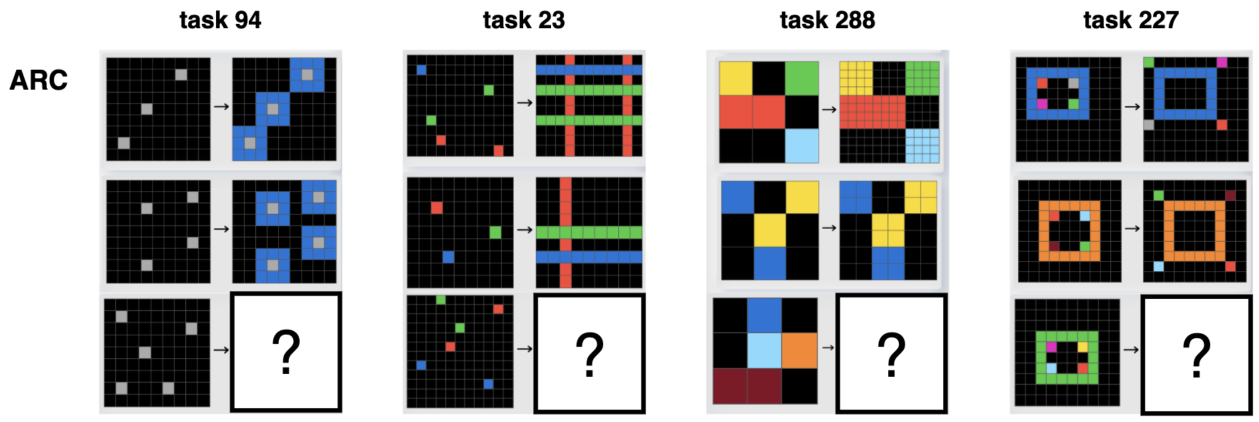
Solving an ARC task entails inferring a rule (a natural program) consistent with a small number of input-output examples and applying that rule to a new input to generate the answer. In humans, such abilities are partlially supported by learning internal models used for generalization, such as learning object categories, discovering common rules behind similar situations, and encoding stories into memory [3-8]. According to the ARC Kaggle challenge[1], the best-performing program synthesis and machine learning systems solve at most 20% of these tasks, while humans can solve over 80% [9].

How can we build intelligent systems that can reason and adapt like humans to novel situations such as the ones in ARC? We hypothesize that the answer may lie in the *language* of the natural programs constructed by humans to solve the ARC tasks: While humans can describe an ARC task in a variety of ways, the best programmer is shackled to the particular domain specific language (DSL) – a library of functions developed to encode specific ARC tasks. To elicit the natural programs from human solvers, we developed a data-collection procedure formulated as a two-player game (see Fig.2). One participant (Describer) solves an ARC task and communicates the underlying program to a different participant (Builder), who must solve the task on the new input using the description alone. As the Builder constructs the output from the description alone, successful construction indicates that the natural language description was *sufficient* in transmitting natural programs.

We collected a corpus of Language-annotated ARC tasks (LARC), which contains a number of attempted task descriptions per task (Figure 1). We found that building ARC tasks from descriptions is harder than building from examples, and so language acts as a lossy compression of ARC examples. Yet, we demonstrate that at least 361 of the 400 training tasks could be solved successfully based on linguistic input alone. Furthermore, the length of natural programs encoded in language correlated with the length of computer programs encoded in a DSL by experienced programmers. Like computer programs, human descriptions relied on common programming concepts such as, numbers, loops, and logical operations. Unlike computer programs, however, natural language descriptions leveraged significantly richer library of primitives, including references to abstract physical concepts such as *windows*, *staircases*, or *gravity*. We perform a preliminary study on if it is possible to leverage natural language annotations to aid existing synthesis tools by ranking the relevant primitives, finding that a naive bayes implementation that predicts the relative likelihood of each primitives given natural language *does not* perform better than the baseline model that simply rank the primitives by their relative frequency of usage.

[1] Massachusetts Institute of Technology
[2] Autodesk AI Lab, yewen.pu@autodesk.com
{samacqua, mkryven, theosech, tessler, jbt}@mit.edu

[1]The solution is hosted on GitHub: https://github.com/top-quarks/ARC-solution



LARC

You should see a black grid with a number of grey boxes

The output grid size is the same as input

Surround each of the grey boxes with a blue border

You should see a grid with red, blue, and green squares.

The output grid size is the same as the input

Draw a vertical red line of squares through any red squares, then make left to right green lines through green squares and left to right blue lines through blue squares. Go over any red squares from before.

You should see a random pattern of pixels

The output grid size is the number of colors in the pattern times 3 for both width and height

Re-create the same pattern with a single pixel becoming a square with sides of length corresponding to the number of colors in the original pattern. In other words, simply increase the resolution of the input by the number of colors it contains.

You should see a grid with a square in one color. There will be four small different colored squares on the inside corners of the square. The rest of the inside of the square and the rest of the grid is black.

The output grid size is the same as the input.

Take the different colored squares from the inside corners of the square and move them to the opposite outside corner of the square. For example if there is a green square on the inside bottom left corner, move it to the outside top right corner. The rest of the squares will be black.

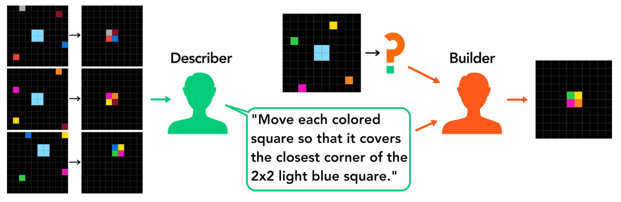
Fig. 1. The ARC dataset (above) and the corresponding LARC language annotation (below). For a set of given input-output examples (2 shown here for each task), the ARC task is to generate the correct output on a new input. In LARC, a *describer* is tasked to describe the pattern present in the input-output examples so that a different person, a *builder*, can generate the correct output using the description *alone*.

This motivates more sophisticated approaches in grounding natural programs to computer programs.

II. The LARC Dataset

We curate the LARC (Language-annotated Abstraction and Reasoning Corpus) dataset, augmenting 361 out of 400 ARC training dataset with natural language annotations that can capture the underlying natural program in its entirety.

A. The Collection Process



"Move each colored square so that it covers the closest corner of the 2x2 light blue square."

Fig. 2. The experimental procedure used to collect natural language annotations for LARC.

We collected the natural-language annotations by recruiting 453 participants on Amazon MTurk. Each participant completed a set of tasks, in which they were randomly assigned to act as a **describer** or as a **builder**. **Describers** saw all of the input-output examples for an ARC task, but not the test-input. The describer generated a description to be used by another person (the builder) of how to construct the correct output from the input. The description had 3 sections: what you should expect to see in the input, what is the output grid size, and what you should do to create the output (See Figure 1). After submitting description, the test-input was revealed to the describer, who then built the output based their own description alone to validate the description. The **builder** saw a description written by a describer, and the test input grid. Both the builder and the describer had three attempts to construct the correct output grid.

The describers describe the task in **generations**, meaning, the n-th describer for a given ARC task has access to all previous n descriptions given before, along with how successful they are. By building descriptions on top of each other, we form a chain of improving descriptions rather than random samples. If a describer fails to validate their description, the description is not shown to subsequent describers.

B. The Bandit Algorithm

The goal of the annotation process is to derive one good natural language description for each ARC task. Given a quanta of focused time of a MTurk participant, how should one allocate this resource? Should one allocate it to *generate* new descriptions for a task, or to *validate* an existing description by building it?

Formally, this is a multi-bandit, infinite-arm best-arm identification problem, in the following sense:

1) *multi-bandit* – each ARC task is a different bandit
2) *infinite-arm* – each task description is a different arm, as there are infinite natural language descriptions, there are infinitely many arms per bandit.
3) *best-arm identification* – for each ARC task, we want to collect the best description (i.e. most builders can generate the correct output with this description).

We use [12] to solve the infinite-arm, best-arm identification problem within a task, and a heuristic that ranks the uncertainties of a particular bandit's best arm's performance to address the multi-bandit problem. This formulation does not take advantage of the generational aspect of descriptions.

C. Dataset Description

The data-set includes in total 1659 descriptions, of which 578 were successfully constructed by a builder. Overall, 361 of the 400 tasks have at least 1 successful communication, or a valid natural program, represented in natural language. For each task we collected on average $M = 4.2, SD = (2.3)$ descriptions (at most 16). The distribution of success rates over Describe and Build tasks are shown in Fig. 3(C-D). Note that success rates shown are not diagnostic of the task difficulty, because of the relatively small numbers of samples per task. On average individuals completed $M = 7.4, SD = (4)$, max = 19, min=1 tasks, which included $M = 4.65(SD = 2)$ Description tasks and $M = 4.2(SD = 3.3)$ Build tasks.

III. Analysis

Natural language as a program. We can think of language as a form of "natural program" that humans can use to communicate procedures and computations. Encoding of the input-output examples into such natural programs is a lossy compression process. Average task build rate from examples in the Describing task is 75% – this rate improves with practice, from 60% to 85%. Average build rate from language in the Building task is 50%, and is not affected by practice. Thus, we lose 25% accuracy in switching the encoding of ARC tasks from visual input-outputs to natural language on average. Fig. 3 (A-B) shows the human success rates as a function of completed trials of the given task. Further, we found that humans can collaborate to assemble and refine a natural language program, to solve particularly difficult tasks. The collectively composed descriptions grow in length from one generation to the next (Fig. 4 C.).

Fig. 6 shows the distribution of words used in successfully built descriptions of ARC tasks, sorted by their frequency in the corpus. The words were pre-processed by singularizing, and grouping together colors names, numbers, and pronouns, resulting in about 800 unique words used to describe the 400 ARC tasks. The most frequently occurring words refered to primitives common in programming languages, such as, numbers, loops and logical operations. At the same time, examining the tail of this distribution reveals that humans also used a variety of complex words that can be read as referring to generalizable policies, for example: *zig-zag, flower, petal, mosaic, magnet, miniaturized, zoom, visualize, inkblot, monochromatic, minority, dominant, vacuum, airplane, snowflake, gravity, spiral, squeeze, arrow, train, window, resemble, floating, staircase*.

Comparison to DSL-based models. We compared human solutions to two architecturally different state-of-the-art DSL models of solving ARC, IC and TH. IC is the DSL used in the top performing algorithm in the Kaggle competition https://github.com/top-quarks/ARC-solution. TH is a DSL developed inhouse that leverages the DreamCoder [10] algorithm for library learning. For each model, we obtained solutions for a sample of ARC problems (81 TH and 73 IC, written in their respective DSLs). The two samples of problems did not overlap. We found that the length of programs required to solve the ARC tasks by each model predicted the length of successfully built language descriptions generated by humans, as shown on Fig. 4 (A-B). This suggests that tasks that take longer to explain to a computer will also take longer to explain to a human.

Procedure vs Specification. We randomly sampled 100 phrases (under 140 characters in length) from all successful descriptions, and coded them by 3 independent raters, using 2 tags: **procedure** and **specification**. Procedure was defined as: *Code crucial to the computation of the task. If you remove it, the task will fail*. Specification was defined as *Description of the output, used to check the validity of the computation. If you remove it, the procedure could still proceed without failure if the builder understood it the right way*. The inter-rater agreement was 88% for whether a statement described a procedure, and 85% for specification. Overall, 30% of the statements in human descriptions were coded as specification, suggesting an important difference between human-human and human-computer communication strategies. Unlike computer programs, which are built from sequences of exact instructions, human language appears to follows a program synthesis-like approach, that begins by outlining the the search space of possible programs, and proceeds to gradually refine it using validation statements.

IV. Leveraging Language for Program Synthesis

We perform preliminary study on whether LARC can aid state of the art program synthesis algorithms used by TH and IC. Both TH and IC leverages enumeration in synthesizing the right programs. Crucial in the enumeration process is the identification of relevant primitives given specification: For instance, given an ARC task that require selection of all red objects, an enumeration order that prioritize the usage of a primitive that contain the concept "red" will find the correct program faster on average. We ask the following question: Can we train a model that takes in natural language annotation, and inform the synthesizer of the relative primitives?

Using LARC and programs generated in solving the ARC tasks in IC and TH, we created a paired dataset $D = \{(w^i, c^i) \dots\}$ consisting of natural language descriptions as a sequence of words $w^i = w^i_1 \dots w^i_n$ and the corresponding programmatic primitives $c^i = c^i_1 \dots c^i_k$ for ARC task i. The

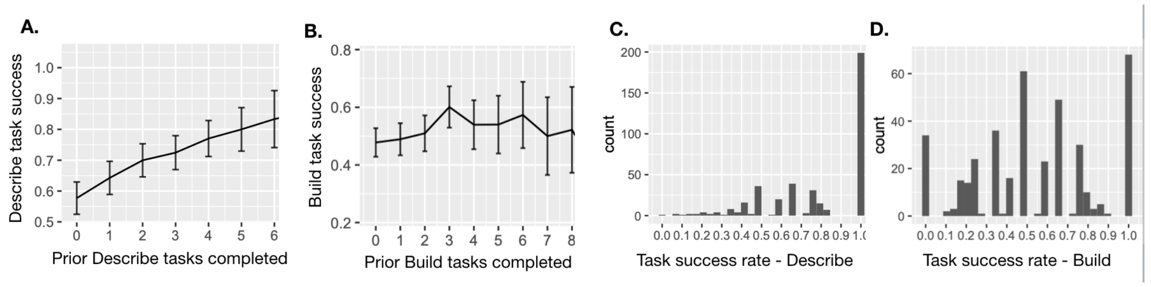


Fig. 3. **A.** The rate of humans solving the Describing task. The average accuracy per task was 75%. **B.** The rate of humans solving the Building task. The average accuracy per task was task was 50%. **C.** Success rates aggregated over individuals in the Describing task, over time. Performance improved with experience. **D.** Success rates aggregated over individuals in the Building task over time.

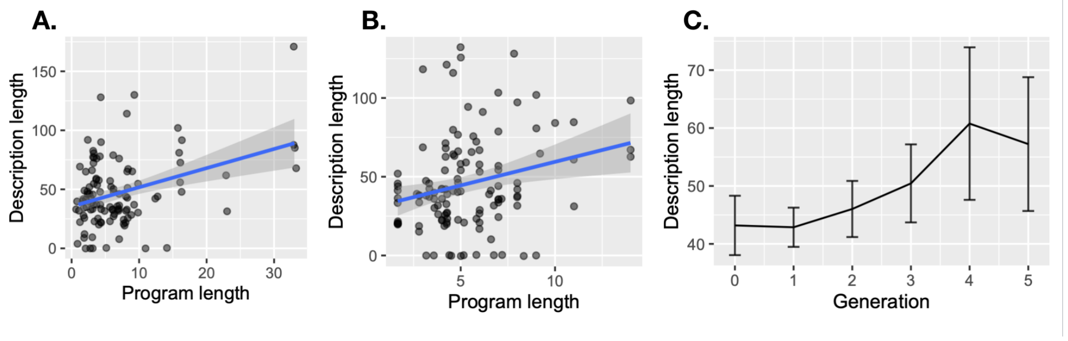


Fig. 4. **A.** IC program length is correlated with human description length $r = .36, p < .0001$. **B.** TH program length is correlated with human description length $r = .25, p < .0001$ **C.** Human descriptions get longer with generation.

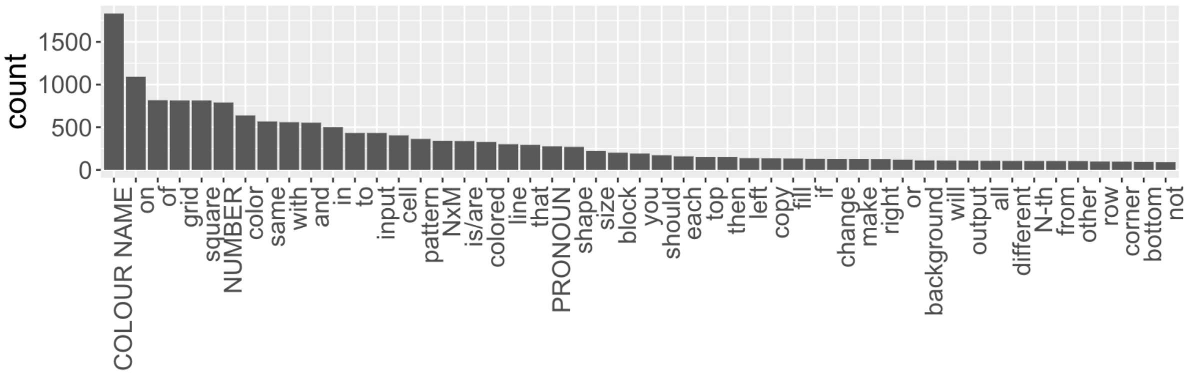


Fig. 5. Words used in successfully built descriptions, sorted by their frequency in the corpus. The words were singularized. Colors names, numbers, and pronouns were grouped together.

train/test split is 50%. Accuracy is defined as the fraction of the k primitives (which may vary from task to task) in the ground-truth program that is correctly predicted by the model's top k predictions.

We compare the performance of the following 3 models. **random** where the top k primitives are ranked at random, **frequency** where the primitives are ranked by relative frequency of usage in the training set, and **naive bayes** computes the likelihood of a primitive according to $P(w|c) = \prod_j P(w_j|c)$, where the term $P(w_j|c)$ is estimated from training data. The result is shown in Figure 6. As we can see, a naive application of natural language does not perform better than simply informing the frequency of primitives. This suggests that if we want to use LARC for better program synthesis, we need more sophisticated approach that can "align" the relevant words with their relevant concepts.

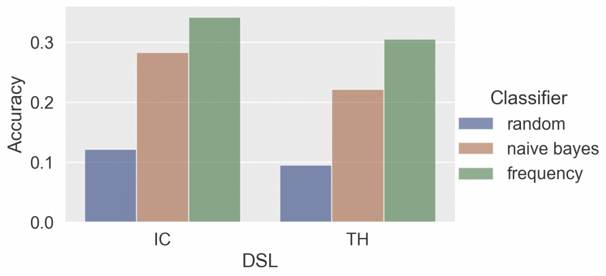


Fig. 6. The predictive accuracy of each classifier predicting the primitives using the natural language descriptions.

References

[1] Chollet, François. ”On the measure of intelligence.” arXiv preprint arXiv:1911.01547 (2019).

[2] Raven J. John. Raven Progressive Matrices. Springer, Boston, MA, 2003.

[3] Michelene TH Chi, Robert Glaser, and Marshall J Farr. The nature of expertise. Psychology Press, 2014.

[4] Harry F Harlow. The formation of learning sets. Psychological review, 56(1):51, 1949.

[5] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. Behavioral and brain sciences, 40, 201 7.

[6] Brenden M Lake and Steven T Piantadosi. People infer recursive visual concepts from just a few examples. Computational Brain Behavior, pages 1–12, 2019

[7] Frederic Bartlett. C.(1932). Remembering: A study in experimental and social psychology. NewYork/London, 1932

[8] Tian, L., Ellis, K., Kryven, M., Tenenbaum, J. (2020). Learning abstract structure for drawing by efficient motor program induction. Advances in Neural Information Processing Systems, 33

[9] Johnson, A., Vong, W. K., Lake, B. M., Gureckis, T. M. (2021). Fast and flexible: Human program induction in abstract reasoning tasks. arXiv preprint arXiv:2103.05823.

[10] Ellis, K., Wong, C., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., Tenenbaum, J. B. (2020). Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. arXiv preprint arXiv:2006.08381.

[11] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815.

[12] Wang, Y., Audibert, J. Y., Munos, R. (2008). Infinitely many-armed bandits. In Advances in Neural Information Processing Systems.