Understanding designed objects by program synthesis

James McDermott
james.mcdermott@nuigalway.ie
National University of Ireland, Galway

Abstract

In this position paper, we argue that short programs in high-level, Turing-complete, human-readable symbolic languages are an attractive representation for designed objects: they have potential benefits both for general agents and for simple AI/ML tools. In principle they allow all available structure to be captured, and they allow for introspection. They may also parallel mechanisms of human thought. Actually finding these programs is a very difficult but worthwhile research problem.

1 Introduction

In every field of design, from art, music and computer programming to civil engineering and product design, designed objects have internal structure and regularity. The same is true of the biological objects *pseudo-designed* by evolution, and even sometimes of natural objects from sand dunes to galaxies which are merely *organised* by physical forces. Seeing this structure is essential to understanding them: our perception and memory use representations which are *smart* (i.e. parsed for structure) rather than *verbatim* [Drescher, 2006].

It seems clear that human intelligence involves both sub-symbolic processes at the neuron level, and conscious reification of these processes in algorithmic, symbolic, causal terms. Pearl and Mackenzie [2018] argue that causal understanding is essential to progress in AI. Meanwhile, Rule et al. [2020] draw on a stream of cognitive literature in arguing that "human learning operates over structured, probabilistic, program-like representations".

For a given object, the representation which best captures its structure and regularity is the shortest: *Compression is understanding* [Grünwald, 2004, Mahoney, 2006]. "A pattern has not been fully understood if the [notation] representing it itself contains a pattern. For that means either that some aspect or the pattern was missed or that the notation lacks the power to characterize that aspect and therefore had to copy it verbatim." – Hofstadter, quoted by Meredith [1986]. Similar ideas are common in developmental encodings for evolutionary algorithms [Woodward, 2003].

When we perceive structure, we understand how an object works and assume that the designer intended it [Schmidhuber, 1997]. Moreover, we are capable of making that understanding explicit, e.g. by writing computer programs that use constants, loops, functional abstraction, and many other methods of reducing repetition and making programs shorter.

In this position paper we will argue that programs in a Turing-complete representation are a good representation (Section 2, next). Following that, in Section 3, we will discuss the problem of actually finding them. Section 4 concludes. We include toy domains in the discussion, because they raise issues and may teach us lessons relevant in more realistic domains [Hofstadter, 1995]. We will not deal with several adjacent problems, such as the problem of perceiving object boundaries.

NeurIPS 2020 Workshop on Object Representations for Learning and Reasoning, Vancouver, Canada.

2 Why programs, why high-level, and why Turing-complete?

By the Church-Turing thesis, only a Turing-complete representation can capture all possible types of regularity in an object, achieving the greatest possible compression.

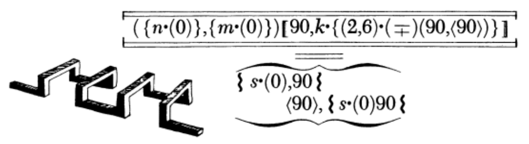


Figure 1: Structural information theory proposes a declarative language capable of capturing much regularity in 2D and 3D designs. From Leeuwenberg [1971].

Declarative languages may achieve most of the same compression. An interesting example is the structural information theory (SIT) model of human perception [Leeuwenberg, 1971], accompanied by a declarative language, which also facilitates efficient computation of the representation for a given object [van der Helm, 2004]. For an example of the flavour of SIT, see Fig. 1. Many authors in evolutionary computation have also investigated ad-hoc generative representations which achieve a large compression between "genotype" and "phenotype". Empirical research is needed to investigate real-world examples which truly require Turing-completeness, but as already stated, it is required in principle.

The choice of language matters. High-level languages can express structure more easily. In algorithmic information theory (AIT) [Kolmogorov, 1965, Li and Vitanyi, 1997, Chaitin, 1977] and minimum description length (MDL) [Grünwald, 2004], the choice of language is regarded as unimportant because any representation can include a "compiler" for any other language as a prefix; but this insight is only useful asymptotically. In bounded rationality the choice between high-level and low-level language has a practical effect. High-level languages are human-readable, with benefits for interpretable and trustable AI, i.e. human users and overseers inspecting representations. But far more important is that high-level languages are amenable to introspection, i.e. AI agents can reason about their own representations. Humans invented symbolic reasoning and formal languages by introspection and formalisation of our pre-existing facilities.

2.1 Object Understanding Tasks

Compression is understanding, but how does this cash out? Let's consider some object understanding tasks that agents may be faced with, whether in "toy" problems or in the real world.

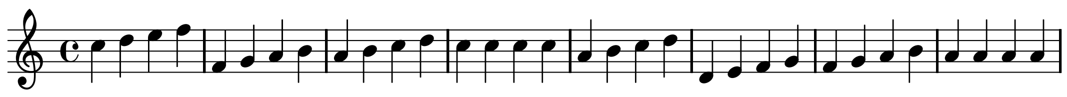
Representing the structure of a single object Program structure parallels object structure. Consider a caterpillar with 30 similar body segments: it may be represented by a program with a loop structure. An agent with such a representation can both use and reflect on it, e.g. discovering that there is important similarity with another caterpillar even if the number of segments differs. Figure 2 is an example in another design domain.

Continuing a sequence Given a sequence-like object, an agent may be required to generate a continuation. This is commonly proposed as a benchmark or toy problem domain, and arises as a real-world problem in music [Larson, 1996]. One approach might be to find a looping program which yields the elements of the given object in turn, and then to "run it forward".

Representing a distribution Intelligent agents often need to carry out classification tasks. Although agents may well use numerical/statistical approaches (analogous to logistic regression) internally, for intelligent agents these can be reflected upon. This requires an understanding of deep/abstract structure in the objects to be classified. An appealing approach is to represent a class distribution as a parametrised program. By varying the parameters, we can generate elements of the distribution.

Representing object transformations Chollet [2019] presents an interesting set of tasks in a "toy" domain of small, abstract 2D images. Each task consists of a few input-output image pairs. The goal is to learn the transformation sufficiently to be able to apply it to new inputs. One approach might create a single parameterised program such that the transformation from input to output corresponds to a specific change in parameters.

Generating variant objects Programs are also useful for design purposes, e.g. creating variants of existing designs. A computer-aided design tool might work as follows. Allow the user



```
[ 10 11 12 13 : 6  7  8  9 : 8  9 10 11 : 10 10 10 10 : 8  9 10 11 : 4  5  6  7 : 6  7  8  9 : 8  8  8  8 ]
```

Figure 2: A short, compressible piece of music, together with a simple integer representation of the pitches. Functional abstraction, including higher-order functions, allow compression. There is an ascending 4-note pattern, repeated multiple times starting on different notes. Given a function `ascend(start-note, n)`, we can call it with $n = 4$ and different `start-note` parameters to produce each of the instances. The second argument could be curried to produce a new function `ascend4(start-note)`. This avoids the duplication of the 4. Similarly, a `repeat4(start-note)` function can create the two instances of repeated notes. The entire piece can now be represented as `ascend4(10) ascend4(6) ascend4(8) repeat4(10)` (etc.). This can be compressed further. With the creation of a new, higher-order function `f(g, h, start-note)` which calls its first argument three times and its second argument once, each at appropriate start-notes relative to its final argument, the piece will be reduced to `f(ascend4, repeat4, 10) f(ascend4, repeat4, 8)`. This suggests another function, `j(f, g, h, start-note, jump)` which calls `f` twice, once at `start-note` and once at `start-note + jump`, passing `g` and `h` in each time. Our piece is now represented by `j(f, ascend4, repeat4, 10, -2)`. Perhaps further abstraction is possible by extracting the constant 4 from `ascend4` and `repeat4`. From McDermott et al. [2010].

to manually create or load a design; find a program which generates it; apply mutations to that program; run the resulting programs to generate variant designs; present them to the user. Both program mutation and recombination methods, and program dissimilarity measures, as commonly used in genetic programming [Koza, 1992, O'Reilly, 1997, Poli et al., 2008], could be used to implement such tools. Program mutation is more general than mere varying of numerical constants.

2.2 Strong objections

Highly-compressed representations can be fragile, e.g. a single bit-flip in a zip-encoded file may make the file unreadable. Using symbolic programs with well-defined mutation operators avoids this problem.

In the Seek-Whence problem [Meredith, 1986], the task is to find a suitable extension of a given integer sequence. The patterns to be found and extended are conceptual rather than arithmetical. In the context of this problem, Hofstadter [1995] has stronger objections: "I absolutely refused to yield to the easy temptation of representing [integer] sequences by computer programs" because (1) programs are *inflexible*, and, relatedly, (2) the variants of an integer pattern that are natural for humans may correspond only to *unnatural* variants of an underlying program. It is worthwhile to revisit these objections. With the right language and principles for finding programs in it, we may hope to overcome them, as follows.

(1) In a computer program subject to mutation there is the problem of *update anomalies* [Eessaar, 2016]. If a numerical constant appears more than once, then a mutation may change it in one place and leave it unchanged in the other, making the resulting object incoherent or ill-formed. It is in this sense that programs are inflexible. The "don't repeat yourself" (DRY) principle of software engineering [Hunt and Thomas, 2000] states that such situations should be avoided. DRY programs are less fragile than non-DRY ones.

(2) This problem may be addressed by using programs which capture human-perceived regularity in the object. With this property, program variations *are* perceived as giving natural object variations: "two objects appear similar when they are likely to have been generated by the same process" [Kemp et al., 2005]. It may be necessary to use all available forms of abstraction in the objects, e.g. higher-order functions, as suggested in Figure 2 [McDermott et al., 2010, 2012]. New empirical research is needed on how program variations align with human perception, for example which program primitives and which program locations are more "slippable" [Hofstadter, 1995, p. 56] in human minds.

Although Hofstadter objected to using computer programs to represent integer sequences, the representation actually used by the Seek-Whence project [Meredith, 1986] is quite program-like, as illustrated in Figure 3. There are custom primitives which perform computation, carry state, and control the flow of execution. Moreover, there is a complex surrounding search/learning infrastructure which actually searches for Seek-Whence representations. The two were designed to interact. The benefit of using an existing text-based language is that of generality, in contrast to the domain-specific design of Seek-Whence. On the other hand, choosing a textual language does not mean rejecting the Seek-Whence search/learning infrastructure, which has important lessons for our program synthesis methods.

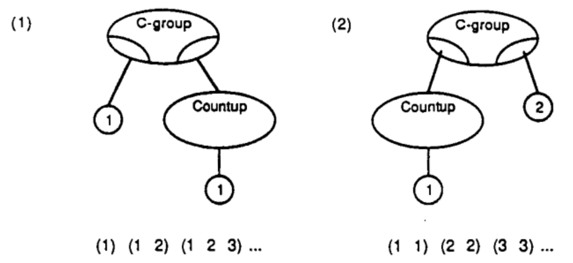


Figure 3: Two integer sequences with possible Seek-Whence representations. From Meredith [1986]. The parentheses show a possible human *interpretation* of the sequence, and it corresponds closely to the program.

There is a further practical difficulty. Small objects with structure may not become shorter when that structure is expressed. Consider the open phrases of Beethoven's 5th: `G G G E♭; F F F D`. Representing the perceived structure here would involve writing a parametric function which outputs four notes, and calling it twice in a loop. The resulting program would not be shorter than the original object by any obvious measure. This objection is unimportant in the asymptotic scenario (in AIT/MDL) but is important in practice. Again, the DRY principle suggests a solution. We can formulate approximate measures of DRY-ness under which a program which expresses the structure here is indeed shorter than the original object. The main idea is to penalise a representation for multiple use of the same primitive (e.g., the `G` and `F` notes above).

3 How can we find programs that represent objects?

Even if programs as representations for objects would be nice to have, can we actually find them? It is a hard program synthesis problem even for small objects and even without considering real-world issues such as noisy data. Program synthesis has been one of the great pipe dreams of computer science and AI since Turing [1950]. Many approaches exist, including metaheuristic search, formal methods, and generative neural approaches. None are yet robust and general, reflecting the difficulty of the problem. Finding the *shortest* program for an object is the problem of Kolmogorov complexity, which is uncomputable. If we relax the goal to that of finding *short* programs, we can retreat to heuristic and approximate methods.

Recent progress with neural approaches [Ellis et al., 2015, Shi et al., 2020] and metaheuristic search [Krawiec et al., 2016] and hybrids [Lynch et al., 2020] suggests that the goal should not be abandoned. For example, in metaheuristic search some authors have begun to develop new search objectives which prevent the "needle in a haystack" problem [Krawiec et al., 2016]; others have argued for smart sampling techniques [Worm and Chiu, 2013, White et al., 2020]. As already mentioned, previous research on cognitive-inspired approaches to representation learning [Meredith, 1986] may inspire improved methods of program synthesis. In particular, the Seek-Whence approach to working with, partially evaluating, mutating, and recombining partial programs may be more appropriate to the program synthesis domain than the typical genetic programming approach or the typical neural network approach.

4 Conclusion

Programs are a good representation for objects as they allow all possible compression, and could enable introspection. A lot of research is needed: (1) understanding how program mutations align with human perception of object variation; (2) finding examples of real-world regularity that cannot be captured without Turing-completeness; (3) robust methods of program synthesis.

References

G. J. Chaitin. Algorithmic information theory. *IBM journal of research and development*, 21(4): 350–359, 1977.

F. Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

G. L. Drescher. *Good and real: Demystifying paradoxes from physics to ethics*. MIT Press, 2006.

E. Eessaar. The database normalization theory and the theory of normalized systems: finding a common ground. *Baltic Journal of Modern Computing*, 4(1):5, 2016.

K. Ellis, A. Solar-Lezama, and J. Tenenbaum. Unsupervised learning by program synthesis. In *Advances in neural information processing systems*, pages 973–981, 2015.

P. Grünwald. A tutorial introduction to the minimum description length principle. In Gruünwald, Myung, and Pitt, editors, *Advances in Minimum Description Length: Theory and Applications*. MIT Press, 2004.

D. R. Hofstadter. *Fluid Concepts and Creative Analogies: Computer models of the fundamental mechanisms of thought*. Basic Books, New York, 1995.

A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

C. Kemp, A. Bernstein, and J. B. Tenenbaum. A generative theory of similarity. In *Proceedings of the 27th annual conference of the cognitive science society*, pages 1132–1137, 2005.

A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of information transmission*, 1(1):1–7, 1965.

J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

K. Krawiec, J. Swan, and U.-M. O'Reilly. Behavioral program synthesis: Insights and prospects. In *GPTP XIII*, pages 169–183. Springer, 2016.

S. Larson. Continuations as completions: Studying melodic expectation in the creative microdomain seek well. In *Joint International Conference on Cognitive and Systematic Musicology*, pages 321–334. Springer, 1996.

E. Leeuwenberg. A perceptual coding language for visual and auditory patterns. *The American journal of psychology*, pages 307–349, 1971.

M. Li and P. Vitanyi. *An introduction to Kolmogorov complexity and its applications*. Springer Verlag, 1997.

D. Lynch, J. McDermott, and M. O'Neill. Program synthesis in a continuous space using grammars and variational autoencoders. In T. Bäck et al., editors, *Parallel Problem Solving from Nature*, Leiden, Netherlands, 2020. Springer.

M. Mahoney. Rationale for a large text compression benchmark, 2006. URL `http://cs.fit.edu/~mmahoney/compression/rationale.html`. Retrieved 17 January 2015.

J. McDermott, J. Byrne, J. M. Swafford, M. O'Neill, and A. Brabazon. Higher-order functions in aesthetic EC encodings. In *CEC*, pages 3018–3025, Barcelona, Spain, July 2010. IEEE Press.

J. McDermott et al. String-rewriting grammars for evolutionary architectural design. *Environment and Planning B: Planning and Design*, 39(4):713–731, September 2012.

M. J. Meredith. *Seek-Whence: A model of pattern perception*. PhD thesis, Indiana University, 1986.

U.-M. O'Reilly. Using a distance metric on genetic programs to understand genetic operators. In *IEEE International Conference on Systems, Man, and Cybernetics: Computational Cybernetics and Simulation*, volume 5, 1997.

J. Pearl and D. Mackenzie. *The book of why: the new science of cause and effect*. Basic Books, 2018.

R. Poli, W. Langdon, and N. McPhee. *A field guide to genetic programming*. Lulu Enterprises UK Ltd, 2008.

J. S. Rule, J. B. Tenenbaum, and S. T. Piantadosi. The child as hacker. *Trends in Cognitive Sciences*, 2020.

J. Schmidhuber. Low-complexity art. *Leonardo*, 30(2):97–103, 1997.

K. Shi, D. Bieber, and R. Singh. Tf-coder: Program synthesis for tensor manipulations. *arXiv preprint arXiv:2003.09040*, 2020.

A. M. Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.

P. A. van der Helm. Transparallel processing by hyperstrings. *Proceedings of the National Academy of Sciences of the United States of America*, 101(30):10862–10867, 2004.

D. R. White, B. Fowler, W. Banzhaf, and E. T. Barr. Modelling genetic programming as a simple sampling algorithm. In *Genetic Programming Theory and Practice XVII*, pages 367–381. Springer, 2020.

J. R. Woodward. Modularity in genetic programming. In *Proceedings of EuroGP*, pages 254–263. Springer, 2003.

T. Worm and K. Chiu. Prioritized grammar enumeration: Symbolic regression by dynamic programming. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1021–1028. ACM, 2013.