doi:https://doi.org/10.4324/9781351212243

To Appear in: In Sune Hannibal Holm & Maria Serban (eds.), Philosophical Perspectives on the Engineering Approach in Biology: Living Machines? London, UK: pp. 40-68 (2020)

A Roomful of Robovacs: How to Think About Genetic Programs

Brett Calcott
University of Sydney

7 June, 2020

Abstract

The notion of a genetic program has been widely criticized by both biologists and philosophers. But the debate has revolved around a narrow conception of what programs are and how they work, and many criticisms are linked to this same conception. To remedy this, I outline a modern and more apt idea of a program that possesses many of the features critics thought missing from programs. Moving away from over-simplistic conceptions of programs opens the way to a more fruitful interplay of ideas between the complexity of biology and our most complex engineering discipline.

1 Introduction

> The genomic sequence encodes the developmental program which determines the progression from fertilized egg to organized body plan (Peter and Davidson 2013).

Peter and Davidson's statement is not unusual; biologists frequently equate or compare genes to computer programs. Their aim is often to identify a special or distinctive role for genes in development. Thus, genes contain the instructions for building an organism in contrast to, say, mere environmental inputs. Despite the frequency of these statements, many philosophers and biologists argue that such comparisons are wrong-headed and ultimately misleading rather than informative (Nicholson 2014; Keller 2001; Pigliucci 2010; Boudry and Pigliucci 2013; Griffiths 2001; Planer 2014; Nijhout 1990; Moczek et al. 2015; Jaeger et al. 2015). The complaints are many, and I won't rehearse them all here. Instead, I want to examine an aspect of the comparison between genes and programs that has had scant attention from philosophers.

The issue is this: if we wish to advocate or dismiss the idea that genes are like a program, then we need a clear idea about what a program itself is like. Yet while the debates about genetic programs draw on the latest insights from molecular and developmental biology, the corresponding talk about programs provides little detail about what programs are and how they function; it is simply assumed that this is common wisdom. But the discussion typically centers around a very narrow interpretation of how programs operate. For example, the debates often assume that a program is a list of instructions telling the computer what to do. This is true of some programming languages (such as C, Java, or Python), but in other languages (such as SQL or Haskell), you describe the results required rather than dictate the instructions and the order they should be done. Many claims in the debates rely on one understanding of a program, rather than anything about programs in general (see Backus 1978 for a critique of these assumptions).

For many critics, this narrow interpretation of programs is irrelevant. Their goal has been to evaluate the use and misuse of the genetic programming metaphor in recent biological history (Keller 2003, for example). Their critiques often hit the mark because advocates of genetic programs deploy concepts that are either similarly narrow or vague enough to be interpreted this way.

But evaluating how scientists have deployed the notion of program is not our only option here. An alternative is to look beyond this narrow interpretation and see if it is possible to build a better analogy. Why bother? One role that engineering analogies play in science is to draw on something well understood to illuminate, clarify, or rethink some puzzling natural phenomenon. If we discard the genetic program analogy without fully exploring what programs are like, we risk losing valuable insights or simply re-inventing them under another name.

In the rest of this chapter, I sketch a way to think about programs that avoids many of the criticisms advanced against genetic programs. This suggests there is still some worth in thinking of genes as programs, as long as we're clear on what programs we have in mind.

2 Avoiding Deep Thought

In Douglas Adams's tale *The Hitchhiker's Guide to the Galaxy*, intelligent mice construct a computer named Deep Thought to compute the answer to the meaning of life, the universe, and everything. The mice then wait 7.5 million years for it to deliver its famous answer. That is an absurdly long time to wait, but it highlights something often attributed to computers and, by extension, programs. The computational task demanded of Deep Thought has what we might call a "ballistic trajectory." We give the computer (or program) some input, set it running, and then stand back and wait for it to produce an answer. The British "bombe" device depicted in the film about Alan Turing (*The Imitation Game*, 2014) demonstrates this same ballistic trajectory, whirring and clicking to crack the German enigma code

(just in time, whew!).

Many programs have this ballistic property. When biologists use phylogenetic software to reconstruct the tree-like relationships between organisms using their DNA, they feed it the raw sequence information and then may wait weeks for these programs to produce results. A program need not be complex or long-running to be ballistic, however. The first programs students are asked to write have this ballistic property, writ small:

EXERCISE 1 : write a program to compute the first N prime numbers.

EXERCISE 2 : write a program that calculates someone's age in days from a given birth date (look out for leap years!).

In each exercise, we're asked to write a program that begins by consuming some input, performs some calculation, and ends by supplying some output. A program like this, which starts with a set of initial conditions and then churns through an ordered series of steps to produce a final output, is sometimes known as a "script" or a "batch program."

Most programs we use today are not batch programs. When I start my word processor, I don't sit around waiting for it to finish its job. The opposite is often true: my word processor is sitting around waiting for me. A word processor, like most programs we encounter these days, is an "interactive program": it takes in various bits of input from the keyboard and mouse, responding with changes to what appears on the screen and the odd beeping noise. Each interaction is brief and evokes a rapid response: a keystroke adds a letter; a menu selection formats the text. A long series of such interactions produce a story or a business report or a scientific paper.

This interplay between the program and user describes a feedback loop. I tap a few words, the word processor highlights a spelling mistake, and I go back and correct it. I read over a few sentences, decide on a better formulation, and then go back and clarify the text. The dynamic coupling between the user and program is a mark of these interactive programs.

The dynamic coupling of the interaction is essential, for interactive programs are not merely programs that require input at times other than when they are started. Consider an installation program from the days of CDs and floppy disks. These programs stopped intermittently with the tedious but necessary request: "please insert next disk." Such a program is interactive in one sense – it asks for something from a user at various stages during its operation. But notice that, in this case, the *program decides* when and what type of interaction takes place. With a word processor, the user is in the driver's seat, and the program awaits commands in the form of mouse clicks and key presses. Something outside the program (us, in this case) decides the order things happen rather than the other way around.

The distinction between batch program and interactive program is not a hard line; a single program might contain aspects of both. For example, we might view com-

mands executed by a word processor, such as "Print" or "Format Bibliography," as tiny batch subprograms in themselves; we start them up and wait for them to finish. But the difference between batch programs and interactive programs is relevant when we make analogies with genes, for it colors our thinking about what programs are, how they operate, and what they explain.

3 Which Program Are You Thinking Of?

Critics of the genomic program often appeal to features of programs that apply only to batch programs. For example, Evelyn Fox Keller thinks of a program as something that starts with some data and finishes by producing some output – following the ballistic trajectory I outlined earlier. Because of this, her only way of envisioning complex gene interaction with feedback is to invoke multiple programs:

> … what counts as "data" for one "program" is often the output of a second "program," and the output of the first is "data" for yet another "program," or even for the very "program" that provided its own initial "data." (Keller 2001)

Yet the notion of a feedback loop in interactive programs renders this unnecessary. A single interactive program can take in input at different times, and that input may include output that it has produced at previous times.

Fred Nijhout, similarly, appears to have batch programs in mind in his critique of the genetic program metaphor. He outlines two conditions for gene expression to be a program (Nijhout 1990, 442):

1. "The gene or its product must be necessary and sufficient for the occurrence of the process, and not be itself provoked by the process itself."
2. "A program must somehow contain information about the temporal sequence of events."

The first claim sounds reasonable for something like a batch program: we start up a program to construct a phylogenetic tree, give it some input, and then let it run. After setting it going, the results are determined by the program alone. Now consider an interactive program: a half-written document, displayed on the screen, provokes the user to change it, perhaps by pointing out a spelling mistake. So the feedback loop between user and program guarantees it will fail Nijhout's first condition.

Nijhout's second condition fares the same. A batch program comprises a set of modular components *and* some order in which to execute these components. The order is important, for the operations later in the program often depend on the completion of earlier operations. An interactive program also has a set of modular components. In a word processor, the various menu options expose some of this modular functionality (such as saving, formatting, numbering, or printing). But the order in which we put together these operations – the "temporal sequence of

events" – is not part of the program. The user of the program chooses how to put them together.

Both of Nijhout's conditions draw on a more basic assumption – that a program contains some intrinsic ordering over how the instructions in the program are executed. Nijhout is not alone in emphasizing the importance of this intrinsic ordering. Another critic of the genetic program, Ronald Planer, finds it problematic that "there is no order in which these instructions can be properly said to be retrieved and executed by the cell during development," so that there is no "beginning, middle, or end to this 'program' " (Planer 2014).

Consider Planer's claim in light of batch programs and interactive programs. A batch program has a clear beginning, middle, and end. Many batch programs even show how far through the various tasks they are: "Processing 50% complete . . ." But how should we respond if asked about the beginning, middle, and end of a word processing program? Given the technical know-how, we might find the first instructions executed by a word processor when it starts and maybe even the last instruction it executes when we quit the program. But where is the middle of this program? The program comprises a set of small actions – typing, formatting, editing – chained together in a variety of ways by the user of the program. There is no way to point to some specific piece of code and say, "This is the middle," like we could with, say, the program that reconstructs phylogenies. This is because an interactive program does not dictate the order in which it must execute its operations. Instead, this order derives from some external input.

Notice that a word processor is useful *because* it does not dictate this order. Without the ability to control the order in which it executes the various components, the flexibility of the word processor would be lost. This exposes assumptions about what programs are good for. If we focus on batch programs, we might think the utility of a program lies in its capacity to encode a complex series of dependent operations. With a word processor, however, rather than a set of dependent steps, we have a set of loosely coupled operations that transform a document. The utility of the program lies in the ease and flexibility with which some external process can recombine these operations. An interactive program resembles a well-designed toolkit of related operations that are assembled at run time rather than a set of ordered instructions that execute a well-designed plan.

This shift in focus turns many common ideas related to programs, and especially *genetic* programs, on their head (see Nicholson 2014, for example). A batch program emphasizes determinism; an interactive program, flexibility and open-endedness. A batch program churns away by itself; an interactive program depends on inputs from some external source. A batch program contains the rules (instructions, procedures, algorithm) for producing its output; an interactive program contains no rules for producing a particular output but a toolkit for generating many outputs. Finally, a batch program often incites agentive thinking: "the phylogenetic program *generated* a tree of related various organisms," while an interactive program does not: Microsoft

Word no more *wrote* my thesis than a typewriter *wrote* Hemingway's *The Old Man and the Sea*.

4 No Spookiness Required

I introduced an interactive program using a word processor – something I assume readers are familiar with. But a word processor interacts with *us* – an intelligent external agent. Am I suggesting there is an external intelligent agent interacting with (or directing!) the genetic program? That would be spooky. So let me describe an interactive program without positing any intelligent external agent.

Instead of a word processor, consider a program that controls a mobile robot, such as a robot vacuum cleaner or "robovac." If you haven't seen one, they are self-contained vacuum cleaners the size and shape of a Frisbee on wheels. They zip around the floor, moving underneath furniture and doing a surprisingly good job of cleaning your house unattended.

A robovac has several sensors that convey information about the local environment – whether it is touching something, whether something is in front, and so on. This information feeds into the program controlling the robovac, and the program responds by maneuvering the robot, such as changing direction or slowing down. By maneuvering, the robot changes its environment, and now its sensors are exposed to a different environment. This results in further maneuvering. And so it goes until (in my experience) the battery runs out or the robovac ingests a stray sock. Here we have the same coupled, fluid feedback loop between a program and some external set of affairs. But instead of an intelligent agent directing the robot, it is the changing environment that serves as the other half of the feedback loop.

The robovac example removes the spookiness, but have we lost the contrast with the batch program I outlined earlier? Unlike the word processor's open-ended output, a robovac's actions are directed to one outcome: cleaning the house. Let's shift our attention from this goal and consider, instead, how the robovac achieves this task. Take the path the robovac robot traced around my house on some particular day. This path was not programmed into the robot – I did not upload a house plan into my robovac and have it precalculate some optimal cleaning path. Rather, the particular path taken arose from the continuous interaction between the layout of my house and the combined set of responses of the program controlling the robot's actions.

This "emergent" behavior is often contrasted with how programs work.[1] Nijhout, for example, differentiates development from a program by describing it thus: "Development is a series of elaborate temporal and spatial interactions that are context dependent. The sequence of gene activation we see in development is an emergent

[1] By "emergent," here (and elsewhere), I mean the weak emergence exhibited by many agent-based systems and cellular automata rather than anything metaphysically hard to understand (Bedau 1997).

property of this interaction" (Nijhout 1990). Yet we can describe the sequence of maneuvers executed by our robovac the same way. The particular path the robovac took is an emergent property of its interaction with the environment. Like the word processor, the ordering of the set of responses it made was not intrinsic but induced by something external to the robot. For the robovac, this external feature was the local environment. This local environment, in turn, resulted from previous decisions made by the robot.

Switching the kind of program we have in mind changes what features we attribute to a program. Yet it is batch programs that critics have focused on, while interactive programs look like a better choice if we wish to build an analogy with genes. In the next section, I show how extending this idea can add some further clarity to constructing a useful analogy.

5 A Roomful of Robovacs

Word processors and robovacs are interactive programs, but what they interact with differs. In the first case, the program interacts with a goal-directed intelligent agent (such as me, on good days at least). But a robovac largely interacts with a *static* environment, a variety of furniture distributed across rooms of various shapes. Dynamic bits of the environment, such as pets and small children, typically present a challenge to it. The variety of input it receives is generated entirely by its own motion in that environment.

Now consider placing several robovacs in the same room. We no longer have a static environment, as each robovac is responding, in part, to other robovacs. To my knowledge, no one has designed robovacs that work in groups, so I doubt this exercise would produce faster or more efficient cleaning (over and above there being just *more* robovacs). But this kind of collective robotic behavior is being actively explored. There are flocks of coordinated aerial robots that can fly in formation, for example (Vasarhelyi et al. 2014). These flocks lack any central controlling system; each robot is independent, responding to local physical parameters, global positioning information, and messages received from other robots. Their ability to fly in formation – a group-level ability – results from the continuous interaction between individual robots.

Would we say the program *created* a particular flying formation? That seems odd but for a different reason than saying the word processor *wrote* my thesis. For the question here is about a group-level capacity: a particular flying formation. A single robot (with a single program) does not create a formation. If we are to attribute this goal to the program, then we need to mention it arose from several copies of that program interacting with one another. We can think of it like this. For the robots to assemble into a formation, each must maneuver itself while continually adjusting to the other robots doing likewise. As with our initial robovac, these paths are not explicitly coded into the robots. Rather, they emerge from each robot's coupled

interaction with other robots navigating their own path. But the maneuvering of individual robots is not equivalent to the assembly of a particular formation. *That* emerges from the collective behavior of all the robots.

So neither the behavior of individual aerial robots nor their group flying formation is encoded in the program. To fully understand how the formation occurs, we need to understand how the program is embedded within the physical context and how the communication and interaction between the multiple versions of it take place. Yet we can still make sense of a program playing a distinctive role in *controlling* a robot. The interactive nature of program is, in part, what enables the robot to respond with appropriate behavior given particular local conditions.

Let me summarize the kind of program we are now dealing with. I'll call it a Collectively-Identical Interactive Program. Abstracting away from the details of our roomful of robovacs (or skyful of aerial robots), we have:

1. A single interactive program (a related but loosely coupled set of operations, which are invoked by, and respond to, incoming input),
2. where there are multiple identical copies of this same program running concurrently,
3. and each copy of the interactive program is coupled to other copies of itself, where the outputs from one can affect the future inputs of others.
4. Furthermore, the program has been designed to interact with other copies of itself,
5. yet the design goal itself is a group-level capacity rather than the individual behavior of a single copy of the program.

This kind of program naturally fits with the goal of controlling swarms of robots, or agent-based simulations. But this is not the only place we find this confluence of properties. One recent example concerns analyzing the social networks that arise in online interactions – a task essential for many big online companies. These social networks comprise "vertices," representing people, and "edges," representing social relationships (capturing friendship, for example). Understanding the structure of these networks can reveal an enormous amount about how people interact (too much perhaps). It turns out, however, that traditional top-down algorithms for analyzing networks cannot deal with networks of this size. One solution to this problem, proposed by employees at Google, is to treat the graph as though each vertex (or node) is running the same program and receiving input and sending output along its edges (Malewicz et al. 2010). Because each program requires only limited local information, we can distribute these programs across many computers and run in parallel. Similar to the way the iterative feedback between programs in interacting aerial robots converges to a particular formation, we can use iterative feedback between many interacting vertex-based programs to identify high-level structure in graphs, such as the complex communities of friendships in a social graph. There are

no robots here, but we still have a program that interacts with itself to produce a collective feature at a higher level.

6 Building a Better Analogy

Given the aim of this chapter, it should not be surprising that I think the kind of program I have just outlined has something in common with genes. The analogy goes like this. In a developing multicellular organism, each cell contains a copy of the same DNA.[2] This DNA encodes a program, and each cell is running a copy of the same program. These are interactive programs, constantly responding to their local environment. In a developing multicellular organism, this environment consists largely of other cells. Like the examples, each copy of the program interacts with other copies in the surrounding cells. On this view, a genomic program is an interactive program that has evolved to interact with copies of itself to generate a higher level of organization, such as prominent features of a multicellular body plan.

This claim might sound familiar and open to the same familiar critiques. But it is essential to see these claims in the light of the preceding discussion about programs. First, although we might say (one copy of) a program is guiding the cell's behavior, this behavior is not encoded as a series of steps in the genome but arises dynamically from the interaction between the program and its changing developmental environment. Second, not only is this cellular behavior not encoded in the genome, but the individual cellular behavior itself is also not equivalent to the higher level of organization that the collective individual-level behavior produces.

This revised view of a genomic program allows us to pull apart some features that are commonly run together. For example, it makes sense to say that this program controls features of development because changing the program will change how the collective interaction of the programs unfolds.[3] But it does not follow that the program itself is sufficient to "compute the embryo." Nor does it mean that this higher-level organization is "in" the DNA or that it can be read off the DNA like the structure of a building can be read off a blueprint. This version of a genetic program is also compatible with views that emphasize the key role of physical self-organization in development (such as Newman and Bhat 2009). As we saw with the aerial robots, the programs by themselves did not explain how the formations were constructed; other information about the interactions amongst the group and were required.

[2] To a rough approximation. There are always exceptions in biology, such as cells without DNA, somatic mutations, and chimeric organisms.

[3] This provides a novel way to pursue the idea that genes are a special kind of difference maker (Griffiths et al. 2015; Weber 2016).

7 Summary

My goal in this chapter was to present a fresh approach to genetic programs. I focused on two key ideas. First, we need to move beyond scripts and batch programs and instead look at how interactive programs work. Second, we need to look at programs that are specifically designed to interact with one another to produce some collective behavior. The first of these is commonplace – almost every program we now use is interactive. The second shift is more specialized, but I identified two areas of active research. More can be said, however. I've focused on describing the behavioral contrasts between batch and interactive programs. But the internal architecture of these programs differs too. I touched on this earlier when I referred to an interactive program as more like a combinatorial toolkit whose pieces are put together at run time. Interestingly, this same architecture enables interactive programs to be extended and modified more easily than batch programs. That looks relevant once we shift our attention to the evolution of development and the subject of evolvability (Calcott 2014).

The notion of a genetic program still has something to offer us. It may not be a complete picture, nor should it be the only source of ideas. It does, however, have additional virtues in contrast to frameworks offered as replacements, such as developmental systems theory (Oyama et al. 2003). For when you borrow from engineering, the ideas you draw on have been applied and are known to work, and their limitations are understood.

References

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8), 613–641.

Bedau, M. A. (1997). Weak emergence. *Noûs*, 31(Suppl 11), 375–399.

Boudry, M., & Pigliucci, M. (2013). The mismeasure of machine: Synthetic biology and the trouble with engineering metaphors. Studies in history and philosophy of science Part C. *Studies in History and Philosophy of Biological and Biomedical Sciences*, 44(4), 660–668.

Calcott, B. (2014). Engineering and evolvability. *Biology & Philosophy*, 29(3), 293–313.

Griffiths, P. E. (2001). Genetic information: A metaphor in search of a theory. *Philosophy of Science*, 394–412.

Griffiths, P. E., Pocheville, A., Calcott, B., Stotz, K., Kim, H., & Knight, R. (2015). Measuring causal specificity. *Philosophy of Science*, 82(4), 529–555.

Jaeger, J., Laubichler, M., & Callebaut, W. (2015). The comet cometh: Evolving developmental systems. *Biological Theory*, 10(1), 36–49.

Keller, E. F. (2001). Beyond the gene but beneath the skin. In S. Oyama, P. E. Griffiths, & R. D. Gray (eds.), *Cycles of Contingency: Developmental Systems and Evolution*, 299–312. MIT Press, Boston.

Keller, E. F. (2003). *Making Sense of Life: Explaining Biological Development with Models, Metaphors, and Machines*. Cambridge, MA: Harvard University Press.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 135–146. SIGMOD '10. New York, ACM.

Moczek, A. P., Sears, K. E., Stollewerk, A., Wittkopp, P. J., Diggle, P., Dworkin, I., Ledon-Rettig, C. et al. (2015). The significance and scope of evolutionary developmental biology: A vision for the 21st century. *Evolution & Development*, 17(3), 198–219.

Newman, S. A., & Bhat, R. (2009). Dynamical patterning modules: A 'Pattern Language' for development and evolution of multicellular form. *The International Journal of Developmental Biology*, 53(5–6), 693–705.

Nicholson, D. J. (2014, December). The machine conception of the organism in development and evolution: A critical analysis. *Studies in History and Philosophy of Science Part C: Studies in History and Philosophy of Biological and Biomedical Sciences*, 48, 162–174.

Nijhout, H. F. (1990). Problems and paradigms: Metaphors and the role of genes in development. *BioEssays*, 12(9), 441–446.

Oyama, S., Gray, R. D., & Griffiths, P. E. (eds.). (2003). *Cycles of Contingency: Developmental Systems and Evolution*. Reprint ed. Cambridge, MA: A Bradford Book.

Peter, I. S., & Davidson, E. H. (2013). Pattern formation in Sea Urchin Endo mesoderm as instructed by gene regulatory network topologies. In V. Capasso, M. Gromov, A. Harel-Bellan, N. Morozova, & L. L. Pritchard (eds.), *Pattern Formation in Morphogenesis*, 75–92. Berlin, Heidelberg: Springer Proceedings in Mathematics.

Pigliucci, M. (2010). Genotype–phenotype mapping and the end of the 'genes as blueprint' metaphor. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 365(1540), 557–566.

Planer, R. J. (2014). Replacement of the 'genetic program' program. *Biology & Philosophy*, 29(1), 33–53.

Vasarhelyi, G., Viragh, C., Somorjai, G., Tarcai, N., Szorenyi, T., Nepusz, T., & Vicsek, T. (2014). Outdoor flocking and formation flight with autonomous aerial robots. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3866–3873. Chicago, IEEE.

Weber, M. (2016). Discussion note: Which kind of causal specificity matters biologically. *Philosophy of Science*, 84.