A Theory of Computational Implementation

Abstract: I articulate and defend a new theory of what it is for a physical system to implement an abstract computational model. According to my *descriptivist* theory, a physical system implements a computational model just in case the model accurately describes the system. Specifically, the system must reliably transit between computational states in accord with mechanical instructions encoded by the model. I contrast my theory with an influential approach to computational implementation espoused by Chalmers, Putnam, and others. I deploy my theory to illuminate the relation between computation and representation. I also address arguments, propounded by Putnam and Searle, that computational implementation is trivial.

§1. The physical realization relation

Physical computation occupies a pivotal role within contemporary science. Computer scientists design and build machines that compute, while cognitive psychologists postulate that the mental processes of various biological creatures are computational. To describe a physical system's computational activity, scientists typically offer a *computational model*, such as a Turing machine or a finite state machine. Computational models are abstract entities. They are not located in space or time, and they do not participate in causal interactions. Under certain circumstances, a physical system *realizes* or *implements* an abstract computational model. Which circumstances? What is it for a physical system to realize a computational model? When does a concrete physical entity --- such as a desktop computer, a robot, or a brain --- implement a given computation? These questions are foundational to any science that studies physical computation.

Over the past few decades, philosophers have offered several theories of computational implementation. I seek to provide a new theory that improves upon prior efforts. My approach differs from most predecessors in a crucial methodological respect. Existing theories typically pursue reductive analysis. They attempt to isolate non-circular necessary and sufficient conditions for a physical system to realize a computation. In contrast, I treat *physical realization* as a primitive concept. Most philosophically interesting concepts resist non-circular reduction: *cause*, *person*, *knowledge*, and so on. I see no reason to expect that *physical realization* will prove more cooperative. Luckily, one can *illuminate* a concept without *reductively analyzing* it. I seek to illuminate computational implementation without offering a reductive analysis.

In §§2-3, I present my theory. In §4, I contrast my theory with a popular approach espoused by Chalmers (1995, 1996, 2011), Putnam (1988), and many other researchers. I then apply my view to two vexed topics: the relation between computation and representation (§5); and Putnam-Searle triviality arguments (§6). I conclude by defending my failure to pursue a reductive analysis (§7).

§2. Descriptivism about computational implementation

The basic idea behind my position runs as follows. A computational model is an abstract description of a system that reliably conforms to a finite set of *mechanical instructions*. The instructions dictate how to transit between states. A physical system implements the computational model when the system reliably conforms to the instructions. Thus,

Physical system *P* realizes/implements computational model *M*

just in case

Computational model *M* accurately describes physical system *P*.

I call my position *descriptivism*. This section elaborates the intuitive rationale behind descriptivism. §3 formulates descriptivism more rigorously.

Many activities are governed by instructions: following a recipe, performing a musical score, constructing a building according to a blueprint, etc. The instructions are sometimes rather vague ("sauté the chicken until it is nicely browned," "*molto agitato*"). Computation is activity that conforms to a finite set of *routine, mechanical* instructions. For example, the Euclidean algorithm allows one to compute the greatest common divisor of numbers m and n, where $n \leq m$. Here is Knuth's formulation of the algorithm (1968, pp. 2-3):

> **E1.** [Find remainder] Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)
>
> **E2.** [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.
>
> **E3.** [Interchange] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1.

Executing the algorithm requires conforming to instructions E1-E3 in the proper order.[1]

To formulate E1-E3, Knuth uses natural language augmented with some mathematical symbols. For most of human history, this was basically the only way to formulate mechanical instructions. The computer revolution marked the advent of rigorous *computational formalisms*, which allow one to state mechanical instructions in a precise, unambiguous, canonical way. The scientific literature features two main types of computational formalism:

- *Programming languages*, such as LISP, C++, or assembly language
- *Machine models*, such as Turing machines, register machines, or finite state machines

I use the phrase "computational model" to include both types of formalism. Whatever the formalism, one specifies a computation by specifying a program. The program encodes

[1] Cleland (2002) also emphasizes that *conformity to instructions* underlies our intuitive notion of computation. However, she does not exploit this observation to offer a theory of implementation for abstract computational models. Instead, she uses it to attack the standard view that abstract models are central to elucidating physical computation. In contrast, I accept the standard emphasis on abstract models.

instructions, to which the computation must conform. For instance, Feferman describes a Turing machine as "an idealized computational device following a finite table of instructions (in essence, a program) in discrete effective steps without limitation on time or space that might be needed for a computation" (2006, p. 203). Similarly, Abelson *et al.* state that a register machine "sequentially executes *instructions* that manipulate the contents of a fixed set of storage elements called *registers*" (1996, p. 490). One can codify a program through a programming language, a machine table, a set-theoretic transition function, or various other means.

Within computer science, a computational model serves primarily as a blueprint. We use the model to build and manipulate physical systems. Within cognitive psychology, a computational model serves primarily as a scientific hypothesis. We postulate that the model accurately describes some biological creature's mental activity. Despite this contrast between computer science and cognitive psychology, computational models play an essentially descriptive role within both disciplines. We seek to ensure that our computational model accurately describes some physical system, so that we can successfully manipulate or explain the system's activity. Depending on our pragmatic or explanatory goals, we may enforce descriptive accuracy either by adjusting the model or else by adjusting the physical system. A computational model advances our pragmatic or explanatory goals because it describes certain physical systems accurately, or with approximate accuracy modulo certain idealizations. By building an artificial system that our computational model accurately describes, we ensure that the system does what we want it to do. By discovering a computational model that accurately describes a biological creature, we explain aspects of the creature's mental activity.

A physical system usually has many notable properties besides those encoded by our computational model: colors, shapes, sizes, and so on. A physical system may even have notable

computational properties besides those encoded by our computational model. For example, when we describe a desktop computer through a high-level programming language, we leave open the particular digital circuitry implemented by the computer. Thus, an accurate computational model need not be *complete*. A similar situation prevails within science more generally. A scientific model of a physical system (e.g. a macrophysical model) usually leaves open many important properties (e.g. the system's particular microphysical constitution).

A computer scientist or cognitive scientist may intentionally employ an inaccurate computational model. For example, she may describe a desktop computer through an idealized model that postulates infinite discrete memory capacity, even though the computer only has finite memory. In similar fashion, physicists frequently employ idealized models that postulate frictionless surfaces or massless strings. An idealized or oversimplified model may be useful for certain purposes. Strictly speaking, though, a physical system implements a computational model only if the model accurately describes the system. Even when an inaccurate computational model of a physical system serves our present explanatory or pragmatic ends, the system does not literally realize the model.

§3. Descriptivism clarified

I now formulate descriptivism more carefully. I focus exclusively on deterministic models. Extending my treatment to stochastic models, while straightforward, would clutter the exposition. My approach is general enough to accommodate diverse formalisms: Turing machines, register machines, finite state machines, higher-level programming languages, and so on. My approach also accommodates certain kinds of analog computation. I assume that computation proceeds in discrete stages, so my discussion does not accommodate analog models

that treat time continuously. Thus, I do not herald my theory as encompassing all important phenomena labeled "computational" in the scientific literature. Nevertheless, my account is general enough to encompass an extremely wide range of important computational phenomena.

Our task is to elucidate the notion *computational model M accurately describes physical system P*. The intuitive idea I will pursue is that *M* accurately describes *P* just in case *P* reliably moves through "state space" according to mechanical instructions encoded by *M*.

We need a canonical notation that can uniformly express the descriptive content of any computational model. In that spirit, I introduce *canonical state space descriptions*. A canonical state space description is an ordered-quadruple

$<S, I, \Omega, s_0>$

whose elements have the following properties:

(1) S is the *state space*: each $s \in S$ is a possible computational state. A state s includes the system's current output, if any. If the system can receive input during computation, then s does *not* include the system's current input.

(2) I is the set of inputs, possibly including the *null input* (corresponding to a situation where the system receives no input during some stage of computation).

(3) The transition function Ω encapsulates how current computational state (including input) determines the next computational state (excluding input). More technically, Ω is a function from $S \times I$ to S. It carries each $s \in S$ and $i \in I$ to an element $\Omega(s, i) = s^*$, where $s^* \in S$.

(4) Many computational models have a privileged *initial state*. In such a case, $s_0 \in S$ is the privileged initial state. If the model has no privileged initial state, then the parameter s_0 is irrelevant to us, so we let it be some fixed entity not belonging to S.

State space description $<S, I, \Omega, s_0>$ *accurately describes* physical system *P* just in case:

(1) For each $s \in S$, *s* is a possible state of *P*.

(2) For each $i \in I$, *i* is a possible input to *P*.

(3) *P* reliably conforms to the transition function Ω. More precisely: if *P* were to enter into state *s* and to receive input *i*, then *P* would transit to state $\Omega(s, i)$ at the next stage of computation.

(4) Absent external inference or internal malfunction, *P* always begins computation in state s_0 (assuming that $s_0 \in S$).

If computational model *M* has determinate descriptive content, then *M* dictates how any physical realizer transits through a determinate space of possible computational states. Thus, *M* corresponds to a unique state space description $<S, I, \Omega, s_0>$. I say that *M induces* $<S, I, \Omega, s_0>$. *M accurately describes P* just in case the induced state space description $<S, I, \Omega, s_0>$ accurately describes *P*. Under precisely those same circumstances, *P realizes* or *implements M*.

The notion of "canonical state space description" is far more general than the notion of computation. Many canonical state space descriptions are not *computational* in any natural sense. This is no problem for my account. What matters is that we can convert an extremely wide range of computational models into canonical state space descriptions, thereby delineating implementation conditions for those models.

Clause (3) employs the *counterfactual conditional*. Whether a physical system implements a computation depends upon how the system *would* behave under various circumstances (Chalmers, 1995), (Copeland, 1996). How should we interpret the relevant counterfactuals? What are their truth-conditions? There is a large philosophical literature on counterfactual conditionals in general (Lewis, 1973), in the specific context of scientific

modeling (Woodward, 2000), and in the more specific context of computational modeling (Chalmers, 1995). Descriptivists can freely deploy the resources offered by this literature. For present purposes, I remain neutral regarding how exactly one should elucidate clause (3).

Descriptivism addresses the conditions a physical system must satisfy to implement a computational model. In any given case, those conditions may or may not be physically satisfiable. For instance, the Turing machine formalism postulates infinite discrete memory capacity, yet it may be that no possible physical machine has infinite discrete memory capacity. Descriptivism elucidates *what it is* to implement a computational model, without offering any guarantee that a given model is implement*able*.

§3.1 Converting computational models into state space descriptions

To apply my descriptivist theory to a computational model *M*, we must convert *M* into a canonical state space description $<S, I, \Omega, s_0>$. This is easier in some cases than others, because computational models vary considerably in precision, detail, and explicitness.

Deterministic finite state machines (FSMs) allow a relatively straightforward application of my theory. An FSM has finitely many "machine states" and finitely many inputs. Current machine state and current input determine the next machine state. A common textbook example is an *elevator controller*, such as the elevator FSM described by Mozgovoy (2010, p. 92). The FSM, which I will call "ELEV", has four buttons labeled "O" (for *open*), "C" (for *close*), "D" (for *down*), and "U" (for *up*). These four inputs comprise the machine's input set. The FSM has four possible machine states:

the door is closed and the elevator is on the first floor (cD)

the door is open and the elevator is on the first floor (oD)

the door is closed and the elevator is on the second floor (cU)

the door is open and the elevator is on the second floor (oU)

The initial state is oD. Transitions among ELEV's states and inputs conform to the machine table given by Table 1. ELEV is overly simplistic in many respects. For instance, ELEV does not specify what happens if the machine receives input while traveling between floors, or if one presses multiple buttons at once, and so on. For a more realistic elevator FSM, see (Vahid and Givargis, 2002, p. 211).

INSERT TABLE 1 ABOUT HERE

It is evident how to convert ELEV's machine table into a transition function Ω_{ELEV}. The canonical state space description induced by ELEV is

<{cD, oD, cU, oU}, {button O, button C, button D, button U}, Ω_{ELEV}, oD>

According to my descriptivist theory, a physical system *P* implements ELEV just in case:

(1) *P* can instantiate all four states cD, oD, cU, oU. Thus, *P* has a door that can open and close, and *P* can travel between the first and second floors of some building.

(2) *P* has four buttons labeled "O", "C", "D", "U".

(3) *P* reliably conforms to the instructions encoded by ELEV's machine table, such as: *If the door is closed and the system is on the first floor, and if button U is pressed, then transit to having the door closed and to being located on the second floor*.

(4) Absent external interference or internal malfunction, *P* begins on the first floor with the door open.

This implementation condition seems intuitively correct.

Most computational models are not so easy to convert into canonical state space descriptions. For example, a complete model of a desktop computer would mention numerous internal components. The corresponding canonical state space description enumerates possible states of these components. In practice, researchers rarely provide anything as detailed or explicit as a canonical state space description. Instead, they informally indicate a state space description through a mixture of English, computer code, diagrams, and so on. What matters for us is that a canonical state space description is possible *in principle* whenever a computational model has a determinate implementation condition. Thus, philosophical theories of computational implementation can legitimately cite canonical state space descriptions.

In straightforward cases such as ELEV, the computational model induces a unique state space description $<S, I, \Omega, s_0>$. In less straightforward cases, there may not be a unique induced $<S, I, \Omega, s_0>$. To illustrate, consider the distinction between *physical* and *virtual* memory. Many computational formalisms incorporate some notion of "memory location" (e.g. the cells on a Turing machine tape, or the registers in a register machine). One might correlate each memory location with a *physical location* in the implementing system. Alternatively, as Chalmers (1996) notes, one might correlate each memory location with an addressable "virtual location" whose physical location changes. These two contrasting approaches yield contrasting state space descriptions. Which approach is correct? I believe that the answer is *indeterminate* for most computational formalisms, including the Turing machine and the register machine. For example, nothing about the descriptive use of register machines within contemporary scientific practice

dictates whether we should interpret talk about "memory registers" in physical or virtual terms. Either interpretation may be appropriate, depending upon our pragmatic or explanatory ends.[2]

This indeterminacy is harmless. If our current use of a computational model does not associate the model with a determinate canonical state space description, then we can stipulate away the indeterminacy as our pragmatic or explanatory needs dictate. For example, a computer designer can stipulate that she intends talk about "memory locations" to be interpreted in physical rather than virtual fashion (or vice versa).

When computational model *M* does not induce a unique state space description, we have several theoretical options regarding *M*'s implementation condition. We might say that *P* implements *M* just in case *some* canonical state space description induced by *M* accurately describes *P*. Or we might say that *M*'s implementation condition is indeterminate, to become determinate only once we stipulate a unique state space description induced by *M*. I suspect that the second option more faithfully captures actual scientific practice, although the first option may be more appropriate in certain cases. For present purposes, I leave the matter unresolved.

What is it for computational model *M* to induce state space description $<S, I, \Omega, s_0>$? I will not attempt to answer this question. Instead, I treat the "inducement relation" as primitive. I exploit our pre-philosophical ability to convert computational models into corresponding state space descriptions. To the extent that one associates a computational model with a determinate implementation condition, this conversion is always possible in principle.

One might worry that my methodology simply shifts the explanatory burden from the "realization" relation between model and physical system to the "inducement" relation between model and state space description. How can a good theory presuppose a primitive inducement

[2] High-level programming languages furnish additional examples in the same vein. As Chalmers (2012, p. 222) notes, there is usually slack between a program couched in a high-level programming language and a more explicit description in terms of states and state-transitions.

relation? I reply that I am not pursuing a reductive analysis. I am trying to offer an illuminating theory that respects how computational implementation figures within contemporary scientific practice. Such a theory may legitimately presuppose a primitive inducement relation.

§3.2 The importance of descriptive practice

Scientists *use* abstract computational models to *describe* physical systems. A model *as used within current descriptive practice* places conditions on which physical systems implement the model. Implementation conditions arise partly from our descriptive employment of computational models within scientific practice.[3]

For example, consider higher-level programming languages such as LISP or Pascal. A program is composed of signs, which lack any inherent meaning in themselves. The signs do not even begin to determine implementation conditions. Implementation conditions arise only because current practice associates the signs with an *intended interpretation*, according to which the signs express mechanical instructions. Only in light of the intended interpretation can we ask whether a physical system implements the program.

An analogous point applies to machine models, including FSMs, Turing machines, and register machines. Consider the standard definition of FSM (Hopcroft and Ullman, 1979, p. 17):

> We formally denote a *finite automaton* by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite *input alphabet*, q_0 is the initial state, $F \subseteq Q$ is the set of *final* states, and δ is the transition function mapping $Q \times \Sigma$ to Q.[4]

[3] Cleland (2002) makes a similar point.

[4] Why does the definition include a privileged set of final states F? So that we can model "language acceptance." An FSM *accepts* a string of symbols just in case sequentially inputting that string to the FSM produces a final state belonging to F. Relations between formal languages and FSMs play a central role in automata theory, but they will not figure in my exposition. For all the FSMs I consider in this paper, we may simply set F=Q.

Suppose that the state space Q for an FSM contains states q_0, q_1, … , q_r. Suppose that $\delta(q_n, i) = q_m$, where i is some member of the input set Σ. Following standard practice, we could interpret this FSM as encoding an instruction to transit from state q_n and input i to state q_m. Or we could employ a deviant interpretation on which the FSM encodes an instruction to transit from state q_n and input i to state $q_{(m+1) \bmod r}$. Or we could employ a deviant interpretation on which the FSM encodes an instruction to transit from state q_n and input i to state $\delta(\delta(q_n, i), i)$. Nothing about the FSM itself, *qua* set-theoretic object, favors one interpretation over the others. Of course, the standard interpretation impresses us as most natural. But that impression does not reflect any intrinsic features of the set-theoretic object. Rather, it reflects how we *use* the set-theoretic object to *describe* physical systems.

Strictly speaking, then, a physical system realizes a computational model only *relative* to a descriptive practice that confers an implementation condition upon the model. Rather than say

> Computational model *M* is implemented by physical system *P*

it would be more appropriate to say

> Computational model *M as used within some particular descriptive practice* is implemented by physical system *P*,

thereby making explicit the relativity to descriptive practice. In practice, context usually makes salient one particular descriptive practice, so that we can safely suppress the relativity. In certain cases, such as computational models that are ambiguous between physical and virtual memory, current descriptive practice may *constrain* implementation conditions without *determining* a single unique implementation condition.

We can rephrase these points in terms of canonical state space descriptions. Assume that we hold fixed the implementation condition associated with each state space description <*S*, *I*, Ω,

s_0>. (Relaxing that assumption would only further accentuate the relativity to descriptive practice.) Then a machine model *qua* set-theoretic object does not even begin to determine an induced state space description <*S*, *I*, Ω, s_0>. Only the set-theoretic object *plus* descriptive practice can determine <*S*, *I*, Ω, s_0>. In certain cases, descriptive practice constrains <*S*, *I*, Ω, s_0> without determining <*S*, *I*, Ω, s_0>.

My heavy appeal to descriptive practice will repulse some readers. How can a rigorous foundation for computational science cite anything as "squishy" as descriptive practice? Shouldn't we study implementation *tout court*, without relativization to our descriptive activity?

I reply that it is a fool's errand to study implementation *tout court*, detached from descriptive practice. An abstract computational model, viewed in detachment from any descriptive use we make of it, cannot magically select certain physical systems as its realizations. For example, an FSM *viewed in isolation from descriptive practice* does not even begin to determine an implementation condition. That my account assigns a central role to descriptive practice is an advantage, not a disadvantage. Most previous accounts either downplay or altogether ignore this crucial aspect of computational implementation.

By relativizing computational implementation to descriptive practice, I do not render the physical realization relation subjective. I do not relativize it to the observer's whim. Computational implementation remains as objective as one could reasonably desire. By analogy, consider *recipe implementation*. A physical process implements a recipe just in case the recipe accurately describes the process. Words in a recipe describe a physical process only by virtue of a descriptive practice, which endows the words with meaning and thereby makes them instructions for manipulating ingredients. In that sense, recipe implementation is relative to a descriptive practice. Nevertheless, it is a perfectly objective matter whether one implements a

recipe *as the recipe figures in some descriptive practice*. For instance, it is an objective matter whether one executes the instruction "add salt" *as that instruction figures in current linguistic practice*. Similarly, it is an objective matter whether a physical system realizes a computational model *as the model figures in current descriptive practice*. Relativity to descriptive practice is harmless and inevitable. It reflects the fact that descriptive tools (e.g. words, diagrams, set-theoretic models) acquire descriptive content only from participating in such a practice.

My emphasis upon descriptive practice echoes a theme familiar from philosophy of science more generally.[5] Many philosophers of science have emphasized that an abstract mathematical model (such as Maxwell's equations) describes the physical world only through the model's role in a larger descriptive practice. A mathematical model does not *in itself* determine what physical phenomena the model depicts. Only the model *as used within current scientific practice* depicts the physical world as being a certain way. van Fraassen (2008, pp. 11-31, pp. 189-190, pp. 238-261) develops this theme in detail, showcasing numerous compelling examples drawn from scientific practice. For present purposes, I continue to focus exclusively upon the special case of computational modeling. That special case raises more than enough complexities to fill an entire paper.

§4. Comparison with structuralist theories of implementation

I now want to compare my descriptivist theory with an alternative approach that I call *structuralism about computational implementation*. The intuitive idea underlying structuralism is that a physical system implements a computational model just in case the system's causal structure "mirrors" the model's formal structure. As Chalmers (1995, p. 401) puts it:

[5] And familiar also from Wittgenstein's later writings: "The arrow points only in the application that a living being makes of it" (1953, §454).

> To implement a computation is just to have a set of components that interact causally according to a certain pattern. The nature of the components does not matter, and nor does the way that the causal links between components are implemented; all that matters is the pattern of causal organization of the system.

He suggests that we codify this intuitive idea along the following lines (1995, p. 392):

> A physical system implements a given computation when there exists a grouping of physical states of the system into state-types and a one-to-one mapping from formal states of the computation to physical state-types, such that formal states related by an abstract state-transition relation are mapped onto physical state-types related by a corresponding causal state-transition function.

Roughly speaking, then, physical system *P* implements computational model *M* just in case:

> (∃*F*)(*F* is an isomorphism from *M*'s formal structure to *P*'s causal structure).

A physical system realizes a computational model when the system instantiates a "causal structure isomorphism type" dictated by the model's formal structure.[6]

Proponents of structuralism, or similar doctrines, include Copeland (1996), Dresner (2010), Godfrey-Smith (2009), Klein (2008), Putnam (1988), and Scheutz (2001). Precise formulations vary considerably. For example, Putnam elucidates the "isomorphism" between formal structure and causal structure by employing the *material conditional*. He demands that the mapping *F* from formal states to physical states satisfy the following constraint:

---

[6] There are affinities between structuralism about computational implementation and broadly "structuralist" views within philosophy of science more generally. The intuitive idea behind such views is that scientific theories represent only "structural" features of the world (Bueno and French, 2011), (van Fraassen, 2008). However, structuralists within general philosophy of science need not endorse structuralism about computational implementation. Their position concerns the representational import of scientific theories, not the implementation relation between computational models and physical systems. To derive a structuralist view of computational implementation, one requires an additional premise linking representational import and implementation conditions, a premise that structuralist philosophers of science might well reject.

> If the model's transition function carries formal state s_1 to formal state s_2, and if the physical system instantiates the physical state to which F maps s_1, then the physical system transits to the physical states to which F maps s_2.

Chalmers (1995, 1996) and Copeland (1996) instead demand that F satisfy *counterfactual conditionals* along the following lines:

> If the model's transition function carries formal state s_1 to formal state s_2, and if the physical system *were* to instantiate the physical state to which F maps s_1, then the physical system *would* transit to the physical state to which F maps s_2.

These counterfactual conditionals are much stronger than Putnam's material conditionals.

Descriptivism is compatible with structuralism. Indeed, structuralists sometimes express descriptivist sentiments. For example, Putnam characterizes physical realization of Turing machines as follows: "A 'machine table' describes a machine if the machine has internal states corresponding to the columns of the table, and it 'obeys' the instruction in the table… Any machine that is described by a machine table of the sort just exemplified is a Turing machine" (1975, p. 365). Even more explicitly, Chalmers writes: "Implementation is the relation that holds between an abstract computational object (a *computation* for short) and a physical system, such that we can say that in some sense the system 'realizes' the computation, and that the computation 'describes' the system" (1995, p. 391).

Alternatively, one can endorse descriptivism while rejecting structuralism. I favor this combination of views. I agree that a mirroring relation between M's formal structure and P's causal structure is *necessary* for P to implement M. But I will now argue that a mirroring relation does not generally *suffice* for P to implement M. A physical system can instantiate the pattern of causal organization dictated by a computational model without implementing the model.

§4.1 Counterexamples to structuralism

Implementing a computational model requires reliably conforming to mechanical instructions encoded by the model. Conforming to instructions requires *doing what the instructions say*. In many cases, this requires instantiating properties that do not reduce to any relevant pattern of causal organization. I illustrate with a parable:

> ***The Parable of the Elevator***: An architect hired an engineer to manufacture and install the elevator for a new building. The architect told the engineer to build a physical system that implemented ELEV. The engineer built a stationary machine in the shape of an elevator. The machine had four buttons labeled "O", "C", "D", and "U". It also contained a red light in its interior. It conformed to the machine table given by Table 2, where cON means *the door is closed and the red light is on*, cOFF means *the door is closed and the red light is off*, and so on. To test the engineer's handiwork, the architect entered the elevator and pressed C. The door closed. The architect pressed U, but the elevator did not move. Instead, the red light turned off. The architect lambasted the engineering: "This so-called elevator cannot even travel between floors," she complained. "Clearly, it does not travel between floors in the manner dictated by my model." The engineer, who had closely studied structuralist theories of computational implementation, replied that his machine implemented ELEV, since its causal structure mirrored ELEV's formal structure. The architect fired the engineer. □

INSERT TABLE 2 ABOUT HERE

ELEV does not just specify a pattern of causal organization. It describes a system's possible states, and it specifies instructions governing how to transit between those states. Implementing the model requires an ability to instantiate states specified by the model. For instance, a physical system implements ELEV only if it can travel between the first and second floors of some building. Only then can it obey an instruction such as:

> If the door is closed and the system is on the first floor, and if button U is pressed, then transit to having the door closed and to being located on the second floor.

The machine from the parable cannot travel between floors, so it does not implement ELEV. It implements a distinct FSM, which I will call "ELEV*", whose machine table is given by Table 2. But it does not implement ELEV, even though it instantiates the causal pattern dictated by ELEV. Thus, ELEV is a counterexample to structuralism.

ELEV is an extremely simplistic FSM. Contemporary science offers numerous more realistic counterexamples that illustrate the same point. For example, the far more sophisticated elevator FSM discussed by Vahid and Givargis (2002, p. 211) encodes instructions that begin as follows: "Move the elevator either up or down to reach the target floor. Once at the target floor, open the door for at least 10 seconds, and keep it open until the target floor change." Similarly, Patterson and Hennessy (2005, p. C-69) introduce a traffic light FSM that includes states such as *the traffic light is green in the north-south direction* and *the traffic light is green in the east-west direction*. Any textbook on embedded systems design discusses additional FSMs with non-structuralist implementation conditions: alarm clocks, seatbelt detection systems, aviation controllers, and so on. Robotics is also a rich source of counterexamples. The robot car Junior (Montemerlo, et al., 2008, pp. 114-115) implements a FSM whose states include:

LOCATE_VEHICLE: “the robot estimates its initial position... and starts road driving or parking lot navigation, whichever is appropriate”

CROSS_INTERSECTION: “the robot waits if it is safe to cross an intersection (e.g., during merging), or until the intersection is clear (if it is an all-way stop intersection)”

UTURN_STOP: “the robot is stopping in preparation for a U-turn”

and so on. Implementing Junior’s FSM requires an ability to drive, to wait at intersections, and so on. Murphy (2000, pp. 174-184) offers several FSMs that illustrate the same point: a robot that moves through an obstacle course; a robot that seeks, retrieves, and relocates trash; and so on. Each FSM encodes instructions, such as *move towards trash* and *grab trash*, that outstrip any relevant pattern of causal organization. Each FSM is a counterexample to structuralism.

A core idea underlying structuralism is that machine states are individuated *functionally*, i.e. by their roles in a pattern of causal organization. Structuralism makes this idea precise by invoking isomorphisms between formal structure and causal structure. A machine state is individuated by its place in a “causal structure isomorphism type” induced by the formal model. The foregoing FSMs undermine this approach. Each FSM includes at least one machine state whose nature outstrips its place in the “causal structure isomorphism type” induced by the formal model. States such as *being located on the first floor* and *waiting at an intersection* have non-functional natures that go beyond any relevant pattern of causal organization. My descriptivist theory accommodates each of these FSMs by postulating a state space description $\langle S, I, \Omega, s_0 \rangle$ whose state space S contains the desired non-functional states.

According to structuralism, *P* implements *M* when *there exist* states of *P* that reliably interact according to *M*’s transition function. According to my descriptivist theory, *P* implements *M* only if *particular states specified by M* reliably interact according to *M*’s transition function.

My position coheres much better with numerous computational models that figure prominently in contemporary science, especially embedded systems design and robotics. Many FSMs *as used within current scientific practice* encode mechanical instructions that cite non-functional properties. These FSMs have non-structuralist implementation conditions.[7]

Objection: FSMs are abstract mathematical entities. A genuine FSM cannot contain non-abstract states such as *being located on the first floor with the door closed*.

Reply: An FSM is a set, and sets are abstract entities. But sets can contain non-abstract entities, as witnessed by the set {Bill Clinton, George W. Bush}. Bearing this point in mind, I invite readers to consult the standard set-theoretic definition of FSM, as presented in §3.2. You will find that the definition places no restrictions on Q, the finite set of states. Q may contain whatever states one likes. In particular, Q may contain non-abstract states (e.g. *being located on the first floor*). So this objection flouts the standard mathematical definition of FSM. The standard definition is perfectly congenial to FSMs that contain non-abstract states. As I argued above, such FSMs occupy a central role within both embedded systems design and robotics. A good theory of computational implementation must accommodate these computational models. To dismiss them is to reject large tracts of current scientific practice.

Objection: ELEV and ELEV* have the same formal structure. More precisely, there exists a mapping between them that respects their transition functions. From a mathematical perspective, all that matters about a computational model is its formal structure. So

[7] Chalmers introduces *complex state automata* (CSAs), which abstract away from idiosyncratic features of particular computational formalisms. He delineates precise structuralist implementation conditions for CSAs, and he specifies a computational model's implementation condition by translating the model into an appropriate CSA. Thus, CSAs play a role within Chalmers's account analogous to the role that canonical state space descriptions play within my account. There are various relatively minor differences between CSAs and canonical state space descriptions, but the major difference reflects Chalmers's structuralist commitments. The states composing a CSA are abstract states with no inherent natures beyond their role in a formal structure determined by the CSA. Implementing a CSA simply requires that a physical system's causal structure mirror the CSA's formal structure. In contrast, the states that compose a canonical state space description can involve non-functional properties (such as *being located on the first floor*) that outstrip any relevant formal or causal structure.

mathematicians would not recognize any significant difference between ELEV and ELEV*. Talk about non-functional properties (such as *being located on the first floor*) is simply a heuristic gloss that helps motivate a given formal structure.

Reply: Let us grant that there is no important difference between ELEV and ELEV* *from a mathematical perspective*. It does not follow that there is no important difference between ELEV and ELEV*. FSMs are objects of mathematical study, but they are not *solely* objects of mathematical study. We also use FSMs to *describe* physical systems. In particular, engineers use FSMs to guide the construction of physical systems with desired properties. From an engineering perspective, the states that compose a ELEV are not just abstract nodes in a formal structure. They have non-functional natures vital to the system's proper operation. The engineer from the parable ignores this fact, so he constructs a machine that deviates wildly from the architect's intended design (as codified by ELEV). Overemphasis on abstract mathematics over practical engineering has nurtured mistaken philosophical theories of computational implementation.

Objection: Since ELEV and ELEV* have the same formal structure, they have the same descriptive import. One can describe an elevator just as easily through ELEV* as ELEV, and one can describe the machine from the parable just as easily through ELEV as ELEV*.

Reply: A computational model's formal structure does not exhaust its descriptive import. Numerous computational models, including numerous models that figure prominently in contemporary science, describe more than a pattern of causal organization. When scientists use an FSM to describe an elevator, a traffic light, a robot, or any other embedded computing system, they do not simply seek to describe a "causal structure isomorphism type." They seek to describe how the system transits between specific non-functional states (e.g. *being located on the first*

floor). Thus, structuralism systematically underestimates the descriptive aspirations of contemporary computational modeling.

Of course, one can alter the implementation condition associated with a given computational model. *We* use computational models as descriptive tools, so we can always change our descriptive use of a model. For example, suppose that two computational models (such as ELEV and ELEV*) have the same formal structure but different implementation conditions. Through explicit stipulation, one can switch the implementation conditions associated with the two models. But our task as philosophers is to elucidate the implementation conditions of computational models *as those models figure in actual scientific practice*, not as those models figure in some imagined scientific practice that differs profoundly from actual practice. In many cases, the implementation condition that practicing scientists associate with a computational model does not depend solely upon the model's formal structure.

Objection: Your theory may provide plausible implementation conditions for an FSM *as used in current scientific practice*. But we should try to provide implementation conditions for an FSM considered in itself, detached from any descriptive use we make of it.

Reply: As I argued in §3.2, it is a fool's errand to study computational implementation in detachment from descriptive practice. A physical system realizes an FSM only *relative* to a descriptive practice that confers an implementation condition upon the FSM. A good theory of computational implementation must emphasize, rather than ignore, the descriptive employment of FSMs within current scientific practice.

Objection: Turing and others have argued convincingly that a Turing machine can execute any possible physical computation. As Minsky (1967, p. 108) puts it: "Any process which could naturally be called an effective procedure can be realized by a Turing machine."

Obviously, a Turing machine cannot move up or down. Yet you incorporate *moving up and down* into the implementation condition for ELEV. You thereby classify as ”computational” a physical process that no Turing machine can execute.

Reply: The Turing machine is an excellent model of *symbolic* computation. But some computational systems (e.g. typical robots) have inputs/outputs that are not strings of symbols. The Turing formalism on its own is inadequate to model such systems, because the inputs/outputs to Turing computation are strings of symbols. Nevertheless, one can readily overcome this inadequacy by *supplementing* the Turing formalism. For example, suppose we want to build a physical Turing machine that replicates ELEV's activity. We can connect the machine to input devices that detect elevator location and door status, and we can connect it to output devices that alter elevator location and door status. Once we supplement physical Turing machines with suitable input/output devices, we can easily write a Turing machine program that mimics ELEV's machine table. Thus, there is no interesting sense in which ELEV outstrips the computational capacities of Turing machines. My analysis is perfectly consistent with the thesis that physical Turing machines (appropriately connected to suitable input/output devices) can execute any possible physical computation.

§4.2 Bounded structuralism

Few philosophers espouse structuralism in the pure form articulated above. Proponents typically weaken structuralism along the following lines: physical system *P* implements computational model *M* just in case

(1) $(\exists F)(F$ is an isomorphism from M's formal structure to P's causal structure $\& \Phi(F))$,

where Φ is a further constraint upon *F*. For example, Copeland (1996), Godfrey-Smith (2009), and Scheutz (2001) demand that *F* carry formal states to physical states that are "natural" rather than "gerrymandered," while Chalmers (1996) demands that *F* respect combinatorial structure by carrying distinct elements of the formal model to distinct components of the physical system. There are many options here, depending upon what one substitutes for Φ. I will not canvass these options. In most cases, it is fairly easy to show that §4.1's counterexamples still apply. But I want to discuss one particularly notable version of (1).

Computational models often impose "boundary conditions" upon inputs and outputs (Chrisley, 1994), (Godfrey-Smith, 2009), (Putnam,1988). For example, a computational model may constrain states of a computer's keyboard and screen or a robot's sensors and motor organs. Accordingly, Putnam recommends that we replace structuralism with an emended view: a physical system realizes a computational model just in case the system instantiates an appropriate pattern of causal organization *and* satisfies desired "boundary conditions" on inputs and outputs. I call this view *bounded structuralism*. According to bounded structuralism, *P* implements *M* just in case

(2) (∃*F*)(*F* is an isomorphism from *M*'s formal structure to *P*'s causal structure & *F* satisfies input/output constraints determined by *M*).

A key challenge here is to elucidate "input/output constraints" so as to preserve structuralist intuitions while simultaneously avoiding §4.1's counterexamples. As I will now argue, this challenge is formidable.

The FSM formalism as articulated in §3.2 features no official notion of "output." There are simply inputs (drawn from Σ) and machine states (drawn from Q). For example, ELEV includes four inputs (corresponding to the four buttons) and four machine states (cD, oD, cU,

oU), but it does not feature any "outputs" beyond these inputs and machine states. One can extend the FSM formalism to include an "output alphabet." There are two standard formalisms (Hopcroft and Ullman, 1979, pp. 42-43): *Moore machines*, where output is determined by current internal machine state; and *Mealy machines*, where output is determined by current input and current internal machine state. ELEV is not a Moore machine or a Mealy machine, because it features no "output alphabet" distinct from the set of machine states Q.

If we employ a notion of "output" drawn from automata theory, then bounded structuralism falls prey to §4.1's counterexamples. The machine from the parable has a causal structure isomorphic to ELEV's formal structure. The machine also has inputs with desired properties: four buttons labeled "O", "C", "D", and "U". ELEV does not feature "outputs" in any sense familiar from automata theory. So (2) wrongly predicts that the machine from the parable implements ELEV, as long as we gloss "outputs" by citing an explicit "output alphabet."

Bounded structuralists may respond by proposing a more liberal notion of "output." They may say that ELEV imposes surreptitious output constraints (e.g. *activating the elevator motor in a certain way*), despite the lack of an explicit "output alphabet." The machine from the parable does not instantiate the requisite outputs, so it does not implement ELEV.

Unfortunately, this proposal is still problematic. ELEV describes how machine states *as individuated non-functionally* combine with inputs to determine subsequent computational developments. For example, ELEV encodes the mechanical instruction:

> If the door is closed and the system is on the first floor, and if button U is pressed, then transit to having the door closed and to being located on the second floor,

which requires that the system respond appropriately to the elevator's current location. The instruction cites non-functional properties not only when describing outcomes, but also when

describing states that reliably cause those outcomes. One cannot generate a correct implementation condition simply by demanding that the system yield suitable outputs (e.g. *activating the elevator motor in a certain way*). One must demand that the system yield those outputs in response to suitable non-functional states (e.g. *being located on the first floor*).

To avoid these worries, bounded structuralists may characterize "input/output constraints" in ever more liberal terms. For example, they may characterize ELEV's outputs along the following lines: *moving from the first floor to the second floor*. Alternatively, they may propose that ELEV imposes surreptitious input constraints, such as *being located on the first floor*. If we gloss "input/output constraints" in such liberal terms, then (2) probably yields the correct implementation condition for ELEV.

I reply that these maneuvers trivialize bounded structuralism beyond recognition. If we employ such a liberal conception of "input/output constraints," then we concede that any implementer of ELEV must transit appropriately between *being located on the first floor* and *being located on the second floor*. We thereby replace the distinctively structuralist emphasis upon patterns of causal organization with a decidedly non-structuralist emphasis upon reliable interaction among specific physical states. The "mirroring" relation between formal structure and causal structure no longer performs any serious work. Instead, what matters is how the system transits between non-functional states dictated by ELEV: cD (*the door is closed and the elevator is on the first floor*), oD (*the door is open and the elevator is on the first floor*), and so on. I have no quarrel with such a view. But neither does the view count as structuralist. The view assigns no significant theoretical role to causal patterns or to "causal structure isomorphism types."

We must not let surface form mislead. Mere conformity to schema (2) does not ensure that a theory is genuinely structuralist. By combining (2) with an overly liberal conception of

"input/output constraints," we abandon the basic idea behind structuralism. To illustrate, suppose we say that ELEV's input/output constraints mandate an elevator that can travel between floors and a door that can open and close. Then we concede that implementation requires appropriate interaction between non-functional states dictated by ELEV. Implementation requires that *specific physical states mentioned by ELEV's machine table* interact appropriately with one another. The existential quantifier over mappings *F* is an idle wheel. The only mapping relevant to ELEV's implementation is the *identity mapping* that carries cD to cD, oD to oD, and so on. Our account may conform to schema (2), but it no longer assigns a significant role to the mirroring relation between formal structure and causal structure.

A good theory must acknowledge that ELEV imposes highly specific non-functional restrictions upon computational states. Calling those restrictions "input/output constraints" does not change the fact that ELEV's machine states are individuated through non-functional properties (e.g *being located on the first floor*), rather than through any causal pattern dictated by ELEV's formal structure. A proper treatment of ELEV's implementation condition will assign no serious work to the mirroring relation between formal structure and causal structure. Hence, one cannot accommodate ELEV while preserving the basic idea behind structuralism.

§4.3 Further reflections on structuralism

Although I reject structuralism, I believe that it contains important elements of truth. Otherwise, it would not have attracted so many adherents. I now highlight three respects in which structuralism seems on the right track.

(1) *Structuralism and bounded structuralism predict correct implementation conditions for some (but not all) computational models*.

Consider an FSM with null alphabet, containing *n* "abstract" machine states S_1, S_2, …, S_n that interact according to the following transition function δ:

$\delta(S_i) = S_{i+1}$, for $i<n$

$\delta(S_n) = S_1$

The machine cycles through its states, never halting. Call this machine "CYCLE." Under what conditions does a physical system implement CYCLE? According to structuralism, a physical system implements CYCLE just in case the system has physical states P_1, P_2, …, P_n that reliably interact according to CYCLE's transition function. This structuralist analysis seems plausible. Since S_1, S_2, …, S_n are "abstract" states, there seems to be nothing to them beyond their functional roles, as dictated by the transition function. Thus, I concede for argument's sake that structuralism yields the correct implementation condition for CYCLE.

I also concede that bounded structuralism yields plausible results for certain FSMs. Consider an example discussed by Godfrey-Smith (2009): a vending machine FSM with two inputs (I_1 = 5 cents, I_2 = 10 cent), three outputs (O_1 = null, O_2 = coke, O_3 = coke & 5 cents), and three "abstract" states S_1, S_2, and S_3, governed by the machine table given by Table 3. This is a Mealy machine, since output depends upon input and current state. Call it "VEND." According to bounded structuralism, a physical system realizes VEND just in case the system has the appropriate "causal organization," constrained at the periphery by input/output states with appropriate non-functional properties. The system must be able to receive 5 or 10 cents as input, and it must be able to output a coke and 5 cents. VEND's inputs and outputs have intrinsic

natures that outstrip any relevant pattern of functional organization, but S_1, S_2, and S_3 have no such non-functional natures. Quite plausibly, there is nothing to S_1, S_2, and S_3 beyond how they interact with one another and with the specified non-functional inputs/outputs.

INSERT TABLE 3 ABOUT HERE

We may instructively compare VEND with ELEV. One difference is that VEND marks a clean distinction between machine states and outputs *inside the computational formalism*. The top row of VEND's machine table contains states S_1, S_2, and S_3 individuated entirely through their functional roles in the machine table. The machine table encodes mechanical instructions that specify how current input *described non-functionally* and current machine state *described functionally* determine subsequent machine state *described functionally* and subsequent output *described non-functionally*. Thus, VEND enshrines the distinction that lies at the heart of bounded structuralism: the distinction between non-functionally individuated inputs/outputs and functionally individuated machine states. In contrast, ELEV does not mark any explicit distinction between machine states and outputs. The top row of ELEV's machine table contains states cD, oD, cU, and oU individuated through properties (e.g. *being located on the first floor*) that outstrip any functional roles dictated by ELEV. The machine table encodes mechanical instructions that specify how current input *described non-functionally* and current machine state *described non-functionally* determine subsequent machine state *described non-functionally*. The machine table secures no evident role for functionally individuated machine states, because it individuates all machine states non-functionally. That is why bounded structuralism yields a plausible analysis for VEND but not ELEV.

In summary, there are many computational models for which structuralism or bounded structuralism predicts plausible implementation conditions. My descriptivist theory can easily accommodate these models, by positing a state space whose elements are individuated functionally. But my theory also accommodates models, such as ELEV, that structuralism handles incorrectly.

(2) *A single physical system can simultaneously implement a computational model for which structuralism (or bounded structuralism) predicts a correct implementation condition and a second computational model for which structuralism (or bounded structuralism) predicts an incorrect implementation condition.*

Consider the standard practice of transforming FSMs into digital circuits (Minsky, 1967, pp. 55-58). For example, suppose we want to construct a digital circuit corresponding to ELEV. The digital circuit is composed of various logic gates. At the periphery, the digital circuit connects to input devices, which detect elevator location and door status, and to output devices, which alter elevator location and door status. Wiring between logic gates and input/output devices ensures appropriate input/output behavior. Call our circuit "ELEV-CIR." A physical system may implement both ELEV and ELEV-CIR. Indeed, the point of constructing ELEV-CIR is to clarify how one might build a physical machine that implements ELEV.[8]

A bounded structuralist analysis of ELEV-CIR seems plausible. Quite plausibly, all "non-functional" properties specified by ELEV-CIR reside in peripheral inputs and outputs. Quite plausibly, there is nothing more to logic gate states beyond their interaction with each other and

[8] Vahid and Givargis (2002, pp. 209-211) sketch a digital circuit along these lines, geared to a far more sophisticated elevator FSM than ELEV. In a similar vein, we might also consider a Turing machine connected to appropriate input/output devices, as discussed at the end of §4.2

with inputs/outputs. Thus, I grant for argument's sake that bounded structuralism yields the correct implementation condition for ELEV-CIR.

How can bounded structuralism yield the right result for ELEV-CIR but not ELEV? Because these are different computational models, with different implementation conditions. ELEV-CIR marks a clear distinction between outputs and internal machine states *inside the computational formalism*. Quite plausibly, ELEV-CIR characterizes outputs non-functionally while characterizing machine states functionally. ELEV enshrines no such distinction between outputs and machine states. It simply posits non-functionally individuated machines states. That is why bounded structuralism yields a plausible analysis for ELEV-CIR but not ELEV.

We can also transform ELEV into a computational model for which structuralism *simpliciter* yields plausible results. Consider the FSM whose machine table is given by Table 4, where I_1-I_4 and S_1-S_4 are abstract states with no intrinsic natures beyond their roles in the machine table. Call this new FSM "ELEV**". Any physical system that implements ELEV also implements ELEV**. For the sake of argument, I grant that structuralism predicts the correct implementation condition for ELEV**. Under that assumption, any physical system that implements ELEV also implements a model for which structuralism predicts the correct implementation condition.

INSERT TABLE 4 ABOUT HERE

Can we generalize the foregoing procedure to all other computational models? Perhaps. It seems plausible that, given any computational model, there is a computational model with an identical formal structure that ignores all non-functional aspects of machine states. So the following principle seems plausible:

(3) *If a physical system implements a computational model, then it implements a computational model for which structuralism predicts the correct implementation condition.*

I do not endorse this principle. But I concede for the sake of argument that it is true. The key point is that my concession provides no support for structuralism. A successful theory of computational implementation must yield the correct result for *all* computational models, including *all* FSMs. Structuralism yields the wrong implementation condition for ELEV, along with numerous more sophisticated examples that figure prominently within contemporary scientific practice.

§4.4 Semantic ambiguity?

Structuralists may greet my argument by suggesting that the phrase "computational implementation" is *semantically ambiguous*. They can concede that I have isolated one viable concept of implementation, but they will insist that the phrase "implementation" sometimes expresses a concept more along structuralist lines. After all, computation figures in diverse scientific fields: mathematical logic, computer science, artificial intelligence, embedded systems design, robotics, cognitive psychology, neuroscience, and so on. These fields are so disparate in aims and methodology that it would not be too surprising if they employed the phrase "computational implementation" to express different concepts. For example, one might suggest that roboticists typically employ the phrase "implementation" in accord with my descriptivist theory but that computer scientists and cognitive scientists typically presuppose a structuralist approach.

I doubt that there is a compelling reason to postulate ambiguity along these lines. As emphasized in §4.3, my view assigns structuralist implementation conditions to numerous computational models. If you want to describe a physical system in structuralist terms, my view lets you do so. You need simply delineate a model (such as ELEV**) with a structuralist implementation condition. Positing semantic ambiguity is unnecessary. We can retain any benefits of the structuralist paradigm while preserving a univocal meaning for "implementation." Why say that the meaning of "implementation" shifts with context, when we can instead say that researchers in certain contexts care mainly about computational models whose implementation conditions are structuralist?

For present purposes, we may set aside debates about the meaning of the term "implementation." Let us simply grant that my account isolates one notable relation between computational models and physical systems (call it *implementation*$_1$) while structuralist theories isolate a distinct notable relation (call it *implementation*$_2$). Let us also grant that scientists sometimes use the phrase "computational implementation" to signify implementation$_2$ rather than implementation$_1$. Finally, let us grant that implementation$_2$ plays an important role in certain explanatory contexts.[9] What I insist is that implementation$_1$ *also* plays a vital role in many explanatory contexts. Specifically, I argued above that implementation$_1$ pervades two sciences directly concerned with physical computation: robotics and embedded systems design. Hence, my account elucidates a pivotal notion of *computational implementation* overlooked in previous philosophical writings. Even if a complete treatment of physical computation assigns central

[9] Chalmers (2011) proposes his structuralist theory of computational implementation as a philosophical foundation for cognitive science. His proposal is an example of *functionalism* about the mind, first espoused by Putnam (1975). If Chalmers's proposal were correct, then it would vindicate the centrality of implementation$_2$ to cognitive science. However, Burge (2007, pp. 376-377) argues that functionalism is "far removed from any explanation that goes on in science." In (Rescorla, 2012), I critique Chalmers's structuralist approach to mental computation. I argue that Chalmers's approach finds no basis within contemporary cognitive science.

importance to implementation$_2$, it should *also* assign central importance to implementation$_1$. Something like my descriptivist theory must figure as a prominent component within any adequate philosophical foundation for physical computation.

In what follows, I further illustrate the theoretical utility of implementation$_1$ by considering two vexed topics: the relation between computation and representation (§5); and Putnam-Searle triviality arguments (§6).

§5. Representation and implementation

Some philosophers hold that a physical system implements a computation only if the system has representational properties (Dietrich, 1989), (Fodor, 1998, p. 10), (Shagrir, 2006), (Sprevak, 2010). In Ladyman's words, "for physical states to count as computational states they must be genuinely representational" (2009, p. 382). Call this *the semantic view of computational implementation*. On the semantic view, *all* physical computational systems have semantic or representational properties.

I reject the semantic view of computational implementation. The FSMs discussed above are all counterexamples to it: ELEV, ELEV*, ELEV**, VEND, and CYCLE. In each case, representational properties do not inform the FSM's implementation condition. For example, a physical system can implement ELEV even if its states lack any representational content. An ordinary physical elevator might implement ELEV, but most philosophers would deny that the elevator's states have representational properties. ELEV's implementation condition does not seem to involve meaning, representational content, or "aboutness."

On the other hand, many computational models have implementation conditions that involve representational properties.

To illustrate, consider a specific register machine discussed in the classic computer science textbook (Abelson *et al.*, 1996, pp. 492-498). The machine, which formalizes the Euclidean algorithm, has three registers: `a`, `b`, and `t`. The machine works as follows (p. 493):

> The basic operations required are testing whether the contents of register `b` is zero and computing the remainder of the contents of register `a` divided by the contents of register `b`… On each cycle of the GCD algorithm the contents of register `a` must be replaced by the contents of register `b`, and the contents of `b` must be replaced by the remainder of the old contents of `a` divided by the old contents of `b`… To accomplish the replacements, our machine will use a third "temporary" register, which we call `t`. (First the remainder will be placed in `t`, then the contents of `b` will be placed in `a`, and finally the remainder stored in `t` will be placed in `b`.)

Abelson *et al.* formalize this description through the following program (p. 497):

```
(controller
   test-b
      (test (op =) (reg b) (const 0))
      (branch (label gcd-done))
      (assign t (op rem) (reg a) (reg b))
      (assign a (reg b))
      (assign b (reg t))
      (goto (label test-b))
   gcd-done)
```

where `rem` expresses the remainder operation.

What implementation condition does descriptivism predict for the Euclidean algorithm register machine? To answer this question, we must decide whether to handle "memory registers" in *physical* or *virtual* terms. As I suggested in §3.1, standard computational practice leaves this decision indeterminate. Certainly, Abelson *et al.* take no stand either way. For the sake of simplicity, I choose a physical construal. Thus, an implementing system must have three distinct physical parts *a*, *b*, and *t* --- corresponding to registers `a`, `b`, and `t` --- whose states evolve in accord with mechanical instructions encoded by the machine's program.

Crucially, those instructions individuate register states in representational terms. For example, Abelson *et al.* write that "[t]he machine has an instruction that computes the remainder of the contents of register `a` and `b` and assigns the result to register `t`" (p. 499). This instruction governs the evolution of machine states *as characterized representationally*. A physical system can execute the instruction only if the system can compute the remainder of one number divided by another. To do that, the system must represent natural numbers. An implementing system must contain three physical locations *a*, *b*, and *t*, each of which can represent natural numbers. The representational properties of *a*, *b*, and *t* must evolve in accord with the above program. Hence, a physical system implements the Euclidean algorithm register machine only if the system can represent natural numbers and perform arithmetical operations.[10]

In that respect, the Euclidean algorithm register machine resembles many other register machines. Indeed, the first register machine in the published literature encodes the instructions "add 1 to the number in register *n*" and "subtract 1 from the number in register *n*" (Shepherdson

[10] The register machine formalism assumes that each memory register can represent any natural number of arbitrary size. As already noted, infinite discrete memory capacity may be physically impossible. If so, then the Euclidean algorithm register machine is not physically realizable. We can circumvent this issue by employing a modified register machine formalism that assumes large but finite memory capacity. (To a first approximation, modern computers are physical realizations of such a formalism.) Programs couched within the modified register machine formalism can still individuate register states through representational relations to (sufficiently small) numbers.

and Sturgis, 1961, p. 219). Executing these instructions requires performing arithmetical operations, which requires a capacity to represent natural numbers. Contemporary CS features numerous additional computational models that individuate computational states in representational terms (Rescorla, forthcoming). In each case, my descriptivist theory yields an implementation condition that crucially involves representational properties.

Overall, then, my descriptivist theory offers an attractive treatment of the relation between computation and representation. A computational model encodes mechanical instructions, which may or may not cite representational properties. Whether a model's implementation condition involves representational properties depends on whether the encoded instructions cite representational properties. If the instructions describe computational states in representational terms (as with the Euclidean algorithm register machine), then representational properties figures in model's implementation condition. If the instructions describe computational states in non-representational terms (as with ELEV), then representational properties do not figure in the model's implementation condition. Thus, representation informs the implementation conditions for *some but not all* computational models.

Structuralism offers a much less satisfying treatment of the relation between computation and representation. In particular, structuralism has difficulty handling models whose implementation conditions *as standardly conceived within current CS* involve representation. For example, the Euclidean algorithm register machine *as described by Abelson et al.* encodes mechanical instructions that individuate computational states through their representational properties. To implement this register machine, a physical system must conform to the encoded representationally-specified instructions. But a physical system can instantiate the causal

structure dictated by the Euclidean algorithm register machine without conforming to the encoded instructions.

To illustrate, imagine two physical machines R_{10} and R_{13} that have the same local, intrinsic physical properties. R_{10} is employed by a society that uses base-10 notation. R_{13} is employed by a society that uses base-13 notation. For example, the numeral "`115`" denotes the number 115 when used by R_{10} but the number 187 when used by R_{13}. We may stipulate that R_{10} implements the Euclidean algorithm register machine.[11] If R_{10} has numerals "`115`" and "`20`" stored in registers *a* and *b*, then it places numeral "`15`" in register *t*, thereby calculating the remainder of the corresponding numbers. R_{13} performs the same syntactic manipulations, but it does not thereby calculate the desired remainder. The base-13 denotation of "`15`" (namely, the number 18) is not the remainder of the base-13 denotation of "115" (namely, the number 187) divided by the base-13 denotation of "`20`" (namely, the number 26). Thus, R_{13} does not conform to mechanical instructions encoded by the Euclidean algorithm register machine. Yet R_{10} and R_{13} have the same "causal structure isomorphism type," since they are intrinsic physical duplicates. So R_{13} instantiates the causal structure dictated by the Euclidean algorithm register machine without implementing the Euclidean algorithm register machine.

The problem here is quite general. Computational models often encode representationally-specified instructions (e.g. *add 1 to the number in register n*). In most cases, instantiating an appropriate causal structure does not ensure that a physical system conforms to these instructions. In any such case, the implementation condition predicted by structuralism diverges from the implementation condition postulated within current CS.

[11] More cautiously, stipulate that R_{10} implements an analogue to the Euclidean algorithm register machine, couched within the modified formalism from note 10. I henceforth ignore this caveat, since it does not affect my argument.

Structuralists might respond to this argument in various ways. In particular, they may recommend that we jettison all talk about register machines representing numbers. When computer scientists indulge in such talk, aren't they really just trying to specify syntactic operations over numerals? Shouldn't we replace the representationally-specified instruction *add 1* with the syntactically-specified instruction *replace decimal numeral "m" with decimal numeral "m+1"*? Why not reinterpret all representational talk in purely syntactic terms?

The syntactic reinterpretation strategy is *revisionary* regarding current CS. The representationally-specified instruction *add 1* and the syntactically-specified instruction *replace decimal numeral "m" with decimal numeral "m+1"* individuate computational states in fundamentally different ways. The former instruction ignores the specific notation through which a system represents numbers, while the latter mentions a particular numerical notation. Thus, the syntactic reinterpretation strategy mandates drastic revisions to numerous computational models that figure prominently within current CS. Any such revisionary proposal faces a high burden of proof. Scientific practice is not sacrosanct, but we should revise it only in light of an excellent argument. In (Rescorla, forthcoming), I critique the syntactic reinterpretation strategy. I argue that there is no sound basis for revising how CS handles the Euclidean algorithm register machine, or other computational models that individuate computational states representationally.[12]

For present purposes, what matters is simply that my descriptivist theory coheres much better with current CS practice than structuralism does. From a descriptivist perspective, there is no pressure to reinterpret the Euclidean algorithm register machine in purely syntactic, non-representational terms. More generally, my theory is perfectly hospitable to models that describe

[12] In (Rescorla, forthcoming), I rebut numerous additional possible structuralist rejoinders to the Euclidean algorithm register machine counterexample. I also argue that such cases militate against bounded structuralism.

computational states representationally. Using my theory, we can effortlessly accommodate a range of computational models that figure prominently within current CS. In contrast, structuralists appear pressured towards the syntactic reinterpretation strategy. Structuralism seems able to accommodate the Euclidean algorithm register machine, along with other important computational models, only by revising current scientific practice.

Let us rephrase these conclusions using §4.4's distinction between implementation$_1$ and implementation$_2$. CS studies numerous computational models that describe representational properties of computational states. By invoking implementation$_1$, we can preserve the implementation conditions ascribed to those models within current CS. In contrast, implementation$_2$ does not mesh well with the standard CS treatment of such models. This contrast provides strong *prima facie* evidence that implementation$_1$ should figure prominently in any adequate philosophical foundation for CS *as it currently stands*.

§6. Triviality arguments

Philosophers sometimes claim that the physical realization relation is *trivial*. For instance, Searle (1990) holds that every physical system implements every computational model. Putnam (1988, pp. 121-125) defends two less extreme triviality doctrines:

- An FSM that lacks inputs and outputs is implemented by every physical system that satisfies certain weak physical constraints.
- An FSM with inputs and outputs is implemented by every physical system that satisfies certain weak physical constraints and that exhibits appropriate input/output behavior.

Putnam-Searle triviality arguments assume some form of structuralism or bounded structuralism: implementation requires only an isomorphism between the model's formal structure and the

physical system's causal structure (perhaps constrained by input/output restrictions). Isomorphic mappings come cheap. Thus, we can institute an isomorphic mapping between a formal model and numerous inappropriate physical systems. As Searle (1990, p. 27) puts it, "the wall behind my back is right now implementing the Wordstar program, because there is some pattern of molecule movements that is isomorphic with the formal structure of Wordstar."

Various philosophers try to defuse Putnam-Searle triviality arguments. They typically follow Putnam and Searle in assuming that something like structuralism or bounded structuralism is correct. They seek to avoid triviality by specifying a suitable restriction $\Phi(F)$ upon the mapping F from formal states to physical states. A few highlights:

(a) Putnam develops structuralism using *material* conditionals. Since material conditionals are so weak, Putnam's approach makes it quite easy to mount a triviality argument. Chalmers (1995, 1996) and Copeland (1996) urge that we should instead employ *counterfactual* conditionals. Once we replace material conditionals with counterfactual conditionals, we can easily rebut the specific triviality argument advanced by Putnam.

(b) Chalmers (1996) holds that sufficiently sophisticated triviality arguments succeed for FSMs, even after we replace material conditionals with counterfactual conditions. But he holds that triviality arguments fail for computational formalisms (such as digital circuits) whose internal states have combinatorial structure.

(c) Chalmers's version of structuralism allows the isomorphic mapping F to carry formal states into arbitrary physical state-types, where the physical state-types may reflect deviant "groupings" of physical states. Godfrey-Smith (2009) and Scheutz (2001) argue that this feature of Chalmers's theory generates a severe triviality problem. Along with Copeland (1996), they recommend that we avoid triviality worries by demanding that F

carry formal states to physical state-types that are "natural" rather than "gerrymandered." Chalmers (2012) tentatively agrees that this recommendation has attractive features.

(d) Chalmers (1996, 2012) contemplates imposing a "uniformity" constraint along the following lines: each formal state-transition of the computational model corresponds to a single uniform causal mechanism in the physical system.

By combining these and other ideas, most structuralists hope to avoid Putnam-Searle triviality arguments. See (Chalmers, 2012) for a state-of-the-art review.

I think that ideas along the foregoing lines should inform any complete response to Putnam-Searle triviality arguments. Descriptivism can readily incorporate all these ideas:

(a) My theory, as presented in §3, emphasizes counterfactual conditionals over material conditionals. So my theory easily evades Putnam's specific triviality argument.

(b) My theory can emphasize that various computational models, such as ELEV-CIR, have combinatorial structure that tightly constrains physical implementation. As Chalmers (1996) argues, combinatorial constraints already suffice to evade numerous putative trivializing implementations.

(c) My approach can impose a "naturalness" constraint upon physical state-types. We could attach this constraint as a supplementary clause to the theory presented in §3. But a better strategy is to preserve my theory as it currently stands, securing a role for "naturalness" by citing descriptive usage of computational models. For example, one can say that normal descriptive usage of ELEV** already embodies a "naturalness" constraint. The FSM *as used in our current descriptive practice* induces a state space containing functional states realizable only by "natural" physical states. On this view, only "natural" physical state-types can realize ELEV**'s formal states.

(d) My descriptivist approach can incorporate Chalmers's proposed "uniformity" constraint on state-transitions. Here a minor emendation is necessary. My theory as presented in §3 demands that appropriate counterfactuals be satisfied, but it does not demand a uniform causal mechanism underlying each counterfactual. However, one can supplement §3's statement with a "uniformity" clause like that suggested by Chalmers.

As (a)-(c) illustrate, my theory as it stands can accommodate diverse implementation constraints from the structuralist literature (including "combinatorial" and "naturalness" constraints). When my theory as it stands does not accommodate some proposed constraint (such as "uniformity"), one can usually append the desired constraint as an additional clause. Thus, my theory is both *general* and *flexible*: general enough to accommodate many widely discussed implementation constraints; flexible enough to admit supplementation by further implementation constraints. In this manner, my theory (or something close to it) can subsume all existing structuralist responses to Putnam-Searle triviality arguments, including all the responses surveyed in (Chalmers, 2012).

My descriptivist theory also provides further ammunition against triviality arguments.

According to my theory, a physical system implements a computational model only if the system can instantiate states belonging to the state space description induced by the model. For many computational models, the relevant states are individuated in highly specific fashion (e.g *being located on the first floor* or *storing a numeral that represents the number 2 in a certain physical memory location*). A system implements the model only if it can instantiate *those specific states*. For instance, a system implements ELEV only if the system can travel between the first and second floors of some building and only if it has a door that can open and close. For that reason, only very special physical systems implement ELEV. Searle's wall does not. Even the machine from the parable does not. There is nothing trivial about the implementation

condition for ELEV. A similar diagnosis applies to any computational model, such as the Euclidean algorithm register machine, that individuates computational states representationally. To implement the Euclidean algorithm register machine, a physical system must bear appropriate representational relations to natural numbers. Most physical systems do not have representational properties, so most physical systems do not implement the Euclidean algorithm register machine. In this manner, my descriptivist theory easily defuses triviality arguments *as applied to a wide range of computational models*, including numerous models that figure prominently within contemporary CS, robotics, and embedded systems design.

As I conceded in §4.3, many computational models individuate computational states through functional role. Implementation conditions for these models seem to accord at least roughly with structuralism or bounded structuralism. The ideas canvassed in the previous paragraph do not help combat triviality arguments *as applied to those models*. One must instead invoke additional implementation constraints, such as (a)-(d).

I have not offered a definitive resolution of all possible triviality worries. No one, including Chalmers, has offered anything along those lines. But I have indicated how my approach can incorporate all constraints upon physical implementation proposed in the structuralist literature. In that sense, my approach is *no worse positioned than existing structuralist theories* to address triviality worries. Moreover, there is one important respect in which my approach is *better positioned than structuralism* to handle such worries: my theory readily entails that ELEV, the Euclidean algorithm register machine, and many other notable computational models have highly non-trivial implementation conditions. So my approach

enables substantial progress regarding triviality arguments, even if it does not conclusively refute all such arguments.[13]

Let us rephrase these conclusions using §4.4's distinction between implementation$_1$ and implementation$_2$. The current philosophical literature offers many compelling ideas for combatting Putnam-Searle triviality arguments *as applied to implementation$_2$*. We can preserve those ideas if we replace implementation$_2$ with implementation$_1$. Moreover, switching to implementation$_1$ allows us to adduce further non-trivializing constraints *for a large but non-exhaustive class of computational models*. Thus, implementation$_1$ resists Putnam-Searle triviality arguments even better than implementation$_2$ does.

§7. Is reductionism mandatory?

I have presented a novel theory of computational implementation, and I have argued that my theory offers important advantages over prior theories.[14] I have not attempted to provide a non-circular reductive analysis of the implementation relation. I doubt that any such analysis is possible. I no more expect a non-circular analysis of what it is to implement a computation than I expect a non-circular analysis of what it is to follow a recipe. In particular, I doubt that one can offer a reductive analysis of the relation *computational model M induces state space description*

[13] Chalmers (2011, 2012) holds that every physical system implements a trivial FSM containing a single abstract machine state. On this basis, he endorses *pancomputationalism*, i.e. the thesis that every physical system implements a computational model. He notes that the resulting pancomputationalist position does not trivialize computational implementation, so long as we deny that every physical system realizes every computational model. Chalmers's discussion of these points strikes me as quite convincing.

[14] See (Piccinini, 2010) for a survey of prior theories. The positions that Piccinini calls *the simple mapping account*, *the causal account*, *the counterfactual account*, and *the dispositional account* are versions of what I call *structuralism* and *bounded structuralism*. The position that Piccinini calls *the semantic account* entails the position that I call *the semantic view*. The position that Piccinini calls *the syntactic account* does not allow representational properties to inform implementation conditions. As I argued in §5, no such theory can accommodate the Euclidean algorithm register machine, among numerous additional computational models that figure prominently within contemporary CS. The only remaining position discussed by Piccinini is the one that he himself espouses, which he calls *the mechanistic account*. Certain aspects of the mechanistic account are congenial to descriptivism. However, Piccinini's version of the mechanistic account does not allow representational properties to inform implementation conditions (Piccinini, 2008).

$<S, I, \Omega, s_0>$. We must simply rely on scientific practice and common sense, which often dictate with tolerable determinacy the state space description induced by a model.

I do not need to argue here that a non-circular reduction is impossible. What matters is that I have not, in fact, offered one. Is the lacuna a problem for me?

Chalmers suggests that, lacking a non-circular theory of computational implementation, "the foundational role of computation in cognitive science cannot be justified" (1995, p. 391). The idea seems to be that cognitive science is entitled to talk about "computational implementation" only given a non-circular analysis. Similar sentiments recur frequently throughout the literature, either explicitly or implicitly.

I see no reason to agree. Science does not usually offer non-circular elucidations of its key notions. Just try to offer a reductive analysis of *force*, *gravity*, *virus*, *interest rate*, or virtually any other important scientific notion. Why hold sciences that study physical computation to a higher standard? Computer science, embedded systems design, robotics, and cognitive psychology routinely invoke the implementation relation without presupposing a reductive definition. I submit that this methodology is entirely legitimate. Of course, we want *some* elucidation of the physical realization relation. But that elucidation need not take the form of a reductive analysis. My discussion elucidates computational implementation without attempting a reductive analysis.[15]

Pursuit of reductive analysis has exerted a distorting influence on the philosophical literature. It has encouraged reflection on abstract set-theoretic models of computation detached from their pragmatic and explanatory moorings. It has deflected attention from the descriptive

[15] Chalmers's own theory is less reductive than it may initially appear. As discussed in note 7, Chalmers specifies a computational model's implementation condition by translating the model into a suitable CSA. Chalmers says nothing to elucidate the "translation" relation between computational models and CSAs. So the "translation" relation plays an unreduced role within Chalmers's account similar to the unreduced role that the "inducement" relation plays within my account.

role that computational models occupy within contemporary scientific practice. By highlighting that role, I have tried to reveal computational modeling as a special case of what all empirical sciences pursue: accurate description of physical systems.

Acknowledgments

I am indebted to David Chalmers, Peter Godfrey-Smith, Gualtiero Piccinini, Eric Rescorla, Mark Sprevak, and two anonymous referees for helpful feedback that greatly improved this paper. My research was supported by a fellowship from the National Endowment for the Humanities. Any views, findings, conclusions, or recommendations expressed in this publication do not necessarily reflect those of the National Endowment for the Humanities.

Works Cited

Abelson, H. and Sussman, G. with Sussman, J. 1996. *The Structure and Interpretation of Computer Programs*. Cambridge: MIT Press.

Burge, T. 2007. *Foundations of Mind*. Oxford: Clarendon Press.

Bueno, O., and French, S. 2011. "How Theories Represent." *British Journal for the Philosophy of Science* 62: 857-894.

Chalmers, D. 1995. "On Implementing a Computation." *Minds and Machines* 4: 391-402.

---. 1996. "Does a Rock Implement Every Finite State Automaton?". *Synthese* 108: 309-333.

---. 2011. "A Computational Foundation for the Study of Cognition." *The Journal of Cognitive Science* 12: 323-357.

---. 2012. "The Varieties of Computation: A Reply." *The Journal of Cognitive Science* 13: 211-248.

Chrisley, R. 1994. "Why Everything Doesn't Realize Every Computation." *Minds and Machines* 4: 403-420.

Cleland, C. 2002. "On Effective Procedures." *Minds and Machines* 12: 159-179.

Copeland, J. 1996. "What is Computation?". *Synthese* 108: 335-359.

Dietrich, E. 1989. "Semantics and the Computational Paradigm in Cognitive Psychology." *Synthese* 79: 119-141.

Dresner, E. 2010. "Measurement-Theoretic Representation and Computation-Theoretic Realization." *The Journal of Philosophy* 107: 275-292.

Feferman, S. 2006. "Turing's Thesis." *Notices of the American Mathematical Society* 53: 2-8.

Fodor, J. 1998. *Concepts*. Oxford: Oxford University Press.

Godfrey-Smith, P. 2009. "Triviality Arguments Against Functionalism." *Philosophical Studies*

145: 273-295.
Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Menlo Park: Addison-Wesley.
Klein, C. 2008. "Dispositional Implementation Solves the Superfluous Structure Problem." *Synthese* 165: 141-153.
Knuth, D. 1968. *The Art of Computer Programming*, vol. 1, Reading: Addison-Wesley.
Ladyman, J. 2009. "What Does it Mean to Say that a Physical System Implements a Computation?". *Theoretical Computer Science* 410: 376-383.
Lewis, D. 1973. *Counterfactuals*. Malden: Blackwell.
Minsky, M. 1967. *Computation: Finite and Infinite Machines*. Englewood Cliffs: Prentice-Hall.
Montemerlo, M., et al. 2009. "Junior: The Stanford Entry in the Urban Challenge." In *The DARPA Urban Challenge*, eds. M. Buehler, K. Iagnemma, and S. Singh. Berlin: Springer-Verlag.
Mozgovoy, M. 2010. *Algorithms, Languages, Compilers, and Automata*. Sudbury: Jones and Bartlett.
Murphy, R. 2000. *Introduction to AI Robotics*. Cambridge: MIT Press.
Patterson, D., and Hennessy, J. 2005. *Computer Organization and Design*, 3rd. ed. New York: Elsevier.
Piccinini, G. 2008. "Computation Without Representation." *Philosophical Studies* 137: 205-241.
---. 2010. "Computation in Physical Systems." In *The Stanford Encyclopedia of Philosophy (Fall 2010 Edition)*, ed. E. Zalta.
Putnam, H. 1975. *Philosophical Papers*, vol. 2. Cambridge: Cambridge University Press.
---. 1988. *Representation and Reality*. Cambridge: MIT Press.
Rescorla, M. 2012. "How to Integrate Representation into Computational Modeling, and Why We Should." *The Journal of Cognitive Science* 13: 1-38.
---. Forthcoming. "Against Structuralist Theories of Computational Implementation." *British Journal for the Philosophy of Science*.
Scheutz, M. 2001. "Computational versus Causal Complexity." *Minds and Machines* 11: 544-566.
Searle, J. 1990. "Is the Brain a Digital Computer?". *Proceedings and Addresses of the American Philosophical Association* 64: 21-37.
Shagrir, O. 2006. "Why We View the Brain as a Computer." *Synthese* 153: 393-416.
Shepherdson, J. and Sturgis, H. E. 1963. "Computability of Recursive Functions." *Journal of the Association of Computing Machinery* 10: 217-255.
Sprevak, M. 2010. "Computation, Individuation, and the Received View on Representation." *Studies in History and Philosophy of Science* 41: 260-270.
Vahid, F. and Givargis, T. 2002. *Embedded System Design*. New York: Wiley.
van Fraassen, B. 2008. *Scientific Representation*. Oxford: Clarendon Press.
Wittgenstein, L. 1953. *Philosophical Investigations*, eds. G. E. M. Anscombe and R. Rhees, trans. G. E. M. Anscombe. Oxford: Blackwell.
Woodward, J. 2000. "Explanation and Invariance in the Special Sciences." *British Journal for the Philosophy of Science* 51: 197-254.

	cD	**oD**	**cU**	**oU**
button O	oD	oD	oU	oU
button C	cD	cD	cU	cU
button D	cD	oD	cD	oU
button U	cU	oD	cU	oU

Table 1. The machine table for finite state machine ELEV.

	cON	**oON**	**cOFF**	**oOFF**
button O	oON	oON	oOFF	oOFF
button C	cON	cON	cOFF	cOFF
button D	cON	oON	cON	oOFF
button U	cOFF	oON	cOFF	oOFF

Table 2. The machine table for finite state machine ELEV*.

	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$
$\mathbf{I_1}$	S_2, O_1	S_3, O_1	S_1, O_2
$\mathbf{I_2}$	S_3, O_1	S_1, O_2	S_1, O_3

Table 3. The machine table for finite state machine VEND.

	S_1	**S_2**	**S_3**	**S_4**
I_1	S_2	S_2	S_4	S_4
I_2	S_1	S_1	S_3	S_3
I_3	S_1	S_2	S_1	S_4
I_4	S_3	S_2	S_3	S_4

Table 4. The machine table for finite state machine ELEV**.