Against Structuralist Theories of Computational Implementation

Michael Rescorla

Abstract: Under what conditions does a physical system *implement* or *realize* a computation? *Structuralism about computational implementation*, espoused by Chalmers and others, holds that a physical system realizes a computation just in the case the system instantiates a pattern of causal organization isomorphic to the computation's formal structure. I argue against structuralism through counter-examples drawn from computer science. On my opposing view, computational implementation sometimes requires instantiating semantic properties that outstrip any relevant pattern of causal organization. In developing my argument, I defend *anti-individualism about computational implementation*: relations to the social environment sometimes help determine whether a physical system realizes a computation.

1 The physical realization relation

What is it for a physical system to *implement* or *realize* a computation? The question is foundational for computer science (CS), robotics, Artificial Intelligence (AI), and cognitive science. All four disciplines emphasize abstract computational models, such as Turing machines or finite state automata. All four disciplines hold that, in certain circumstances, a physical system such as a brain or desktop computer can implement an abstract computational model.

According to many philosophers, implementing a computation involves instantiating an appropriate pattern of causal interaction among internal states. As Chalmers ([1995], p. 392) puts it, '[a] physical system implements a given computation when the causal structure of the physical system mirrors the formal structure of the computation.' I will call this view *structuralism about computational implementation*.[1] Besides Chalmers, proponents of structuralism (or closely related doctrines) include Copeland ([1996]), Dresner ([2010]), Egan ([1992]), Godfrey-Smith ([2009]), Kim ([2005], pp. 130-132), and Scheutz ([2001]).

According to structuralism, a physical system realizes a computation just in case there is an 'isomorphism' between the computation's formal structure and the physical system's causal structure. Chalmers ([1995], p. 392) clarifies the relevant notion of 'isomorphism' as follows:

> 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.

Chalmers later offers an emended version of structuralism, but the emendations do not matter for us. I want to address the intuitive idea behind structuralism, not the specifics of how Chalmers or anyone else develops it. Chalmers ([1995], p. 401) formulates the intuitive idea as follows:

> 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.

On this view, a physical system implements a computation if it instantiates some relevant 'pattern of causal organization.' The computation imposes no constraints on physical systems that implement it, save that they exhibit the desired pattern. What matters is the pattern, viewed as an 'isomorphism type.' Of course, a physical system's states have many properties, such as shapes and colors, that outstrip any causal pattern dictated by the computation. According to structuralism, such properties do not inform whether the system implements the computation. What matters are the causal patterns, not the particular states composing the patterns.

I will argue that structuralism predicts incorrect implementation conditions for some, though perhaps not all, computations. I concede that an appropriate pattern of causal organization is necessary for implementing a computation. I deny that it is sufficient. My argument hinges upon the relation between computation and *representation.* Intuitively, a physical system *represents* some subject matter just in case its states are *about* that subject matter. Following standard philosophical usage, I say that such a system has *semantic* properties. I will argue that certain computational models individuate computational states through their semantic properties. A physical system implements such a model only if it has appropriate semantic properties. These ‘appropriate semantic properties’ do not supervene upon any relevant pattern of causal organization. Thus, instantiating the specified pattern of causal organization does not suffice to implement the model.

§2 reviews some relevant philosophical literature. §§3-5 present my basic anti-structuralist argument. §6 extends the argument to a weakened variant of structuralism. §7 addresses Putnam-Searle *trivial arguments*. §8 connects my discussion with Burge’s *anti-individualism about mental content*.

2 Semantics and computational implementation

Some philosophers hold that a physical system implements a computation only if the system has representational properties (Crane [1990]; Fodor [1998], p. 10; Ladyman [2009b]; Sprevak [2010]). In Ladyman’s words, ‘for physical states to count as computational states they must be genuinely representational’ ([2009b], p. 382). Similarly, Sprevak ([2010], p. 260) claims that ‘appeal to representational content is inescapable when attributing computations to physical systems.’ Call this *the semantic view of computational implementation*. On the semantic view, *all* physical computational systems have semantic or representational properties.

At the opposite extreme, philosophers such as Chalmers ([1995]), Egan ([1992]), and Piccinini ([2008]) deny that semantics *ever* informs computational implementation. Call this *the non-semantic view of computational implementation*. As Chalmers ([1995], p. 399) puts it, ‘when computer designers ensure that their machines implement the programs that they are supposed to, they do this by ensuring that the mechanisms have the right causal organization; they are not concerned with semantic content.’ Proponents of the non-semantic view can acknowledge that certain computational systems *have* semantic properties. They merely deny that we should cite semantics or representation when specifying what it is for a physical system to implement a given computation.

I reject both the semantic and the non-semantic views of computational implementation. Many computational models are implementable by physical systems that lack representational properties. But *other* computational models have implementation conditions that essentially involve representation. My position steers a middle course between two extreme views: that implementation conditions *always* involve semantics, and that implementation conditions *never* involve semantics. On my position, the implementation conditions for *some but not all* computational models essentially involve semantic properties.

As a potential counter-example to the semantic view, consider a simple finite state vending machine discussed by Godfrey-Smith ([2009]). The machine has 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 internal states S_1, S_2, and S_3, governed by the following transition table:

INSERT TABLE 1 HERE

Call this machine 'VEND.' The implementation condition for VEND does not seem to involve meaning, representational content, or 'aboutness.' A physical system can implement VEND even if its states lack any semantic interpretation. Of course, one might *impose* representational talk upon the system. For instance, one might say that a system entering into state S_2 thereby 'represents' that 5 cents more are required for a Coke. At best, such representational attributions reflect a Dennettian 'stance' towards the system (Dennett [1987]), not a genuine constraint the system must satisfy to implement VEND. Nothing about VEND itself seems to require that we attribute representational import to states S_1, S_2, and S_3. Nothing about VEND's transition table assigns any essential role to semantics, representation, or content.

In what follows, I focus on presenting counter-examples to the non-semantic view. I will highlight the implementation conditions attributed by computer scientists to specific computations, and I will argue that those implementation conditions feature semantic properties in an essential way. Furthermore, I will argue that the relevant semantic properties outstrip any pattern of causal organization specified by the computation. Thus, the counter-examples are also counter-examples to structuralism. My heavy emphasis upon contemporary CS is a distinctive feature of my methodology. No previous discussion so thoroughly highlights how structuralism conflicts with entrenched CS practice.

3 Conforming to instructions

Computational implementation is grounded in a more general notion: *conforming to instructions*. Many activities are governed by instructions: following a recipe, performing a musical composition, and so on. Conforming to instructions requires *doing what the instructions say*. For instance, if a recipe instructs me to add salt, then conforming to the recipe requires that I add salt.[2] Computation falls under special instructions that are 'mechanical,' 'effective,' or 'algorithmic.' As Knuth ([1968], p. 6) emphasizes, the instructions are precise and mindless, in contrast with the instructions found in recipes ('sauté until the chicken is nicely browned'). A physical system implements a computation only if it reliably conforms to the relevant instructions. Conforming to the instructions requires *doing what the instructions say*, which requires transiting appropriately between states with the properties mentioned by the instructions.

Consider an informal example. The oldest known algorithm, from Euclid's *Elements*, describes how to compute the greatest common divisor of m and n, where $n \leq m$. Here is Knuth's formulation ([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.

where '"$m \leftarrow n$" means the value of variable m is to be replaced by the current value of variable n.' To execute the Euclidean algorithm, I must conform to instructions E1-E3 in the proper order. How do I conform to the instruction 'divide m by n'? I must represent n and m, either in my thought or else through a symbolic representation, such as pencil marks on the page. I then perform the arithmetical operation of division upon those numbers. Thus, executing the Euclidean algorithm requires representing numbers and performing arithmetical operations on those numbers. Conforming to the algorithm's component instructions requires bearing appropriate semantic relations to the natural numbers.

Following Peacocke ([1994]), I say that an instruction is *content-involving* just in case it individuates states partly through their representational properties. A content-involving instruction is specified, at least partly, in semantic or representational terms. On this usage, the instructions that compose the Euclidean algorithm are content-involving. The instructions taxonomize computational states through representational relations to natural numbers.

In what follows, I will extend the foregoing analysis to the rigorous computational formalisms employed by computer scientists. A good theory of computational implementation should address both *programming languages* (such as C++, LISP, or Prolog) and *machine models* (such as the Turing machine or the register machine). I will discuss programming languages in §4 and machine models in §5. In both cases, I will offer content-involving counter-examples to structuralism.[3]

4 Implementing a computer program

A programming language has no inherent meaning in itself. However, programmers associate a programming language with an *intended interpretation*, according to which a program contains instructions to perform certain tasks. As McCarthy puts it, '[p]rograms are symbolic expressions representing procedures' ([1962], p. 21). A programming language is not simply an uninterpreted formal calculus. It is a *language* whose expressions are *meaningful* instructions to which a physical machine may or may not conform. It makes no sense to discuss implementation of an uninterpreted computer program, but it makes perfect sense to discuss implementation of a computer program *interpreted according to its intended meaning*. By analogy, the English words that compose a recipe have no inherent meaning. Nevertheless, our linguistic conventions associate the recipe with a standard interpretation. Under this interpretation, the recipe issues instructions, to which a human chef may or may not conform. When elucidating what it is to execute some recipe, we ask what instructions the recipe issues, *as understood with its intended meaning*. When elucidating what it is to execute some computer program, we ask what instructions the program issues, *as understood with its intended meaning*.

In this spirit, consider how a leading CS textbook renders the Euclidean algorithm within Scheme, a dialect of LISP (Abelson and Sussman [1996], p. 49):

```
define (gcd a b)
    (if (= b 0)
         a
         (gcd b (remainder a b)))
```

where ‘`remainder`’ is taken as primitive. To initiate a computation, we apply the Scheme program to particular numerical inputs. For example, we can supply a physical system that implements the Scheme program with input

```
(gcd 115 20)
```

in which case the system eventually yields output

```
5
```

Under what conditions does a physical system implement the Scheme program? Abelson and Sussman (p. 493) write that

> [a] machine to carry out this algorithm must keep track of two numbers, *a* and *b*, so let us assume that these numbers are stored in two registers with those names. 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 they contents of register `b`.

According to Abelson and Sussman, a machine implements the Scheme program only if it can compute the remainder of one number divided by another. To do that, the machine must represent numbers. Thus, the Scheme program contains content-involving instructions, to which a physical system conforms only if the system represents natural numbers.

The Scheme program is a counter-example to the non-semantic view of computational implementation. Computer designers who seek to implement the Scheme program must ensure that their machine executes suitable arithmetical operations and hence that it bears suitable representational relations to numbers. To ensure that a physical machine represents natural numbers, one can employ numerals that already have standardized semantic properties through their role in linguistic practice. Alternatively, one can employ new symbols and explicitly stipulate that those symbols have desired semantic properties. Thus, computer designers can secure desired semantic properties either by implicit appeal to linguistic convention or else by overt stipulation. So it is easy for human users to ensure that a physical machine bears suitable semantic relations to numbers. This may explain why Chalmers overlooks semantic constraints when analyzing computational implementation. The ease of fulfilling semantic constraints should not mislead us into dismissing the constraints as inessential.

To deploy these observations against structuralism, consider two physical machines M_{10} and M_{13} that have the same local, intrinsic physical properties. M_{10} is employed by a society that uses base-10 notation. M_{13} is employed by a society that uses base-13 notation. We may stipulate that M_{10} executes our Scheme program. This stipulation is legitimate, because our own society can easily build such a machine. M_{10} and M_{13} have the same 'causal structure isomorphism type,' since they are intrinsic physical duplicates. Nevertheless, M_{13} does not execute our Scheme program. Not, at least, if Abelson and Sussman correctly state what it takes to execute the program. When M_{10} converts input numerals '`115`' and '`20`' into the output '`5`', it thereby calculates the greatest common divisor of the corresponding numbers. When M_{13} performs the same syntactic manipulations, it does not thereby calculate the greatest common divisor of the corresponding numbers. The base-13 denotation of '`5`' (namely, the number 5) is not a divisor of the base-13 denotation of '`20`' (namely, the number 26). M_{13} does not execute the correct arithmetical operations, so it does not execute the Scheme program.

To explain why M_{10} executes the Scheme program while M_{13} does not, we must cite semantic properties of machines states. States of M_{10} have base-10 semantic interpretation, while states of M_{13} base-13 semantic interpretation. This difference does not reduce to relevant structural facts about the respective machines. The *patterns* of causal interaction are the same in both cases. What differs are not the *patterns* but the *states* composing the patterns. A machine implements the Scheme program only if it instantiates various states (*representing the number m*) and performs various operations (*dividing m by n*) whose natures outstrip any relevant pattern of causal organization. Thus, M_{13} is a counter-example to structuralism.

The issue here is not whether semantic properties such as *representing the number m* are ‘non-structural’ in some metaphysically absolute sense. Perhaps semantic properties ultimately reduce to *some* physical system’s ‘causal structure isomorphism type.’ For instance, the causal organization of the surrounding human linguistic community may suffice to confer determinate semantic properties on M_{10}’s computational states. My point is that representational properties do not reduce to any causal pattern *specified by the Scheme program*. The program’s formal structure does not even begin to fix a unique semantic interpretation. Implementing the program requires more than instantiating a causal structure that mirrors relevant formal structure.

Some readers may worry that the keyboard for M_{13} should have three extra keys, to serve as extra primitive digits for base-13 notation. In that case, M_{10} and M_{13} would not be intrinsic physical duplicates. However, we can avoid this worry in various ways. We might stipulate that M_{13} does not have three extra keys, perhaps due to a mishap at the factory, or the whims of an eccentric inventor. In such a scenario, M_{13} does not represent all the natural numbers, so members of the surrounding linguistic community will not find it useful for arithmetical calculation. Nevertheless, it executes the same syntactic manipulations as M_{10}. Alternatively, we might stipulate that M_{10} is built with three functionally redundant extra digits on the keyboard. Finally, we might change the example by comparing base-10 not with base-13 but with some alternative mapping from Arabic numerals to numbers, which would again ensure that the doppelganger machine does not execute our Scheme program.[4] Given the foregoing options, we may safely ignore any worries about missing extra keys on M_{13}’s keyboard.

I have developed my argument with respect to a single counter-example. However, we could extend the argument to numerous other programs formulated in Scheme or another programming language. CS features content-involving programs for computing square roots, factorials, prime factorizations, and so on (Abelson and Sussman [1996], pp. 31-90, pp. 384-385). We could apply the above argumentative strategy to any such program.

But how can executing a computer program require having certain representational properties, when the program itself is just a meaningless string of signs? Aren't M_{10} and M_{13} programmed with the same Scheme instructions? How can there be any significant difference between the computations executed by M_{10} and M_{13}?

I respond that there is more to a program than meaningless signs. The signs have an intended interpretation as instructions to do something, just as a recipe has an intended interpretation as instructions to do something. The intended meaning of 'add salt' is that one should add salt. The intended meaning of our Scheme program is that the machine should perform certain arithmetical operations. M_{10} and M_{13} are programmed with the same strings, but those strings contain numerals that have different meanings within the respective linguistic practices in which M_{10} and M_{13} are embedded, so the strings have different intended meanings. Despite syntactic overlap, M_{10} and M_{13} do not execute the same instructions. To claim otherwise suggests a use-mention confusion between the numerals composing a programming language and the numbers denoted by those numerals (as used by the surrounding linguistic community).

Chalmers might dismiss content-involving descriptions of computer programs as heuristic remarks that serve a useful pedagogical function without being literally true. Isn't talk about numbers and arithmetical operations just a loose way of describing syntactic manipulation of numerals? Wouldn't a more rigorous description eschew semantic locutions?

I will now argue that this suggestion flouts entrenched CS practice, which assigns a crucial, non-heuristic status to content-involving instructions.

4.1 The denotational semantics of Scheme

Programming languages are artificial constructs. In contrast with natural languages, their meanings reflect explicit stipulation rather than tacit convention. For instance, the *Revised^n Report on the Algorithmic Language Scheme* (R*n*RS) standardizes the syntax and semantics of Scheme. The latest revision is R6RS (Sperber, et al. [2009]), although R5RS (Kelsey, et al. [1998]) still enjoys wide popularity. R*n*RS stipulates the official meanings of Scheme programs, so it provides unrivaled insight into those meanings.

R*n*RS straightforwardly endorses a content-involving construal of Scheme instructions. It explicitly distinguishes numerals from numbers. It emphasizes that implementation of Scheme programs requires representing numbers and performing arithmetical operations. For instance, R6RS includes the following key passage: 'it is important to distinguish between the mathematical numbers [and] the Scheme objects that attempt to model them… In this report, the term *number* refers to a mathematical number, and the term *number object* refers to a Scheme object representing a number' (p. 24). The report enumerates various arithmetical operations executed by Scheme (pp. 81-96), such as 'number-theoretic integer division' and 'the greatest common divisor' (p. 92), and it explicitly distinguishes those arithmetical operations from mere syntactic manipulation of numerals ('number objects').

That Scheme's designers intend their talk about numerical representation literally is confirmed by the official formal semantics offered in R5RS. The formal semantics codifies talk about numerical representation through a rigorous *denotational semantics*, based on work of

Scott ([1972]) and Strachey ([1966]). The basic idea is that a computer program denotes a function from one state to another, where a state is modeled roughly as an assignment of data values to memory locations. The function denoted by a program is determined compositionally, by assigning denotations to primitive syntactic items and then offering rules that determine a complex string's denotation in terms of the denotations of its parts. Mitchell ([2003], p. 70) illustrates with a simple language of binary terms and arithmetical function signs:

$$E[[0]] = 0$$

$$E[[1]] = 1$$

$$E[[nb]] = E[[n]] * 2 + E[[b]]$$

$$E[[e_1 + e_2]] = E[[e_1]] + E[[e_2]]$$

$$E[[e_1 - e_2]] = E[[e_1]] - E[[e_2]]$$

where E[[…]] maps an expression to its denotation. Expanding this approach to a full programming language is complex task that requires serious mathematical machinery (Stoy [1977]). For the specifics regarding Scheme, see (Kelsey, et al. [1998], pp. 84-95).

The key point for us is already implicit in Mitchell's toy example: just as the name suggests, 'denotational semantics' codifies representational relations to extra-syntactic entities such as numbers. As Mitchell ([2003], p. 70) describes his example:

> [o]n the right-hand side of the equal signs, numbers and arithmetical operations *, +, and – are meant to indicate the actual natural numbers and the standard integer operations of multiplication, addition, and subtraction. In contrast, the symbols + and – and expressions surrounded by double square brackets on the left-hand side of the equal signs are symbols of the object language, the language of binary expressions.

Obviously, Mitchell recognizes the distinction between numerals and numbers, and he holds that denotational semantics assigns numbers as denotations to numerals. Similar remarks, with similar attention to use-mention distinctions, appear frequently in the denotational semantics literature (Allison [1986], pp. 4-5; Stoy [1977], pp. 26-31). That literature provides the background for Scheme's official denotational semantics. The semantics takes numerals to represent numbers, and it takes certain computational states to involve assignment of numbers to memory locations (Kelsey, et al. [1998], pp. 84-95).

In summary, Scheme's designers explicitly stipulate that executing certain Scheme programs requires instantiating suitable semantic properties. They present this stipulation both through informal remarks and through a formal denotational semantics. There can be no reasonable doubt that Scheme's designers construe many Scheme programs as instructions to perform various arithmetical operations. Executing those instructions requires bearing suitable representational relations to numbers.

Of course, a physical computer executes content-involving instructions only because it performs certain syntactic manipulations. Computer scientists capture the requisite manipulations through an *operational semantics*, which describes the meaning of a program through elementary syntactic operations that any implementation must execute. For example, in the standard operational semantics for Scheme, one elementary syntactic operation is a version of β-reduction for the lambda calculus (Sperber, et al. [2009], p. 126). Operational semantics provides indispensable insight into computation. Thus, I do not claim that denotational semantics *exhausts* a program's meaning. I claim only that it captures *one important aspect* of a program's meaning. (Compare: one can hold that referential semantics captures a vital aspect of natural language meaning while acknowledging that it misses other aspects, such as Fregean sense.) Operational and denotational semantics are complements, not competitors.

Content-involving descriptions offer distinctive advantages not offered by purely syntactic descriptions. For instance, the informal Euclidean algorithm is ‘notation-free.’ It ignores details about how we represent numbers. It applies whether our notation is unary, binary, decimal, etc. The algorithm invokes arithmetical operations, not syntactic mechanisms. As Peacocke ([1999]) observes, notation-free description offers indispensable insight into physical computation. It individuates computational states partly by their semantic properties (e.g. *storing the number m in a memory location*), rather than their syntactic properties (e.g. *storing some decimal numeral in a memory location*). It thereby facilitates homogeneous description of systems that are heterogeneous under syntactic and physical description. It captures what diverse physical systems have in common despite radically different numerical notations. Thus, content-involving descriptions yield a distinctive degree of generality not otherwise available. That is one main reason why computer scientists frequently operate at a content-involving level.[5]

Our discussion reveals that structuralism about computational implementation is *revisionary* regarding CS practice. Computer scientists routinely individuate computational states by citing representational properties that outstrip any relevant pattern of causal organization. If we dismiss computer scientists’ appeals to representational properties of computational states, we do not merely dismiss a few extraneous asides. We dismiss entrenched features of scientific practice. We must argue that computer scientists are pervasively mistaken about the meanings of instructions formulated in a programming language that they themselves invented. We must hold that CS as currently practiced is radically mistaken, requiring systematic correction by philosophers. We must apply an extreme error theory to the CS community.

Are there compelling grounds for embracing this radically revisionist agenda? I will now consider and reject two possible grounds.

4.2 Worries about intentionality

Beginning with Quine ([1960]), many philosophers have insisted that representationality, or *intentionality*, is suspect. Quine argued that intentionality cannot be reduced to the non-intentional. He concluded that intentional locutions deserve no place in scientific discourse.

Deploying Quinean worries about intentionality, one might argue that semantics deserves no place in our theory of computational implementation. Chalmers argues along these lines, although he does not explicitly mention Quine. He warns that '[i]f we build semantic considerations into the conditions for implementation, any role that computation can play in providing a foundation for AI and cognitive science will be endangered, as the notion of semantic content is so ill-understood that it desperately needs a foundation itself' ([1995], p. 399). He does not say what it is for computation to provide a 'foundation' for AI and cognitive science, or what it would be to provide a 'foundation' for the notion of semantic content. But the idea seems to be that intentional notions are so obscure that science should cite them only once we have reduced them to non-intentional notions.

I disagree. I will not review the well-known Quinean and Quine-inspired arguments that intentionality is legitimate only if reducible to the non-intentional. However, I agree with Burge ([2010], pp. 296-298) that those arguments are uniformly unconvincing. There is no principled reason why a science that employs primitive intentional notions counts as irredeemably flawed. There is no clear reason why scientific appeals to intentionality must await the completion of some reductive philosophical enterprise. I do not dismiss the problem of intentionality as a pseudo-question. There is a genuine puzzle about what confers content on linguistic expressions or mental states. We should try to solve that puzzle. But why must we solve it in order for semantics to occupy a legitimate role within current science? There is no clear reason why scientists who cite semantic properties must explain the facts by virtue of which an entity has semantic properties, any more than economists who cite monetary value must elucidate the facts by virtue of which an entity has monetary value. For many purposes, we can hypothesize that an entity has certain semantic properties (just as economists hypothesize that an entity has certain monetary properties), deploying that hypothesis within our theorizing.

In effect, Quine and Chalmers impose an ‘external’ norm of clarity upon scientific practice. They allow science to cite intentionality only once intentionality meets proprietary philosophical standards of perspicuity, standards that find no grounding in current science itself. This methodology has a disreputable history. For instance, the same methodology guided those who argued that Newtonian gravity was occult because it violated their mechanistic world view. My paper employs an opposing methodology. I scrutinize the current science of physical computation (specifically, CS), and I ask what standards of clarity and success that science embodies. The standards derive from scientific practice, not from some external standpoint. Given this methodology, we should not dismiss intentionality as obscure or illicit. Computer scientists routinely individuate machine states by their semantic properties (e.g. *representing the number n*). We therefore have *prima facie* reason to accept semantic properties as scientifically legitimate. The attacks on intentionality launched by Quine and his followers do not even begin to defeat this *prima facie* reason.

4.3 Worries about the natural numbers

Another possible rationale for revising CS emphasizes not representation *in general,* but representation *of the natural numbers* specifically. One might worry that numbers do not exist, or that my analysis presupposes an overly Platonist conception of them. One might also worry about the ability of a human or machine to represent the numbers, given that we cannot causally interact with abstract entities (Benacerraf [1973]). Citing such worries, Turner ([2007]) maintains that denotational semantics is far more problematic than operational semantics. More generally, one might argue that numerical representation deserves no place in implementation conditions for computer programs, despite whatever computer scientists may say.

A key point to recall here is that classical mathematics features widespread apparent ontological commitment to abstract entities, including the natural numbers. Thus, it is no devastating objection to contemporary CS that it likewise features such ontological commitment. On the contrary, the burden lies with nominalists to explain away or otherwise accommodate ubiquitous talk about abstract entities within classical mathematics. Whatever maneuvers nominalists deploy towards that end can presumably be extended towards talk about abstract entities within CS. The widespread CS commitment to content-involving instructions raises no *special* ontological problems beyond those raised by classical mathematics more generally.

I do not want to downplay the vexing questions raised by mathematical representation. There are important problems concerning the ontological status of the natural numbers, not to mention our representational access to them. I claim only that my argument in this paper is compatible with any remotely plausible solution to these problems.

On any plausible view, there is an important difference between base-10 and base-13 notation. Mathematical activity is not simply a game during which we manipulate a meaningless formal calculus. Mathematical symbols are *meaningful*. To capture the meanings of mathematical symbols, Platonists cite denotational relations to mind-independent abstract entities. Anti-Platonists reject this approach. But any plausible view, including even a nominalist view, must acknowledge that the numeral '`20`' has a different meaning in base-10 notation than base-13 notation. This difference in meaning is all that my argument requires.[6] To rephrase my argument:

> **(1)** It is possible for there to exist intrinsic physical duplicates M_{10} and M_{13}, the former appropriately embedded in a linguistic community that employs base-10 notation, the latter embedded in a linguistic community that employs base-13 notation. Since M_{10} and M_{13} are intrinsic physical duplicates, they have the same causal organization.
>
> **(2)** M_{10} manipulates numerals whose meanings are given by base-10 notation, and it executes appropriate syntactic manipulations, so it thereby implements the Euclidean algorithm Scheme program. M_{13} manipulates numerals whose meanings are given by base-13, so it does not implement the Euclidean algorithm Scheme program, even though it executes the same syntactic manipulations as M_{10}.

Taken together, (1) and (2) suffice to rebut structuralism. Ultimately, then, my argument does not enshrine an overly controversial conception of the numbers or our representational access to them. It is true that I initially formulated my argument in fairly Platonist terms. But my basic anti-structuralist point persists through the more metaphysically neutral formulation (1)-(2).[7]

5. Implementing a machine model

I now turn attention to *machine models of computation*. A machine model explicitly specifies possible states of a computing system (e.g. configurations of the Turing machine scanner and tape), and it specifies a transition function over those states. The transition function encodes mechanical instructions dictating how to transit among computational states. Implementing the machine model requires reliably conforming to the instructions. Thus, a machine model is an idealized description of a system that reliably executes mechanical instructions. For instance, Feferman ([2006], p. 203) 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.’ Similarly, Abelson and Sussman ([1996], p. 490) state that a register machine ‘sequentially executes *instructions* that manipulate the contents of a fixed set of storage elements called *registers*.’

Crucially, machine models can encode content-involving instructions. Consider a Euclidean algorithm register machine program discussed by Abelson and Sussman (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)
```

As Abelson and Sussman note, ‘[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). Thus, executing the register machine program requires representing numbers and performing arithmetical operations. Needless to say, one can easily encode the above program through an explicit transition function.

We can now consider register machine analogues to M_{10} and M_{13}. The only wrinkle is that the register machine formalism explicitly assumes infinite discrete memory capacity. Infinite discrete memory capacity is impossible in practice and perhaps in principle. So it is doubtful that a physical system can literally implement a register machine. I avoid this wrinkle by considering a modified formalism that assumes large but finite memory capacity in each register. (To a first approximation, modern computers are physical realizations of such a formalism.) Let us henceforth consider a Euclidean algorithm 'register machine' subject to these finitary memory restrictions. Then we can implement the machine with an actual physical computer R_{10}.

Imagine a system R_{13} with the same local, intrinsic physical properties as R_{10}, but employed by a society that uses base-13 rather than base-10 notation. Then R_{10} implements the Euclidean algorithm register machine, but R_{13} does not. The two systems instantiate the same 'causal structure isomorphism type,' but only one of them implements the Euclidean algorithm register machine. The difference, as with M_{10} and M_{13}, reflects the linguistic environments in which the machines are respectively embedded. Once again, this particular counter-example is representative of an infinite class of counter-examples, since we can describe infinitely many register machines that perform arithmetical operations over numbers. I conclude that implementing a machine model sometimes requires instantiating suitable semantic properties, which do not reduce to any relevant pattern of causal organization.[8]

Talk about register machines performing arithmetical operations is no mere pedagogical crutch. It figures prominently in the original paper that introduced register machines (Shepherdson and Sturgis [1961]). That paper's first register machine contains infinitely many registers, 'each of which can store any natural numbers 0, 1, 2, …' (p. 219). The machine's basic instructions include 'add 1 to the number in register *n*,' 'subtract 1 from the number in register *n*,' and 'place 0 in register *n*' (p. 219). Lest one dismiss such talk as careless or confused allusions to syntactic operations on numerals, Shepherdson and Sturgis later distinguish numerals from numbers (p. 226), offering a new definition of register machines defined over formal languages. Such passages are not decisive, but they further demonstrate that structuralism is revisionary regarding the aims and methods of CS. To embrace CS as it currently stands, we must reject structuralism as a general theory of computational implementation.

Objection: The Euclidean algorithm register machine takes the remainder operation as primitive. That operation has computational structure, since it involves iterated application of multiplication and subtraction. You have biased your case against structuralism by choosing an example with 'hidden' computational structure.

Reply: We can easily modify the example so as to reveal the hidden computational structure. We can employ a modified Euclidean algorithm register machine that features only the primitive arithmetical operations posited by Shepherdson and Sturgis: *add 1*, *subtract 1*, and *replace with 0*. Those primitive operations do not hide computational structure. Shifting towards this more elementary register machine does not alter our basic moral. We can still construct intrinsic physical duplicates S_{10} and S_{13} embedded in contrasting linguistic environments, with S_{10} executing the register machine and S_{13} not. We need merely stipulate that the respective linguistic communities associate appropriately different semantic interpretations with S_{10} and S_{13}. For instance, if S_{10} performs a syntactic manipulation that replaces numeral '`9`' with numeral '`10`', then S_{10} conforms to the instruction *add 1*. In contrast, when S_{13} performs that same syntactic manipulation, it instead conforms to the instruction *add 4* (since '`10`' denotes 13 in base-13 notation). Thus, decomposing our register machine into 'unstructured' computational steps does not weaken our anti-structuralist conclusion.

Objection: Structuralists intend their account to apply only to computational models specified in non-representational terms. You define 'computational model' more broadly to include semantic interpretation as part of the model, so that semantic constraints naturally become relevant to implementation. The difference is a purely terminological one regarding what counts as a 'genuine computational model.'

Reply**:** This objection trivializes structuralism by dismissing all putative counter-examples as 'non-genuine' computational models. Nothing in contemporary CS suggests that models such as the Euclidean algorithm register machine are second-class or otherwise undeserving of the label 'computational.' On the contrary, to reject content-involving instructions is to reject large swathes of entrenched scientific practice. For instance, the inventors of the world's first stored-program electronic digital computer, the Manchester 'Baby,' explicitly described it as performing arithmetical operations on numbers, including division and factorization (Williams and Kilburn [1948]). Structuralists will dismiss this description as a confused allusion to underlying syntactic operations. Yet they offer no convincing grounds for the dismissal. The burden of proof lies with structuralists to defend their revisionary stance towards CS. Thus far, they have not even begun to meet that burden.

6. Bounded structuralism

Philosophers have proposed several variants of structuralism. For instance, Copeland ([1996]) and Godfrey-Smith ([2009]) suggest that the isomorphism between formal models and physical systems must map formal states to physical state types that are 'natural' rather than 'gerrymandered.' I will not canvass such variants. In virtually all cases, it is straightforward to show that the variant encounters essentially the same counter-examples presented in §§3-5. However, I want to address one notable variant, which I call *bounded structuralism*.[9]

According to bounded structuralism, a physical system realizes a computation just in case: (i) the system instantiates an appropriate pattern of causal organization; (ii) the system has inputs and outputs with desired properties. Implementing a computation requires *not only* instantiating a causal structure isomorphic with the computation's formal structure *but also* satisfying certain 'boundary conditions.' To motivate bounded structuralism over structuralism *simpliciter*, Chrisley ([1994]), Godfrey-Smith ([2009]), and Putnam ([1989], p. 124) note that computational description of a physical system usually cites inherent, non-functional properties of inputs and outputs: e.g. configurations of a computer's keyboard, mouse, or screen, or of a robot's sensors or motor organs.

Bounded structuralists can concede that semantics informs computational implementation 'at the periphery,' i.e. by constraining input-output states. For instance, they can say that implementing the Euclidean algorithm register machine requires: (i) instantiating an appropriate pattern of causal organization; (ii) having inputs and outputs whose semantic interpretation is given by base-10 (rather than, say, base-13). Since R_{13} does not satisfy clause (ii), bounded structuralists can say that R_{13} does not implement the Euclidean algorithm register machine. In this manner, bounded structuralism avoids the particular counter-examples canvassed above. Nevertheless, I will argue that bounded structuralism encounters modified counter-examples.

Say that semantic interpretation of a physical computer's states is *homogenous* just in case it assigns the same meaning to a syntactic item throughout the entire course of computation. If semantic interpretation is homogenous, then we fix the meanings of internal states by fixing the meanings of input or output states (assuming that all syntactic items figuring in internal states also figure as possible inputs or outputs). Yet why must semantic interpretation be homogenous? Homogenous interpretation may be more convenient or natural. It may cohere better with our goals and practices. But there is no principled bar to non-homogenous interpretation. A community might impose the following convention: numerals figuring in a machine's input-output states fall under a base-10 interpretation, but numerals figuring in the machine's internal states fall under a base-13 interpretation. The convention may strike us as deviant, but that does not prevent it from successfully conferring determinate contents on machine states.

Suppose now that an intrinsic physical duplicate of R_{10} is embedded in a community that employs the proposed non-homogenous semantics. Call the duplicate '$R_{10/13}$.' Does $R_{10/13}$ implement our Euclidean algorithm register machine? Implementing the machine requires conforming to its instructions, which include content-involving descriptions of the machine's internal calculations. For instance, if we provide the register machine with inputs 115 and 20, then its program dictates that it should compute the remainder of 20 divided into 115. This is a purely 'internal calculation' that does not directly manifest as an output. R_{10} conforms to the instruction by assigning numeral '`15`' to register `t`. By virtue of making that assignment, and by virtue of falling under an appropriate semantic interpretation, R_{10} computes the remainder of 20 into 115. $R_{10/13}$ will also assign '`15`' to register `t`, but in doing so it does not compute the remainder of 20 divided into 115. Given the interpretation associated with $R_{10/13}$, '`15`' denotes 18 when it figures in internal states (such as the assignment of '`15`' to register `t`). Thus, $R_{10/13}$ does not conform to the requisite content-involving instruction (*compute the remainder of 20 divided into 115*). So $R_{10/13}$ does not implement the desired register machine. Yet its causal structure is isomorphic to the register machine's formal structure, and its inputs and outputs have appropriate meanings. Hence, $R_{10/13}$ is a counter-example to bounded structuralism.

As this example illustrates, bounded structuralism encounters modified versions of the counter-examples that confront structuralism *simpliciter*. CS features instructions that describe internal states, not merely input-output states, in content-involving terms. Thus, current scientific practice supports the following conclusion:

(*) In certain cases, implementing a computation requires instantiating internal states with suitable semantic properties.

Given the possibility of non-homogenous semantic interpretations, (*) undermines bounded structuralism just as readily as it undermines structuralism *simpliciter*. One might argue that current practice is mistaken in embracing (*). But such radical revisionism requires extensive defense, which structuralists have not even begun to provide.

Non-homogenous semantic interpretations will strike some readers as intuitively deviant, unnatural, or contrived. For that reason, $R_{10/13}$ exerts less intuitive force against bounded structuralism than R_{13} exerts against structuralism *simpliciter*. But there are two reasons to ignore the intuitive diminution in force. First, CS frequently employs non-homogeneous semantic interpretations. In decision algorithms, for example, it is common to interpret inputs '|' and '||' as meaning 0 and 1 when they figure as inputs but as meaning 'yes' or 'no' when they figure as outputs. Second, and more importantly, intuitive 'unnaturalness' is irrelevant to our discussion. $R_{10/13}$, however bizarre, is surely *possible*. So the only question is whether it constitutes a counter-example to bounded structuralism. I claim that it does. That it seems contrived or deviant does not render it any less a counter-example.

7. Triviality arguments

Philosophers sometimes claim that the implementation relation is *trivial*. Most dramatically, Searle ([1990]) argues that *every* physical system implements *every* computation. Putnam ([1988], pp. 121-125) defends a somewhat weaker triviality thesis. Triviality arguments typically assume the structuralist view that computational implementation requires only an 'isomorphic mapping' between the computation's formal structure and the physical system's causal structure (perhaps constrained by input/output restrictions). We can erect a gerrymandered isomorphism between a computation and numerous inappropriate 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.'

Many structuralists try to the block the inference from structuralism to pancomputationalism. For instance, Copeland ([1996]) and Godfrey-Smith ([2009]) hope to block the inference by excluding gerrymandered isomorphisms between formal state-types and physical state-types. More concessively, Chalmers ([1996]) holds that triviality arguments succeed for finite state automata but not for computational formalisms whose internal states have combinatorial structure. In contrast, Fodor ([1998], pp. 11-12), Ladyman ([2009b]), and several other discussants attempt to rebut triviality arguments by replacing structuralism with the semantic view of computational implementation.

My analysis helps defuse triviality arguments *for certain computations*. I have argued that certain computations are implemented only by physical systems with appropriate representational properties. Most physical systems do not have representational properties, so most physical systems do not implement the relevant computations. For instance, a physical system implements the Euclidean algorithm register machine only if it bears appropriate semantic relations to numbers. Searle's wall does not satisfy this constraint. Neither does R_{13}.

Thus, the implementation condition for the Euclidean algorithm register machine is not trivial. I do not recommend my analysis as a universal rebuttal to triviality arguments. I claim only that my treatment defuses triviality arguments *for an important class of computational models*: namely, those models that individuate computational states in content-involving terms.

Searle might retort that I have played into his hands by emphasizing representation. We can change R_{13}'s representational properties simply by altering the surrounding linguistic environment. Doesn't it follow that R_{13}'s semantic properties are indeterminate, subjective, trivial, or otherwise suspect? Doesn't my analysis entail that the implementation relation itself is indeterminate, subjective, trivial, or otherwise suspect?

The proposed retort shows that we can *change* semantic properties. It does not even begin to show that there is anything *suspect* about semantic properties. For instance, the word 'dog' currently denotes dogs. We could change our linguistic practice to change the word's meaning. *Until we do so*, 'dog' has a stable, determinate meaning. Similarly, there is a stable, determinate contrast between R_{10} and R_{13}. R_{10} manipulates numerals whose meanings are given by base-10, while R_{13} manipulates numerals whose meanings are given by base-13. There is nothing indeterminate, subjective, trivial, or otherwise suspect about this semantic difference. The linguistic practices in which R_{10} and R_{13} are currently embedded engender a determinate semantic difference between the two physical systems. We could eradicate this difference by embedding R_{13} in a different linguistic practice, such as the practice in which R_{10} is currently embedded. *Until we do so*, R_{13}'s states fall under the stable, determinate semantic interpretation given by base-13. *Until we do so,* R_{13} does not implement the Euclidean algorithm register machine.

Searle may object that a single linguistic practice can fail to determine a unique semantic interpretation. A society might simultaneously employ conflicting interpretations of a physical computer's states. For instance, a society might employ binary strings to represent numbers on some occasions and to represent graphs on other occasions. Thus, semantic interpretation might be underdetermined even within a single linguistic practice.

This latest objection does not purport to establish the pernicious triviality advanced by Searle's original argument. The objection provides no hint that R_{13}, let alone a wall, implements the Euclidean algorithm register machine. At best, the objection establishes a kind of underdetermination: facts about a system's physical properties and the surrounding computational practice may not fully determine whether the system implements a computation.

I agree that this kind of underdetermination can arise. I also claim that it is harmless. By analogy, the phonological item 'bank' can mean either *financial institution* or *river bank*. Linguistic practice does not itself fully determine the meaning of a given utterance featuring that phonological item. Nevertheless, in any ordinary context of use, speaker intentions and expectations fix a unique, determinate meaning. Similarly, suppose that states of physical system *P* are subject to two distinct interpretations within a society: the states can denote numbers, or they can denote graphs. In most normal contexts, a human user will intend only one of these interpretations. If the user wants to compute a number-theoretic function, she will intend the first interpretation. If she wants to compute a function over graphs, she will intend the second. Her intentions fix determinate semantic properties for *P*'s states *during that particular context of use*. During any such context, there is a determinate fact about which computations *P* implements. There may be certain unusual contexts in which *P*'s semantic and computational properties are underdetermined. But no such underdetermination prevails in typical contexts of use. The proposed objection simply shows that facts about computational implementation can depend upon the context of use *as well as* general linguistic conventions. The objection does not show that the implementation relation is pervasively underdetermined.

It is not the case that every physical system realizes every computation within every context of use. There is no context of use in which R_{13} *as currently employed by its actual surrounding society* executes the Euclidean algorithm register machine program. By analogy, there is no context of use in which 'bank' *as currently employed by our actual society* means Bill Clinton. We could alter our current linguistic conventions. *Until we do so*, there is no context of use in which 'bank' means Bill Clinton. We could alter R_{13}'s surrounding linguistic environment to confer base-10 semantic interpretation upon it. *Until we do so*, there is no context in which R_{13} falls under base-10 interpretation. *Until we do so*, there is no context in which R_{13} implements the Euclidean algorithm register machine. So there is nothing trivial or indeterminate about the register machine's implementation condition.

8. Anti-individualism about computational implementation

I conclude by relating my discussion to *anti-individualism about mental content*. This view, espoused by Burge ([2007]), holds that mental content is individuated partly by relations to the surrounding environment, and hence that mental content does not supervene upon internal neurophysiology. Burge defends anti-individualism in several ways. Applying Putnam's Twin Earth thought experiment, he argues that relations to the *physical* environment help individuate mental content ([2007], pp. 82-99). Applying his own arthritis thought experiment, he argues that relations to the *social* environment help individuate mental content ([2007], pp. 100-181). He also supports his position by citing explanatory practice within cognitive science, especially *perceptual psychology* ([2007], pp. 221-253, [2010], pp. 61-105).

My position addresses computational implementation, not the individuation of mental contents. One could accept my position while rejecting Burge's, and vice-versa. Yet the positions are analogous. Burge argues that relations to an embedding social environment help individuate a thinker's propositional attitudes. I argue that relations to an embedding social environment help individuate a machine's computational states, and hence which computations the machine executes. R_{10} executes the Euclidean algorithm register machine, but intrinsic physical duplicate R_{13} does not. The difference does not reflect any 'internal' differences in circuitry. It reflects the fact that R_{10}'s states have different semantic import than R_{13}'s, which reflects the differing linguistic environments within which R_{10} and R_{13} are embedded. Thus, I endorse what one might call *anti-individualism about computational implementation*: relations to the external physical and/or social environment sometimes inform whether physical system P implements a given computation. A system's 'social environment' may include both general linguistic conventions and specific contexts of use.

The opposing position is *individualism about computational implementation*, according to which computational properties supervene on intrinsic physical properties. As Egan ([1992], p. 446) puts it, 'if two systems are molecular duplicates then they are computational duplicates.' Structuralism entails individualism, since intrinsic physical duplicates share the same 'causal structure isomorphism type.' Most commentators, although not all, endorse individualism.[10] I have argued that individualism conflicts with the contemporary science of physical computation. CS routinely individuates computational states and processes through relations to the embedding social environment. CS routinely deploys computational models whose implementation conditions are anti-individualist.

Burge pioneered the thesis that relations to the social environment can help individuate content-bearing internal states. However, he defends that thesis principally by studying folk psychology, not scientific practice. When he cites perceptual psychology to support anti-individualism, he emphasizes relations to the physical rather than the social environment. Perhaps for that reason, philosophers are generally much warier regarding individuation by relations to the social, as opposed to physical, environment. I have identified a successful scientific discipline (CS) that individuates content-bearing internal states of a physical system partly through relations to the surrounding social environment. Few other authors, if any, offer detailed arguments that an ongoing scientific enterprise employs such an individuative scheme.

Please distinguish my position from a much weaker thesis: namely, that a physical system's semantic properties sometimes depend upon the surrounding social environment. Most philosophers, including Chalmers and Egan, would accept the weaker thesis. But my position goes well beyond the weaker thesis. I claim also that the surrounding social environment can inform whether a physical system implements a computational model. Chalmers and Egan deny this stronger claim.

A physical system can implement more than one computational model. If physical system *P* implements a model with an anti-individualist implementation condition, I can concede that *P* also realizes a model *M* with an individualist implementation condition. *M* might type-identify computational states in purely syntactic, non-semantic terms. Alternatively, *M* might type-identify computational states through some kind of 'narrow content' that supervenes on intrinsic physical properties. For present purposes, I can even concede that *M* enjoys causal or explanatory priority over models with anti-individualist implementation conditions. Perhaps *M* captures *P*'s true 'causal structure.' Perhaps *M* offers explanatory benefits not otherwise available. This paper is neutral on such questions. I am addressing *implementation*, not *causation* or *explanation*.[11] My claim is simply that implementation conditions for some computations are anti-individualist. Thus, structuralism misdescribes the implementation relation. It provides an inadequate general framework for studying physical computation.

I have not offered my own alternative framework. But I have articulated two constraints on any viable alternative: it must allow semantic properties to inform implementation conditions; and it must embrace anti-individualism about computational implementation. Only by satisfying these constraints can we securely ground the philosophical analysis of computational implementation in contemporary scientific practice.

Department of Philosophy

University of California

Santa Barbara, CA 93106

rescorla@philosophy.ucsb.edu

Acknowledgments

I am greatly indebted to David Chalmers for comments that substantially improved the paper. I also received helpful feedback from Tyler Burge, Mark Sprevak, and three anonymous referees for this journal.

References

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

Ackerman, D. [1979]: ‘*De Re* Propositional Attitudes Toward Integers’, *Southwestern Journal of Philosophy*, **9**, pp. 145-153.

Allison, L. [1986]: *A Practical Introduction to Denotational Semantics*, Cambridge: Cambridge University Press.

Benacerraf, P. [1965]: ‘What Numbers Could Not Be’, *Philosophical Review,* **74**, pp. 47-73.

---. [1973]: ‘Mathematical Truth’, *The Journal of Philosophy*, **70**, pp. 661-679.

Bontly, T. [1998]: ‘Individualism and the Nature of Syntactic States’, *The British Journal for the Philosophy of Science*, **49**, pp. 557-574.

Burge, T. [2007]: *Foundations of Mind*, Oxford: Oxford University Press.

---. [2010]: *Origins of Objectivity*, Oxford: Oxford University Press.

Chalmers, D. [1995]: ‘On Implementing a Computation’, *Minds and Machines*, **4**, pp. 391-402.

---. [1996]: ‘Does a Rock Implement Every Finite State Automaton?’, *Synthese*, **108**, pp. 309-333.

Chrisley, R. [1994]: ‘Why Everything Doesn’t Implement Every Computation’, *Minds and Machines*, **4**, pp. 403-420.

Cleland, C. [2002]: ‘On Effective procedures’, *Minds and Machines*, **12**, pp. 159-279.

Copeland, J. [1996]: ‘What is Computation?’, *Synthese,* **108**, pp. 335-359.

Crane, T. [1990]: ‘The Language of Thought: No Syntax Without Semantics’, *Mind and Language*, **5**, pp. 187-212.

Dennett, D. [1987]: *The Intentional Stance*, Cambridge, MA: MIT Press.

Dresner, E. [2010]: ‘Measurement-Theoretic Representation and Computation-Theoretic Realization’, *The Journal of Philosophy*, **107**, pp. 275-292.

Egan, F. [1992]: ‘Individualism, Computation, and Perceptual Content’, *Mind*, **101**, pp. 443-459.

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

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

Godfrey-Smith, P. [2009]: 'Triviality Arguments Against Functionalism', *Philosophical Studies*, **145**, pp. 273-295.

Halbach, V., and Horsten, L. [2005]: 'Computational Structuralism', *Philosophia Mathematica*, **13**, pp. 174-186.

Horowitz, A. [2007]: 'Computation, External Factors, and Cognitive Explanations', *Philosophical Psychology*, **20**, pp. 65-80.

Kelsey, R., Clinger, W. and Rees, J. [1998]: 'Revised^5 Report on the Algorithmic Language Scheme', *Higher-Order and Symbolic Computation*, **11**, pp. 7–105.

Kim, J. [2005]: *Philosophy of Mind*, Boulder: Westview Press.

Kripke, S. [1984]: *Wittgenstein on Rules and Private Language*, Cambridge, MA: Harvard University Press.

Knuth, D. [1968]: *The Art of Computer Programming*, vol. 1, Reading: Addison-Wesley.

Ladyman, J. [2009a]: 'Structural Realism', in E. Zalta (*ed.*), *Stanford Encyclopedia of Philosophy*, <plato.stanford.edu>

---. [2009b]: 'What Does it Mean to Say that a Physical System Implements a Computation?', *Theoretical Computer Science*, **410**, pp. 376-383.

McCarthy, J. [1963]: 'Towards a Mathematical Science of Computation', in C. M. Popplewell (*ed.*), *Information Processing 1962: Proceedings of DFIP Congress 62*, Amsterdam: North-Holland, pp. 21-28.

Mitchell, J. [2003]: *Concepts in Programming Languages*, Cambridge: Cambridge University

Press.

Peacocke, C. [1994]: ‘Content, Computation, and Externalism’, *Mind and Language*, **9**, pp. 303-335.

---. [1999]: ‘Computation as Involving Content: a Response to Egan’, *Mind and Language*, **14**, pp. 195-202.

Piccinini, G. [2008]: ‘Computation Without Representation’, *Philosophical Studies*, **137**, pp. 205-241.

Putnam, H. [1988]: *Representation and Reality*, Cambridge, MA: MIT Press.

Quine, W. V. [1960]: *Word and Object*, Cambridge, MA: MIT Press.

Rescorla, M. [2007]: ‘Church’s Thesis and the Conceptual Analysis of Computability’, *Notre Dame Journal of Logic*, **48**, pp. 253-280.

---. [Forthcoming (a)]: ‘Are Computational Transitions Sensitive to Semantics?’, *Australasian Journal of Philosophy*.

---. [Forthcoming (b)]: ‘How to Integrate Representation into Computational Modeling, and Why We Should’, *Journal of Cognitive Science*.

Scheutz, M. [2001]: ‘Computational versus Causal Complexity’, *Minds and Machines*, **11**, pp. 544-566.

Scott, D. [1972]: ‘Lattice Theory, Data Types, and Semantics’, in R. Rustin (*ed.*), *Formal Semantics of Programming Languages*, Englewood Cliffs: Prentice-Hall, pp. 65-106.

Shepherdson, J. and Sturgis, H. E. [1963]: ‘Computability of Recursive Functions’, *Journal of the Association of Computing Machinery*, **10**, pp. 217-255.

Sperber, M., Dybvig, K., M. Flatt, and van Straaten, A. [2009]: ‘Revised^6 Report on the Algorithmic Language Scheme’, *Journal of Functional Programming*, **19 (S1)**, pp.

1-301.

Sprevak, M. [2010]: ‘Computation, Individuation, and the Received View on Representation’, *Studies in History and Philosophy of Science*, **41**, pp. 260-270.

Stoy, J. [1977]: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Cambridge, MA: MIT Press.

Strachey, C. [1966]: ‘Towards a Formal Semantics’, in T. B. Steel (*ed.*), *Formal Language Description Languages for Computer Programming*, Amsterdam: North-Holland, pp. 198-218.

Turner, R. [2007]: ‘Understanding Programming Languages’, *Minds and Machines*, **17**, pp. 203-216.

Williams, F., and Kilburn, T. [1948]: ‘Electronic Digital Computers’, *Nature*, **162**, p. 487.

Notes

[1] My target is *structuralism about computational implementation*, not *structuralism about the natural numbers* (Benacerraf [1965]; Halbach and Horsten [2005]), *structural realism* (Ladyman [2009a]), or any other view that may deservedly be called ‘structuralist.’ My arguments are specific to a structuralist view of the realization relation between abstract computational models and physical systems.

[2] Cleland ([2002]) likewise emphasizes that *conformity to instructions* underlies our intuitive notion of computation, and she also draws the analogy with recipe execution. She seeks to undermine the standard view that abstract computational models are central to elucidating physical computation. In contrast, I accept the standard view. My goal is to illuminate the physical realization relation between abstract models and physical systems.

[3] Peacocke ([1994], [1999]) also defends the thesis that some computational models feature content-involving instructions. My discussion goes beyond Peacocke's in several respects. First, I defend the thesis through detailed analysis of computer science practice. Second, I apply the thesis in ways that Peacocke does not: to attack structuralism (§§4-6); to address Putnam-Searle triviality arguments (§7); and to argue that social factors can inform computational implementation (§8). Peacocke ([1994], p. 320) accepts the semantic view of computational implementation, which I reject. Despite these differences, Peacocke's discussion has decisively influenced my treatment.

[4] There are infinitely many functions from Arabic numerals to the natural numbers, infinitely many of which are intuitively computable. In principle, any intuitively computable function from numerals to the numbers can serve as a viable semantic interpretation for the numerals. In contrast, semantic interpretations that are not intuitively computable do not seem legitimate. For further discussion, see (Rescorla [2007]).

[5] Operational semantics offers several advantages over denotational semantics, including increased mathematical accessibility. Accordingly, R6RS replaces the denotational semantics from earlier reports with an operational semantics. But this shift does not signify rejection of a content-involving perspective on Scheme instructions. If anything, the informal remarks from R6RS even more explicitly endorse the content-involving perspective.

[6] Some philosophers argue that humans achieve *de re* reference to the natural numbers through canonical numerals (Ackerman [1979]; Burge [2007], pp. 70-75). My discussion is consistent with this view, but I do not presuppose it. I assume only that there is a difference in meaning between base-10 and base-13. I deploy that assumption to argue that M_{10} and M_{13} implement different computations.

[7] My position is consistent with a structuralist view of the natural numbers (Benacerraf [1965]). On this view, arithmetic is about a structure determined by the Dedekind-Peano Axioms, perhaps along with other constraints, such as the computability restriction advocated in (Halbach and Horsten [2005]). Natural numbers are ‘positions’ in the structure. Structuralists must still differentiate between base-10 and base-13. Presumably, they will do so by contrasting how the respective notations correlate numerals with positions in the relevant structure. Thus, structuralists should still acknowledge an important semantic difference between M_{10} and M_{13}.

[8] Inspired by Wittgenstein, Kripke ([1984]) argues that the notion *following a rule* is problematic. Even if this is right, Kripkenstein rule-following worries do not affect my anti-structuralist argument. Kripke’s argument focuses on rules such as *add 2,* which apply to infinitely many cases. Given my focus on register machines with finite memory capacity, we can concentrate entirely on rules with finite scope. For instance, rather than considering the remainder operation as defined over all numbers, we can consider a modified operation that applies only to numbers below some upper bound determined by the register machine’s finite memory capacity. We can enumerate how this operation applies to all numbers below the upper bound. The enumeration might take longer than any actual human lifetime, but it is possible in principle for an idealized human. In principle, then, we can explicitly enumerate what a physical machine must do in order to realize the Euclidean algorithm register machine (with restricted memory). No Kripkensteinian worry arises, because there is no need to extend the programming instructions to larger numbers beyond our explicit enumeration. It is easy to show that R_{10} executes the desired instructions but that R_{13} does not.

[9] Thanks to David Chalmers for suggesting that I consider this variant position.

[10] Bontly ([1998]), Horowitz ([2007]), Peacocke ([1994], [1999]), and Sprevak ([2010]) espouse anti-individualism about computational implementation. Bontly does not ground his treatment in semantic constraints upon implementation. Sprevak grounds his treatment in the semantic view of computational implementation, which I rejected in §2. None of these four authors explicitly mentions, let alone endorses, individuation of computational states by relations to the *social* (as opposed to *physical*) environment.

[11] For related discussion, see (Rescorla [forthcoming (a)], [forthcoming (b)]).

TABLE 1 (to be inserted as indicated on p. 5):

	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$
$\mathbf{I_1}$	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