Why do we need a theory of implementation?

André Curtis-Trudel

April 8, 2020

Abstract

The received view of computation is methodologically bifurcated: it offers different accounts of computation in the mathematical and physical cases. But little in the way of argument has been given for this approach. This paper rectifies the situation by arguing that the alternative, a unified account, is untenable. Furthermore, once these issues are brought into sharper relief we can see that work remains to be done to illuminate the relationship between physical and mathematical computation.

1 Introduction: The Received View of Physical Computation

Many contemporary scientific disciplines, including computer science, cognitive science, and artificial intelligence, explain the behavior and capacities of certain physical systems in terms of the computations those systems perform. Accordingly, one task is to develop a theory of physical computation suitable for this work.

The received view characterizes physical computation in terms of mathematical computation. Ordinarily this is taken to be a matter of relating mathematical computations to the physical world:

> Determining what conditions a physical system must satisfy in order to compute is the focus of theories of computational implementation, or physical computation . . . An account of implementation aims to specify the conditions under which a physical system performs a computation defined by a mathematical formalism – it is a theory of physical computation. (Ritchie and Piccinini [2019], pp. 192-3)

Though I will say more about this in due course, the crucial point is that this view begins with a mathematical notion of computation, and applies it to physical systems by way of an implementation relation. Call this 'implementationism' about physical computation.[1]

Implementationist theories must negotiate a number of further issues, concerning both the nature of implementation, and what constraints, if any, there are on which physical systems

[1]Particular implementationist views are developed and defended in (Chalmers [1994]; Chalmers [1996]; Chalmers [2011]; Egan [2010]; Matthews and Dresner [2017]; Millhouse [2019]; Rapaport [1999]; Rescorla [2014]; Scheutz [2001]; Schweizer [2019]), among others. Ritchie and Piccinini ([2019]) and Sprevak ([2019]) survey the major issues.

may implement a computation. Perhaps unsurprisingly, there is widespread disagreement about how these issues should be addressed. But this is disagreement over details—how to develop implementationism, and not whether we should. What is the rationale for characterizing physical computation this way in the first place? The standard answer, roughly, is that we already have a mathematical theory of computation, so we might as well use it. Here's Chalmers on the matter:

> The mathematical theory of computation in the abstract is well-understood, but cognitive science and artificial intelligence go beyond the domain of abstract objects to deal with concrete systems in the physical world. The difficult questions about computation are largely questions about the relationship between these domains. How does the abstract theory of computation relate to a science of concrete, causal systems, and how might it help us explain cognition in the real world? To answer these questions, we need a bridge between the abstract and the concrete domains. (Chalmers [1994], pp. 341-2)

As Chalmers goes on to suggest, a theory of implementation is just what is needed to 'bridge' the gap between mathematical and physical reality.[2]

It seems to me that this 'mathematics-first' approach rests on a more basic attitude towards the relationship between the theories of mathematical computation and physical computation. Implementationist theories say nothing about mathematical computing systems, such as Turing machines, because they are assumed to be dealt with by a prior mathematical theory. Given such a theory, the implementationist's task is to use it to develop an account of physical computation. Thus, by taking a mathematical theory of computation for granted, implementationism divides the theory of computation into two parts, each of which plays a different explanatory role. One part explains computation in the mathematical case, while the other explains computation in the physical case. Call this 'bifurcationism' about the theory of computation. I shall say more about bifurcationism shortly.

Bifurcationism has been tacitly assumed by much of the literature on physical computation. This is striking, for a couple reasons. First, what little argument exists for it rests on

[2]Chalmers' attitude persists today. Here's Rescorla ([2017], p. 288): "Philosophical discussion of computation should ground itself in the mathematical theory of computation." And Sprevak ([2010], p. 263): "A realization [implementation] function maps physical nuts and bolts to the formal symbols employed by computation theory. It is the link between the abstract world of mathematical computation and the empirical world." Similar remarks may be found throughout the literature.

questionable assumptions about the nature and character of the mathematical theory of computation. Second, and perhaps more seriously, it seems that considerations from both the general philosophy of science, as well as from the philosophy of computer science in particular, augur against bifurcationism. These considerations point towards a theory which offers the same account of computation in both the physical and mathematical cases. I call this alternative 'unificationism'.

The upshot is that if we are to endorse bifurcationism, much less implementationism, a new argument is needed. The main contribution of this paper is to provide one. I argue that the prospects for a unificationist theory which does not trivialize physical computation are grim. To the extent that a non-trivial theory of physical computation is desirable—I will assume that it is—and to the extent that bifurcationist accounts stand a better chance of avoiding triviality—I will argue that they do—we ought on balance to prefer bifurcationism to unificationism.

As we will see, however, bifurcationism does not mandate implementationism. That is, bifurcationism as such does not require that we take Chalmers' 'mathematics-first' approach to physical computation. In fact, bifurcationism is compatible with at least two other views about the relationship between the theories of physical and mathematical computation. These other views have not received the attention they deserve, and I submit that this partly because bifurcationism itself has not yet been isolated for independent treatment. Thus, a second contribution of this paper is to clarify the possible relationships between theories of physical and mathematical computation, and suggest two views for further investigation.

Here is the plan for the rest of the paper. Section 2 fleshes out the distinction between unificationism and bifurcationism. Section 3 argues that extant motivation for bifurcationism, such as it is, is uncompelling. Section 4 makes a *prima facie* case for unificationism. Sections 5 and 6 argue that, initial attractions aside, unificationism faces serious problems. Section 7 concludes.

2 Unificationism and Bifurcationism

I take it that any theory of computation must answer two questions. The first concerns the computational status of a system: what is it for a system to compute, and what in general distinguishes systems that compute from those that don't? The second concerns computational identity and individuation: among systems that compute, what distinguishes those that compute the same thing from those that don't? As we might put it, we wish to know whether, and if so what, a system computes. I will call these the status and identity questions, respectively.

The contrast between unificationism and bifurcationism can be understood as a disagreement about how these questions should be answered. To a first approximation, the bifurcationist holds that the theory which furnishes answers to the status and identity questions concerning physical computing systems needn't also furnish answers to these questions as they concern mathematical computing systems. The unificationist, by contrast, holds that one and the same theory which furnishes answers to these questions concerning physical systems must also furnish answers to their mathematical counterparts.

So construed, unificationism and bifurcationism are methodological claims about which questions a theory of computation ought to address. It will be useful to have a 'metaphysical' version of the contrast on the table, too, although it seems to me that the methodological claims are more fundamental. On this second way of drawing the distinction, a theory is unificationist if it supplies the same kind of answer to the status and identity questions for both sort of computing system, and a theory is bifurcationist if it doesn't.[3]

How are the methodological and metaphysical claims related? For now it's enough to notice that if unificationist or bifurcationist theories are in general untenable, it's likely that the corresponding methodologies are misguided too. That is, the methodological versions of either unificationism or bifurcationism should be abandoned if it can be shown that their metaphysical counterparts are unsustainable. Accordingly, even though I focus on the

[3]Additionally, while I've drawn the distinction at the level of theories, we might also characterize it in terms of a variety of more fine-grained notions, such as the truth-conditions of computational statements, the applicability conditions of computational concepts, or the instantiation conditions of computational properties. For our purposes, however, the theory-centric characterization will suffice.

prospects for unificationist and bifurcationist theories, the methodological views are implicated by my arguments as well.

One might doubt that the unificationism–bifurcationism distinction marks a substantive contrast, on the grounds that it is very easy to turn any putatively bifurcationist theory into a unificationist theory. Suppose that M is a theory which answers the status and identity questions for mathematical systems and that P is a theory which answers these questions for physical systems. Then $M \wedge P$ is a unificationist theory, for one and the same theory, namely $M \wedge P$, answers the status and identity questions for both sorts of system.

However, this sort of construction doesn't get to the heart of the matter. What's at issue is not just the form a theory of computation takes, but how it answers the status and identity questions concerning different sorts of system. Moreover, this sort of construction can be ruled out easily enough: let unificationism be as before, but require that there be no proper sub-theory which supplies answers to the status and identity questions for physical systems alone.

To help bring out the intended contrast, it is instructive to compare some particular theories of physical computation. The first is Chalmers' implementationist theory.[4]

> A physical system P implements a CSA M if there is a decomposition of internal states of P into components $[s^1, s^2, ...]$, and a mapping f from the substates s_j into corresponding substates S_j of M, along with similar decompositions and mappings for inputs and outputs, such that for every state-transition rule $([I^1, ..., I^k], [S^1, S^2, ...]) \rightarrow ([S'^1, S'^2, ...], [O^1, \ldots, O^l])$ of M: if P is in internal state $[s^1, s^2, ...]$ and receiving input $[i^1, ..., i^n]$ which map to formal state and input $[S^1, S^2, ...]$ and $[I^1, ..., I^k]$ respectively, this reliably causes it to enter an internal state and produce an output that map to $[S'^1, S'^2, ...]$ and $[O^1, \ldots, O^l]$ respectively. (Chalmers [1994], p. 394)

Thus on this view whether a physical system computes is a matter of implementing a CSA computation, and what it computes is determined by the CSA (or CSAs) it implements. This approach relies on a prior mathematical theory of CSA computation, specifying what mathematical entities count as CSAs and what computation a given CSA performs, which is then put to work to define physical computation. The account thus offers different kinds of answers to the status and identity questions as they concern physical and mathematical

[4]Chalmers' account is couched in terms of combinatorial state automata (CSAs), a generalization of finite state automata.

systems. The latter are addressed by the mathematical theory of CSA computation; the former by the account of implementation just cited. For this reason Chalmers' account seems to me a paradigmatic bifurcationist account of computation. And to the extent that Chalmers' view is representative of implementationist accounts generally, implementationism overall incurs a commitment to bifurcationism too.

Contrast this with the account of function computation found in (Copeland [1996]). According to Copeland's account:

> Entity e is computing function f if and only if there exists a labelling scheme L and a formal specification *SPEC* (of an architecture and an algorithm specific to the architecture that takes arguments of f as inputs and delivers values of f as outputs) such that (e, L) is a model of *SPEC*. (Copeland [1996], p. 338)

The relevant notion of modeling, taken from model theory, is Tarskian satisfaction. Importantly, this notion allows that both physical and mathematical structures may model a given architecture–algorithm specification. In order for an entity e to be a model, all that is required is that there be a way to use it to identify a domain of objects and a suite of relations on them. As far the modeling relation is concerned, these entities and relations may be physical, mathematical, or indeed just about anything you like. Moreover, this flexibility with respect to physical and mathematical models is not just a technical quirk. That both physical and mathematical structures may stand as models is an important part of Copeland's account, a point about which he is quite clear:

> The construction [i.e. the labelling scheme] must be applicable not only to real hardware but also to merely conceptual machines. For example, we wish to say that each action of a Turing machine is the result of its configuration (i.e. the combination of its state and the scanned symbol). Unless we intend to speak metaphorically, the phrase 'is the result of' cannot be replaced here by 'is caused by', for the machine is a purely abstract entity. (Copeland [1996], p. 341)

I submit that this account of computation is unificationist, not bifurcationist. On Copeland's view, whether and what a system computes is determined by the architecture-algorithm specifications of which it is a model. Consequently, the status and identity questions for both physical and mathematical computing systems are answered in terms of algorithm-architecture specifications.

To my knowledge, Copeland's view is perhaps the clearest illustration of unificationism in the literature. But at times other philosophers appear to flirt with unificationism, too. For instance, in some of his moods Piccinini appears to offer the mechanistic account of computation in a unificationist spirit. This account holds that computing systems are a kind of mechanism with teleological functions—or a 'functional mechanism', for short—whose function is computing.[5] At least in some places, Piccinini appears to suggest that the account applies in equal measure to both physical computing systems, such as digital computers, and mathematical computing systems, such as Turing machines:

> All paradigmatic examples of computing mechanisms, such as digital computers, calculators, Turing machines, and finite state automata ... perform digital computations. Thus, the mechanistic account properly counts all paradigmatic examples of computing mechanisms as such. (Piccinini [2015], p. 143)

> Whether abstract or concrete, Turing machines are mechanisms, subject to mechanistic explanation no more and no less than other mechanisms. (Piccinini [2010], p. 290)

> [a] similar notion of functional mechanism [to that which applies to physical computing mechanisms – A.C.] applies to computing systems that are defined purely mathematically, such as (unimplemented) Turing machines. Turing machines consist of a tape divided into squares and a processing device. The tape and processing device are explicitly defined as spatiotemporal components. They have functions (storing letters; moving along the tape; reading, erasing, and writing letters on the tape) and an organization (the processing device moves along the tape one square at a time, etc.). (Piccinini [2015], pp. 119-20)

Thus on this view, the status and identity questions are answered in mechanistic terms. Whether and what a system computes amounts to being a certain sort of functional mechanism.[6,7]

[5](Piccinini [2015]).

[6]An anonymous reviewer wonders, not unreasonably, whether this is fair to Piccinini, for in other moods he seems downright hostile to unificationism (see, for instance, (Piccinini [2015], pp. 8-9)). Yet, even if this is right, it is not clear how to reconcile this with the outlook expressed in the previously cited passages, which seem to me clearly unificationist. In any event, while I think this points towards an interesting tension in Piccinini's view, I don't propose to get bogged down in exegesis at this stage. I'll return to this point again in Section 6.1, where I argue that Piccinini is committed to unificationism by his own lights.

[7]These views raise another issue worth clarifying. Much of the literature on physical computation takes a realist attitude towards the relevant mathematics: witness Chalmers' talk of computational objects, for instance. But we shouldn't read too much into this. Given the variety and subtlety of extant nominalization strategies, such as those developed in (Burgess and Rosen [1997]), I see no obvious reason why this literature couldn't be developed in nominalist terms without distorting the major issues. Accordingly, we can set aside questions about

I think that this is enough to get a feel for the distinction. As I remarked above, it seems to me that much work on physical computation has been guided by a tacit commitment to bifurcationism. Chalmers, for instance, simply assumes bifurcationism without argument. But once we isolate the unificationism–bifurcationism contrast we can begin to see that matters are not so straightforward. Indeed, as I will argue next, there is a real worry that bifurcationism is misguided.

3 Troubles with Bifurcationism

If I'm right that bifurcationism has been endorsed only tacitly, it is perhaps unsurprising that we find little explicit argument for it. However, those who appear to endorse it oftentimes emphasize the distinctively 'mathematical' subject matter of computability theory. Sprevak ([2010], pp. 262-3) expresses this idea vividly:

> Mathematical computation theory is a branch of pure mathematics and concerns relations between mathematical structures and objects. The 'computers' it studies are mathematical entities not physical systems. According to mathematical computation theory, a Turing machine is not a physical system; it does not 'perform' a computation in the sense that a physical system does.

And a bit later:

> The computations studied in mathematical computation theory are independent of how things are in the physical world. They are independent of empirical ink-marks, they do not 'take place' in time, or depend on the physical possibility of infinitely long tapes. Computers in mathematical computation theory are mathematical entities that bear certain relations, studied by that theory, to other mathematical entities, the functions they compute ... physical entities are not the subject matter of the relevant mathematical claims. Mathematical computation theory does not say anything about physical systems.[8]

These remarks suggest the following sort of argument: (1) the mathematical theory of computation is concerned solely with mathematical entities, such as Turing machines,

fundamental mathematical ontology and focus on issues internal to the philosophies of computer and cognitive science. For convenience, I will continue to adjudicate matters in realist vocabulary.

[8]It should be noted that what Sprevak calls 'the received view' is the claim that all physical computation essentially involves representation. This is distinct from how I use the term above, to refer to implementationist theories of physical computation generally. Sprevak's received view is just one kind of implementationist view, that is, just one instance of my 'received view'.

recursive functions, and the like. But (2) these are not physical entities – they are 'independent of how things are in the physical world'. So (3) a theory of such systems is not automatically a theory of physical systems, in which case (4) bifurcationism follows.[9]

However, if this is the master argument for bifurcationism, it leaves much to be desired. One issue is that premise (1) comes perilously close to begging the question. Our topic is the extent to which the mathematical and physical theories of computation share a subject matter. To assume that the mathematical theory is concerned only with mathematical computing systems appears to assume that the status and identity questions concerning physical systems cannot be answered by that theory, which is just what we are wondering about. Another issue is that (2) presupposes a substantial view about the nature of mathematical entities, in effect that they are not physical entities of some sort or another. Yet it is not clear why one must endorse this claim in order to be a bifurcationist, for surely the unificationism–bifurcationism distinction cuts across issues of fundamental mathematical ontology. That is, it would be rather surprising if endorsing bifurcationism (much less implementationism) required one to take this particular attitude towards the relevant mathematics.

But even setting these worries aside, the crucial move from (2) to (3) is suspect. Suppose it's true that the primary subject matter of computability theory are highly idealized mathematical computing systems. That doesn't show that computability theory can't also be a theory of certain non-idealized physical systems, too. Certainly it is not clear that this reflects how computability theorists conceive of their own subject matter, for at times they certainly seem to be describing actual or possible physical devices. This outlook goes right back to Turing's foundational work on effective calculability, for Turing's focus was, in the first instance, on the sorts of problems that could be solved by a human working effectively. In this respect the situation is quite unlike that found in, for example, the more exotic branches of set theory, where there is typically no pretension that the objects of interest are in any sense physical, idealized or not.

Moreover, it seems that computability theory actually has the resources to describe physical systems directly. This can be illustrated by looking at the standard definition of a deterministic

[9]It would be unfair to attribute this argument, at least in this form, to Sprevak. But it does seem to me that this kind of reasoning lies behind bifurcationism.

finite automaton. On that definition, DFAs are five-tuples $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, $\delta : Q \times \Sigma \rightarrow Q \cup F$ is a transition function, $q_0 \in Q$ is a start state, and F is a set of final states. As Rescorla ([2014]) observes, this definition imposes no restrictions on the members of Q or F: all that is required that they be sets of states. Now, we might take the states to be purely abstract, but then again we might also take them to be states of a particular physical system. And in the latter case it's unclear why computability theory wouldn't describe the computational characteristics of physical systems. Indeed, it would be surprising if it didn't!

In light of all this, I am skeptical that bifurcationism can be motivated by an appeal to the supposedly 'mathematical' subject matter of computability theory. If we are going to endorse it, we'll need a more sophisticated argument. That's coming up. But first let's what can be said in favour of unificationism.

4 Three Arguments for Unificationism

This section argues that unificationism satisfies a variety of plausible adequacy conditions on a theory of physical computation. Of course, that a theory satisfies some adequacy conditions only counts towards endorsing it if all else is equal. I don't pretend that the considerations presented here are demonstrative. In fact, I'll eventually argue that all else is not equal, and that for other reasons unificationism should ultimately be abandoned. But, taken together, I submit that these considerations make a strong *prima facie* case for unificationism. Strong enough, at any rate, to give the bifurcationist pause.

4.1 The value of unification

Many philosophers hold that more unified theories are preferable to less unified theories. This is a familiar point, so I'll be brief.[10] Unificationist theories of computation promise to unify computation, with a twist. Normally unified scientific theories are taken to unify diverse physical phenomena. But unificationism, as a thesis about the theory of computation, would unify both physical and mathematical phenomena. I see no reason why this sort of unification

[10]See, for instance, (Kitcher [1981]; Kitcher [1989]; Swoyer [1999]).

would be any less preferable to the usual sort, in which case a unificationist theory of computation is preferable to a bifurcationist one.

4.2 Extensional adequacy

It is widely accepted that a theory of computation should be extensionally adequate.[11] Minimally, this means that it should correctly classify paradigmatic computing systems as well as paradigmatic non-computing systems. Theories which correctly classify paradigm cases, of either sort, are preferable to those which don't.

Clearly this condition depends on the choice of paradigms and anti-paradigms. Since it is unlikely that these are to be determined *a priori*, we should look to scientific practice as a rough and ready guide. And as Piccinini ([2015], p. 12) points out, computer scientific practice suggests that plausible paradigms are "digital computers, calculators, both universal and non-universal Turing machines," while plausible anti-paradigms are "planetary systems, hurricanes, and digestive systems," among many others. But notice that both mathematical computing systems, such as Turing machines, and physical computing systems, such as digital computers, feature among the paradigms. Thus it appears that an extensional adequacy requirement, coupled with this choice of paradigms, points towards a single account which captures both mathematical and physical computation. Unificationism naturally fits the bill.

4.3 Computer scientific practice

Perhaps the strongest case for unificationism comes from computer scientific practice. If that practice marks no clear distinction between the theory of computation as applied to physical versus mathematical systems, then that is some reason to accept unificationism. And indeed we find this in two places. First, the ordinary thought and talk of computer scientists marks no clear distinction between computations carried out by mathematical computing systems and physical computing systems. Computer scientists routinely apply what appear to be the same computational notions to systems of either type. Both mathematical and physical systems are routinely described as computing a function or following this or that algorithm, for instance.

[11](Ritchie and Piccinini [2019], p. 194).

Algorithm 1 Mul(x, y)

```
if $x = 0$ then
    return 0
end if
$p \leftarrow y$
while $x > 1$ do
    $p \leftarrow p + y$
    $x \leftarrow x - 1$
end while
return $p$
```

We find similar tendencies in the explanatory practice of computer science. Oftentimes the same explanations are offered in order to explain the computational features of both mathematical and physical systems. Since this seems to me the more revealing datum, I'll take a minute to gnaw on it.

To fix ideas, focus on an explanation of a system's behavior in terms of the function it computes. And consider a mathematical computing system, such as an abstract register machine, that computes multiplication by repeated addition.[12] Let's suppose that the explanation why the device outputs $x \times y$ when given x and y is that it follows something like Algorithm 1.[13] This simple explanation is familiar from elementary computability theory, and others like it are readily come by. The point to notice is that this pattern of explanation may equally well be applied to a contemporary digital computer. Just like an abstract register machine, contemporary digital computers manipulate strings of digits in finite memory locations. Given such a device, we can explain that it too outputs $x \times y$ when given x and y because it follows Algorithm 1.

In neither case does the explanation advert to the character of the items manipulated. In the register machine's case the strings are mathematical objects, perhaps defined in terms of pure sets, whereas in the digital computer's case the strings are physical, perhaps realized as voltage levels in silicone. But these differences are irrelevant to the explanation why the devices multiply; what matters, from the perspective of computing multiplication, is that the algorithm takes x and y to $x \times y$. Whether this happens in sets or silicone is simply beside the point.

Of course, computer scientists are not just concerned to explain why a system computes

[12]See (Cutland [1980]) for such devices.

[13]Strictly speaking, I should say that the device outputs a numeral for $x \times y$ given numerals denoting x and y.

some function. They also wish to explain why certain functions are uncomputable in principle, or are at least uncomputable in a feasible amount of time. Here too we find patterns of explanation applicable to both mathematical and physical systems.

A vivid example comes from work on performance bounds for comparison sorting algorithms.[14] It is known that any comparison sorting algorithm must make approximately n lg n comparisons to sort a list of length n. [15] A standard explanation notes that comparison sorting is equivalent to the task of guessing which permutation of a list one is given, if one can only 'see' two elements at a time. There are $n!$ permutations of an n-element list, and each guess, or comparison, eliminates half of the remaining permutations from consideration. The guessing procedure can be represented by a binary tree whose internal nodes are comparisons, and whose leaves are permutations. A path from root to leaf corresponds to a sequence of comparisons—a sorting—in which case a lower bound on the number of comparisons required to sort the list corresponds to the minimum height of such a binary tree, and this is more or less n lg n.

Details aside, the important point is again that this explanation can be applied to any comparison sorting system, physical or mathematical. Anything that sorts by comparison must compare list members, and differences in the nature of the items compared are irrelevant as far as the explanation is concerned. That these differences are irrelevant suggest that unificationism, or so the thought goes.

So far, then, I've argued that the case for bifurcationism is uncompelling, and that the alternative, unificationism, is attractive. To the extent that the received, implementationist view tacitly endorses bifurcationism, this argument threatens it as well. Nonetheless, as I argue next, unificationist theories face a serious challenge of their own.

[14]Given an unsorted list $l_1, l_2, ..., l_n$ of items and a linear order $<$ on them, a comparison algorithm sorts by checking whether $l_i < l_j$ holds, and manipulating the list depending on the outcome.

[15]See (Cormen *et al.* [2001], pp. 166-7) and (Knuth [1998], 180-2).

5 No Unificationism without Trivialization

One familiar objection to computational explanation is that physical computation is trivial.[16] Triviality follows from two claims. The first is pancomputationalism (sometimes called universal realization), the claim that at one and the same time, every (or at least many) physical system performs every (or at least many) computations. The second claim links pancomputationalism to computational explanation, and holds that computational explanation succeeds only if a unique, or at any rate a select few, computations are performed at a time. Now, we might worry about one or another of these moves, but here I'll assume that this pattern of argument is valid and that triviality ought to be avoided.[17]

For the most part triviality arguments have been deployed against the received view.[18] And with the received view in their sights, these arguments purport to show that every physical system implements every, or at least many, computations.[19] I mention this only so that I can distinguish it from what I'm up to. The question I'm interested in is whether, and to what extent, triviality worries emerge for unificationism in particular. My claim is that they do, and that they do so in a particularly insidious form.

Here's the problem in brief. Unificationism requires that the account of computation applied to mathematical computing systems also apply to physical computing systems. Let's say that an account that satisfies this requirement 'accommodates' unificationism. Now, it seems clear that a theory which accommodates unificationism must characterize computation only in terms of characteristics shared by different kinds of computing systems. If it didn't, it's hard to see how it would be a genuinely unificationist theory in the first place. But here's the rub: it appears that there is no account of computation which at once appeals only to shared

[16]See (Putnam [1987]) and (Searle [1992]) for classic triviality arguments. (Sprevak [2019]) is a recent survey.

[17]This way of formulating triviality issues ignores a few distinctions sometimes made between different kinds of pancomputationalism (Piccinini [2015], ch. 4). While it is surely problematic if a given system performs every computation, it is perhaps tolerable if only a select few computations are performed at once. For instance, a given digital circuit may reasonably be said to compute both logical AND and logical OR, depending how one looks at it (Dewhurst [2016]). However, this observation is of no help to the unificationist. Even if we can tolerate some degree of pancomputationalism, my claim will be that unificationism entails an unacceptably strong version of it. Thanks to an anonymous referee for urging me to clarify this point.

[18](Copeland [1996]) is an exception.

[19]Most implementationist attempts to avoid triviality target the first, pancomputationalist, premise. (Chalmers [1996]) is representative of this maneuver. Other strategies, such as that in (Schweizer [2019]), accept pancomputationalism but defuse the problem by denying the second claim.

characteristics, but which is also sufficient to block triviality. One can accommodate unificationism, or avoid triviality, but not both.

To argue for this in any detail we must get into the weeds on various accounts of computation. That happens in the next section. But to get a feel for the problem, the rest of this section sketches a triviality result for a simple structuralist theory of computation. Of course, few endorse this account any more.[20] But it nicely brings out the threat posed by triviality arguments, and the bind in which unificationists find themselves.

On this theory, to compute is just to have a certain structure. This account straightforwardly accommodates unificationism, for it is plausible to think that both mathematical and physical systems have structural features of the right sort. But the problem is that structure is cheap, which leads directly to pancomputationalism and hence triviality.

To illustrate, consider a variation on Putnam's classic argument. We'll describe computations in terms of finite state automata (FSAs). Given the structuralist account under consideration, we can think of FSA computations as describing computational structures. For instance, the two state automaton which moves from one state to another and then halts describes a computational structure, instantiation of which is a matter of having two parts (states, for instance) which stand in some sequential relationship.

We wish to establish that every physical system performs every FSA computation. This amounts to showing that every physical system goes through the state-transitions described by every FSA. So consider the particular FSA M with two states, A and B, which proceeds through the following state transitions: $A \rightarrow B \rightarrow A \rightarrow B \rightarrow A$. We define a 'maximal state' of some physical object O at a time to be its total intrinsic state at that time. Next, we consider five sequential maximal states of O: $M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4$. The idea is to define two new states of O, A_O and B_O, so that these new states evolve in a way that corresponds to the state transitions described by M.

The following will do: $A_O = M_0 \vee M_2 \vee M_4$; $B_O = M_1 \vee M_3$. Thus O proceeds through a series of state transitions capture by M: $A_O \rightarrow B_O \rightarrow A_O \rightarrow B_O \rightarrow A_O$. So O performs M's computation. But since this kind of construction can be had for any FSA and any physical

[20](Schweizer [2019]) is a recent exception.

entity, the argument generalizes and triviality follows.

Of course, nothing about this construction is novel. The important point is how it interacts with the background theory of computation: while the structuralist account smoothly accommodates unificationism, it does so at the cost of triviality. The question I take up next is whether some other account can be found which both accommodates unificationism and resists trivialization.

6 Four Accounts of Computation

The implementationist literature contains a variety of strategies for avoiding triviality, and this section explores whether any of them can be appropriated in the service of unificationism. I focus on accounts which hold that computation essentially involves causation, counterfactual dependencies, teleological functions, or representation, respectively. In each case I argue that the proposal fails to avoid the problem outlined in the last section. Some fail to avoid triviality, while others accommodate unificationism only at great cost. The lesson I draw is that the prospects for unificationism seem grim.

I should make a couple more clarificatory points before getting down to business. First, my aim is not to show the impossibility of a unificationist theory of computation. I see no way to establish such a strong result. Rather, and more modestly, my aim is to shift the burden onto the unificationist to articulate a theory that at once accommodates unificationism and avoids triviality. The proposals I investigate below seem to me to be the most plausible options, but they are not the only ones available. I leave open the question whether some other account can do the job. But to the extent that the options canvassed here stand the best chance of avoiding triviality, unificationism is in trouble.

Second, it is worth highlighting how the responses to triviality investigated below differ from those employed by the received, implementationist view. Because they endorse bifurcationism, proponents of implementationism are under no pressure to employ constraints that apply to mathematical computations. For them, the problem is to just say which physical objects are allowed to figure in the implementation relation. Accordingly, they are free to impose constraints on the physical side which may make no sense in the context of mathematical computation. The unificationist, by contrast, has an altogether different task to pull off. They must impose constraints that apply equally to physical and mathematical computing systems. And this, we will see, is the source of their troubles.

6.1 The causal-mechanical account

A natural first response to triviality requires that the computational structure of a physical system track its causal structure.[21] A related suggestion is found in the mechanistic account of computation, which holds that computing systems are mechanisms.[22] Given their close resemblance, we can treat these proposals together. On this view, to have a computational property is just a matter of having a certain causal-mechanical structure. The reason why an arbitrary physical object doesn't perform every computation is that the computational 'states' cited by triviality arguments are not causally related, or are not genuine mechanistic components of the system. Pancomputationalism (hence triviality) fails because 'pan-causation' and 'pan-mechanism' fail, or so the thought goes.

But the question is whether the causal-mechanical account accommodates unificationism. And it seems not, at least on the face of it. Nowadays Turing machines are defined as consistent sets of 5-tuples $(Q_i, \sigma_1, \sigma_2, m, Q_j)$ where Q_i and Q_j are states, σ_1 and σ_2 are letters in the machine's alphabet, and m is an instruction for moving the read/write head. Sets of 5-tuples fix a set of state-space transitions, which capture the behavior of the machine at every possible stage of operation. Now, whether we identify Turing machines with their sets of 5-tuples or with a set of state-space transitions, each of these items is constructed, or at least constructible, out of pure sets. Such pure set theoretic entities are abstract, non-causal, and non-spatiotemporal. But given this view of Turing machines it is hard to see how the causal-mechanical theory accommodates unificationism, for the simple reason that Turing machines, so construed, lack causal-mechanical characteristics altogether.

One response to this is to throw up one's hands and abandon unificationism. Piccinini takes this tack in his more pessimistic moods, writing that he is "after an account of computation in

[21] See (Chalmers [1996]), (Godfrey-Smith [2009]), and (Scheutz [2001]) for proposals in this vein.

[22] (Milkowski [2011]); (Piccinini [2015]). Depending on one's views about causation and mechanism, the mechanistic conception might collapse into the causal conception; see (Glennan [2017]) and (Woodward [2013]) for discussion.

the physical world" and that “[i]f Turing machines and other mathematically defined computational entities are abstract objects ... they fall outside the scope of my account" (Piccinini [2015], p. 9). But it is unclear how to square this with his later claim, in the very same book, that the mechanistic account correctly counts Turing machines as computing mechanisms. Moreover, it seems that this move is unavailable to Piccinini by his own lights, for “[t]he primary aim of the mechanistic account is doing justice to the practice of computer scientists and engineers” (Piccinini [2015], p. 118). As I argued in Section 4, that practice supports unificationism, at least *prima facie*. So this sort of response is unavailable to anyone impressed by computer scientific practice, as Piccinini appears to be.

A more radical response denies that Turing machines are purely mathematical entities. Instead, we should regard them as idealized physical entities, which retain some, but not all, of the physical properties of their unidealized counterparts. Copeland and Shagrir call this view ‘Turing–machine realism’. As they explain:

> Turing-machine realism recognizes an ontological level lying between the realization level and the level of pure-mathematical ontology. We term this the level of notional machines. At this level are to be found notional or idealized machines that are rich with spatiotemporality and causality.(Copeland and Shagrir [2011], p. 234)[23]

Such a view could allow the unificationist to endorse a causal-mechanical theory of computation, unifying computation in causal-mechanical terms.

Yet it is doubtful that this maneuver will succeed. One issue concerns the status of the idealized machines, for it is far from clear how we are to make sense of their ‘in between’ ontological status. Are they abstract objects, or mental entities, or what?[24] This question demands an answer before we can legitimately claim that Turing machines ’have’ causal features.

However, even if we can make sense of notional machines, there appears to be a general problem which undercuts any attempt to recharacterize Turing machines, spatiotemporally or otherwise. The trouble is that the computational explanations surveyed in Section 4 apply not just to physical systems, but also to purely mathematical computing systems. Those systems,

[23]It should be noted that Copeland and Shagrir don’t endorse Turing–machine realism, but merely flag it as a live option.

[24]See (Thomson-Jones [2010]) for critical discussion of a related proposal found in (Giere [1988]).

such as the pure set-theoretic counterparts of Turing machines, haven't gone anywhere. And if unificationists take this practice seriously, which they presumably do, they must furnish a theory which accounts for Turing machines in their pure set-theoretic guise too: adding a new kind entity 'in between' physical systems and the level of pure mathematics doesn't discharge that task. For this reason it seems to me that Turing machine realism is a dead end, and that we should explore other options.

6.2 The counterfactual account

Copeland ([1996]) explicitly disavows a causal-mechanical theory of computation in light of the difficulties raised in the last section. He complains that causal-mechanical theories are "intolerably narrow", because they don't capture abstract Turing machine computations.[25] But despite this, he maintains that causal-mechanical accounts do capture something important about computation, namely that later computational states counterfactually depend upon earlier states. This is true even for Turing machines, for there certainly seems to be some sense in which a Turing machine's later states depend on its earlier states (plus tape contents).

One immediate question is how we should understand the notion of counterfactual dependence Copeland has in mind. The suggestion seems to be that it is a generalization of causal dependence. But whereas causal dependence relations holds only between spatio–temporal particulars, such as physical events, counterfactual dependence relations may hold between either spatio–temporal particulars on the one hand, or between non-spatiotemporal particulars, such as the 'events' in a Turing machine computation, on the other.

Now, this doesn't clear everything up. In what sense do Turing machine operations involve 'events', for instance? But Copeland thinks we can skirt these sorts of questions. The target notion of counterfactual dependence is logically equivalent to causal dependence, in the sense that both relations underwrite counterfactual assertions about the behavior of computing systems. So, to determine whether the target dependence relation obtains it is enough to determine whether a system satisfies certain 'computational counterfactuals'. Thus, on this

[25](Copeland [1996], p. 353). See also Chrisley ([1994], p. 408), who considers but ultimately rejects a similar thought.

account whether a system computes reduces to the question whether the system satisfies certain counterfactual assertions about its behavior.

How does this avoid triviality? We already remarked that most of the physical 'states' appealed to by the triviality argument are not causally related; by similar reasoning it is not hard to see that they will fail to support counterfactuals as well. For example, recalling the construction from above, the following is plausibly false of the physical object O:

(1) If it were the case that O was in state A_O, it would transition into state B_O.

In which case O doesn't compute, as desired. And to see that the proposal correctly captures mathematical computations, consider an abstract register machine M with an 'instruction register' whose contents contain a code for the next operation to be performed. M operates by reading the contents of the instruction register and then performing the encoded operation. The device is defined to be sensitive to contents of the instruction register, so that differences in register contents make for differences in the operations performed. Accordingly, M satisfies computational counterfactuals of the form:

(2) If it were the case that at some time M's instruction register held such-and-such instruction, M would perform so-and-so operation.

Thus the counterfactual proposal accommodates unificationism while avoiding triviality.

But it seems to me that this proposal runs up against two difficulties. First, Copeland's own version succumbs to a revenge triviality argument. Standard triviality arguments can be tweaked to show that every sufficiently complex physical system computes every computable function while satisfying appropriate computational counterfactuals.[26] So Copeland's account fares no better than the simple structuralist account from before.

And it gets worse. There are reasons to be pessimistic that any other counterfactual account will be forthcoming any time soon. The trouble is that an account of computational

[26]I develop one such argument in Curtis-Trudel (unpublished). In brief, the trouble is that Copeland supplies a standard satisfaction-based semantics for his counterfactual conditionals. Absent further constraints on satisfaction—and Copeland doesn't supply any that even remotely do the trick—it's not hard to devise deviant physical models of these counterfactuals. Moreover, on reflection this isn't even all that surprising, since completeness guarantees the existence of models, and it's not hard to manipulate particular physical models in order to come up with arbitrarily many distinct physical interpretations of the counterfactuals.

counterfactuals general enough to apply to the operations of abstract computing systems must also be an account of counterpossibles, and it is not at all clear how to understand counterpossible assertions about the behavior of mathematical computing systems.

To illustrate, consider a Turing machine M described by the instructions:

$$(Q_0, 1, 0, R, Q_0); (Q_1, 1, 1, R, Q_1)$$

whose operation is to erase a contiguous block of 1s and then halt. In line with the counterfactual account, we presumably want M to satisfy counterfactuals such as

(3) If M had been in state Q_1 and read 1, it would write 1 and go into state Q_1.

However, if we make the standard assumption that M starts in state Q_0, there is no possible sequence of state transitions which will take it into state Q_1. So the antecedent of (3) is impossible in a strong mathematical sense, making (3) not merely counterfactual, but counterpossible.

This is fine as far as it goes, but how should we understand (3)? It seems inappropriate to endorse the view that all such counterpossibles are vacuously true.[27] This is because we presumably wish to distinguish (3), which is intuitively true, from (4), which is intuitively false:

(4) If M had been in state Q_1 and read 1, it would write 0 and go into state Q_1.

Yet we cannot capture the apparent truth of (3) and falsity of (4) in the usual possible worlds semantics, for the familiar reason that there is no possible world in which M is in state Q_1.

Of course, the unificationist might make various maneuvers at this point. One is to introduce impossible worlds to distinguish between (3) and (4).[28] Such moves are not unheard of, but it is no understatement to say that impossible worlds are problematic and perplexing. It is far from clear whether a counterfactual conception of computation can be worked out in such a framework. But I think it is enough for the present, burden-shifting argument, to notice

[27](Lewis [1973]; Williamson [2007]).

[28](Berto and Jago [2019]).

that if this is the road to non-triviality, then friends of unificationism have their work cut out for them. One can be forgiven for exploring other routes.

6.3 The teleological account

Some versions of the mechanistic account hold that computing systems are not just mechanisms, but mechanisms with teleological functions. While the causal-mechanical aspects of this view sits poorly with unificationism, perhaps teleological functions alone suffice to avoid triviality. According to the view explored next, possession of certain teleological functions is necessary for computing and sufficient to avoid triviality.[29]

One task is to specify the notion of teleological function at play. I will focus on the account developed in (Piccinini [2015], ch. 6), since it is apparently designed to apply to both digital computers and Turing machines. A first-pass account is couched in terms of objective goals, which include things such as survival and reproduction:

> A teleological function is a stable contribution to an objective goal of organisms by a trait or an artifact of the organisms.

But even if we can make sense of the thought that Turing machine operations are a 'trait' or 'artifact' of an organism, it is frankly incredible to think that such operations stably contribute to anyone's survival and reproduction, even, alas, the professional computer scientist's.

Perhaps recognizing this, Piccinini ([2015], p. 116) broadens the account to include 'subjective goals'. These include desires, among other things, so that something contributes to a subjective goal if it stably contributes to the satisfaction some desire:

> A teleological function (generalized) is a stable contribution to a goal (objective or subjective) of organisms by either a trait or an artifact of the organisms.

So while Turing machine operations might not stably contribute to computer scientists' survival and reproduction, it's not completely unreasonable to think that they might help to satisfy certain of their desires for example, their desires for knowledge about computable sets, Turing degrees, and so on.

[29]An anonymous reviewer asks whether anyone nowadays would really endorse a teleological view of mathematical computing systems. The short answer is apparently 'yes', for Piccinini appears to; see (Piccinini [2015], ch. 7) and the passages quoted in Section 2.

But it seems to me that this account faces a number of problems. One is that there are not enough desires to go around. There are only finitely many actual desires, but denumerably many Turing machines. If Turing machines compute only by virtue of satisfying some actual desire, then there will be denumerably many Turing machines which, whatever else they do, don't compute. Surely a bad result!

We might try to get around this by appealing to the desires of possible agents, so that a Turing machine computes if there is some possible agent some of whose desires would be stably satisfied by that machine's operations. Now the trouble is that the suggestion overshoots. There are many possible agents—enough, let us grant, that every Turing machine's operations satisfy some possible agent's desires. If stably satisfying the desire of a possible agent is sufficient to have a teleological function, then given the abundance of possible agents, any object can have a teleological function. In particular, now paradigmatic non-computing systems such as rocks, walls, and pails of water will have teleological functions in this generalized sense, which is just what we want to avoid.[30]

The proper response to this worry, it seems to me, is to grant that generalized teleological functions are in principle universally realizable, but nevertheless deny that this poses a problem for a theory of physical computation in particular. A system computes some mathematical function f only if computing f is one of its teleological functions. As long as the conditions on having computing f as a teleological function are stringent enough, an abundance of generalized teleological functions does not threaten computational explanation. Triviality is avoided not because generalized teleological functions are hard to come by, as it were, but because the particular teleological function of computing f is.

But whether this maneuver succeeds turns crucially on just what it is to compute f. A notion couched in terms of formal string manipulations plausibly applies to Turing machines, but encounters a problem familiar from the structuralist account: how do we determine which parts of a system are strings? There are many ways to arbitrarily identify parts of a physical entity as strings, and it is straightforward to reverse-engineer deviant interpretations according to which computes every function. So we're back to where we started. Moreover, we cannot

[30]This is more or less just Searle's ([1992]) old complaint that psychologistic accounts of computation lead to pancomputationalism.

impose causal-mechanical constraints on string identification, for in this case we're back to the problems encountered with the causal-mechanical account. So, absent some other account of what it is to compute f, it seems the teleological account fares no better than the others.

6.4 The representational account

The last account I will consider holds that to compute is, at least in part, a matter of having certain representational features.[31] The version considered here takes this to involve representing entities in some external domain, such as a system's distal environment or a set of mathematical objects.

One point in favour of this account is that it is plausible to think that representational notions feature in the mathematical theory of computation.[32] Turing machines directly compute string-theoretic functions. Computation over non-linguistic domains is characterized relative to a mapping from linguistic entities to non-linguistic entities. When such a mapping is in place, it is natural to regard the linguistic entities as representing the non-linguistic entities. In the case of computation over the naturals, for instance, strings of digits are taken to represent natural numbers.

However, the notion of representation that emerges from the mathematical theory of computation is quite thin: all that is required for a Turing machine to represent is that there be an effective mapping from its language to some non-linguistic domain.[33] Such mappings are abundant; indeed, if the linguistic and non-linguistic domains are denumerable, there will be denumerably many. But plainly pancomputationalism is unavoidable if this is the notion of representation underwriting physical computation, for there are simply too many mappings from physical systems and their states to external entities.

At this point we could cast about for a more stringent account of representation, so that physical systems stand in determinate representational relations to comparably few entities. A causal constraint is natural but unavailable for familiar reasons: a causal notion of representation will not obviously apply to abstract Turing machines, and at any rate there

[31]A number of philosophers have endorsed accounts of computational implementation which are either partly or wholly representational; see (Rescorla [2014]), (Sprevak [2010]), and (Shagrir [2018]).

[32](Rescorla [2015]).

[33](Shapiro [1982]).

seems to be little sense in which a microprocessor's states stand in causal relations to abstract numbers. Similar points apply to other robust conceptions of representation too, whether they appeal to functional roles, teleological functions, or whatever.[34]

A different option is to take representational notions as primitive.[35] From this perspective, we shouldn't attempt to characterize representational notions in non-representational terms. Rather, representational notions earn their keep because they play an indispensable explanatory role in our scientific theories.

I'll make just one remark about this strategy. If attributions of representational properties are warranted in virtue of the explanatory work they do, then an immediate problem is that there appear to be physical computing systems, such as simple embedded systems in ordinary appliances, whose behavior and character can be exhaustively explained without appeal to representational notions.[36] If this is right, then by the primitivist's lights we have no reason to ascribe representational properties to them. But then it begins to look very difficult to use representational notions to solve the unificationist's woes, for we re–encounter the problem that not every computing system can be brought under a single (primitive representational) rubric. Once again, there seems to be no way to hold on to both unificationism and a substantive theory of physical computation.

6.5 Summary

The counterfactual and representational accounts accommodate unificationism but at the cost of triviality. The causal-mechanical and teleological accounts might avoid triviality, but don't accommodate unificationism. Perhaps the unificationist has another trick up their sleeve, but I think by now we've seen enough. It is doubtful that a single theory can supply satisfactory answers to the status and identity questions for both physical and mathematical machines. Unificationism has got to go. Instead, we should be bifurcationists about computation.

[34]Note that I am not asking for a naturalization of representation, whatever that amounts to, but a guarantee that representation is non-trivial. Such a guarantee might come on the heels of a naturalistic account of representation, but then again it might not. The worry is that absent an appeal to these other notions, no such guarantee is forthcoming. Naturalization is a separate issue.

[35]See (Burge [2010]), (Rescorla [2013], pp. 692-3).

[36](Rescorla [2014]). See also (Piccinini [2008]) for relevant discussion.

7 Life After Bifurcation

Given bifurcationism, should we go on to be implementationists too? One point to notice is that bifurcationism alone does not tell us that physical computation should be characterized in terms of implementation. In fact, it is neutral between three different views: the received view, which uses the mathematical theory to develop the physical theory; one which reverses this order of things; and a third which denies that there should be any deep relationship between the two. Hence further argument is needed before we endorse the received view over these latter alternatives. The issues involved are complex, and I don't propose to settle the matter here. Instead, I will briefly sketch out how each of these options might be pursued. My aim is to say just enough to show that the latter two are genuine alternatives to the first.[37]

The first option has obvious advantages. We already have a well defined mathematical notion of computation, and the task is to apply it to physical systems. Moreover, while such theories face triviality worries of their own, they have a competitive advantage over unificationist alternatives, because they needn't apply to both physical and mathematical computing systems. Rather, they are free to impose whatever constraints they like on physical computation, even if those constraints make no sense in the context of mathematical computation. Of course, this is not to say that triviality objections can be definitively solved by these techniques. But it is to say that such a view stands a better chance of avoiding triviality than unificationist competitors.

What of the second option? It has received comparably little attention in the literature on physical computation.[38] This is somewhat striking, since it closely tracks the historical development of computability theory. As we know, the task facing Church, Gödel, Turing, and others in the 1930's was to give a precise mathematical statement of the intuitive idea of a worker proceeding effectively. Turing's landmark characterization proceeded, in the first instance, by careful reflection on the abilities of actual human workers.[39] Historically, at least,

[37]Clearly the full story will have to include an account of the scientific practice discussed in Section 4. Quite likely the different accounts will offer different takes on that practice.

[38]But see (Joslin [2006]) and (Cleland [2001]; Cleland [2002]) for partial moves in this direction. Could Piccinini's mechanistic account fit in here? Perhaps, but it seems he cannot shake the 'mathematics-first' attitude. For instance: "the mathematical notion of digital computation is clear enough. The remaining question is how to apply it to concrete mechanisms" (Piccinini [2015], p. 127).

[39](Sieg [2009]).

mathematical notions of computation emerged out of consideration of computation in actual physical systems, not the other way around.

We might work this observation up into a theory of computation in different ways, and I'll just mention one. On the envisioned theory, we begin with a notion of physical computation—of a human working effectively, say—and idealize and refine it to arrive at a mathematical notion of computation—of a Turing machine computation, say—suitable for mathematical work. Further idealization may generalize and refine this first-pass mathematical notion, delivering quite general mathematical computing objects.[40] Crucially, these mathematical objects count as computational only to the extent that they are idealizations or abstractions of physical computing systems.

One advantage of this approach is that it straightforwardly avoids triviality. It is plausible, at least *prima facie*, that our pre-theoretical notions of physical computation are non-trivial. Plainly not every physical system is a human working effectively, for instance. But because notions of mathematical computation emerge from prior notions of physical computation, there is no question of 'applying' them to the physical world, in which case there is no threat of over application either.

The third option departs more radically from the others. On this approach neither the mathematical nor the physical theories of computation should be developed in terms of the other. Of course, this is not to say that these theories are completely unrelated, or that they are different subject matters. It is just to deny that either is more fundamental than the other.

This approach sits naturally with the idea that mathematical computations are models of physical ones, in something like the way that mathematical models encountered elsewhere in science may model certain physical systems.[41] Mathematical computations inform us about physical ones to the extent that one resembles the other. But to hold this is not to hold that the theory of physical computation should be developed by appropriating or applying notions from the mathematical theory, any more than to hold that the fact that a physical system can be described by some real function means that the fundamental theory of such systems should

[40]Such as *k*-graph machines (Sieg and Byrnes [1996]) or abstract iterators (Moschovakis [1998]).

[41]In fact matters are more complicated than this, since we can also use a physical system to model a mathematical computation. See (Fletcher [2018]) for some relevant discussion. Thanks to an anonymous referee for flagging this issue.

take its lead from, say, real analysis. In general we don't take models to ground the features of the systems which they model; nor, by the same token, do we take the modeled systems to ground the features of their models. Instead, models are useful tools for investigation in certain circumstances and certain respects, but bear no more theoretical weight than this.

Here too it is hard to see how triviality would be an issue, for, like the second view, it doesn't look to ground the physical theory in the mathematical theory. Thus to the extent that antecedent notions of physical computation are non-trivial, and plausibly they are (for why else would we think triviality is a problem?), there is again no worry about over-applying the mathematics.

This, at any rate, is enough to get a feel for the alternatives. Plainly much more needs to be done to fill in the details. This remains on the agenda for further work, and I have no firm conclusions to offer at this point. By way of closing, I will instead note that, absent any suitable exploration of these alternatives, it is premature to endorse the received view. Whether we need a theory of implementation remains to be seen.

Acknowledgements

I would like to thank Stewart Shapiro, Richard Samuels, and Chris Pincock for invaluable feedback and support throughout the writing of this paper. Additional thanks go to Ethan Brauer, Steven Dalglish, Preston Lennon, and Damon Stanley for helpful comments and discussion. Early versions of this paper were presented to audiences at the 2019 Ohio State Fink Ceremony and the 2019 meeting of the Society for Exact Philosophy. I wish to thank the members of each audience for their questions. Finally, I wish to thank three anonymous referees whose insightful comments and suggestions significantly improved the final version.

Department of Philosophy
The Ohio State University
350 University Hall, 230 N. Oval Mall
Columbus, Ohio, 43210 USA
curtistrudel.1@osu.edu

References

Berto, F. and Jago, M. [2019]: *Impossible Worlds*, Oxford: Oxford University Press.

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

Burgess, J. P. and Rosen, G. [1997]: *A Subject with No Object: Strategies for Nominalistic Interpretation of Mathematics*, Oxford: Oxford University Press.

Chalmers, D. J. [1994]: 'On implementing a computation', *Minds and Machines*, **4**(4), pp. 391–402.

Chalmers, D. J. [1996]: 'Does a Rock Implement Every Finite-State Automaton?', *Synthese*, **108**(3), pp. 309–33.

Chalmers, D. J. [2011]: 'A Computational Foundation for the Study of Cognition', *Journal of Cognitive Science*, **12**, pp. 323–357.

Chrisley, R. L. [1994]: 'Why everything doesn't realize every computation', *Minds and Machines*, **4**(4), pp. 403–20.

Cleland, C. E. [2001]: 'Recipes, Algorithms, and Programs', *Minds and Machines*, **11**, pp. 219–237.

Cleland, C. E. [2002]: 'On Effective Procedures', *Minds and Machines*, **12**(2), pp. 159–179.

Copeland, B. J. [1996]: 'What is Computation?', *Synthese*, **108**, pp. 335–359.

Copeland, B. J. and Shagrir, O. [2011]: 'Do Accelerating Turing Machines Compute the Uncomputable?', *Minds and Machines*, **21**(2), pp. 221–239.

Cormen, T. H., Stein, C., Rivest, R. L. and Leiserson, C. E. [2001]: *Introduction to Algorithms*, Cambridge, MA: McGraw-Hill, 2nd edition.

Curtis-Trudel, A. [unpublished]: 'Counterfactuals for Computation', .

Cutland, N. [1980]: *Computability: An Introduction to Recursive Function Theory*, Cambridge: Cambridge University Press.

Dewhurst, J. [2016]: ‘Individuation without Representation’, *The British Journal for the Philosophy of Science*, **69**(1), pp. 103–116.

Egan, F. [2010]: ‘Computational models: a modest role for content’, *Studies in History and Philosophy of Science Part A*, **41**(3), pp. 253–259.

Fletcher, S. C. [2018]: ‘Computers in Abstraction/Representation Theory’, *Minds and Machines*, **28**(3), pp. 445–463.

Giere, R. N. [1988]: *Explaining Science: A Cognitive Approach*, Chicago: University of Chicago Press.

Glennan, S. [2017]: *The New Mechanical Philosophy*, Oxford: Oxford University Press.

Godfrey-Smith, P. [2009]: ‘Triviality arguments against functionalism’, *Philosophical Studies*, **145**(2), pp. 273 – 295.

Joslin, D. [2006]: ‘Real realization: Dennett’s real patterns versus Putnam’s ubiquitous automata’, *Minds and Machines*, **16**(1), pp. 29 – 41.

Kitcher, P. [1981]: ‘Explanatory unification’, *Philosophy of Science*, **48**(4), pp. 507–531.

Kitcher, P. [1989]: ‘Explanatory unification and the causal structure of the world’, in P. Kitcher and W. Salmon (*eds*), *Scientific Explanation*, vol. 8, Minneapolis: University of Minnesota Press, pp. 410–505.

Knuth, D. [1998]: *The Art of Computer Programming 3: Sorting and Searching*, Reading, Mass.: Addison-Wesley, second edition.

Lewis, D. K. [1973]: *Counterfactuals*, Malden, MA: Blackwell.

Matthews, R. J. and Dresner, E. [2017]: ‘Measurement and Computational Skepticism’, *Nous*, **51**(4), pp. 832–854.

Milkowski, M. [2011]: ‘Beyond Formal Structure: A Mechanistic Perspective on Computation and Implementation’, *Journal of Cognitive Science*, **12**(4), pp. 361–383.

Millhouse, T. [2019]: 'A Simplicity Criterion for Physical Computation', *The British Journal for the Philosophy of Science*, **70**(1), pp. 153 – 178.

Moschovakis, Y. N. [1998]: 'On Founding the Theory of Algorithms', in H. G. Dales and G. Oliveri (*eds*), *Truth in Mathematics*, Clarendon, Oxford: Oxford University Press, pp. 71–102.

Piccinini, G. [2008]: 'Computation without Representation', *Philosophical Studies*, **137**(2), pp. 205–241.

Piccinini, G. [2010]: 'The Mind as Neural Software? Understanding Functionalism, Computationalism, and Computational Functionalism', *Philosophy and Phenomenological Research*, **81**(2), pp. 269–311.

Piccinini, G. [2015]: *Physical Computation: A Mechanistic Account*, Oxford, UK: Oxford University Press.

Putnam, H. [1987]: *Representation and Reality*, Cambridge, MA: MIT Press.

Rapaport, W. J. [1999]: 'Implementation is Semantic Interpretation', *The Monist*, **82**(1), pp. 109–130.

Rescorla, M. [2013]: 'Against Structuralist Theories of Computational Implementation', *The British Journal for the Philosophy of Science*, **64**(4), pp. 681–707.

Rescorla, M. [2014]: 'A theory of computational implementation', *Synthese*, **191**(6), pp. 1277–1307.

Rescorla, M. [2015]: 'The Representational Foundations of Computation', *Philosophia Mathematica*, **23**(3), pp. 338–366.

Rescorla, M. [2017]: 'From Ockham to Turing – and Back Again', in J. Floyd and A. Bokulich (*eds*), *Philosophical Explorations of the Legacy of Alan Turing: Turing 100*, Cham: Springer International Publishing, pp. 279–304.

Ritchie, J. B. and Piccinini, G. [2019]: ‘Computational Implementation’, in M. Sprevak and M. Colombo (*eds*), *Routledge Handbook of the Computational Mind*, London: Routledge, pp. 192 – 204.

Scheutz, M. [2001]: ‘Computational versus Causal Complexity’, *Minds and Machines*, **11**(4), pp. 543–566.

Schweizer, P. [2019]: ‘Triviality Arguments Reconsidered’, *Minds and Machines*, **29**, pp. 287 – 308.

Searle, J. R. [1992]: *The Rediscovery of the Mind*, Cambridge, MA: MIT Press.

Shagrir, O. [2018]: ‘In defense of the semantic view of computation’, *Synthese*. <10.1007/s11229-018-01921-z>

Shapiro, S. [1982]: ‘Acceptable notation’, *Notre Dame Journal of Formal Logic*, **23**(1), pp. 14–20.

Sieg, W. [2009]: ‘On Computability’, in A. Irvine (*ed.*), *The Philosophy of Mathematics*, Burlington, MA: North-Holland Elsevier, pp. 535–630.

Sieg, W. and Byrnes, J. [1996]: ‘K-graph machines: generalizing Turing’s machines and arguments’, in P. Hájek (*ed.*), *Gödel ’96: Logical foundations of mathematics, computer science and physics*, vol. 6, Berlin: Springer-Verlag, pp. 98–119.

Sprevak, M. [2010]: ‘Computation, individuation, and the received view on representation’, *Studies in History and Philosophy of Science Part A*, **41**(3), pp. 260–270.

Sprevak, M. [2019]: ‘Triviality arguments about computational implementation’, in M. Sprevak and M. Colombo (*eds*), *Routledge Handbook of the Computational Mind*, London: Routledge, pp. 175 – 191.

Swoyer, C. [1999]: ‘How ontology might be possible: Explanation and inference in metaphysics’, *Midwest Studies in Philosophy*, **23**(1), pp. 100–131.

Thomson-Jones, M. [2010]: ‘Missing systems and the face value practice’, *Synthese*, **172**(2), pp. 283–299.

Williamson, T. [2007]: *The Philosophy of Philosophy*, Malden, MA: Wiley-Blackwell.

Woodward, J. [2013]: ‘Mechanistic Explanation: Its Scope and Limits’, *Aristotelian Society Supplementary Volume*, **87**(1), pp. 39–65.