Three Challenges to Chalmers on Computational Implementation

Published in *Journal of Cognitive Science* (2012) 13: 107–143. mark.sprevak@ed.ac.uk

Three challenges to Chalmers on computational implementation

Mark Sprevak
University of Edinburgh

12 June 2012

The notion of computational implementation is foundational to modern scientific practice, and in particular, to explanation in cognitive science. However, there is remarkably little in the way of theoretical understanding of what computational implementation involves. In a series of papers, David Chalmers has given one of our most influential and thorough accounts of computational implementation (Chalmers 1995, 1996, 2012). In this paper, I do three things. First, I outline three important desiderata that an adequate account of computational implementation should meet. Second, I analyse Chalmers' theory of computational implementation and how it attempts to meet these desiderata. Third, I argue that despite its virtues, Chalmers' account has three shortcomings. I argue that Chalmers' account is (i) not sufficiently general; (ii) leaves certain key relations unclear; (iii) does not block the triviality arguments.

1 Introduction

What does it mean for a physical system (e.g. a brain, a desktop computer) to implement a computation? The first step in answering this question is usually to talk about a special relation—*implementation*—between abstract mathematical computations and physical systems. Implementation is understood as the bridge between the realm of abstract mathematical computations and the nuts and bolts of concrete physical systems. But what is the implementation relation, and how does it work? What are the necessary and sufficient conditions for computational implementation to obtain? Is computational implementation a matter of objective fact, or is it only something that is in the eye of the beholder? Despite our widespread use of the notion of computational implementation in explanation, and despite its foundational role in cognitive science, the concept has received remarkably little attention. Often computational implementation is treated as an explanatory primitive: other things are explained in terms of it, but it itself remains

unexplained. In a series of papers, David Chalmers has done more than anyone else to provide an account of what computational implementation involves (Chalmers 1995, 1996, 2012).

In this paper, I aim to do three things. First, I outline three desiderata that an account of computational implementation should meet. Second, I describe Chalmers' theory of computational implementation and how it attempts to meet those desiderata. Third, I argue that despite the virtues of Chalmers' account, there are three challenges that it faces. I argue that Chalmers' account is (i) not sufficiently general; (ii) leaves certain key relations unclear; (iii) does not block the triviality arguments. These critical remarks should not take away from the spectacular progress that Chalmers has achieved in articulating the notion of computational implementation. My claim is only that, as it stands, Chalmers' account falls short of a complete account. Some distance has yet to be travelled before we reach an adequate account of implementation, and in particular, before we have a clear view of the metaphysical commitments involved in using computational implementation in explanations in cognitive science.

2 What is at stake in a theory of implementation?

Before starting, it is worth pausing to consider what is at stake when one gives an account of computational implementation. As noted above, the notion of computational implementation is often treated as an unanalysed explanatory primitive. This is particularly evident in the day-to-day practice of cognitive science where the notion rarely receives explicit articulation. Over the past forty years, cognitive science has scored spectacular explanatory and predictive successes without explicitly articulating the notion of computational implementation, and cognitive science appears to have the potential to score plenty more successes in the future. So one might wonder why one should even bother looking for a theory of computational implementation. If many cognitive scientists have not found the nature of computational implementation a particularly pressing problem, why should we?

There are at least three inter-related issues which jointly motivate a theory of computational implementation.

(R1) *Clarity*:

> Ultimately, the foundations of our sciences should be clear. Although it might be acceptable to treat computational implementation as an explanatory primitive for the nonce (e.g. for the purposes of model-building), eventually we are owed an explanation of that notion. If cognitive science is successfully to explain the mind, the foundations of cognitive science should be clear and transparent. The foundations will, of course, ultimately resort to appealing to certain basic *sui generis* concepts—concepts that cannot be reductively explained in terms of other concepts. But at the very least we should understand clearly what those basic concepts involve. The notion of computational implementation—despite being a concept of which we have a rough-and-ready grasp—is not clear and transparent in the required

sense. There is no consensus on the content of the concept of computational implementation (as (R2) and (R3) below illustrate). The concept of implementation, and whether it picks out an objective feature of the world, is fundamentally obscure. As it stands, the notion of computational implementation does not function as a suitable *sui generis* concept for the foundation of cognitive science. Some sort of explanation is owed of what computational implementation involves.

This explanatory motivation—a desire for clarity about the conceptual foundations of cognitive science—is mirrored by a metaphysical motivation—a desire for minimising needless commitments on the part of our best science to basic ontological furniture in the universe. The catalogue of ultimate ontological constituents of the universe may end up containing the elementary particles and forces of physics, but it should not end up also containing the *sui generis* relation of *computational implementation*. If computational implementation is real, it must really be something else. But if that is the case, we need some story about how facts about computational implementation are made true by other facts in the world. If we are completely in the dark about this, then we will have no choice but to be lumbered with *computational implementation* as a basic ontological ingredient alongside the fundamental constituents of physics.

(R2) *Response to triviality arguments*:

As we will see in Section 4, our conventional understanding of the notion of computational implementation is threatened by triviality arguments. These triviality arguments claim that almost every physical system implements every computation. The triviality arguments are an open challenge to anyone who wants to make the notion of computational implementation do explanatory work in cognitive science. The intention of the triviality arguments is to show that the notion of computational implementation is completely unsuited for use in cognitive science. There are many motivations for developing an explicit theory of a concept. But perhaps the most powerful motivation is the threat of a challenge to our conventional understanding of that concept. Typically, such threats aim to show that a concept that we thought was perfectly in order, and capable of functioning in explanations, is in fact trivial, empty, or otherwise problematic in a way that makes trouble for its presumed role in explanation. For example, there are powerful motivations to come up with an explicit theory of *knowledge*, *meaning*, and *personal identity* arising from a series of challenges—sceptical arguments, reference indeterminacy arguments, fission arguments—that appears to show that our ordinary understanding of those concepts has unacceptable consequences. Such challenges prompt us to articulate the relevant concept in a way that allows it to justifiably deliver the explanatory work that we require (or as much as possible). The challenges provide a motivation for theory building, and a wealth of constraints on such theories. The triviality arguments of Section 4 concerning computational implementation operate in precisely this way.

(R3) *Naturalistic foundations*:

The ultimate aim of cognitive science is to offer, not just any explanation of mental phenomena, but a *naturalistic* explanation of the mind. The objective is to explain

> how a system can be mental in terms that *do not already presuppose* mental life. This naturalistic character is one of salient features of modern cognitive psychology, and why its explanatory ambitions, in contrast to much of past psychological theorising, are so revolutionary. Cognitive science promises to deliver an explanation of the mind that folds mental life into a broader synthesis of physics, chemistry, evolutionary biology, molecular biology, and the other natural sciences.
>
> To a first approximation, cognitive science's strategy for achieving this goal is to explain mental life in terms of computations implemented by the brain. If this strategy is to work, then explanation in terms of implementation of computation had better be explanation in non-mental terms. The alternative would be incompatible with the naturalistic project. It would mean that, rather than explaining mental life in non-mental terms—in terms of computations implemented by the brain—cognitive science would ultimately be explaining mental life in terms of, *inter alia*, other mental properties. If it turns out that computational implementation itself needs to be explained in terms of mental properties like our beliefs, interests, attitudes, then the naturalistic aim of cognitive science—explaining mental life in non-mental terms via the notion of computation—is doomed to failure.
>
> Call the claim that computational implementation can be explained in non-mental terms, *realism* about computation. Call the claim that computational implementation has to be explained in mental terms (e.g. in terms of our beliefs, interests, and attitudes), *anti-realism* about computation. In order for cognitive science to be able to naturalise the mind, *realism* about computation must be true. The problem is that it is far from obvious whether realism about computation is true or not. There appear to be strong reasons that pull in both directions.[1] A major motivation for a theory of computational implementation is to settle this question: to discover what is involved in computational implementation, and clarify whether computational implementation has eliminable or ineliminable anti-realist commitments. The status of cognitive science as a natural science depends on it. It is hard to imagine higher stakes.

(R1), (R2), and (R3) are three major motivations for a theory of computational implementation. (R1), (R2), and (R3) also provide three desiderata for such a theory to serve the needs of cognitive science. A notion of computational implementation that is adequate to the needs of cognitive science should at least be: (D1) clear, (D2) avoid the triviality arguments, and (D3) provide a naturalistic foundation. If an account of implementation falls short in any one of these areas, then we have reason to complain.

Chalmers achieves a great deal of progress in all three areas. However, I argue that his account does not fully satisfy all three desiderata. It falls short in three main areas: (i) it is not sufficiently general and leaves certain features of the implementation relation unclear (D1), (ii) it does not block the triviality worry (D2), and (iii) it does not secure naturalistic foundations for cognitive science (D3). Chalmers makes a major step forward in explaining the nature of computational implementation, but the resulting account is not complete.

1. For examples of the pull of anti-realism about computation, see Bringsjord (1995); Hardcastle (1996); Putnam (1988); Searle (1992).

Before assessing Chalmers' account of implementation, it is important to have two other pieces in play. In Section 3, I describe the account on which Chalmers builds: what I will call the Standard Position on computational implementation. In Section 4, I describe the triviality arguments that render the Standard Position untenable, and which motivate Chalmers' position.

3 The Standard Position on implementation

Despite the widespread use of computational implementation as an explanatory primitive, it would not be right to say that there are no widely-held theoretical beliefs about the nature of computational implementation. On those occasions when computational implementation is called into question, practitioners tend to produce a proto-theory—a theory that is almost certainly correct in many respects. The proto-theory says that computational implementation involves a *mirroring relation* between an *abstract mathematical formalism* and the *physical states and transitions* of a physical system. Chalmers provides a nice statement of the view:

> A physical system implements a given computation when the causal structure of the physical system mirrors the formal structure of the computation.
>
> (Chalmers 2012, p. 326)

I will call this the *Standard Position* (SP) on computational implementation. SP provides a plausible starting point for a theory of computational implementation. Chalmers' account can be understood as a sophisticated elaboration of SP.

How does SP apply to a particular case? Chalmers applies SP to finite state automata (FSAs):

> A physical system P implements an FSA M if there is a mapping f that maps internal states of P to internal states of M, inputs to P to input states of M, and outputs of P to output states of M, such that: for every state-transition relation $(S, I) \rightarrow (S', O')$ of M, the following conditional holds: if P is in internal state s and receiving input i where $f(s) = S$ and $f(i) = I$, this reliably causes it to enter internal state s' and produce output o' such that $f(s') = S'$ and $f(o') = O'$ (p. 327)

A physical system implements an FSA just in case the formal structure of that FSA is mirrored in the physical structure of that system. The notion of mirroring is identified with that of a *structure-preserving mapping*, a move which I take to be part and parcel of the Standard Position. Physical states of physical system P are paired (mapped) to each formal state of the FSA M. If P's physical states follow the same sequence of transitions as the counterpart formal states of M to which they are mapped, then P implements M. In other words, if a mapping exists between P and M that preserves the structure of the state-transitions of M, then P implements M.

Straight off one can see that SP meets two of the desiderata on a theory of implementation: *clarity* (D1) and *naturalistic foundations* (D3). SP appears to give clear conditions for computational implementation. It explains computational implementation in terms of other concepts, primarily that of a structure-preserving mapping. At the very least, this unifies computational implementation with other notions involving structure-preserving mappings: *measurement* (Dresner 2010), and *model-theoretic interpretation* (Copeland 1996).

SP also secures *naturalistic foundations* for cognitive science. Whether a physical system implements a computation depends only on whether a structure-preserving mapping *exists* between the physical system and the formal computation. It does not depend, for example, on whether someone *judges* such a mapping to exist, or whether it *suits their interests* to look for such mapping, or whether the mapping appears to them *perspicuous*. SP does not require reference to any subject or agent at all. Computational implementation is a purely objective matter: either a mapping between the computational formalism and the physical system obtains, or it does not.

Unfortunately, SP spectacularly fails to meet the third desideratum: *blocking the triviality results* (D2). Chalmers' theory of computational implementation can be seen as a way of revising SP to get around this problem while keeping the other two virtues. Before considering the way in which Chalmers revises SP, it is important to get the triviality arguments clearly in view.

4 Triviality arguments

There are two major triviality arguments: an informal argument from Searle (1992) and a formal argument from Putnam (1988).

4.1 Searle's informal triviality argument

Searle (1992) asks one to imagine one's desktop computer running Microsoft Word. What is happening? There are many physical state transitions inside the desktop computer: transitions in electrical activity, transitions in thermal activity, transitions in vibrational activity, transitions in gravitational activity. According to SP, the computer implements Microsoft Word because one of these patterns of activity—the pattern of electrical activity—has the right structure. If one were to build another physical system, perhaps made out of different materials (e.g. brass wheels and cogs), which had physical transitions with the same structure, then it too would implement Microsoft Word. Now consider a brick wall behind the computer. Despite its static appearance, on a microscopic level a brick wall is teeming with physical state transitions. Within the wall there are physical transitions of vibrational activity, electromagnetic activity, atoms changing state, subatomic particles moving around—a typical wall contains more than 10^{25} atoms, which for one thing are all in microscopic motion. Searle claims there is *so much* physical activity inside a brick wall that there is almost certain to be at least one pattern of activity with the same structure as that inside the desktop computer. Therefore, according to SP, a brick wall implements Microsoft Word.

Similar reasoning appears to show that almost any macro-size physical system implements any computation one likes. Chalmers (2012) identifies two important computational theses in cognitive science: *computational sufficiency* and *computational explanation*. *Computational sufficiency* claims that implementing the right computation is a sufficient condition to possess a mind. If the triviality argument is right, then computational sufficiency would entail an absurdly strong form of panpsychism: brick walls would have minds just as much as we do. *Computational explanation* describes the methodology above: one explains why we have the particular cognitive processes we do by appeal to the fact that our brains implement particular computations.[2] If the triviality argument is right, then this explanatory methodology has to be abandoned. One could not explain our distinctive cognitive processes in terms of the brain implementing particular computations because the brain, like every macro-sized physical system, trivially implements almost every computation.

4.2 Putnam's triviality argument

One might have two immediate concerns about Searle's argument. First, one might be unmoved by his claim that a brick wall contains some pattern of physical transitions with the same structure as Microsoft Word. One might reasonably insist that the burden of proof is on Searle to demonstrate that such a pattern exists. Second, one might think that his triviality argument only applies to macro-sized physical systems. Perhaps if one restricts attention to smaller or simpler physical systems, one can regain a non-trivial form of computational implementation.

Putnam (1988) presents a triviality argument that neatly scotches both of these worries. Putnam offers an argument that finds the relevant pattern of physical transitions that mirror almost any computation one likes in almost any physical system one likes.

Putnam states his argument in terms of finite state automata (FSAs). Pick an arbitrary FSA. Putnam chooses a simple FSA, M, which transits between two computational states, A and B, with the following formal transitions, $A \rightarrow B \rightarrow A \rightarrow B$. Pick an arbitrary open physical system (say, a rock), and a finite time interval, t_0 to t_n. For Putnam's argument, an open physical system is a physical system that is not shielded from, and so remains in causal interaction with, its environment. Almost all physical systems in which we are interested are open in this sense. Next, consider the *phase space* of the rock over time. The phase space is the space of every possible value of every one of the rock's physical parameters. Over time, and even though the rock may appear to our eyes to be unchanging, the rock will trace a path through its phase space as its microscopic physical parameters evolve. Its microscopic physical parameters will evolve both due to endogenous physical factors (internal atoms changing state, vibrations, atomic decay, etc.), and because of external causal influences (gravitational, electromagnetic, vibrational influences, etc.) Putnam argues that among these external influences are external 'clocks': causal influences that cause the rock never to return to *precisely* the same set of values of its microphysical parameters in the course of its evolution.

2. Computational sufficiency, although once an important principle of AI, has now fallen into the background. However, computational explanation remains utterly central to explanation in cognitive science.

Consider the rock's trajectory through its phase space from t_0 to t_n. Since the rock will not revisit precisely the same point in phase space, its trajectory from t_0 to t_n will not contain loops. Putnam divides the rock's path through its phase space into a journey through four regions, labelled r_1, r_2, r_3, r_4. These regions pick out the set of the rock's physical state in four time intervals between t_0 and t_n. Since the rock's physical state is entirely characterised by its position in phase space, regions in phase space provide a description of the physical state of the rock. We can say that in the first time interval, the rock is in physical state r_1, in the second, it is physical state r_2, in the third, in physical state r_3, and in the fourth, in physical state r_4.

We can now ask about the physical *transitions* that the rock exhibits over the interval. One sequence of physical transitions that the rock exhibits is: $r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow r_4$. But, Putnam observes, this is *not the only* sequence of transitions exhibited by the rock. The rock also exhibits the pattern: $r_1 \vee r_3 \rightarrow r_2 \vee r_4 \rightarrow r_1 \vee r_3 \rightarrow r_2 \vee r_4$. In other words, as well moving between four regions of phase space (r_1, r_2, r_3, r_4), the rock also oscillates between two disjoined regions of phase space ($r_1 \vee r_3$ and $r_2 \vee r_4$). In principle, there is nothing wrong with identifying a physical state with a disjunction of unconnected regions of phase space. That is how many perfectly legitimate physical states types are characterised, e.g. net thermal energy and electric charge of the rock. It is also how the physical states of many electronic PCs are characterised, e.g. as disjunctions of electrical signals occurring in various electronic components in different internal locations.[3] Now map physical state $r_1 \vee r_3$ to computational state A, and map physical state $r_2 \vee r_4$ to computational state B. Putnam has demonstrated a structure-preserving mapping between the physical transitions of the rock and the formal transitions of FSA M. The physical transitions in the rock mirror the formal transitions: $A \rightarrow B \rightarrow A \rightarrow B$. Therefore, according to SP, the rock implements FSA M. Putnam's mapping trick could be repeated for other FSAs and other open physical systems. Therefore, an unvarnished version of SP appears to commit one to the claim that every open physical system implements every FSA.

5 Chalmers' solution

The two triviality arguments above show that SP in its bare form cannot be right. Computational implementation must involve more than a simple mirroring between formal transitions and physical transitions. Chalmers revises SP to block the triviality result while aiming to keep SP's virtues of clarity (D1) and naturalistic foundations (D3).

It is helpful to divide Chalmers' revision of SP into three steps.

3. An anonymous referee helpfully points out that many objections to Putnam's argument hinge on objecting to taking disjunctions of phase space regions. The point above is only that there is nothing wrong in taking *disjunctions per se*. The objections are rather that some disjunctions are legitimate bases for computational implementation, while others are not. In Section 6, I consider an objection of this kind based around an element of Chalmers' account that I call the INDEPENDENT-COMPONENTS condition.

5.1 Step 1: Transitions must support counterfactuals

One feature of the triviality arguments is that they assume that mirroring a pattern of actual physical activity is sufficient for computational implementation. Putnam's argument identifies a structure-preserving mapping between the actual evolution of physical states between t_0 and t_n and the evolution of an FSA. But what if the physical state of the rock had been slightly different: e.g. the rock had been hit by one photon extra at t_0, or its temperature one trillionth of a degree warmer, or the Sun 1 cm further away? Putnam's construction is silent about whether the mapping to M would still obtain, and there appears to be no reason to assume that it would. We appear to have identified a potential weakness in the triviality arguments: they exploit *accidental*, not *counterfactually robust*, patterns of physical activity.

Chalmers argues that we should introduce two counterfactual requirements into SP.

First, in order for a physical system to implement a computation, the relevant physical transitions should be *reliable*. The transitions should not be at the mercy of small physical changes. If an extra photon arrives that should not disrupt the mirroring between physical states and formal states. Some work would need to go into spelling this out—for example, what counts as reliable is likely to vary with context—but a reliability condition of some sort is clearly required. Second, it should be true that certain physical transitions *would* have occurred even if, as a matter of contingent fact, they did not. For example, for a physical system to be an *adder*, it is not enough that it gives the right answer to every pair of numbers it is *in fact* asked. It should also be true that *were* it to have been asked a different sum, it *would* have given the correct answer. In the case of FSAs, even if certain formal transitions do not figure in the actual history of the system, they should be mirrored in counterfactuals about what the physical system would have done.

Initially, Chalmers presented these counterfactual requirements as doing 'all the work' in blocking triviality arguments (Chalmers 2012, p. 331), (Chalmers 1995, p. 398). This was a view that enjoyed widespread currency in the literature at the time: it was generally assumed that counterfactual conditionals deal a knock-down blow to the triviality arguments (for example, see Block (1995) and Maudlin (1989)). Interestingly, Chalmers later showed that these kind of considerations *cannot* block the triviality arguments. Even if SP is strengthened with the counterfactual conditions above, a similar triviality result still obtains (Chalmers 1996, pp. 316–319). Beefing up one's account of computational implementation to include counterfactual conditionals is not a silver bullet to deal with triviality worries. It is instructive to see why.

In the revised argument, Chalmers (1996) defines a 'clock' as a physical component that reliably transits through a sequence of physical states over the time interval. He defines a 'dial' as a physical component with an arbitrary number of physical states such that when it is put into one of those states it stays in that state during the time interval. The triviality result for the counterfactually-strengthened version of SP is that every physical system with a clock and a dial implements every FSA.

The argument involves a similar construction to Putnam's, but over possible, not actual, trajectories in phase space. In one respect the construction is simpler, since the only states that need to be considered are the physical system's clock and dial; the other physical

states can be safely ignored. Chalmers' strategy is to identify a mapping between each formal FSA state and a disjunction of physical states $[i, j]$ of the implementing system, where i corresponds to a numbered clock state, and j to a numbered dial state, and the relevant physical states stand in the right counterfactual relations to each other. Here is the argument.

Suppose the system starts in physical state $[1, j]$, then it will reliably transit to $[2, j]$, $[3, j]$, and so on, as the clock progresses. Suppose, without loss of generality, that the system starts its actual run in dial state 1. The start state of the FSA can then be mapped to $[1,1]$, and the subsequent formal states in the evolution of the FSA to $[2,1]$, $[3,1]$, and so on. At the end of this mapping process, if some FSA states have not come up, then choose one of those formal states as the new start state of the FSA and map $[1,2]$ to it. Then pair physical states $[2,2]$, $[3,2]$, and so on with the formal states that follow in the evolution of the FSA. Continue until all of the un-manifested states of the FSA have been covered. Now, for each formal state of the FSA, we will have a non-empty set of associated physical states $\{[i_1, j_1], [i_2, j_2], \ldots, [i_n, j_n]\}$. Assign the disjunction of these states to each FSA state. The resulting mapping between formal and physical states satisfies the counterfactually-strengthened version of SP.

It is worth noting that almost all physical systems in which we are interested will have a clock and a dial. A clock could simply be any law-like sequence of physical changes inside the system. A dial could be the entire trajectory of phase space through which the system travels on a particular run. As Chalmers notes, a clock and a dial could also be easily added just by placing a wristwatch inside the physical system. Clearly, some extra condition needs to be added to solve the triviality problem.

5.2 Step 2: Add input and output constraints

Another striking feature of the triviality arguments is that the computations they consider lack inputs and outputs. Chalmers argues that the triviality results can be avoided, or at least attenuated, if inputs or outputs are added. SP should require that a physical system not only mirror the internal states of the formal computation, but also have appropriate inputs and outputs. There is a *weak* and *strong* way of reading the input-output condition.

On the *weak* reading, all that it means to have appropriate inputs and outputs is that *there exists a structure-preserving mapping* between the inputs and outputs of the physical system and the inputs and outputs of the formal FSA. Just as a structure-preserving mapping is sufficient to implement internal formal states, a structure-preserving mapping is also sufficient to implement formal inputs and outputs. It is not hard to see that this reading of the condition would do almost nothing to solve the triviality problem. Consider a thin spatial boundary around the implementing physical system. This boundary itself is an open physical system, and it will trace its own (non-looping) trajectory through phase space over the time interval. As before, one can map disjunctions of regions of its phase space to formal inputs and outputs. It is not hard to construct a similar triviality result.

The *strong* reading of the input-output condition requires more than the existence of a structure-preserving mapping. It also requires that the physical inputs and outputs be of a *certain physical type*. For example, an implementation of a word processor should take

physical key strokes as input, and yield written physical text as output—something a brick wall clearly fails to do. An implementation of a calculator should take button-presses as input, and yield written physical numerals as output. The strong reading, unlike the weak reading, does appear to provide defence against a Putnam-style construction. As a matter of brute fact, a brick wall does not have the right type of physical input and output to be an implementation of Microsoft Word, regardless of the mappings that might obtain.

Nevertheless, even on the strong reading, a triviality result still obtains. This triviality result is that any physical system that implements *some* FSA with a certain input-output behaviour, will implement *every* FSA with that behaviour. We must distinguish between physical systems with different *internal computational structures*. Two physical systems may have the same (counterfactually-robust) pattern of input-output behaviour, but different computational methods for achieving that behaviour. Internal structure matters a great deal to cognitive science. Disagreements often concern the internal computational structure of cognitive processes (classical, connectionist, etc.), rather than their inputs and outputs. A strong input-output condition constrains implementation only up to the level of inputs and outputs. It places no constraints on internal structure, leaving it open to Putnam's attack. As Putnam observes, the input-output response would, in effect, collapse cognitive science into a form of behaviourism. Therefore, even on the strong reading, the input-output condition still leaves us with the meat of the triviality challenge intact.[4]

5.3 Step 3: Move to CSA architecture

The final revision to SP proposed by Chalmers is to replace the FSA architecture with a more complex computational architecture. Chalmers claims that the triviality arguments can be resisted for a type of formal architecture that he calls *combinatorial state automata* (CSAs). He claims that once we refocus attention on CSAs, non-trivial conditions on implementation can be obtained.

Chalmers concedes that Putnam is right that the implementation conditions of FSAs are trivial in the ways described above.[5] Nevertheless, this would be tolerable if non-trivial implementation are secured *for all computational architectures relevant to cognitive science*. It was originally a worry about cognitive science that motivated a theory of implementation satisfying (D1), (D2), (D3). If this worry could be dealt with, then much of the heat would go out of the debate. It would then be a further question whether the same desiderata need to be met in other contexts where we rely on computational explanation (arguably,

4. Chalmers (1996, pp. 320–323) claims that SP should be supplemented with the additional condition that inputs *would reliably cause* the right internal states, even if they do not actually cause such states (effectively combining Step 1 + Step 2). He argues that this allows the input-output condition to get a toe-hold on constraining internal structure, and helps defend it from Putnam's attack. However, Chalmers goes on to show that a similar triviality result still obtains: any physical system with an *input memory* and a dial implements any FSA with a given input-output behaviour. An *input memory* is a physical component that goes into a unique physical state for every sequence of inputs. Having an input memory is again not hard to satisfy (Chalmers gives the example of adding a tape recorder to the system). See Godfrey-Smith (2009, Section 2) for a more detailed discussion of how a triviality result for a combined Step 1 + Step 2 condition obtains under even less exacting conditions. A combined Step 1 + Step 2 condition therefore does not by itself solve the triviality problem.

5. Chalmers (1995), pp. 394–395; Chalmers (2012), p. 334.

(D3) in many cases would not). Consequently, Chalmers claims that CSAs cover all those computational architectures relevant to cognitive science.[6]

Combinatorial state automata are just like finite state automata except that their states have a combinatorial structure rather than a monadic state structure. Instead of having a single internal state, S, the internal state of a CSA is a vector of sub-states, $[S_1, S_2, \ldots, S_n]$, where the ith component of the state vector is the ith sub-state of the system. The state transitions of a CSA are defined by specifying, for each component of the state vector, how its new value depends on the old state vector and an input vector.

Chalmers claims that a physical system implements a CSA when the following conditions are met:

> A physical system implements a given CSA if there is a decomposition of its internal states into sub-states $[s_1, s_2, \ldots, s_n]$, and a mapping f from those sub-states onto corresponding sub-states S_j of the CSA, along with similar mappings for inputs and outputs, such that: for every formal state transition $([I_1, \ldots, I_k], [S_1, \ldots, S_n]) \rightarrow ([S_1', \ldots, S_n'], [O_1, \ldots, O_l])$ of the CSA, if the system is in internal state $[s_1, \ldots, s_n]$ and receiving input $[i_1, \ldots, i_n]$ such that the physical states and inputs map to the formal states and inputs, this causes it to enter an internal state and produce an output that map appropriately to the required formal state and output. (Chalmers 1996, p. 325)

This completes Chalmers' account of computational implementation. Call it SP-C. Chalmers claims that SP-C meets the three desiderata: it is clear (D1), blocks the triviality arguments (D2), and it provides naturalistic foundations for cognitive science (D3).

6 Three challenges to Chalmers

I will argue that there are three problems with Chalmers' theory of computational implementation, SP-C. These problems are: (i) SP-C does not cover all architectures relevant to cognitive science; (ii) SP-C leaves certain key features of implementation unclear; (iii) SP-C does not block the triviality arguments. I argue that SP-C cannot simultaneously satisfy (D1), (D2), (D3).

6.1 SP-C does not cover all architectures relevant to cognitive science

Chalmers observes that CSA architectures are *more* relevant to explanations in cognitive science than FSAs: FSAs have unstructured internal states, which seem poor formal analogues of human cognitive states (p. 326).[7] However, even if CSAs are *better* models of human cognition than FSAs, that does not show that CSAs are the *only*, or the *best*,

6. Ibid.; Chalmers (1996), p. 324.

7. Nevertheless, see Brooks 1991 for an argument that cognition should be modelled via a series of nested FSAs. If Brooks is right, then FSAs, and their implementation conditions, matter a great deal to cognitive science.

computational architectures for cognitive science. The possibility appears to be open that there are other computational architectures, which are equally, or *more*, relevant to cognitive science than CSAs, and for which (non-trivial) implementation conditions have not yet been secured.

Chalmers attempts to guard against this worry by claiming that the CSA formalism is capable of accurately describing *any* computational architecture. SP-C is therefore a perfectly general account of computational implementation. According to Chalmers, it is possible to *translate* any other computational formalism into the CSA formalism. SP-C can then be applied to that CSA translation, yielding the implementation conditions of the original architecture:

> This definition [of CSA implementation] can straightforwardly be applied to yield implementation conditions for more specific computational formalisms. To develop an account of the implementation-conditions for a Turing machine, say, we need only redescribe the Turing machine as a CSA. The overall state of a Turing machine can be seen as a giant vector, consisting of (a) the internal state of the head, and (b) the state of each square of the tape … A similar story holds for computations in other formalisms. Some formalisms, such as cellular automata, are even more straightforward. Others, such as Pascal programs, are more complex, but the overall principles are the same. In each case there is some room for maneuver, and perhaps some arbitrary decisions to make (does writing a symbol and moving the head count as two state-transitions or one?) but little rests on the decisions we make. … The theory of implementation for combinatorial-state automata provides a basis for the theory of implementation in general. (Chalmers 2012, p. 330)

It is worth emphasising that Chalmers' claim is not the relatively modest claim that the CSA formalism can reproduce the *input-output behaviour* of any other computational formalism.[8] Chalmers' claim is stronger: that the *internal workings* of any of computational formalism *can be adequately expressed in the CSA formalism*. Call the former claim *weak-equivalence*, and the latter claim *strong-equivalence*.

The weak-equivalence claim concerns *what* function a given machine computes: the machine's input-output behaviour. The strong-equivalence claim concerns *how* that function is computed: the mechanism by which the machine moves from input to output.[9] The weak-equivalence claim says that, for any computational architecture, there is some CSA which has the same input-output behaviour—that solves the same overall computational task—as the original architecture. Typically, weak-equivalence claims are justified by the kind of evidence that Chalmers provides: translation rules, which involve a finite number of steps, that take one from the original architecture to some CSA. Weak-equivalence proofs (often called 'simulation' or 'computational equivalence' proofs) play a major role in mathematical computation theory.

8. Like Chalmers, I will restrict attention to computational architectures with finite storage (e.g. Turing machines with finite tape). As Chalmers notes, finite storage architectures are the ones most relevant to modelling human cognition. Chalmers sketches how SP-C can be extended to apply to architectures with unbounded storage, but will do not need to consider his extension here.

9. Cf. Pylyshyn (1984), Ch. 4.

Weak-equivalence is one thing, strong-equivalence is another. The strong-equivalence claim requires that at least one of the CSAs which 'solves the same computational task' also accurately describes—without loss or distortion in its algorithmic description—*how* that task was solved. For architectures to be strongly equivalent, it is crucial that the CSA description accurately captures all computationally-relevant properties of the original formalism. As discussed above, cognitive science not only cares about the input-output behaviour, but also about the method by which the system gets from input to output. Therefore, a weak-equivalence proof alone would not be sufficient to justify SP-C as a general theory of computational implementation. One would also need to show that the CSA formalism describes, without loss or distortion, the internal computational mechanism by which the original architecture operates. After all, if asked for the implementation conditions of a computer X, it would be no good to reply with the implementation of a *different* (albeit input-output equivalent) computer Y. We want to know what are the implementation conditions *of a machine that works like* X, not the implementation conditions of another computer that solves the same problem in a different way.

Whether SP-C succeeds as a general account of computational implementation hinges on the truth of the strong-equivalence claim: on whether translation of any computational method into a CSA is an accurate description (without loss or distortion) of that computational method. I think that there are good reasons for doubting this claim.

Let us start by examining Chalmers' poster-case of strong-equivalence: CSAs and Turing machines.[10] Chalmers argues that a Turing machine can be re-described, without loss or distortion, as a CSA. In the quotation above, he gives a number of translation rules that take one from a Turing machine to an equivalent CSA. For example, the state of the *head* of the Turing machine is mapped to the state of some of the elements of a CSA's state vector. The state of the *tape* of the Turing machine is mapped to the state of other elements of the CSA's state vector. The *transition table* of the Turing machine (how the head acts on the tape) is mapped to how some of the state of some of the CSA's state-vector elements depend on the state of other elements inside the CSA's transition table. Using Chalmers' translation rules, it seems possible to construct a weak-equivalence proof for CSAs and Turing machines. But do the translation rules also establish *strong-equivalence*? Does the specified CSA accurately capture, without loss or distortion, all the computationally-significant features of the original Turing machine? Arguably not.

There are a number of computationally-significant features of Turing machines that are lost in the CSA translation. One such feature is a distinction between *data* and *control*. The Turing machine formalism separates data from control: the formalism marks a distinction between the data on which the computer operates (tape states), and the control states that govern changes to that data (head states). The distinction between data and control plays a major role in theoretical and engineering computer science. This distinction is one of reasons why the Turing machine formalism (rather than a purely state-based formalism such as a CSA) is often used to express and categorise different computational methods. Different computational methods are categorised, at a first pass, by the different ways in which they manage data and control. A touchstone of engineering practice that there are trade-offs between investment of computational resources in control mechanisms versus data structures. These trade-offs can take subtle forms. But a crude example would be

10. Again, restricting attention to Turing machines with finite storage.

that a Turing machine with a handful of head states could 'off-load' control information onto data on its tape, thereby performing its computation with a very simple control structure but rich data representations. In contrast, a Turing machine with a large number of head states could do more computational work in the head and get away with sparser and simpler data representations on the tape. Computer science treats these two kinds of Turing machine as imposing different demands on implementation. Data and control structures are assumed to govern *distinct physical features* of the implementing system. Changes to one may involve modifying the physical logic unit (e.g. the CPU), the other the physical memory bank (e.g. the RAM). They are assumed to involve qualitatively distinct physical components in the system. In short, it is important to both theoretical and engineering computer science to keep data and control elements in the Turing machine formalism distinct, and that this distinction should be reflected in the implementation.

Pylyshyn argues that a distinction between data and control is also important to cognitive science:

> To understand what is essential about computing—or, at least, the aspect of it that is relevant to cognitive science—it is mandatory that we preserve a number of distinctions. … The difference between an extremely complex device characterized merely as proceeding through distinguishable states (but not processing symbols) and what I call a "computer" is precisely the difference between a device viewed as a complex finite-state automaton and one viewed as a variant of a Turing machine … the fundamental distinction that I also want to press, [is] namely, the distinction between a strictly finite control *mechanism* (the Turing machine's finite-state "control-box") and a finite but unbounded string of *symbols* … Describing the computer at the symbol level is to make the crucial distinction I have been arguing must be made: separate the finite mechanism from the (to a first approximation) arbitrarily expandable symbol structures. (Pylyshyn 1984, pp. 70–72)

The distinction between data and control is a *computationally-significant* feature of Turing machines that matters to both computer science and cognitive science.

The fundamental idea expressed by SP is that *formal structure should be mirrored in physical structure*. Computationally-significant distinctions and similarities in the computational formalism should be mirrored by physically-significant distinctions and similarities in the implementation. What we saw above was that a distinction between *tape states* and *head states* is a computationally-significant distinction for a Turing machine: tape states and head states are treated as formally distinct from each other, but similar amongst themselves. An implementation of a Turing machine should therefore have its *tape states* and *head states* implemented in ways that are physically distinct from each other, but physically similar amongst themselves. For example, the head and tape states should be implemented by distinct *types of physical component*. A physical system only implements a Turing machine if it works like a Turing machine. And a physical system only works like a Turing machine if its head states and tape states are implemented in distinct physical ways that are physically similar amongst themselves.

The problem is that the distinction between data and control is entirely lost in the CSA translation. Both head states and tape states are just sub-states alike of a giant undifferentiated state vector. SP-C places no constraints that head states and tape states be implemented in distinct physical ways that are physically similar amongst themselves. Indeed, SP-C does not even have the resources to state such a condition, since the distinction between a Turing machine's data and control elements disappears in the CSA translation. Strong-equivalence—the claim that the CSA translation captures *every computationally-significant feature* of the original Turing machine formalism—appears to be false. The CSA translation does not preserve a critically important feature of Turing machine formalism: its distinction between data and control.

A natural response would be to augment the CSA architecture so that it encodes the distinction between the data and control elements of the original Turing machine. For example, one might introduce within the CSA formalism a distinction between two types of sub-state of a CSA: *data* sub-states and *control* sub-states. The monolithic state vector of the CSA $[s_1, s_2, \ldots, s_n]$ would then contain certain 'highlighted' elements, which are marked out within the CSA formalism as 'data' elements or 'control' elements. SP-C could then be supplemented with a condition that requires that the 'data' and 'control' elements of a CSA's state vector be implemented in distinct physical ways, e.g. by distinct types of physical component.

This response is good as far as it goes, but other computationally-significant differences still threaten a strong-equivalence claim. One such feature that a Turing machine's data is not *random access*: if the Turing machine's head is on square 1, and the machine wishes to read square 15,000, then it must read the state of all intermediate squares first. Non-random access memory is a computationally-significant feature of Turing machines that marks out their computational methods from, say, those of a von Neumann machine. This formal property—non-random access memory—does not hold true of CSAs. A CSA's subsequent state vector can be *immediately* determined by any of its current sub-states without stepping through 'intermediate' elements first.

A natural response again would be to require that a CSA translation of a Turing machine step through a sequence of intermediate 'data reading' states—each corresponding to the original Turing machine reading an intermediate tape square—before the CSA reaches the state that corresponds to the Turing machine 'reading' the desired square. This would not be the most efficient way for a CSA to operate, but it would appear to replicate the non-random-access property of the Turing machine inside the CSA formalism.

The problem is that this response locates the formal property—non-random access memory—in the wrong place: it locates the formal property in the CSA's *control element* (its contingent rules and transition table), not in the *functional architecture* of the computer. This reveals a wider problem with the CSA translation method described above: it collapses the difference between two qualitatively distinct aspects of the CSA—those that encode the *transition table* of the Turing machine, and those are used to simulate the Turing machine's background *functional architecture*. Just as the CSA's state vector collapsed the distinction between the Turing machine's head state and tape state, so the CSA's transition table collapses the distinction between the Turing machine's transition table and its background functional architecture. Both are merely features alike of the CSA's undifferentiated transition table. But the difference between the two matters, both

to computer science and cognitive science. For something to implement a Turing machine, it should work like a Turing machine. And for something to work like a Turing machine, there should be a distinction between its *control element* (its transition table), and its background *functional architecture*. The implementation conditions of a Turing machine should require that the Turing machine's transition table and its background architectural features be implemented in physically distinct ways that are physically similar amongst themselves. These two properties of Turing machines—functional architecture and control element—are treated as formally distinct but similar among themselves in the formalism, and they should be treated as physically distinct but physically similar among themselves in the implementation. However, there is no way SP-C can require this to be true. The difference between these two elements of a Turing machine—functional architecture and control element—simply disappears in the CSA formalism.

A natural way to respond is to repeat the 'highlighting' trick above. One could augment the CSA formalism to encode a distinction between transitions within the giant CSA transition table. For example, one might explicitly distinguish between *simulation* transitions and *target* transitions. A *simulation* transition is one that is needed to reproduce some aspect of the Turing machine's background functional architecture (e.g. to reproduce properties like non-random-access memory as above). A *target* transition would be a transition of that encodes the Turing machine's transition table (i.e. its control element). A condition could then be added to SP-C that these different types of transitions should be implemented in physically distinct ways that are physically similar among themselves (e.g. by different types of physical mechanism). The properties of the physical implementation would then mirror, and preserve the relevant difference between, computationally-significant features of the Turing machine formalism.

This fixes two translation problems, but plenty more remain. There are no shortage of computationally-significant features of Turing machines that are lost or distorted in the CSA translation: what counts as an atomic operation, what can happen synchronously, and at what stages input or output is permitted. All of these matter to way in which Turing machines work. All characterise the distinctive methods by which Turing machines achieve their behaviour. All should be preserved and reflected in the implementation conditions of Turing machines. And all are distorted or lost in the CSA translation.

A hard-headed solution would be to keep replaying the 'highlighting' trick above, augmenting the CSA formalism to capture each and every computationally-significant feature of Turing machines until all of the relevant formal distinctions and similarities of the Turing machine formalism are captured in an enriched CSA formalism. Conceivably, at the end of this procedure one would represent the computational methods of a Turing machine within an enriched CSA formalism without loss or distortion. SP-C could then be revised in light of this CSA formalism, and appropriate constraints placed on physical implementation. We would then have achieved our goal of stating the implementation conditions of a Turing machine using the CSA formalism—albeit a heavily modified and augmented version of the CSA formalism.

But all this has a serious cost.

First, it loses the simplicity and generality of Chalmers' original proposal. The only way to capture the computationally-significant features of Turing machines appears to be to

revise the CSA formalism to such an extent as to effectively recreate the Turing machine formalism inside it. It appears that there is little or no redundancy lurking in the Turing machine formalism for the original CSA formalism to exploit. But if this is so, then it appears that the strong-equivalence claim as originally proposed is not, in any interesting sense, true.

Second and more seriously, the CSA formalism was claimed to be capable of expressing without loss or distortion the computational methods, not just of Turing machines, but of *any* computational architecture. We have seen that the CSA architecture needed major revision to capture the computational properties of the Turing machine formalism. However, the translation problems encountered in the case of Turing machines are nothing in comparison to those of other computational architectures.

As a first step, consider switching from the original Turing machine architecture to a multi-head Turing machine architecture, or a multi-tape Turing machine architecture. The CSA formalism and SP-C would need to be revised again. Different features of the target formalism will need to be 'highlighted' in the CSA translation. For example, the distinction between different heads and tapes in a multi-head/multi-tape Turing machine will need to preserved in the CSA translation and reflected in the implementation conditions. Each head and tape should be implemented by a distinct type of physical component that is similar amongst themselves (*heads*), and different in kind again from the others (*tapes*). More highlighting and tweaks to the CSA formalism and SP-C would be required.

Now consider moving to an architecture that departs more dramatically from that of Turing machines: register machines, Church's λ-calculus, quantum computers, dataflow computers, billiard-ball computers, enzymatic computers. These formalisms have radically different ways of splitting control and data, different clocking paradigms, different parallelisms, different synchronous natures, different atomic operations, and different ways of handling input and output. They differ from CSAs, Turing machines, and each other, in major and often incompatible ways. They employ different computational methods, and introduce new and incompatible computationally-significant properties. Accurately translating each formalism, without loss or distortion, to the CSA formalism would require the CSA formalism 'highlight' the computationally-significant properties of that particular architecture. Given the *open-ended* nature of the list of possible alternative architectures, and the *incompatible* nature of the decisions they make about computationally-significant properties, it is hard to see how this could be done. Tweaking the CSA formalism to achieve strong-equivalence with *all* such architectures simultaneously without that formalism becoming massively disjunctive and open-ended seems impossible. Moreover, the brunt of the work of a theory of implementation seems to be done by how the CSA formalism should be modified in each case, not by what the CSA formalism itself says.

Therefore, replaying the 'highlighting' trick—hard-wiring the desired formalism inside the CSA formalism—may achieve strong-equivalence between CSAs and Turing machines, but it would be a short-sighted move. Indeed, tailoring the CSA architecture to Turing machines has the cost that it moves us *further away* from accurately modelling other computational architectures. As the CSA formalism becomes a better model of Turing machines, it becomes a worse model for other architectures (register machines, Church's λ-calculus, quantum computers, dataflow computers, enzymatic computers). The hard-wired modifications proposed to the CSA architecture to model Turing machines—a

Turing machine-style control/data split, non-random access memory, atomic operations, etc.—are precisely the wrong sorts of modifications to the CSA formalism to accurately represent other architectures.

This worry has particular bite in the case of the computational models in cognitive science. Contemporary computational models in cognitive science have little resemblance to *either* CSAs *or* Turing machines. It is worth bearing in mind just how distant they are, and the incompatible nature of the decisions they make about computationally-significant properties. They help to bring into sharp focus the magnitude of the challenge faced by a strong-equivalence claim for cognitive science.

For example, Marr (1982)'s computational architecture for the early visual system is a series of discrete nested computational filters that pass signals to each other in serial or parallel. Marr's formal architecture differs in numerous ways from both Turing machines and CSAs. It differs in terms of its control/data split, atomic operations, introduction of nesting relations between filters, requirements on what can happen synchronously, and at what stages input and output are permitted. Neither the original CSA formalism nor the modified CSA formalism above accurately capture the computationally-significant properties of Marr's formal architecture. Anderson (2007)'s ACT-R architecture is different, but just as challenging for a CSA architecture to model. ACT-R is tailored to explain different cognitive capacities from Marr's architecture and has different computationally-significant properties: a difference between declarative and procedural data, a difference between chunks and buffers, an organisation into modules, a production-driven rather than state-driven control system. Again, CSAs seem a poor model: they would fold all these properties into the workings of a giant transition table and state vector, with no guarantee that the relevant distinctions and similarities in the original architecture would be preserved by distinctions and similarities in kind between the components of the implementation. Wolpert and Kawato (1998)'s MOSAIC architecture is different again, and requires different formal distinctions. MOSAIC has a highly modularised structure, it has continuous dependence of output on input, it has computational relations that are best described by differential equations, it has error comparison and error summation operations as atomic steps, it is fully asynchronous, it uses probabilistic generative models among its basic representations, and it has a radically different way of managing control to traditional computers (Wolpert, Doya and Kawato 2003). The CSA formalism does not appear capable of expressing the methods of the MOSIAC architecture without loss or distortion. If strong-equivalence between Turing machines and the original CSA architecture was hard to achieve, strong-equivalence with the computational models in cognitive science appears even harder. And if one tries to secure strong-equivalence by departing from the original CSA formalism by using the highlighting trick, then one faces the problem that different models depart from the CSA formalism in open-ended and incompatible ways.

There is a claim that is closely related to strong-equivalence which is almost certainly true, and which may be the source of possible resistance to the concerns above. It is almost certainly true that a physical system that is described as, say, a MOSIAC system can *also* be described as a CSA. If Chalmers (2012) is right, almost any physical system can be described as a CSA merely in virtue of having a causal structure of some kind or other (p. 341). But the fact that two computational descriptions are simultaneously true of the

physical system does not establish that a MOSIAC formalism and a CSA are the same (or strongly-equivalent) computational formalisms. Just as a quantum mechanical description and molecular biological description of a cat can both be true of the same physical system does not establish that the two descriptions express the same information. It is often the case that multiple distinct computational descriptions are satisfied by the same physical system. The same physical system (e.g. an electronic PC) may simultaneously implement an FSA, a CSA, a Turing machine, a register machine, and Microsoft Word. But this in no way shows that all these computational descriptions are strongly-equivalent. Just as a cat's quantum mechanical description and the molecular description have neither the same content nor the same satisfaction conditions, so a CSA and MOSAIC computational description have neither the same content nor the same implementation conditions, even if both are (non-accidentally) satisfied by the same physical system.

Finally, it is worth wondering why, if strong-equivalence *were* true, we would feel the need for talking about different architectures at all. If strong-equivalence were true, why would computer science even bother using other computational architectures? Pragmatic factors may provide part of the answer: some formalisms are simply easier for humans to manipulate than others. But pragmatic factors only go so far, and a plausible deeper explanation, which fits with the assertions made in computer science, lies at hand. We have such a rich variety of computational formalisms because of their different *expressive resources*. Different architectures allow us to describe different computational methods for solving problems. The same overall effect may be achievable with another formalism, but not in the same way. We pick our architecture with an eye to its control structure, basic operations, data structures, whether it is synchronous or asynchronous, etc. These choices enable different computational tricks, distinctive twists and turns in moving from input to output. This is seen in the differences between algorithms that are enabled by von Neumann machines, λ-calculus, quantum computers, DNA computers, enzymatic computers.[11] Strong-equivalence would render these other architectures superfluous: there is no method that cannot be expressed without loss or distortion in the CSA formalism. But why then does computer science and cognitive science place such a premium on the resources offered by a switch in architecture? Isn't the point of such a switch to open up a space for expressing new computational methods?

6.2 What is SP-C's mapping relation?

The first problem with SP-C is that it is not sufficiently general as a theory of implementation, and in particular, that SP-C does not secure the implementation conditions of computational models in cognitive science. The second problem concerns the mapping relation between physical states and abstract machine states—the relation that SP-C inherits from SP. So far we have treated this mapping relation as an explanatory primitive. We also said that SP satisfied (D1) on clarity because it unified computational implementation with model-theoretic interpretation and measurement. But what is this mapping relation? What metaphysical commitments does it bring with it?

This may seem like a strange question, but it should be pressed. SP's mapping relation plays an absolutely central role in computational implementation. One of the desiderata

11. See Backus (1978) for how formal architecture determines the available computational methods.

for a theory of computational implementation is that it provide a naturalistic foundation for cognitive science (D3). We saw that, to a first approximation, this means that computational implementation has to be explicable in wholly *non-mental* terms. But then it looks like SP is hostage to the fortunes of the mapping relation. If the mapping relation turns out to be mind-dependent, then this will infect computational implementation too. And it is not clear whether the mapping relation is innocent in this regard. Indeed, it is far from clear what the mapping relation is, and what commitments it introduces into computational implementation.

One might claim that the mapping relation is an *internal relation* and hence does not introduce any extra commitments over and above the metaphysical commitments brought by its relata. There are two problems with this response. First, it is not obvious that the mapping relation is an internal relation. Merely claiming that it is does not settle it; stipulations cannot make it so. Second, an internal relation requires the existence of its relata. Therefore, in order for a mapping relation to obtain, *both* the physical state and the abstract mathematical formalism that is the CSA need to exist. But this appears to commit one to the existence of mathematical objects, which is far from uncontroversial. Explaining the mapping relation in terms of other relations such as *correspondence* or *pairing* between individuals runs into similar problems. Either move seems to commit one to the existence, not only of individual physical objects, but also of abstract objects described in the mathematical formalism.

One might object that this is a general problem, not one that is specific to SP.[12] The mapping relation that SP employs is shared by other domains, not just computational implementation. If the nature of the mapping relation introduces worrisome commitments, or is unclear, then that is a problem not just for SP, but for a wide range of other areas. However, the general nature of the worries should not blunt their force. A parallel can be drawn with the treatment of the representation relation. Chalmers (2012) argues that the representation relation should not figure in an account of computational implementation because it is obscure and poorly understood (violating (D1)).[13] Searle (1992) argues that the representation relation should not figure because it introduces illicit mind-dependency (violating (D3)). One might take issue with either of these claims, but both employ general concerns about the representation relation to place constraints on computational implementation. If general worries about representation justify keeping it out of an account of implementation, then general worries about the mapping relation should have force too.

6.3 SP-C does not escape the triviality result

A final problem for SP-C is that, even for the specific case of CSA computations, SP-C does not block Putnam-style triviality arguments. We saw in Section 5 that Step 1 and Step 2 of SP-C were neither individually nor jointly sufficient to block the triviality arguments. The work of blocking the triviality arguments therefore falls almost entirely on Step 3. The problem is that it is not clear how Step 3—switching from an FSA to a CSA architecture—helps to solve the triviality problem at all.

12. Thanks to an anonymous referee for pressing this point.
13. Chalmers (1995), pp. 399–400; Chalmers (2012), p. 334.

One worry is that CSAs immediately fall prey to Putnam's triviality argument. Formally, it is easy to translate between CSAs and FSAs. If one is persuaded by the line of reasoning that Chalmers gives to justify strong-equivalence above, one might be inclined to think that CSAs and FSAs are not genuinely different formalisms, but just notational variants.

Without loss of generality, denote the sub-states that the elements of the CSA state vector can take, $S_1, S_2, \ldots, S_n$, with numerical values (1, 2, 3, etc.). Now consider an FSA with states $S = 2^{S_1}3^{S_2} \ldots {p_n}^{S_n}$ for every possible sub-state S_i, where p_i is the ith prime number, and transitions $S \rightarrow S'$ iff $[S_1, \ldots, S_n] \rightarrow [S_1', \ldots, S_n']$. The FSA so defined is a state-based automaton with exactly the same states and transitions. The only difference is that FSA states are described by a scalar and the CSA states are described by a vector. Both CSA and FSA descriptions appear to identify the same state-based automata, just with different notations. If as Putnam argues, the implementation conditions of FSAs are trivial (which Chalmers admits), then so too are the implementation conditions of CSAs.

Chalmers is of course aware of this problem. He knows that an extra constraint must be added to avoid collapsing the CSA case into the FSA case. The key move is to flag the vectorial nature of the CSA notation as *computationally-significant* with specific implications for implementation, which differ from those of a scalar notation. Chalmers does this by proposing that 'each element of the [CSA] vector corresponds to an *independent element of the physical system*' (Chalmers 1996, p. 325, my italics). Call this the INDEPENDENT-COMPONENTS condition. To my mind, the INDEPENDENT-COMPONENTS condition is the single most important element of SP-C. It does the lion's share of the work in blocking the triviality arguments. Without this condition, it is unclear how switching to a CSA architecture offers any gain in blocking the triviality arguments over SP.

Given the amount of work that the INDEPENDENT-COMPONENTS condition does, it is frustrating that it is not easier to spell out the content of the condition. What does it mean for something to be an *independent element of the physical system*? One answer can be ruled out straightaway: 'independent element' cannot mean *treated as independent by us*, or some such, since that would immediately concede anti-realism about computation, and lose us naturalistic foundations for cognitive science (D3). There must be an objective, naturalistic, answer to what makes something an *independent element of the physical system*. But it is not obvious what this is. Any candidate proposal has to strike a delicate balance. It has to be strict enough to block the Putnam-style triviality arguments. But it also has to be liberal enough to accommodate the vast number of legitimate way of dividing up and deploying physical properties in genuine computations.

Chalmers proposes an answer that attempts to strike this balance: each component of the state vector of a CSA should correspond to a *distinct physical region* of the implementing system.[14] Call this the SPATIAL-REGIONS proposal. SPATIAL-REGIONS aims to spell out the content of the INDEPENDENT-COMPONENTS condition. The SPATIAL-REGIONS proposal has the virtue of being clear (D1) and naturalistic (D3), but unfortunately it is neither necessary nor sufficient for computational implementation of CSAs.

The SPATIAL-REGIONS proposal is not necessary because a system could implement a CSA even if its sub-states occupy the same spatial regions. There are many ways in which this could happen. First, a system could use *properties* to encode its sub-states, and distinct

14. Chalmers (1996), p. 325, Chalmers (2012), p. 328.

properties can be instantiated at the same spatial location. For example, a system might use different vibrational frequencies (e.g. different electromagnetic frequencies) to implement different elements of its state vector, even if those frequencies are instantiated in the same spatial locations (as in AM radio). Second, a system might use *pulses* that travel back and forth over the same spatial regions, but which are individuated by their *delays* to implement different elements of its state vector (for example, as in a mercury delay line (Eckert 1997)). Finally, as Chalmers suggests, the sub-states of a CSA could change their implemented spatial region over time as part of the computation. Two sub-states could swap over to occupy each other's spatial regions. An example would be the use of pointers in PCs which allow the physical memory location of data to be changed without affecting the computation (Chalmers 1996, pp. 329–330).

Perhaps more worrying is that the SPATIAL-REGIONS condition is not sufficient for implementation. Even with the SPATIAL-REGIONS condition in place, a triviality result for CSAs still obtains. Choose an open physical system P and pick within it n spatial regions, $e_1, \ldots, e_n$, which are (i) spatially distinct and (ii) share some spatial border with each other (say, via a connective region). Each spatial region is itself an open physical system. Hence, by Putnam's result, each spatial region e_i implements any FSA. Therefore, at any moment in time, each region e_i can be associated with an arbitrary state S_j of any FSA. The spatial regions $e_1, \ldots, e_n$ share common borders. Define the borders of these connective regions as the weak inputs and outputs to each spatial region e_i. A weak input or output merely requires that there exists a structure-preserving mapping between the physical inputs and outputs and the inputs and outputs of the abstract FSA. As we saw above, weak inputs and outputs impose almost no constraints on implementation. Since the regions e_i implement any FSA, they implement any FSA with weak inputs and outputs so defined. Therefore, physical system P implements a group of n FSAs such that if those FSAs are in states $S_1, \ldots, S_n$ at one moment, they will be in states $S_1', \ldots, S_n'$ at the next moment, with the transition governed by all the previous S_i states via the inter-region weak inputs and outputs. Therefore, the physical system as a whole implements the state transitions $[S_1, \ldots, S_n] \rightarrow [S_1', \ldots, S_n']$. Modifying the result so that it also accepts overall input and output (Step 2), and its transitions have modal force (Step 1) is not hard. Similarly, the requirement that there be at least n distinct spatial regions that border with each other inside the physical system is easy to meet.

Even if one were to reject SPATIAL-REGIONS, the INDEPENDENT-COMPONENTS condition still seems to be a fundamentally correct thought about the nature of computational implementation. The basic idea of INDEPENDENT-COMPONENTS also generalises beyond the specifics of the CSA formalism (as we saw in Section 6.1). That idea is that *distinct computational elements* in the abstract computational formalism should be implemented by *independent physical components* in the physical system. This is an instance of the general idea explored in Section 6.1 that computationally-significant differences and similarities in the abstract formalism should be mirrored by physically-significant differences and similarities in the implementation. The INDEPENDENT-COMPONENTS condition appears to offer the seed of a response to the triviality arguments. Presumably, not just any physical system has physical components that are independent in the right sense and wired up in the right ways. However, to make good on this, one must spell out the content of the INDEPENDENT-COMPONENTS condition. In particular, one must give a naturalistic, object-

ive characterisation of the conditions under which two physical features are independent components that strikes the correct balance above between being too liberal and too strict. In the absence of such an account, using the INDEPENDENT-COMPONENTS condition to block the triviality arguments remains a promissory note.

If INDEPENDENT-COMPONENTS cannot be spelt out as SPATIAL-REGIONS, how should it be understood? My own view is that INDEPENDENT-COMPONENTS should be understood as placing constraints on what various physical features represent. This brings extra resources into play, and I believe allows one to strike the right balance described above. However, this representational approach to implementation differs significantly from that of Chalmers, and it brings further challenges, which I will not discuss here.

7 Conclusion

We identified three desiderata on an account of computational implementation. These were that an account should be (D1) clear, (D2) avoid the triviality arguments, and (D3) provide naturalistic foundations for cognitive science. Chalmers' account of implementation, which I have called SP-C, can be understood as an attempt to meet these three desiderata.

I raised three challenges to SP-C. These were that SP-C is (i) not sufficiently general, (ii) leaves certain key relations unclear, (iii) does not block the triviality arguments. We saw a trade-off between meeting the desiderata. Individually, each desideratum is easy to meet. Even if one gives up one of the three desiderata, meeting the other two is relatively easy. For example, a common way to block the triviality arguments (D2) and keep clarity (D1) is to allow non-naturalistic factors into the facts that determine computational implementation (e.g. our *judgements*, *interests*, and *attitudes* concerning the merits of different mappings) (giving up (D3)). Meeting all three desiderata simultaneously is hard. Chalmers' account provides the best attempt to do so, but even his proposal falls short. In each of the three challenges above (i)-(iii), we saw some or other trade-off was pressed on us—either giving up clarity (D1), the response to the triviality arguments (D2), or naturalistic foundations (D3).

Even if one is convinced by the challenges above, Chalmers' account remains of absolutely central importance. Chalmers' account presents insightful and plausible necessary conditions on computational implementation. What I have called Chalmers' INDEPENDENT-COMPONENTS condition expresses an important insight: that different elements of the computational formalism should be implemented by *independent physical components*. Tantalisingly, this thought appears to contain the seeds of an answer that meets all three desiderata. Cashing out what *independent physical component* means in the context of an account of computational implementation is one of the major challenges facing future work on implementation.

Acknowledgements

I would like to thank two anonymous referees for helpful comments on a previous draft of this paper.

References

Anderson, J. R. 2007. *How Can the Human Mind Occur in a Physical Universe?* Oxford: Oxford University Press.

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

Block, N. 1995. 'The mind as the software of the brain'. In *An Invitation to Cognitive Science, Vol. 3, Thinking,* edited by E. E. Smith and D. N. Osherson, 377–425. Cambridge, MA: MIT Press.

Bringsjord, S. 1995. 'Computation, among other things, is beneath us'. *Minds and Machines* 4:469–488.

Brooks, R. A. 1991. 'Intelligence without representation'. *Artificial Intelligence* 47:139–159.

Chalmers, D. J. 1995. 'On implementing a computation'. *Minds and Machines* 4:391–402.

———. 1996. 'Does a rock implement every finite-state automaton'. *Synthese* 108:309–333.

———. 2012. 'A computational foundation for the study of cognition'. *Journal of Cognitive Science* 12:323–357.

Copeland, B. J. 1996. 'What is computation?' *Synthese* 108:335–359.

Dresner, E. 2010. 'Measurement-theoretic representation and computation-theoretic realization'. *The Journal of Philosophy* 107:275–292.

Eckert, J. P., Jr. 1997. 'A survey of digital computer memory systems'. First published 1953. *Proceedings of the IEEE* 85:184–197.

Godfrey-Smith, P. 2009. 'Triviality arguments against functionalism'. *Philosophical Studies* 145:273–295.

Hardcastle, V. 1996. 'Computationalism'. *Synthese* 105:303–317.

Marr, D. 1982. *Vision.* San Francisco, CA: W. H. Freeman.

Maudlin, T. 1989. 'Computation and consciousness'. *The Journal of Philosophy* 86:407–432.

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

Pylyshyn, Z. W. 1984. *Computation and Cognition.* Cambridge, MA: MIT Press.

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

Wolpert, D. M., K. Doya and M. Kawato. 2003. 'A unifying computational framework for motor control and social interaction'. *Philosophical Transactions of the Royal Society of London, Series B* 358:593–602.

Wolpert, D. M., and M. Kawato. 1998. 'Multiple paired forward and inverse models for motor control'. *Neural Networks* 11:1317–1329.