The Varieties of Computation: A Reply

David J. Chalmers

I am grateful to the twelve commentators on my article "A Computational Foundation for the Study of Cognition". The commentaries reflect a great deal of thought by authors many of whom have thought about this topic in more depth than I have in recent years. In many cases, they raise issues that I did not anticipate when I wrote the article and that require me to rethink various aspects of the account I gave there.[1]

Publication of this symposium, almost twenty years after writing the paper, has encouraged me to dig into my records to determine the article's history. The roots of the article lie in a lengthy e-mail discussion on the topic of "What is Computation?", organized by Stevan Harnad in 1992. I was a graduate student in Doug Hofstadter's AI laboratory at Indiana University at that point and vigorously advocated what I took to be a computationalist position against skeptics. Harnad suggested that the various participants in that discussion write up "position papers" to be considered for publication in the journal *Minds and Machines*. I wrote a first draft of the article in December 1992 and revised it after reviewer comments in March 1993. I decided to send a much shorter article on implementation to *Minds and Machines* and to submit a further revised version of the full article to *Behavioral and Brain Sciences* in April 1994. I received encouraging reports from BBS later that year, but for some reason (perhaps because I was finishing a book and then moving from St. Louis to Santa Cruz) I never revised or resubmitted the article. It was the early days of the web, and perhaps I had the idea that web publication was almost as good as journal publication.

Over the years, the article has been discussed here and there, and as it turns out it has been cited about as much as the shorter published article. I have always viewed this article as the more important of the two, and it is good to have it finally published. Over the last two decades the discussion of computation has continued apace and has become much more sophisticated, but as

[1]This article was published in *Journal of Cognitive Science* 13: 211-48, 2012. The papers I am replying to are available at http://j-cs.org (issues 12:4, 13:1, 13:2). I would like to thank Gualtiero Piccinini for conceiving and organizing the symposium and also for his comments on a draft of this reply.

usual with philosophy, there is still not much consensus on the underlying issues. This symposium is a good chance to revisit the issues and see how things look in light of progress since then.

It is worth saying that I never saw this article as putting forward a radically original view. As I said in a footnote, I viewed the account of implementation as a version of what I saw as the "standard" definition, with roots in the work of Putnam and others. But it was surprising how little attention this notion had received in the philosophical literature. I saw my main contribution as (i) putting forward a detailed account of implementation in this framework that could avoid the triviality results of Putnam, Searle, and others, and (ii) using this account to draw explicit and detailed connections to the foundations of cognitive science, via theses of computational sufficiency and computational explanation, and the key notion of organizational invariance. In this way it seemed to me that many criticisms of computationalism that were extant at the time could be answered.

I come to this symposium in the position of still being fairly sympathetic to the spirit of the account I gave in the original article, but certainly not wedded to every detail, and certainly open to other perspectives. My metaphilosophical views have evolved in the direction of pluralism. I am inclined to think that as with most philosophically interesting notions, there are many important notions of computation, with different notions playing different roles. Disputes over what counts as "computation" and "implementation" may be verbal unless we pin them down by specifying relevant roles that we want these notions to play. I am also inclined to be pluralistic about explanation, so there may be many sorts of explanation in cognitive science rather than just one basic sort. That said, I would like to think that an account of computation and implementation in the spirit of the account I give in the article can play a number of useful explanatory roles.

In the article, I first give an account of what it is to implement a computation in terms of causal organization: roughly, a computation by a physical system is implemented when the formal structure of the computation is mirrored by the causal structure of the physical system. To make this more precise, I introduce the notion of a combinatorial-state automaton (CSA), a sort of abstract computational object with the power and flexibility to capture the structure of many other computational objects such as Turing machines. I give an account of what it is for a physical system to implement a CSA in terms of mappings from physical states to formal states such that physical state-transitions mirror corresponding formal state-transitions. I suggest that this account captures roughly the conditions under which computer scientists and other theorists would count a physical system as implementing a certain complex computation.

I go on to make two substantive claims linking computation and cognition. A thesis of Computational Sufficiency says that there are computations such that implementing those computations

suffices for the possession of a mind, and for the possession of a wide variety of mental properties. A thesis of Computational Explanation says that computation so construed provides a general framework for the explanation of cognitive processes and of behavior. Two central planks in arguing for both theses are the notions of causal organization and organizational invariance. I claim that computational descriptions of physical systems serve to specify their causal organization (or causal topology), and that cognitive processes and properties are organizational invariants that are determined by the abstract causal organization of the system.

In what follows I will largely respect this structure. I will start by addressing some general issues about conceptions of computation: first issues about abstract and concrete computation (raised by Brendan Ritchie and Brandon Towl), and then issues about syntactic, semantic, and functional conceptions of computation (raised by Frances Egan, Colin Klein, Gerard O'Brien, and Michael Rescorla). I then address issues specific to combinatorial-state automata, including the issue of whether they are a useful formalism (raised by Curtis Brown, Marcin Milkowski, and Mark Sprevak) and the issue of whether my account of their implementation is subject to trivialization (raised by Brown, Sprevak, and Matthias Scheutz). Finally, I address issues about computational sufficiency (raised by Stevan Harnad and Oron Shagrir) and about computational explanation (raised by Egan, O'Brien, and Rescorla).

1 Abstract and Concrete Computation

Ritchie and Towl both focus on the question of whether computations are abstract objects or concrete processes. I confess that I see this as a nonissue: it is a paradigmatically verbal dispute. It is obvious that there are abstract objects (Turing machines, C programs, and so on) that play a central role in computing, and it is obvious that there are concrete processes (e.g. the emacs process running on my iMac) that also play a central role in computing. Which of these better deserves the name "computation" is not something that we really need to worry about. The abstract objects play the central role in the mathematical theory of computation, while the concrete processes play a central role in computation "in the wild". The really interesting issue, and the focus of my article, is the bridge between the two.

Towl says that these two ways of conceiving of computation are "mutually exclusive". That sounds to me like saying that there are two mutually exclusive ways to conceive of shapes: as abstract objects or as concrete objects. But we have long ago learned that we need not make a choice here. There is an important mathematical domain of abstract geometry, studying the

properties of Euclidean and nonEuclidean geometries. There is also concrete or physical geometry, studying the geometry of the world we live in. And there is a bridge between the two: concrete objects can realize (or implement) certain abstract shapes and other geometric structures. For example, it turns out that objects in our world do not strictly realize the structures of Euclidean geometry, but they do realize the structures of nonEuclidean geometry.

Towl criticizes the abstract conception of computation as incomplete, but this is just what we should expect: for a full understanding of computing, we need both notions. He also suggests that "computation" is used more often in practice for concrete processes: perhaps he is right, but nothing of substance turns on this. We might simply use the different terms "a-computation" and "c-computation" for the abstract objects and for the concrete processes.

Of course there is a strong link between abstract and physical geometry: in effect, we count physical objects as having certain physical shapes whenever they realize certain abstract shapes. The same goes for a-computation and c-computation: we count a process as a certain sort of c-computation whenever it realizes a certain sort of a-computation. In both cases, the abstract objects play a crucial role in understanding the concrete objects. This is usually how it is in applied mathematics. Of course, those with nominalist scruples may worry about whether the abstract objects really exist, and may want to find some way of paraphrasing our talk of them away, but in practice the science will continue appealing to them whenever it needs to, and in any case the issues regarding computation here are no different from the issues in any other parallel areas of pure and applied mathematics.

Towl asks whether computations are just descriptions. Certainly, I think that when a physical system implements an a-computation, the a-computation can be seen as a description of it: the property of implementing that a-computation is an organizational property that the physical system really has. We can also talk of the system as undergoing or performing c-computation if we like: this is also a description of the system, with very much the same content.

Towl then worries that things become overliberal, with every system (even a pool table) describable as implementing a-computations or as performing c-computations. I think this issue can be separated from the previous one, as it depends on just how we are understanding computation and especially on the constraints on a specific account of implementation (such as my account of CSA implementation). In any case, for reasons given in the target article, I do not think that objection is much of a worry. On my account, a pool table will certainly implement various a-computations and perform various c-computations. It will probably not implement interesting computations such as algorithms for vector addition, but it will at least implement a few multi-state

automata and the like. These computations will not be of much explanatory use in understanding the activity of playing pool, in part because so much of interest in pool are not organizationally invariant and therefore involve more than computational structure. I note that (*pace* Towl) locations, velocities, relative distances, and angles are certainly not organizational invariants: systems with the same causal topology might involve quite different locations, velocities, and so on. As in the target article, what matters in practice is not much *whether* a system is computing as *what* it is computing and what that computational structure explains.

Ritchie says that I do not adequately distinguish between what he calls *modeling* descriptions and *explanatory* descriptions. He does not give a clear account of the distinction, but as far as I can tell it involves three elements. Explanatory descriptions (i) require strict mappings, (ii) involve real mechanisms, functions, and causal structure; and (iii) explain a system's behavior, whereas modeling descriptions allow approximate mappings, involve mere state-transitions, and need not explain a system's behavior. On dimension (i) my account certainly falls with explanatory descriptions, as it requires strict mappings (between precise formal states and coarse-grained physical states), though I think there is almost certainly a place for a more flexible notion that allows approximate mappings. On dimensions (ii) and (iii), my definitions do not explicitly require claims about mechanisms, functions, causal structure, or explanation, so I suppose they fall with Ritchie's modeling descriptions. I would like to think that the sort of state-transitional structure I invoke is often explanatory, but as we see in the case above it need not always be. It all depends on what we want to explain. There is certainly room for more constrained notions of implementation that invoke stricter constraints on dimensions (ii) and (iii), building things like teleology and explanation into the definition, though I think the consequence is likely to be a much less precise definition.

Ritchie thinks that the modeling/explanatory distinction somehow undercuts my inference claims about from "implementing CSAs" (his CS_2) to corresponding claims about "implementing computations" (his CS_3), presumably because he takes the former notion to involve modeling descriptions and the second to involve explanatory descriptions. But I use a univocal notion of computation (as in the previous paragraph) throughout: insofar as the former is understood using "modeling" descriptions, so is the latter. Ritchie suggests that this "trivializes" claims of the latter sort, but he gives no argument for this suggestion. Perhaps Ritchie thinks that theses such as computational sufficiency and computational explanation are false when the weaker "modeling" notions are involved, but if so he gives no response to my arguments for those theses.

Ritchie also says I fail to distinguish functionalism from computationalism. Here he is quite correct, as the target article never mentions functionalism. But of course traditional Lewis-Armstrong

functionalism differs from the computationalism I advocate in many respects: the former invokes coarse folk-psychological causal structure where the latter invokes fine-grained organizational structure, the former typically invokes an identity thesis whereas the latter invokes a sufficiency or supervenience thesis, and so on. That said, one can use traditional functionalism to help support the claim that mental properties are organizational invariants, which can then be used to help support computationalism as construed above.

At the end of his article Ritchie suggests that concrete computation is more important than abstract computation. In particular he says that my claims would be more plausible if cast in terms of systems "having computational structure" (his CS_1) rather than "implementing a computation" (CS_2) or "implementing a CSA" (CS_3). I do not know what the distinction between the first two notions can come to: as far as I can tell, for any reasonable notion of a concrete system's having computational structure there is a corresponding and coextensive notion of its implementing an abstract computation. Of course, for different versions of the former there will be different versions of the latter: stronger and weaker notions of having computational structure will correspond to stronger and weaker notions of implementating a computation. But once again, abstract and concrete computation still go hand in hand.

My sense is that the issue that Ritchie and Towl want to raise does not really concern the primacy of abstract or concrete computation, but instead concerns how implementation should be understood. They think that my notion of implementation is too unconstrained, so that too many systems end up implementing too many computations, in a way that does not reflect their real causal structure and does not explain their behavior. The only real argument I can find for this claim in their papers is Towl's "pool table" argument, but I may have missed something. I think the issue is an important one: I am quite open to the idea that the notion of implementation could usefully be constrained, and I am interested in all arguments to that effect. In the meantime I will revisit this issue in the context of trivializing implementations in section 4.

2 Syntactic, Semantic, and Functional Conceptions of Computation

In the target article I offered a "syntactic" conception of computation: abstract computations are characterized in terms of their formal combinatorial structure, and implementation requires only that causal structure of an implementing system reflect this formal structure. Semantic notions such as that of representation played no role in that account. I said that this better captured standard notions of implementation and computation for Turing machines, for example, and also al-

lows computation to better play a foundational role, as the semantic notions stand in need of a foundation themselves.

In response, a number of commentators defend alternatives to the syntactic conception of computation. In particular, O'Brien and Rescorla defend semantic conceptions of computation, while Egan and Klein (as well as Rescorla) defend what we might call functional conceptions of computation, which gives individuation in terms of functions (in the mathematical rather than the teleological sense) a central role.

None of these commentators suggest that theirs is the only tenable conception of computational or that the syntactic conception is incoherent. They suggest simply that the alternative conceptions are also available and can play a central role in understanding computational practice and especially in understanding cognitive science. In effect, we are left with a pluralist position on which there is both syntactic computation and semantic computation, but on which the latter notion can play explanatory roles than the former cannot.

I like the spirit of this pluralism, and I certainly allow that semantic notions of computation can be defined in principle. The key challenges for such a notion, as I see it, are (i) to clarify the relation to the standard mathematical theory of computation, involving Turing machines and the like, which seems to be largely nonsemantic, and (ii) to spell out a corresponding account of implementation, especially one in broadly naturalistic terms.

O'Brien sketches a picture on which there are two very different traditions, a mathematical tradition centering on syntactic computation and an engineering tradition centering on semantic computation. The engineering tradition stems from building control systems to function in the real world and centrally involves representation. As O'Brien sketches things, though, these traditions are so separate from each other that the use of "computation" for both might as well be a pun. He does not attempt to connect the engineering/semantic tradition to the formal theory of computation at all: there is no real role for abstract computations such as Turing machines or for theoretical notions such as universality. Given this degree of separation from the bread and butter of computer science, I wonder whether it is best to avoid confusion by using a term such as "information-processing" or "representational mechanisms" for O'Brien's notion of semantic computation. Certainly these are important topics, but I take my topic here to involve the role of formal computation.

Rescorla's account of semantic computation is much more closely integrated with the formal theory of computation. Rescorla argues that there can be *semantically permeated* abstract computations: those in which certain states are individuated in part by their representational contents. He

gives the example of numerical register machines, in which a certain formal state might represent the number 2 by definition. He also argues that functional programming languages, although typically not regarded as semantically permeated, can be understood in that way. All this is at least reasonably well integrated with the formal theory of computation.

Rescorla's account of semantically permeated computation is certainly coherent. One can raise questions about it, however. It is striking that Rescorla says nothing about what it takes to implement a semantically permeated computation. Presumably it is required that an implementing state itself has the relevant semantic content: e.g. it must represent the number 2. Now we are far from a reductive or naturalistic account of implementation, however. The question immediately arises: under what conditions is the number 2 represented? Rescorla's account gives no answer to that question, instead taking representation as an unreduced notion. So a full understanding of computation then awaits a proper understanding of representation.

This is a far cry from the ambitions of many proponents of a computational theory of mind, according to which a theory of computation is intended to partly ground a theory of representation. It also tends to defang the thesis of computational sufficiency, as representation is already a quasi-mentalistic notion, and there is not much leverage in the claim that having an appropriate representational structure suffices for having a mind. Even those who take themselves to be opponents of computationalism, such as John Searle, could happily agree with this claim.

It is also not easy to see how this conception interfaces with the design and implementation of computer programs in practice. To implement a semantically permeated computation, one will have to independently ensure that states have the right representational content. When it comes to mathematical representation, it is arguable that this does not require too much more than getting the causal structure of the computation right. But when it comes to representation of concrete matters in the external world (especially matters more distal than inputs and outputs), then Rescorla's own externalist views entail that this will not suffice: one will have to ensure an appropriate connection to relevant entities in the external world. It is not clear that anything like this is part of standard practice in the implementation of computation in the wild. In practice, programmers are happy with something like interpretability of symbols and processes as having a certain content (where they could equally be interpreted as having many other contents), and it is not clear that they are in a position to meet the much stronger constraint whereby symbols and processes are really required to have those contents.

Egan and Klein advocate what we might call a *functional* conception of computation: very roughly, a conception on which computation is all about the evaluation of functions. Egan notes

that many computational theories in cognitive science proceed by arguing that systems compute various functions: for example, Marr's theory of vision postulates that edge detection works by computing the Laplacian of the Gaussian of the retinal array. These theories are often neutral on the algorithms by which these functions are computed, andon the causal structure of the underlying process.

Egan does not spell out the relationship between this conception and the formal theory of computation and does not give a corresponding account of implementation. On one way of thinking about things, the relevant functions correspond to input-output relationships in a computation such as an FSA or a CSA. Functions might then be specified simply as a dependence of outputs on inputs, with neutrality on intermediate processes. A function will then be implemented presumably if there is a mapping from physical inputs to formal inputs and from physical outputs to formal outputs such that whenever a formal input produces a formal output, a corresponding physical input will produce a corresponding physical output. Perhaps there might be constraints on what counts as a proper mapping here (Egan puts weight on the fact that outputs must be used by later processes), but the matter is not entirely clear.

Again, I think this conception of computation is a coherent one. There are certainly mathematical functions, and these can be used to characterize input-output relationships. I think this conception loses a great deal of what is central to standard thinking about computation, however. In particular, the notion of an algorithm plays no role here: all that matters to individuating a computation is the dependence of outputs and inputs, with mediating processes being irrelevant. Correspondingly, the formal theory of algorithms is left off to one side. I think it is clear that Egan's functional conception does not capture standard practice of computing in the wild: here input-output dependencies play a useful role in characterizing what we want systems to do, but in software design and implementation the use of algorithms is where much of the central action is found. Furthermore, a thesis of computational sufficiency applied to the functional conception will probably be rejected by anyone who rejects behaviorism about the mind: there is no function such that computing that function suffices for having a mind. That said, it may be that this functional conception is well-suited to capture certain sorts of explanation in cognitive science, especially applied to the analysis of cognitive modules and processes rather than to the cognitive system as a whole. I return to this matter in the final section.

Klein advocates a different functional conception of computation, one tied to functional programming languages. He calls this the "Church paradigm" for computation, as opposed to the "Turing paradigm", as it originates in Church's lambda calculus. Functional programming is much

more fine-grained than a mere specification of a function to be computed, as it breaks down complex functions into a structure of primitive functions. So where Egan's functional conception lies at Marr's computational level, Klein's functional conception lies much closer to Marr's algorithmic level. But unlike the Turing paradigm, the Church paradigm does not specify algorithms at the fine-grained level of transitions between states. Instead, it specifies them at the level of primitive functions computed, which leaves a large amount of flexibility in state-transitional structure. In effect this is a level intermediate between Egan's and mine, one that Klein thinks may have special relevance to explaining cognitive processes.

There is no question that functional programming languages play an important role in computational practice. My own first programming language was APL. (In the late 1970s, members of my high school's computing club found that APL's highly compressed programs were ideally designed to send on punched cards to the computer across town!) Much work in AI proceeds in Lisp. Both of these are largely (although not purely) functional programming languages. That said, there are interesting and difficult questions about the relationships between programming languages and other computational formalism. Any such language requires an interpreter or a compiler to be executed, and almost any interpreter or compiler leaves some slack between a program and the resulting set of state-transitions. This applies even to imperative languages, though declarative languages typically leave more slack here. So one might think of high-level programs quite generally as a level intermediate between the state-transition level and the input-output level. But it is reasonable to think of functional programs as somewhat further from the state-transition level than imperative programs, while still counting as algorithmic in a broad sense.

An account of implementation for imperative programming languages is fairly straightforward. One can translate a program into patterns of state-transitions, in effect yielding a class of CSAs for any given program. Then one can understand a physical system as implementing a given program if it implements a corresponding CSA. Perhaps something similar applies to functional programming languages, except that the translation from programs to state-transitions will have somewhat fewer constraints, so that more CSAs will correspond to a given program. If this is the right way to think about things, CSA-style formalisms will still play a central role in understanding implementation. The discussion of abstract-state machines in the following section lends further support to this view.

Rescorla also discusses functional programming languages, and in effect suggests that a functional conception of computation can be combined with a semantic conception. Here the idea is that the arguments and values of the relevant functions can be seen as entities in some represented

domain such as numbers. Then the relevant functional programs can be seen as semantically permeated, and there will then presumably be additional semantic constraints on implementations, as before. Rescorla notes that this semantic conception of functional programming is not obligatory (and Egan and Klein say little about semantics in their discussion of functional conceptions), but he seems correct to say that it is available. Of course, it will then share both the upsides and the downsides of both the semantic and the functional conceptions taken alone.

In any case, functional programs are certainly important abstract objects in computer science, and any full account of computation should accommodate their role. They may well provide a useful intermediate level of description, and combined with an adequate account of implementation they might plausibly feed into theses of computational sufficiency and explanations. Something similar applies to semantically permeated computation, though my grip on this notion and its role is somewhat less secure. In any case, as a pluralist, I am inclined to think that intermediate levels of explanation certainly have an important role to play, and these conceptions seem to characterize promising intermediate levels on which to focus. I return to this matter in the section on computational explanation.

3 Are Combinatorial-State Automata an Adequate Formalism?

In the target articles I introduced combinatorial-state automata (CSAs) as the abstract computations used to illustrate my account of implementation. The reason for not using more familiar abstract objects such as Turing machines and finite-state automata is that most of these are idiosyncratic in some way: Turing machines use the same mechanism (a tape) for inputs, outputs, and aspects of internal state; finite-state automata lack states with combinatorial structure; cellular automata lack inputs and outputs as standardly conceived; and so on. Combinatorial-state automata are simply a natural way of capturing an abstract structure involving distinct inputs, outputs, and complex internal states. Furthermore, I suggested, almost any standard computational formalism can be translated into the CSA formalism.

Brown, Milkowski, and Sprevak all suggest that the CSA formalism is in some way inadequate. Brown says that CSAs are too powerful to serve as a computational model in the way that Turing machines do, as CSAs can compute noncomputable functions. In addition, all three of them argue that CSAs are not as general as I suggested: although one can translate Turing machines and the like into CSAs, the translations are often complex and unperspicuous and lose important distinctions, yielding accounts of what it is to implement a Turing machine that is correspondingly

unperspicuous.

On the first point: I think that here Brown reads more into my ambitions for CSAs than I intended. I did not intend to be making a contribution to the mathematical theory of computation by introducing them. I simply intended an easy-to-understand formalism in which the key issues about implementation would be made clear.

Certainly, CSAs with unrestricted structure are more powerful than Turing machines, not least because they can encode an infinite amount of information. I do not think that this means they are not a "model of computation", as Brown suggests: after all, there are many different sorts of computation, including Turing-level computation and super-Turing-level computation. One might view unrestricted CSAs as a super-Turing model that subsumes Turing-level computation as a special case. To restrict CSAs to Turing-level computation, one needs certain restrictions on their structures: for example, one can require a certain sort of uniformity in all but a finite number of state-transition rules and starting states. But in any case, none of this matters for the account of implementation. If I have given an account of implementation that works for both Turing-level and super-Turing-level computation, all the better!

The second point, about problems translating from other formalisms into CSAs, is somewhat more worrying. It is true that in the target article I suggested that such translations could be accomplished straightforwardly, and I implied that these translations would involve no loss of generality. This is a substantive claim and open to counterexample, however.

Brown objects that CSA specifications will in general be extremely complex compared to corresponding TM characterizations. That is right, if we simply treat the CSA state-transitions as a huge list. Those CSAs that correspond to TMs will have a much more uniform structure, however: roughly, there will be uniform transition-rules for each substate corresponding to a square of the TM tape. That uniformity will enable us to represent the CSA state more briefly and perspicuously. Brown also objects that CSA descriptions will not generalize across changes in tape size, but I think these compressed descriptions that exploit uniformity will naturally generalize.

Sprevak worries that CSAs obscure certain distinctions that are crucial to TM architecture. The distinction between control structure (machine head) and data (tape) that is central to TM architecture is not reflected in CSA architecture. Furthermore, CSAs allow a sort of random access parallel computation whereas TMs involve local-access serial computation. I think that loss of distinctions and constraints is an inevitable result of moving from relatively specific models to relatively general models. Of course loss of constraints makes a difference at the level of formal theory. But it is not obvious to me why it is a problem where the account of implementation is

concerned. Again, those CSAs that correspond to TMs will merely be a tiny subclass of the class of CSAs, and will have a strongly constrained sort of structure. The substates will in effect be divided into two very different sorts, the machine head substate and the tape substates. The machine head substate can directly affect each tape substate, whereas tape substates can only affect the machine head substate (and indirectly, neighboring substates). The effects of machine head substates on tape substates are constrained to operate in a certain sequential order. All of these constraints must be reflected in an implementation of the CSA, just as they must be reflected in an implementation of the TM.

It is not clear, then, that the CSA-translation account gives any incorrect results as an account of what it is to implement a TM. Sprevak suggests that in a TM implementation, the machine head and the tape must be implemented in physically distinct ways. I think this is false. One can certainly implement a TM by encoding the machine head states and tape states in a physically uniform array I would guess that the vast majority of existing implementations of Turing machines, constructed via the mediation of data structures in high-level programming languages, take this form. One will then need surrounding connection structures to make sure that the head state can affect the tape states, but the result will be an implementation for all that. So I do not think there is a problem for the CSA account here.

Milkowski objects that a TM and its translation into a CSA may differ in their causal complexity. As a result, a TM implementation may be susceptible to certain patterns of breakdown (a tape head wearing out the paper for example) that CSA implementations need not be. However, he does not give any reason to think that the class of TM implementation differs from the class of translated CSA implementations, so it is hard to see how this difference can come about. Perhaps Milkowski is focusing on certain famous examples of TM implementations (with paper tape, for example). Once we recognize (i) that there is a vast class of alternative implementations, as in the previous paragraph, and (ii) that the relevant (TM-translating) class of CSA implementations will be a tiny subset of all CSA implementations, as in the previous-but-one paragraph, there is no longer any clear reason for believing in a principled difference in causal complexity among these classes.

Something similar applies to Milkowski's discussion of spatiotemporal characteristics. Certainly the spatiotemporal characteristics of an implementation are important, and a CSA specification will underdetermine these characteristics just as it underdetermines other implementational features, but precisely the same goes for a TM specification. So we have not yet uncovered any problems for the CSA-translation account of TM implementation.

It is worth noting that the notion of "translation" is somewhat stronger than is needed for present purposes. To exploit the CSA formalism to understand TM implementation, we need not claim that for every TM there is a "translated" CSA such that a physical system implements the TM iff it implements the corresponding CSA. We can allow instead that there is a many-to-one relation between CSAs and TM, so that multiple CSAs can implement a TM, and we can then say that a physical system implements the TM iff it implements a CSA that implements the TM. Of course this requires spelling out an implementation relation between CSAs and TMs, but this need not be much more difficult than spelling out a translation relation. It has the advantage of allowing that there can be arbitrary choices in "translation"—for example, is machine head location coded in an aspect of tape substates in the CSA or an aspect of head substates?— so that more than one kind of translated CSA can implement a TM.

All that said: suppose that Brown, Sprevak, and Milkowski are right that there are some distortions in the translation from a TM to a CSA, so that the class of TM-implementations and corresponding CSA implementations are not coextensive. Even so, this would not cause major problems for the spirit of the current account. CSAs are really serving as a useful illustration of a model of computation and an associated account of implementation. Once one understands the account of implementation given here, one can readily adapt it to give an account of implementation for alternative models such as TMs: not via translation to CSAs, but by implementation constraints in their own right. In the case of TMs, there will need to be elements in an implementation implementing machine head state, elements implementing tape states, and causal relations between these, all mirroring the formal relations between the corresponding formal states. This account of TM implementation will not explicitly mention CSAs, but it will nevertheless be an account in terms of causal organization, and one to which the main points of my discussion will apply.

More generally, my account of CSA implementation can readily be extended to handle other models of computation. Milkowski mentions analog computation. To handle this, we need only extend CSAs to allow real-valued states, with appropriate formal dependencies between these states, and to allow physical implementations to have real-valued parametrized substates, with corresponding causal or counterfactal dependences between these substates. This will not literally be a translation from an analog computation into a CSA, but it will be a natural extension of the current account, and one to which the main consequences still apply. Something similar applies to the framework of quantum computation.

Likewise, Sprevak mentions more abstract cognitive architectures such as ACT-R and MO-

SAIC. It may be that any problems in translating these to CSAs can be handled much as I handled the problems for TM translation above. But if there are more serious problems, then we can simply modify a CSA-style account of implementation to handle implementation of ACT-R and MOSAIC systems directly.

Milkowski makes the very useful suggestion that my appeal to CSAs could be replaced by an appeal to ASMs: abstract state machines. ASMs were only just becoming prominent when I wrote the target article (the first international conference on them was held in 1994), and I was only dimly aware of them before reading Milkowski's article. But I think that they can certainly play a very useful role here. Like CSAs, they serve as a general formalism, but they have a much more fine-grained structure that allows a variety of architectures and data structures to explicitly be modeled. Furthermore, ASMs can be cast at multiple levels of abstraction, all the way from highly detailed low-level models that specify every state-transition, as a CSA might, to much more high-level models that specify computational structure at the levels of functions or commands. And crucially, the theory of ASMs comes with an account of when a low-level ASM implements a high-level ASM.

I think that the ASM framework is just what is needed to make the framework of the target article more general and powerful. In effect, it formalizes and extends the informal idea that computations of all sorts can be "implemented" in a CSA (here, a low-level model). We can then give an account of the conditions under which physical systems implement low-level models, very much along the lines given in the target article. Combined with the ASM framework's account of when low-level models implement high-level computations, this will yield a general account of when computations of all sorts are implemented.

This is unquestionably a very useful thing. It helps us to answer difficult questions (as arose in the previous section) about the relationship between high-level computational structures such as functional programs to the account of computation and implementation given in the target article. And insofar as the "ASM thesis" holding that one can use ASMs to model any algorithm in any framework is correct, we will then have a general account of implementation that applies to any algorithm.

4 My account of implementation

The centerpiece of the target article is my account of the conditions under which a physical system implements a computation. I first offer a rough gloss: a physical system implements a computation

when the causal structure of the physical system mirrors the formal structure of the computation. I go on to offer a more detailed gloss in terms of mappings from groups of physical states to formal states:

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

Finally, I offer a fully detailed version of the account that applies to CSAs. Here a vectorization is a way of specifying the internal state of a physical system as a vector of substates, one substate for each of a number of elements of the system, where each element of the vector corresponds to a spatially distinct element of the physical system.

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

Before getting to the crucial issue of whether this account lets in trivializing implementations, I will address some more minor concerns. Scheutz makes heavy weather of my use of "mirrors" in the rough gloss, noting that the physical system will have a more fine-grained causal structure than the corresponding computation, so that the relation is more one of homomorphism than isomorphism. I think the right way to think about things is to conceive of physical systems as having multiple causal structures, corresponding to different ways of grouping states of the system into state-types. Then my rough gloss might be better put by saying that implementation obtains when *a* causal structure of the physical system mirrors the formal structure of the computation. In any case, there is no problem for the more precise definitions here.

Scheutz also discusses Kripke's (1981) objection that no system perfectly implements a formal computation as it will eventually wear out or malfunction in other ways. For what it is worth,

the account in the target article certainly does not require that for a system to count as an implementation, it must mirror the computation forever. The key notion is really one of a system implementing a computation *at a time*. The constraints all concern the system's structure at a given time: in particular, it must satisfy various conditionals of the form *if* it is in a certain state, it will transit to another specified state. As far as I can tell, Kripke's objection gives no reason to think that these conditions cannot be satisfied. At best the objection gives reason to think that a system that satisfies these conditions at one time will not continue to satisfy them forever. But that simply yields the reasonable conclusion that a system that implements a given computation will not implement it forever. That is just what we expect for ordinary physical systems such as computers and brains.

Note that on this account, there is no need to invoke counterfactuals about "going successively through the states of a sequence" if the system "satisfies conditions of normal operation for long enough", as on the account of Stabler (1987). All that is required is the simple counterfactuals above. If a system satisfies those counterfactuals and continues to satisfy them over time, it follows that it will always go successively through the states of relevant sequence: if it satisfies A $\rightarrow$ B and B $\rightarrow$ C conditionals, and continues to satisfy them, then a transition from A to B will always be followed by a transition from B to C. (Note that on my account, the relevant conditionals are not "nearest A-world" counterfactuals, which might violate this requirement, but "all A-world" counterfactuals, which will not). There is no need to build in this condition about sequences explicitly. And there is no need to say anything about conditions of normal operation (although see later for a somewhat different reason to appeal to background conditions). Where Stabler's account says that a malfunctioning system may still be implementing the computation that it would have implemented under normal conditions, my account says that it is no longer implementing the computation. That seems to be the right result to me.

Sprevak asks "What is the mapping relation, and what metaphysical commitments does it bring?". I confess that I do not understand either question. The definition of implementation does not appeal to any specific mapping relation: rather, it quantifies over mapping relations, which can be any function from physical states to formal states. I also do not know what it is for a relation to have metaphysical commitments. I suppose that there is a question about whether Platonism or nominalism is true about relations, but the current account is not really any more committed to Platonism than standard mathematics is; one might well formulate a nominalist version of the implementation conditions. Sprevak also talks about whether relations are mind-dependent, but I do not know what it is for a set of ordered pairs of physical and formal states to be mind-dependent.

I now turn to the crucial question of whether there are "trivializing" implementations of CSAs: implementations that satisfy the official conditions while lacking the sort of complex structure that a CSA implementation should have. The locus classicus of trivialization objections is Hilary Putnam's argument, in the appendix to *Representation and Reality* (1987), that every ordinary open system implements every finite-state automaton. I responded to that argument at length in "Does a Rock Implement Every Finite-State Automaton" (written in April 1993, around the same time as the target article). There I suggested that Putnam's argument could be evaded by requiring the system to satisfy state-transition counterfactuals. One could then devise more complex trivializing implementations by requiring systems to contain a "clock" (an element that goes into different states at each different time), a "dial" (an element that can be set to various different readings and stays there). I suggested that if the opponent of trivialization appeals to inputs and outputs to block these moves, this addition can be trivialized by adding an "input memory" (an element that contains a record of all inputs to date). However, I argued that all of these trivializations are defeated by requiring automata to have combinatorial structure: hence CSAs.

Even in that paper, I conceded that there are residual issues about trivializing implementations. First, to avoid CSAs collapsing into FSAs and hence being trivializable, I had to impose a "spatial independence" constraint on the substates of a CSA; this condition rules out the relevant trivializations, but at cost of ruling out some reasonable implementations as well. Second, there are still some astronomically large false implementations, derived by having every state of the machine record every input and every previous state of the machine (an "input/state memory"). These implementations will be infeasible in practice due to a combinatorial explosion in the size of an input/state memory, but even as hypothetical systems it seems incorrect to count them as implementations. So in that article I entertained the possibility of adding further constraints, including further constraints on the physical state-types (e.g. a naturalness constraint) or on the physical state-transitions (e.g. a uniformity constraint), although I left the matter open.

Of the commentators, Brown, Scheutz, and Sprevak all offer trivializing implementations for CSAs. These are all intended to be realistic false implementations rather than the astronomically large false implementations that I offered.

Scheutz puts forward a complicated trivializing implementation that involves states that are temporally individuated. I think that temporally individuated states should certainly be excluded in an account of implementation, so there is no real problem so far. Scheutz points out that the role of temporal individuation can be handled nontemporally simply by adding a clock to a system to play the role of the time. If only a single clock is added and it is used to individuate all physical

substates, however, this implementation will violate the condition that all physical substates need to be spatially independent. Scheutz might reply in turn by adding n clocks to the system, one for each physical element of the system. But there will now be total physical states in which these clocks are set to different values, and under which the system then transits inappropriately. So Scheutz's construction does not yet yield false implementations. Scheutz might try further moves such as requiring that the clocks be kept synchronized. This in effect will reduce his construction to a variant of Brown's simpler construction below, so I will concentrate on that construction here.

Sprevak also raises triviality issues. His discussion is mainly devoted to the spatial independence constraint discussed above. He first argues that spatial independence of components is not *necessary* for implementation, an issue that I concede above. He also briefly argues that these conditions are not sufficient for implementation, giving a sort of triviality argument. On my reconstruction of this brief argument, there seems to be a hole in it. Sprevak makes a case that every component of a CSA can be construed as an FSA with "weak" inputs and outputs (that is, where any physical inputs and outputs could realize these under some mapping), and he notes that any such FSA is realized by trivial systems, so he concludes that the CSA as a whole is realized by trivial systems made up of these trivial components. However, this last step does not seem to follow. To implement the CSA as a whole the realizers of the weak outputs from one component will have to cause or coincide with the realizers of the corresponding weak inputs to another component. As far as I can tell Sprevak's construction cannot guarantee this. I may be missing something, but in any case I will focus on Brown's construction here.

Brown's trivialization is an extension of my argument that a system with a clock and a dial implements every inputless FSA. He suggests that to implement an inputless CSA with n elements, we need only have n components each consisting of a clock and a dial, and a mechanism to ensure that all n clocks are always in the same state, as are the n dials. (He also suggests that each element be distinguished by a separate marker with a number i between 1 and n, but this marker plays no essential role.) In the same way that I mapped every FSA state to a disjoint set of (clock, dial) states, he maps every total CSA state to a disjoint set of (clock, dial) states. Once this is done, one can map every CSA substate (of the ith formal component) to the set of (clock, dial) states (of the ith physical component) that map onto total CSA states in which that substate occurs. Then all the conditionals required for an inputless CSA will be satisfied.

Now, the CSAs that I consider have inputs and outputs, a requirement that Brown does not explicitly discuss. Presumably he will handle these the same way I handle FSAs with inputs and outputs: by adding an input memory to each of the n components (where all n are required to

have the same state), and then proceeding as above. The result is close to a potential trivialization involving synchronized input memories that I consider in the 1995 article. I responded to that trivialization by noting that this system would not satisfy required counterfactuals about how the system would behave when in states in which the input memories for each component were not synchronized. Brown in effect counters that response by requiring the input memories (or the clocks) to be synchronized as a matter of physical necessity.

This ingenious move of Brown adheres to the letter of the definition of implementation. That definition does not require that physical substates can be arbitrarily recombined: if some recombinations of physical substates are physically impossible, then they will be irrelevant to assessing the truth of the counterfactual. In fact, the definition I gave does not explicitly require that formal substates can be arbitrarily recombined, in that there is a total physical state mapping to every possible total formal state, although that requirement should certainly be there. Brown's implementation in effect satisfies the second requirement without satisfying the first. Because recombinations of input memory states (or clock states) are physically impossible, they are irrelevant to assessing the truth of the counterfactuals, and my response is circumvented.

My first inclination is to respond to Brown by requiring that physical substates be arbitrarily recombinable. After all, they are already required to be spatially independent, and given a spatial recombination principle (setting worries about quantum mechanical entanglement aside) this implies that they are recombinable. Brown suggests that this is too strong a requirement:

> Perhaps we could implement a Turing machine on a PC in such a way that the contents of the tape can be stored either in RAM or on the hard drive. Then for every square there would be two ways to implement the state in which the square contains a stroke. But it might be impossible to combine the first implementation of square 1 containing a stroke with the second implementation of square 2 containing a stroke: perhaps the contents of the entire tape must be stored in the same location.

However, it will certainly not be *physically* impossible to combine the RAM implementation of the first square with the hard drive implementation of the second square. It is just that if we do so, we will not get the right results. The upshot is that the system that Brown describes, under the mapping from RAM/hard-drive disjunctive states to tape states, does not qualify as a true implementation of a Turing machine on my original definition. Of course it might still qualify as an implementation under a simpler mapping from RAM states or hard-drive states alone to tape states. Strictly speaking, something similar applies to Brown's trivializing implementation. In a real

physical system, it will not be *physically* impossible for the clocks and dials to be unsynchronized: one need only imagine that the bars that keep them synchronized are broken, for example. So Brown's system does not really qualify as an implementation after all.

Now, Brown might object that under conditions of normal functioning, or under normal background conditions, the clocks and dials will always be synchronized. Likewise, he could object that under normal background conditions, only the hard drive or the RAM will be active. This requires that the definition in the target article be restricted to include "conditions of normal functioning" or "under certain background conditions", thereby complicating the definition. But it is arguable that some such complication is needed anyway, in order to avoid physically possible situations in which a system is in the right physical substates but goes completely awry due to unusual background conditions.

For example (simplifying away from structure, inputs, and outputs), the definition might require that there be a mapping M *and* conditions C [that currently obtain] such that for every formal state-transition rule $S_1 \rightarrow S_2$, if conditions C obtain and the system is in a total physical state that M maps to S_1, it transits to a total physical state that M maps to S_2. To capture "formal recombinability", we might also require that for every formal state there be some physical state compatible with conditions C that map onto it. But conditions C need not permit full physical recombinability.

Obviously this matter is quite complex. The need to invoke background conditions adds a degree of complexity to the definition, and one might worry that it lets in further room for trivializing implementations. Without it, we can arguably escape trivialization, but at cost of arguably overstrong requirements on implementation in the form of conditionals that apply to all physically possible configurations of a system. Furthermore, the claim that spatially distinct elements of a system are physically recombinable seems at best to be a contingent physical truth, not one that an account of implementation can rely on.

One might at this point think it is best to add further constraints to our account of implementation: constraining either physical state-types (via naturalness) or physical state-transitions (perhaps via uniformity), as suggested earlier. Either of these sorts of constraints might rule out the trivializing implementation. However, it is not obvious how to draw these distinctions in a clear and precise way.

An alternative option is instead to appeal to a familiar consideration: inputs and outputs. In particular, on the definition given above, internal states must be causally and counterfactually connected to outputs. On Brown's construction, the internal physical states that realize formal states will not cause the relevant outputs at all. He gives the clocks and dials (and input memories)

no causal role in producing outputs. Instead it is simply assumed that outputs are produced by some part of the system. But as the construction stands they are not produced by the clocks, dials, and input memories.

The problem here goes back to an error in my arguments in "Does a Rock Implement Every Finite-State Automaton". There I argued that every FSA is implemented by every system with the right input-output dependences, as long as we adjoin a dial and an input memory to it. I said that the FSA would satisfy the conditionals connecting internal states to outputs. But this was incorrect. On the construction I gave, there is no reason to think that the relevant internal states (of dial and input memory) are causally connected to the output at all. The output will instead be produced by a state S of the original system, independent of the dial and input memory. Furthermore, the construction does not satisfy the relevant counterfactual conditional, even shorn of the causal requirement. There will be possible states of the system that recombine state S of the original system with arbitrary states of the dial and input memory. All of these states will produce the same output, whereas in fact many states of the dial and input memory will need to produce different outputs to implement vthe FSA. This error in my argument (which I discovered only by reflecting on Brown's related argument) is unfortunate, but it helps rather than hurts the case against trivialization.

Of course one could simply adjoin an additional artificial output produced by the input memory. On Brown's construction, for example, one might adjoin yet another input memory synchronized with the other input memories and brought about by them, for example. One could then map a disjunction of states of the output input memory to formal outputs in the familiar way. Scheutz's conjunction involves a disjunctive output state in a similar spirit. However, this will only work if we allow what Sprevak calls *weak outputs*: physical outputs that map onto formal outputs by any mapping. As Sprevak notes, weak outputs do not add significant constraints to the usual CSA constraints. For output constraints to make a difference, we must require *strong outputs*, with physical constraints on what sort of physical outputs are allowed. In my 1994 paper I noted that constraints of this sort would be needed, though I did not go into detail. These constraints will block input memories and the like from qualifying as appropriate realizers of output states.

There are two ways such constraints might be spelled out. First, we might individuate the inputs and outputs to an abstract CSA as physical state-types, not as formal states. Then a physical realizer will be required to handle those states as input and outputs. This would turn abstract computations into an unusual hybrid of formal and physical states, however. Second, we can individuate the inputs and outputs to an abstract CSA formally, for example as 1's and 0's, and

then we can constrain what sort of physical state-types count as realizing these formal outputs. Here one could choose an arbitrary realizing physical state-type (say, low voltage for 0 and high voltage for 1). Or perhaps better, one could impose a naturalness constraint, requiring that states be relatively natural (or a related constraint such as requiring that that outputs be perceptually distinguishable by an ordinary human cognitive agent).

Of course this last move inherits some of the obscurity of the appeal to naturalness in understanding internal states, but at least the appeal is restricted to inputs and outputs, for which the constraint is especially well-motivated. In fact, merely invoking naturalness for output realizations suffices to rule out the relevant false implementations. Still, once the appeal is made, it would also be possible to invoke it more widely in restricting internal states as well. All this tends to bring out that there is some inevitability in the appeal to naturalness in understanding the notion of implementation. This will not be a surprise to those who have witnessed the widespread use of naturalness in analyzing many philosophically important notions.

5 Computational sufficiency

We now come to issues that connect computation and cognition. The first key thesis here is the thesis of computational sufficiency, which says that there is a class of computations such that implementing those computations suffices to have a mind; and likewise, that for many specific mental states here is a class of computations such that implementing those computations suffices to have those mental states. Among the commentators, Harnad and Shagrir take issue with this thesis.

Harnad makes the familiar analogy with flying, digestion, and gravitation, noting that computer simulations of these do not fly or digest or exert the relevant gravitational attraction. His diagnosis is that what matters to flying (and so on) is *causal* structure and that what computation gives is just *formal* structure (one which can be interpreted however one likes). I think this misses the key point of the paper, though: that although abstract computations have formal structure, implementations of computations are constrained to have genuine causal structure, with components pushing other components around.

The causal constraints involved in computation concern what I call causal organization or causal topology, which is a matter of the pattern of causal interactions between components. In this sense, even flying and digestion have a causal organization. It is just that having that causal organization does not suffice for digestion. Rather, what matters for digestion is the specific bi-

ological nature of the components. One might allow that there is a sense of "causal structure" (the one that Harnad uses) where this specific nature is part of the causal structure. But there is also the more neutral notion of causal organization where it is not. The key point is that where flying and digestion are concerned, these are not organizational invariants (shared by any system with the same causal organization), so they will also not be shared by relevant computational implementations.

In the target article I argue that cognition (and especially consciousness) differs from flying and digestion precisely in that it is an organizational invariant, one shared by any system with the same (fine-grained) causal organization. Harnad appears to think that I only get away with saying this because cognition is an "invisible" property, undetectable to anyone but the cognizer. Because of this, observers cannot see where it is present or absent—so it is less obvious to us that cognition is absent from simulated systems than that flying is absent. But Harnad nevertheless thinks it is absent and for much the same reasons.

Here I think he does not really come to grips with my fading and dancing qualia arguments, treating these as arguments about what is observable from the third-person perspective, when really these are arguments about what is observable by the cognizer from the first-person perspective. The key point is that if consciousness is not an organizational invariant, there will be cases in which the subject switches from one conscious state to another conscious state (one that is radically different in many cases) without noticing at all. That is, the subject will not form any judgment—where judgments can be construed either third-personally or first-personally – that the states have changed. I do not say that this is logically impossible, but I think that it is much less plausible than the alternative.

Harnad does not address the conscious cognizer's point of view in this case at all. He addresses only the case of switching back and forth between a conscious being and a zombie; but the case of a conscious subject switching back and forth between radically different conscious states without noticing poses a much greater challenge. Perhaps Harnad is willing to bite the bullet that these changes would go unnoticed even by the cognizer in these cases, but making that case requires more than he has said here. In the absence of support for such a claim, I think there remains a prima facie (if not entirely conclusive) case that consciousness is an organizational invariant.

Shagrir makes a different sort of case against computational sufficiency, saying there are cases in which a single system will implement two mind-appropriate CSAs and will therefore have two minds. I am happy to say there are such cases, as I did in the original paper. These cases can even arise when the CSAs are entirely different from each other. Most obviously, one could simply put

the two CSAs next to each other in one skull, or implement them in some interwoven way where the circuits of one crosscut the other. In such a case, the system as a whole will have twice the complexity, of the CSAs taken alone. I see no problem with there being two minds present here any more than in the case of conjoined twins with a single skull. These cases may be somewhat different from the cases Shagrir has in mind, but I mention them for completeness.

Shagrir raises three objection to systems supporting two minds. One concerns competing control over the same outputs—but in the case I have in mind the systems will have quite different outputs, so there is no problem here. Another concerns supervenience—but it is a familiar point from the metaphysics literature that two distinct systems (the statue and the lump), for example can depend superveniently on the same portion of matter. What counts for supervenience here is that when one duplicates the portion of matter, one duplicates both minds, and there is no obvious obstacle to that claim. Shagrir's third objection is that theorists will deny that there are two minds here—but I think that in cases like the one I have in mind they will have no problem.

Now, the sort of two-CSA cases that Shagrir describes are somewhat different from the two-independent-CSA cases that I describe. They involve two structurally similar CSAs implemented using tristable instead of bistable gates, so that relative to one way of dividing up the structure one gets one CSA and relative to another way, one gets the other. In this case, unlike the case above, the overall system may have complexity only somewhat greater than the original CSAs taken alone, as there is so much overlapping structure between the systems.

Now this sort of cases is far less general than the cases above—it will only work for some pairs of CSAs. So we cannot simply take any two CSAs that support minds and assume that this construction will work. Most of the time the systems will be just too different. So it is far from obvious that there will be such cases in which the CSAs support two distinct minds. It may be that in any such cases, either one supports a mind and the other does not, or alternatively they both support the same mind (minds might be multiply realized by CSAs, after all). Either of these results would be no problem at all for the framework.

Even if there are cases of Shagrir's sort where both CSAs are mind-appropriate and both support distinct minds, I do not see a strong objection to allowing both minds here. Of course the system in question would be an unusual one—Shagrir does not try to make the case that something like this is going on with our brains. We could give the same response to the objection from supervenience as above. On the objection from control, in this case the outputs will overlap, but it appears to be a case in which the outputs are always synchronized with each other so there is no real competition. On the objection from explanation, I agree that it is far from obvious that

theorists would allow that there are two minds here. Indeed, I am not sure that I would allow two minds in a given case—but this would be because I would not be sure whether it counted instead as one of the cases in the previous paragraph.

One might worry that there are Shagrir-like cases closer to home, perhaps even involving human brains. A brain presumably implements a complex mind-supporting computation, but perhaps it also implements various other computations that would support a mind in their own right if they were implemented in another context? For example, perhaps a duplicate of the prefrontal cortex (with its inputs and outputs) taken alone would support a conscious mind that differs from our own? I think it is far from obvious that there are such computations, and if there are it is far from clear why, if they would support a conscious mind alone, they should not also support the same conscious mind in the context of a whole brain. It is also worth noting that cases of this sort pose the same sort of obstacle to (local) microphysical supervenience thesis, saying that physically identical systems have the same mental states, as they do to the thesis of computational sufficiency.

Still, if one worries about this sort of case, a small modification to the principle of computational sufficiency is possible. We could say that a mind-supporting computation *c*is *derivatively implemented* when it is implemented in virtue of some other mind-supporting computation c' being implemented, where c' implements c. Then in the case above, the prefrontal cortex derivatively implements the relevant computation in the context of a whole brain but nonderivatively implements it when taken alone. Then one could modify the principle to say that *nonderivative* implementation of the relevant computation suffices for a mind. (One could then either deny that derivative implementation ever yields the same sort of mind or simply stay neutral on that matter.) I am not sure whether such a modification is necessary, but in any case it does not seem far from the spirit of the original principle.

Before moving on, it is worth addressing a related worry for my account of implementation that is often raised. This is the objection that my account leads to pancomputationalism (Piccinini 2007), the thesis that everything implements some computation, and that this thesis is implausible or trivializes the appeal to computation. In the target article I noted that it does not trivialize the account, as any given complex computation will be implemented only by a few systems, so the claim that a system appeals a specific complex computation will be nontrivial. Still, the objection has continued to arise, so it is worth taking a moment to mount a counterargument.

One might say to opponents of pancomputationalism: there are certainly trivial abstract automata, such as a one-state FSA. Question: Does a rock, say, implement the one-state FSA? It seems implausible that it does not. If it does, then presumably just about any system also imple-

ments this FSA. And if so, we have a sort of pancomputationalism. Perhaps an opponent might respond by saying that this FSA is too simple to count as a computation. To which we can respond: now the issue concerns what criterion to use to divide the abstract objects that are computations from those that are not, rather than concerning the theory of implementation. Given such a criterion, we can straightforwardly combine it with the current theory of implementation and thereby avoid pancomputationalism after all. If so there is no objection to the current theory here. An opponent might instead allow that a one-state FSA is a computation but invoke further constraints that rule out the rock as an implementation, including etiological constraints about its design or purpose. Such constraints will clearly be superfluous for many explanatory purposes, however. Perhaps they will be useful for some purposes, but in that case they can also be added to the current account as supplementary conditions. Either way something like the current account will plausibly be at the core of a theory of implementation, perhaps supplemented by further constraints for certain purposes.

6 Computational Explanation

My second key thesis in the target article was the thesis of computational explanation, saying that computation (as I have construed it) provides a general framework for the explanation of cognitive processes and of behavior.

I think that this claim in the target article was overly strong, or at least needs to be read in an attenuated way. First, there are aspects of cognition that computation as I construe it does not explain well. Second, there are forms of explanation in cognitive science that my model of computation does not capture well. Already in the single footnote added to the target article, I suggested that the thesis should be modeled to say "*insofar as cognitive processes and behavior are explainable mechanistically*, they are explainable computationally.". That modification is intended to in effect allow for both of the limitations mentioned here.

Egan and Rescorla focus especially on forms of explanation in cognitive science that my model of computation does not capture well. Egan focuses on function-theoretic explanation: for example, Marr's explanation of edge detection in terms of computing the Laplacian of the Gaussian of the retinal array. Rescorla focuses on representational explanation: for example, Bayesian models in perceptual psychology cast in terms of the confirmation of hypotheses about the scene in front of one.

I concede right away that these are reasonable forms of explanation in cognitive science and

that they do not give a central role to computation as I construed it. I do not think I would have denied these things when I wrote the article, either. Most likely I would not have counted them as distinctively computational explanations. I intended to put representational explanations in a different class, and the same may have gone for function-theoretic explanations. Certainly there is a sense in which these strike me as at best incomplete qua computational explanations. But of course the issue of what counts as "computational" explanation is verbal, and I would not be inclined to put too much weight on it now.

In any case, however we label them, I take representational and function-theoretic explanation—like social explanation, purposive explanation, and other forms of explanation—to be higher-level forms of explanation that are quite compatible with computational explanation in my sense. A representational explanation or a function-theoretic explanation can be filled out with a causal/computational story. I take it that while these explanations are reasonably good explanations on their own, filling them out makes for a better and more complete explanation. In particular (to take the line suggested by the added footnote), the representational and functional story lack a specification of the *mechanisms* by which hypothesis confirmation proceeds or by which functions are computed. A computational story (in my sense) allows these mechanisms to be specified.

Still, one might wonder whether computational explanation has any especially privileged role as a preferred framework for explanation. Around here a lacuna in the target article reveals itself. I argued for the thesis of computational sufficiency by arguing:

> "insofar as mental properties are to be explained in terms of the physical at all, they can be explained in terms of the causal organization of the system. We can invoke further properties (implementational details) if we like, but there is a clear sense in which they are not vital to the explanation. The neural or electronic composition of an element is irrelevant for many purposes; to be more precise, composition is relevant only insofar as it determines the element's causal role within the system. An element with different physical composition but the same causal role would do just as well. This is not to make the implausible claim that neural properties, say, are entirely irrelevant to explanation. Often the best way to investigate a system's causal organization is to investigate its neural properties. The claim is simply that insofar as neural properties are explanatorily relevant, it is in virtue of the role they play in determining a systems causal organization."

I think that what I said here was correct, but it does not go all the way to establishing the the-

sis. It makes a case that computational explanation is at least as important as neural explanation. But it does not really make a case concerning the importance of these forms of explanation compared to representational explanation and the like. So far as this argument is concerned, it leaves open the possibility that there are other higher-level forms of explanation that stand to computational explanation roughly as computational explanation stands to neural explanation. One way to construe Egan's and Rescorla's arguments is as making a case that that function-theoretic and representational explanation (respectively) are precisely such forms of explanation. That is, they capture the most essential elements in a computational explanation while leaving out inessential implementational details concerning the fine details of causal organization.

Around here I am inclined to take the pluralist line. There are many forms of explanation in cognitive science, and computational explanation in my sense is not more important than neurobiological or representational explanation, say. I do think computation captures one important kind or level of explanation, though.

In particular, I think computational explanation as I understand it is the highest-level form of fully specified mechanistic explanation. Here, a mechanistic explanation is one that provides mechanisms by which cognitive and behavioral functions are performed. Such an explanation is fully specified when it provides a full specification of the mechanism, in the sense of providing a recipe that could be copied to yield a system that performs the function in question. One explanation is higher-level than another when it abstracts away from details that the other invokes. So the claim is that computational explanation in my sense yields a sweet spot of being detailed enough that a fully specified mechanism is provided, while at the same time providing the minimal level of detail needed for such a mechanism.

Neurobiological explanation, for example, can presumably yield a fully specified mechanism. But it is a relatively low-level form of explanation, and computational expanation in my sense can provide such a mechanism at a higher level. Representational explanation, on the other hand, is even higher-level, but it does not provide a fully specified mechanism. One cannot go straight from representational explanation to building a mechanism; one has some hard working to do in figuring out the right mechanism. The same goes for function-theoretic explanation.

Rescorla suggests that his representational explanations are mechanistic. Here I would distinguish the Bayesian models of perception early in his article from the register machine programs late in his article. The former are representational models but not yet mechanistic models in that they need to be supplemented by an algorithmic specification of how the various Bayesian probabilities are computed. The latter give fully specified algorithmic details, so have more claim to

be called mechanistic. One might worry as before that the representational requirement makes it much harder to know when these programs are being implemented, and in a way goes beyond the level of pure mechanism. But in any case these models can be seen as augmented computational models in my sense: the algorithm in effect specifies a causal topology and we additionally require that various places in this topology are filled by representations. So I think the phenomena that Rescorla points to are consistent with the claims about mechanistic explanation above.

Rescorla also questions whether we need computational explanations in my sense—those in terms of causal topology—at all. He suggests that we have a clear need for neural and representational explanations but that the role of the intermediate level is not clear. He notes that we do not try to give explanations of digestion, for example, in terms of causal topology. I think that this last point ignores the crucial difference between cognition and digestion: the former is an organizational invariant (setting externalism aside for now) while the latter is not. Causal topology does not suffice for digestion, so no explanation of digestion wholly in terms of causal topology can be vadequate. But causal topology suffices for cognition, so we can expect an explanation of cognition in terms of causal topology to be adequate. Such an explanation has the potential to cut at the joints that matter where a mechanistic explanation of cognition is concerned. Compared to representational explanation, it gives an account at the level of mechanisms. Compared to neurobiological explanation, it abstracts away from noncrucial implementational detail and also specifies a structure that can be shared by many systems, neurobiological or not.

I also note that once we combine the framework of the target article with the framework of abstract state machines, as suggested earlier, we will be able to use it to capture various more abstract (and therefore more general) levels of explanation. Functional explanation can be captured within the ASM framework, for example. ASM explanations will not be fully mechanistic in the sense above, as they leave some details of causal structure unspecified, but they at least involve something like mechanism-ready explanation.

One could argue that computational explanations in my sense apply more generally than representational and function-theoretic explanations. The latter are useful when they are possible, but it is likely that various aspects of cognition are not especially amenable to these forms of explanation. By contrast, the considerations in the target paper suggest that any aspect of cognition is in principle amenable to computational explanation in my sense. These explanations are also more general than neurobiological explanations for obvious reasons. So computational explanation at least promises to be an especially general explanatory framework, one that can be applied in principle to any cognitive domain in any organism (setting worries about consciousness aside,

and allowing supplementation in the case of environmental phenomena). This might do at least a little to make sense of the claims about a "general framework" in the target article.

Still, I remain a pluralist about explanation. Computational explanation (as I construe it) is just one form of explanation in cognitive science among many, distinctive especially for its role in mechanistic explanation and its generality.

References

Brown, C. 2012. Combinatorial-state automata and models of computation. *Journal of Cognitive Science*.

Egan, F. 2012. Metaphysics and computational cognitive science: Let's not let the tail wag the dog. *Journal of Cognitive Science*.

Chalmers, D.J. 1994. On implementing a computation. *Minds and Machines* 4:391-402.

Chalmers, D.J. 1995. Does a rock implement every finite-state automaton? *Synthese* 108:310-33.

Harnad, S. 2012. The causal topography of cognition. *Journal of Cognitive Science*.

Klein, C. 2012. Two paradigms of implementation. *Journal of Cognitive Science*.

Kripke, S. 1981. *Wittgenstein on Rules and Private Language*. Harvard University Press.

Milkowski, M. 2012. What is implemented? *Journal of Cognitive Science*.

O'Brien, G. 2012. Defending the semantic conception of computation in cognitive science. *Journal of Cognitive Science*.

Piccinini, G. 2007. Computational modeling vs. computational explanation: Is everything a Turing machine, and does it matter to the philosophy of mind? *Australasian Journal of Philosophy* 85:93-115.

Putnam, H. 1987. *Representation and Reality*. MIT Press.

Rescorla, M. 2012. How to integrate representation into computational modeling, and why we should. *Journal of Cognitive Science*.

Ritchie, J.B. 2012. Chalmers on implementation and computational sufficiency. *Journal of Cognitive Science*.

Scheutz, M. 2012. What it is not to implement a computation? A critical analysis of Chalmers' notion of computation. *Journal of Cognitive Science*.

Shagrir, O. 2012. Can a brain possess two minds? *Journal of Cognitive Science*.

Sprevak, M. 2012. Three challenges to Chalmers on computational implementation. *Journal of Cognitive Science*.

Stabler, E. 1987. Kripke on functionalism and automata. *Synthese* 70:1-22.

Towl, B. 2012. Home, pause, or break: A critique of Chalmers on implementation. *Journal of Cognitive Science*.