Implementation and Indeterminacy

Curtis Brown

Department of Philosophy
Trinity University
One Trinity Place, San Antonio, TX 78212, USA

cbrown@trinity.edu

Abstract

David Chalmers has defended an account of what it is for a physical system to implement a computation. The account appeals to the idea of a "combinatorial-state automaton" or CSA. It is unclear whether Chalmers intends the CSA to be a computational model in the usual sense, or merely a convenient formalism into which instances of other models can be translated. I argue that the CSA is not a computational model in the usual sense because CSAs do not perspicuously represent algorithms, are too powerful both in that they can perform any computation in a single step and in that without so far unspecified restrictions they can "compute" the uncomputable, and are too loosely related to physical implementations.

Keywords: Combinatorial-state automaton, computational model, implementation, Turing machine.

1 Introduction

It is a commonly held view in the cognitive sciences that cognition is essentially computation. If this idea is to be explanatorily useful, however, there must be an objective account of when a physical process implements a particular computation. Philosophers such as Hilary Putnam and John R. Searle have questioned whether such an account is possible. Searle has raised two related objections (Searle 1992, Chapter 9): first, that physical facts do not suffice to determine what computation a process implements ("physics does not determine syntax"), and second, that computation is an observer-relative property of physical processes, not an intrinsic property. The second point requires the first, since if physics constrained the computations a system could be interpreted as performing so tightly that only a single interpretation was possible, there would not be multiple possible interpretations for observers to select from.

David Chalmers has responded to these criticisms by developing an account of implementation according to which it is an objective relation between a physical system and an abstract model (Chalmers 1994, 1996a, 1996b). The account relies on the notion of a "Combinatorial-State Automaton" or CSA. I will argue that the CSA cannot be regarded as a model of computation in the sense that Turing machines, for example, are. This does not show that Chalmers' account of implementation is inadequate, but it may make it less attractive than it would be if the CSA were a full-fledged model of computation.

2 Chalmers on Implementation

Chalmers (1996a) provides several proofs that plausible definitions of implementation for some computational models, in particular finite-state automata either with or without input and output, make implementations far too easy to come by, thus trivializing the notion of the implementation of a computation and partially vindicating the critiques of Putnam and Searle.

Where exactly does the problem lie? Chalmers suggests that the root of the problem is that finite-state automata are too simple and unstructured. Chalmers writes: "Even simple FSAs with inputs and outputs are not constrained enough to capture the kind of complex structure that computation and cognition involve. The trouble is that the internal states of these FSAs are *monadic*, lacking any internal structure, whereas the internal states of most computational and cognitive systems have all sorts of complex structure."

Chalmers then introduces a model to attempt to capture this internal structure, the model of the "combinatorial-state automaton," or CSA. The CSA can be described in exactly the same way as an FSA, except that each input state, internal state, and output state are described as vectors rather than structureless states; that is, each state is regarded as being composed of substates. A given internal state S will be viewed as a vector $[S^1, S^2, \ldots, S^n]$, and similarly for input and output states.[1] For a physical system to implement a CSA, it must have states with substates that map to substates of the CSA, and state-transition rules in the CSA must correspond to reliable causal dependencies in the physical system. More precisely, to again quote Chalmers,

> A physical system *P* implements a CSA *M* if there is a decomposition of internal states of *P* into components $[s^1, \ldots, s^n]$, and a mapping *f* from the substates s^j into corresponding substates S^j of *M*, along with similar decompositions and mappings for inputs and outputs, such that for every state-

[1] To capture the full power of a Turing machine, the internal states must be allowed to have infinitely many components, but Chalmers considers primarily the finite case.

transition rule $([I^1, \ldots, I^k],[S^1, \ldots, S^n]) \rightarrow ([S'^1, \ldots, S'^n],[O^1, \ldots, O^l])$ of M: if P is in internal state $[s^1, \ldots, s^n]$ and receiving input $[i^1, \ldots, i^k]$ which map to formal state and input $[S^1, \ldots, S^n]$ and $[I^1, \ldots, I^k]$ respectively, this reliably causes it to enter an internal state and produce an output that map to $[S'^1, \ldots, S'^n]$ and $[O^1, \ldots, O^l]$ respectively (Chalmers 1996b: 318, with one small typo corrected).

Chalmers offers this as a general account of implementation: any abstract computation can be redescribed in terms of CSA state-transitions, so that the above definition of implementation can be applied to any abstract computation whatsoever. And the new model avoids the triviality proofs for implementations of finite-state automata.

3 Against the CSA as a Computational Model

There are two ways one might interpret the CSA model. First, it could be intended to be a general model of computation, in the same way that Turing machines or register machines are models of computation. Second, it could be intended, not as a computational model in its own right, but merely as a convenient formalism for redescribing computations from a variety of specific models, in order to be able to state conditions on implementation in a way that will apply to all of them. I will argue that the CSA cannot play the former role, and that, although it may be able to serve the latter, more modest role, this may not be as advantageous as it first appears.

In many ways the former interpretation of the CSA, as a full-fledged computational model, is a very attractive one. There have been many proposals for making the abstract idea of a computation precise, including Turing machines, register machines, abacus machines, Post production systems, and more. All of these have turned out to be equivalent, in the sense that they can compute exactly the same functions. In another sense, though, they are not equivalent: although a Turing machine and a register machine can each compute, say, $f(x) = x!$, the procedures used to compute the function will be quite different in the two cases. Each specific model of computation suggests a fairly restrictive physical implementation; for example, a Turing machine is thought of as having a read/write head that travels back and forth on a tape that is divided into squares. The idea of a CSA could be seen as abstracting away from such details, offering a completely general account of computation that is not restricted to any particular kind of physical implementation. On this interpretation, the CSA would have two important characteristics: (a) it would respect the differences between different computational models: the CSA transcription of a TM that computes a given function will be different from the CSA transcription of a register machine that computes the same function. But, unlike the familiar models, (b) it would be general enough to encompass them all. A TM cannot in any natural way be represented as a register machine (although it could be simulated by one), but either can be represented as a CSA. So the CSA seems to provide an attractive way of expressing the core of a computation without commitment to any specific kind of physical implementation.

3.1 First Problem: Lack of Perspicuity

If we consider how to translate a TM description into a CSA description, however, we may start to wonder whether something has gone wrong. Consider the following very simple two-state Turing machine. If started on the leftmost of a block of one or more strokes, it will move to the right, add a stroke, and then move back to the left to halt on the first blank space before the strokes. We could think of it as computing the function $f(n) = n + 1$. (For simplicity I ignore the usual convention that the machine must end on the leftmost stroke of the resulting block.)

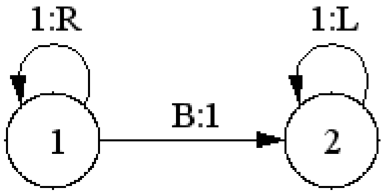

Figure 1: Simple Turing Machine

How should this Turing machine be described in the CSA formalism? A state of the CSA that represents this TM will be a vector with components for each square of the TM and a component for the internal state of the TM. To keep things simple, let us restrict our TM to a tape with only three squares. Each square will either be blank or contain a 1, and any combination of blanks and 1s will be a possible state of the tape. This gives us eight possible states so far. States must also have a component to represent the internal state of the TM; since our TM has two possible internal states, we now have 8 * 2 = 16 states. Finally, a CSA state needs to indicate the position of the TM's read/write head. The head must be on one and only one square of the tape, so we have a grand total of 16 * 3 = 48 distinct states the CSA can be in.[2]

In the general case, CSAs may have inputs and outputs as well as internal states. But this is not required to represent a Turing machine. There is no output aside from the final state of the tape. Chalmers suggests that the TM be regarded as having input only once, when it starts, but we can equally well regard it as having no input at all if we treat every state as a starting state, since the input is also simply a distribution of symbols on the tape.

Finally, in addition to state vectors (and input and output vectors if necessary), a CSA must have a state-transition

[2] The simplest way to represent the position of the head would be to add another component to the state vector and use it to indicate the number of the square on which the head is located. But this would not work if we allowed infinite vectors, which we need to fully represent a TM. Chalmers suggests letting the components for squares of the tape be ordered pairs of a symbol and a yes/no value indicating whether the head is on the square. If we do this we need to add a restriction specifying that only one square can have the value “yes.”

function. Since we do not need inputs and outputs for our TM representation, we can regard this function as simply a function from state vectors to state vectors. The most obvious way to represent such a function, and the way that is standard for other computational models such as Turing machines or FSAs, is simply an exhaustive list. A function is simply a set of ordered pairs, so we can list every such pair. Equivalently, we can regard each pair as a rule stating that the first member of the pair must be followed by the second member. Call a description of a CSA by means of such a complete list an exhaustive listing. In the present case we will have 48 such rules, one for each state of the CSA.

The first thing to notice about this listing is that it seems rather long as a way of characterizing a Turing machine that we could describe very briefly and simply! (The TM formalism itself requires only three state-transition rules.) And of course this is the description for a machine with a tape only three squares long; every additional square of tape will double the number of possible states, so that to represent a machine with a tape of, say, 1000 squares, we will need more than 2^{1000} states, or around 10^{300}, and a similar number of state-transition rules in an exhaustive listing.

Now, what is the significance of this example? Let us consider two cases, first the case of a finite CSA such as the example we have been considering, and second, an infinite CSA that represents the TM with an infinite tape. In the finite case it is tempting to say that the CSA does not represent a general algorithm at all in the way that the TM does, because information is actually lost in the redescription of a TM as a CSA. If you extend the tape of the TM, the very same TM description will now characterize a machine that computes the same function over a larger domain. But you cannot deduce from the state-transition function of a CSA how it should behave if we add more substates to represent additional squares of tape. We could say that the state-transition function of the CSA does not determine what mathematical function the CSA is computing. It is true that we could try to find the simplest description of the general principles the CSA is applying, and then use those general principles to project how the CSA should behave if extended to represent a larger tape. But the state-transition function itself does not determine this.

If we have an infinite CSA representing our TM with an infinite tape, then we will have all the information we need to determine what mathematical function is being computed. In this case, it may still be reasonable to say that the CSA does not represent an algorithm at all; certainly it does not represent one perspicuously. The information about the TM algorithm is present only in the same way that the laws of motion and gravitation would be present in a complete description of all the possible trajectories of objects in the universe. We have a complete listing of what the TM will do under every possible circumstance, but we have no easy or automatic way to determine the general principles that underlie these actions.

A closely related way to look at the matter is to notice that a state of the CSA that describes the Turing machine represents what Turing called a "complete configuration" of the machine, and what is now often called the state of a computation. The state transition rules relate complete computational states, and taken as a whole they specify every possible course the computation could take. So the state transition function in a sense gives us the results of applying an algorithm rather than the algorithm itself.

3.2 Second Problem: Excessive Power

Without severe unspecified restrictions, the CSA is simply too powerful to count as a computational model. There are at least two ways to see this point. First, recall that Turing machines and other computational models were originally introduced to try to provide a precise interpretation of the idea of an effective procedure or algorithm for computing a function. Turing machines (and other models) have the following property: if we can find a Turing machine that computes a given function, then we have found an effective procedure for computing the function, and the TM description is a description of this procedure. But this is simply not true for CSAs in general. There will always be a CSA which finds the value of a function for any argument in some finite range in a single step. For instance, in the case of the function $f(x) = x + 1$, which our simple Turing machine computes, we could dispense with the component that lists the position of the TM head, keeping the n components that represent squares of the tape and the component for the TM state. For every CSA state in which the TM state component is 1 and m consecutive tape components contain ones while the rest contain blanks, we will simply have a state-transition rule stating that the subsequent state of the CSA has $m + 1$ ones on an otherwise blank tape and the TM-state component is 2. Thus all the work is done in state 1; state 2 is simply a halting state. The resulting CSA is no longer a Turing machine, since we have dispensed with the head and can change more than one square of the tape at a time. But it still satisfies the definition of a CSA, even though what it is doing hardly seems to count as computation at all. (It amounts to looking up the answer in a lookup table, except that the lookup table is stored in the state-transition rules rather than in some sort of memory.)

Second, consider the case of a CSA whose states have an infinite number of components. Chalmers explicitly allows this, as indeed he must if it is to be possible to have a CSA transcription of a TM with an infinite tape. But now the state-transition function will need to be able to take infinitely many arguments (so that an exhaustive listing would have infinitely many state-transition rules). But once we allow the state-transition function to have infinitely many arguments, it is hard to see how to prevent CSAs from being able to "compute" functions that are in fact not computable! And clearly a model that permits "computation" of uncomputable functions is not a good candidate for a model of computation. (A closely related observation is that without further restrictions, a diagonalization argument will show that the CSA has nondenumerably many possible states.)

I hesitate to place too much weight on this point, since Chalmers only briefly mentions infinite CSAs, and he does state that “restrictions have to be placed on the vectors and dependency rules, so that these do not encode an infinite amount of information” (Chalmers n.d.: section 2.1). Chalmers does not state what these restrictions might be, though he says that specifying them “is not too difficult.” Clearly one way to specify such restrictions would be to require that the CSA conform to the limitations of a Turing machine: the only square that can change is the one the head is on, and the position of the head can only change by one square at a time. But we certainly do not want to impose the constraints specific to Turing machines on the general notion of a CSA, since this would deprive it of its ability to transcribe other computational models as well.

In some ways the most natural way to limit the class of CSAs to those that compute functions that are “computable” in the usual sense might be to require that there be a way to give a finite specification of the state-transition function. More precisely, it would be natural to require that the state-transition function be effectively computable -- for instance, by requiring that it be definable from very basic functions by composition, primitive recursion, and minimization. But this solution would seem to rob the CSA formalism itself of its interest as a computational model, since the work of guaranteeing that what the CSA is doing is computable would in fact be done by an independent conception of computability.

The problem of excessive power can be put in another way. Traditional computational models begin with a highly restricted set of abilities, and then show that more and more complex tasks can be performed by combinations of these basic abilities. It is precisely that fact that complex tasks can be accomplished by complex applications of simple abilities that shows that the tasks are computable. However, the CSA model in a sense moves in the exact opposite direction. It begins with the ability to move from absolutely any state to absolutely any other state, so that to guarantee that only computable functions can be captured, we have to impose restrictions.

3.3 Third Problem: Aloofness from Implementational Details

A third problem with viewing the CSA as a model of computation is that it is too aloof from implementational details. This may seem odd, since it is precisely its level of abstraction that is intended to be its chief advantage. But precisely this level of abstraction removes one of the chief attractions of models like the TM, namely that they show us how a computation could be accomplished by a system we have a good idea how to implement. For any set of TM instructions, it is easy to imagine an actual physical implementation of a system that follows these instructions. On the other hand, the CSA would seem to provide us with the possibility of describing computations or pseudocomputations that there might be no natural way of implementing, or perhaps even no possible way of implementing at all. For instance, Fredkin has suggested that the fact that a TM can change only the square of the tape that is under the head reflects the physical principle that there is no action at a distance (Fredkin and Toffoli, 1982). But nothing like this constraint is built into the CSA model. For instance, we could write CSA rules that would correspond to an extended TM that could change squares very distant from the one the head is on, or for that matter could change arbitrarily large numbers of squares at once. The computation this CSA represents may well be implementable in some way, but it would not be straightforwardly implementable as a TM whose read/write head can act at a distance! So unlike other abstract computational devices, the fact that a given CSA exists gives us no guidance about how it might be implemented.

4 The CSA as a Transcription Device

I have argued that the CSA does not constitute a computational model in its own right, at least as presently described. (It is possible that a revised version with restrictions imposed on the allowable states and transitions might be.) It is entirely possible, however, that it was not Chalmers’ intention to provide such an account. It may be that he intends the second interpretation mentioned above, construing the CSA merely as a convenient formalism into which more specific abstract machines can be translated.

If this were the case, then, since each TM state-transition rule (for instance) corresponds to a large number of CSA state-transitions (in fact an infinite number if we are representing a TM with an infinite tape), we could abandon the exhaustive listing as a way of characterizing a CSA, and translate each TM rule by a universal quantification over CSA states. (Some sentences in Chalmers 1996a: section 6 may be read as suggesting something like this.)

For the example we have been considering, we could say that the function that maps a state S onto its successor S' is the unique function such that:

1. $(\forall i\text{: } 1 \leq i \leq n)((S^i = \text{‘}B\text{’} \wedge S^{n+1} = i \wedge S^{n+2} = 1) \rightarrow$ $(\mathrm{S}'^{i} = \text{‘1’} \wedge S'^{n+1} = i \wedge S'^{n+2} = 2))$
2. $(\forall i\text{: } 1 \leq i < n)\ ((S^i = \text{‘1’} \wedge S^{n+1} = i \wedge S^{n+2} = 1) \rightarrow$ $(S'^{n+1} = i + 1 \wedge S'^{n+2} = 1))$
3. $(\forall i\text{: } 1 < i \leq n)\ ((S^i = \text{‘1’} \wedge S^{n+1} = i \wedge S'^{n+2} = 2) \rightarrow$ $(S^{n+1} = i\text{-}1 \wedge S'^{n+2} = 2))$
4. In all other respects, S' is identical to S.[3]

If we took this approach, then constructing informative descriptions of rules underlying the state transition function would be straightforward, since they would simply transcribe the general rules guiding the more specific abstract machine. We could be sure that we were considering only CSAs representing computable

[3] n is the number of squares on the TM tape -- three, in the case we have been considering. This description treats the first n components of the state vector as representing the states of the tape's n squares, the next component as representing the index of the square the head is currently over, and the final component as representing the current internal state of the TM.

functions, since they would all be transcribed from other models which guarantee computability. But we would entirely lose the attractive idea of the CSA as a model that represents the concept of computability in a completely general way.

Moreover, once we lose the idea that a physical system implements a computation if and only if there is a CSA that it implements, it becomes less clear what the advantage of using CSAs to define implementation is. For on this more limited understanding of the significance of the CSA, we will need to decide how to translate each more specific computational model into a CSA, a task which may prove to be just as difficult as defining implementation directly for each specific model. And we will not have a completely general account of computation unless and until we have discovered every possible computational model in the full-fledged sense, and provided a way to translate each of them into CSAs. This is not only a challenging project, it is poorly enough defined that it is not clear what would count as success! So abandoning the idea of the CSA as a general, all-purpose model of computation removes some of the attractiveness of regarding it even as merely a convenient formalism.

5 Conclusion

It would be delightful to have a general account of computation, an account fine-grained enough to distinguish between different ways of computing the same function, and general enough that it can describe any abstract computation. It is possible that when Chalmers provides details that were only hinted at in his earlier papers, in particular about the restrictions that need to be placed on allowable CSA states, the CSA will in fact turn out to be such a model. But it cannot yet be regarded in that light.

6 References

Chalmers, D. (1994): On Implementing a Computation. *Minds and Machines* **4**: 391-402.

Chalmers, D. (1996a): Does a Rock Implement Every Finite-State Automaton? *Synthese* **108**: 309-333.

Chalmers, D. (1996b): *The Conscious Mind*. Oxford, Oxford University Press.

Chalmers, D. (n.d.): A Computational Foundation for the Study of Cognition. http://www.u.arizona.edu/~chalmers/papers/computation.html. Accessed 20 Feb 2004.

E. Fredkin and T. Toffoli (1982): Conservative Logic. *International Journal of Theoretical Physics* **21**: 219-253.

Putnam, H. (1998): *Representation and Reality*. Cambridge, MIT Press.

Searle, J. (1992): *The Rediscovery of the Mind*. Cambridge, MIT Press.