Smith, Brian Cantwell (1987), "The Correspondence Continuum," *Report CSLI-87-71* (Stanford, CA: Center for the Study of Language and Information).
_____ (1991), "The Owl and the Electric Encyclopedia," *Artificial Intelligence* 47: 251–88.
_____ (1997), "One Hundred Billion Lines of C++," *CogSci News* (Bethlehem, PA: Lehigh University Cognitive Science Program), vol. 10, no. 1 (Spring): 2–6
Soare, Robert I. (1996), "Computability and Recursion," *Bulletin of Symbolic Logic* 2: 284–321.
Wartofsky, Marx W. (1979), *Models: Representation and the Scientific Understanding* (Dordrecht, Holland: D. Reidel, 1979).

WHAT IS COMPUTER SCIENCE ABOUT?

1. Introduction

What is computer-science (CS) about? CS is obviously the science of computers. But what exactly are computers? We know that there are physical computers, and, perhaps, also abstract computers.[1] Let us limit the discussion here to physical entities and ask: What are physical computers? What does it mean for a physical entity to be a computer? The answer, it seems, is that physical computers are physical dynamical systems that implement formal entities such as Turing-machines. I do not think that this answer is false. But it invites another, and troubling, question: What distinguishes computers from other physical dynamical systems? The difficulty is that, on the one hand, every physical system implements abstract formal entities such as sets of differential equations, while on the other hand we certainly do not want to count every dynamical system as a computer. After all, if CS is somehow distinctive, then there must be a difference between computers and other systems such as solar systems, stomachs, and carburetors. But what is the difference?

There are two familiar answers to the latter question. One is that computers are systems that execute algorithms, meaning that they implement formal specifications of algorithms.[2] The other is that computers are systems whose processes are sensitive to the syntactic or formal properties of the representations over which they are defined, meaning that the semantic properties have no role in the computational identity of the system.[3] My thesis, however, is that both answers are at fault. In Sections 2 and 3 I argue that being algorithmic is not necessary for being computational. In Section 4, I argue that content does affect the computational identity of a system. In Section 5, I outline a semantic conception of computation. In Section 6, I get back to answer, more informatively, what CS is about.

2: Nonrecursive computation

Let $f_1, f_2, \ldots$ be a list of all the total recursive functions whose domain is the natural numbers and whose range is $\{0, 1\}$.[4] Let T_i be the code of a Turing-machine that computes f_i.[5] Imagine now a universal machine S that possesses an infinite list $T_1, T_2, \ldots$ Given a natural number n, the machine "goes to" the n-place in the list, extracts its content, and simulates the operation of T_n on the argument n. The final operation of S is to switch the end result of $T_n(n)$. If $T_n(n) = 1$, S will produce 0 as its output, and if $T_n(n) = 0$, S will produce 1 as its output. In short, we have no trouble seeing S as computing the anti-diagonal function:

$$h(n) = \begin{cases} 1 \text{ if } f_n(n) = 0 \\ 0 \text{ if } f_n(n) = 1 \end{cases}$$

But h is not a recursive function. If it were, it should have been listed as f_k, for some k. But this cannot be. For: $h(k) = f_k(k) = 1$ if $f_k(k) = 0$, and $h(k) = f_k(k) = 0$ if $f_k(k) = 1$.

The fact that h is nonrecursive may seem surprising since the operations $T_n(n)$, + 1, and -1 are recursive operations. The nonrecursivity, then, results from the fact that $f_1, f_2, \ldots$ are not effectively enumerable, i.e., there is no Turing-machine that, for any n, can compute the code T_n. But unlike any Turing-machine, S need not compute the codes. Whereas a Turing-machine can access, at each moment, only finite lists, S has the access to an infinite list, fixed in advance by an oracle or the like. With the aid of this super-source, S computes h.

The claim that S *computes* h does not falsify the Church-Turing thesis. The Church-Turing thesis asserts that a function is effectively computable if and only if it is partially recursive (Turing-computable), whereas "effectively computable" refers to the class of functions that can be computed *via* an algorithm, i.e., a finite list of instructions that access finite information. This means that the defined values of these functions can be reached in a finite number of steps by following the algorithm. But these constraints need not be put on any computing machine. There is no reason why we cannot have computing machines, such as S, with privileges that no "effective computation" or Turing-machine has, e.g., an

access to an infinite given list. Thus, the claim that S computes h is perfectly consistent with the Church-Turing thesis (In fact, the claim that S computes a function that is not "effectively computable" assumes the Church-Turing thesis—for it was really shown that h is not recursive). The Church-Turing thesis is the claim that certain classes of computing machines, e.g., Turing-machines, compute effectively computable functions. It is not the claim that all computing machines compute solely effectively computable functions.[6]

The claim that S computes h also does not falsify the "Physical Church-Turing thesis" (that all mathematical equations describing physical systems are recursive).[7] S is not an actual physical system, nor is it implemented by a physical system. My claim is conceptual: we have no difficulty in imagining a system that computes functions that cannot be computed just by following an algorithm. This demonstrates that algorithmic computations do not occupy the whole space of computations. It shows that being algorithmic is not necessary for being computation.

One may suggest that S follows an algorithm after all. Though S includes an infinite list, the *rule* it follows *is* finite. The suggestion, in other words, is to stretch the notion of algorithm with the notion of computation. We apply the notion of algorithm to processes that operate according to finite rules, no matter whether their memory is finite or infinite. On this suggestion, then, I have not shown that we could have nonalgorithmic computation, but that there are algorithmic computations of nonrecursive functions. I think that this is reasonable. Still, it reaffirms my claim that the current notion of algorithmic computation has been too restrictive. Put differently, this suggestion confirms my claim that the notion of algorithmic computation, as it is customarily understood in CS, does not exhaust all kinds of computations. The difference is just that this suggestion keeps algorithms and computations together. I now turn to show that algorithms and computations should be kept apart if we want to retain the idea that some physical dynamics are not computations.

3. Analog computation

Analog computers have been around for many decades. They are nowadays more noticeable because of their close connection to the rapidly developing field known as "neural computation."[8] Today, we can build, in principle, an analog-VLSI computer consisting of $n \times n$ units that implements an abstract attractor neural net for the n-queens problem.[9] And, yet, it seems that the existence of analog computers completely undercuts the role of algorithms in drawing the line between computers and other physical dynamics. If analog processes are not algorithmic, then, it seems, we have instances of computations (e.g., "neural computations") that are not algorithmic. And if analog processes are algorithmic, it is hard to see what physical dynamics are not algorithmic. After all, attractor neural nets and the stochastic magnetic (spin-glasses) systems are, from a formal point of view, close cousins.[10] So if the attractor nets compute, it is hard to explain, in algorithmic terms, why dynamical systems such as magnetic systems do not compute. In short, anyone who assumes that being algorithmic is essential for being computational must choose between the exclusion of analog computations, including "neural computations," from the computational domain, and the inclusion of practically every physical system in the computational domain.

Let me explicate the nature and scope of this dilemma with the aid of Pitowsky's (1990) simple but forceful example. Consider the following machine for averaging three numbers (Figure 1). We take an insulated container divided into three equal parts by insulated removable barriers. We put a thermometer in each section, and we set the temperature in each section to a degree equal to the corresponding three numbers, k, m, n. We now remove the barriers simultaneously and wait until the temperatures equalize. This process is a thermodynamic one and is usually described by a set of differential equations. Still, its output is (k + m + n)/3.

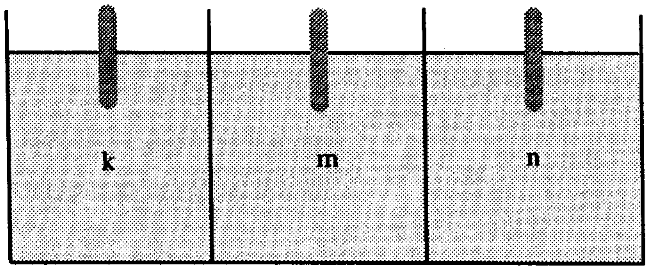

Figure 1. A Thermal machine for averaging numbers (from Pitowsky, 1990).

Consider now another scenario. We repeat the process with k, m, n. But now we (or some other mechanism) remove only one barrier, and wait for the temperature to equalize ((k + m)/2). Only then do we remove the second barrier. The output is evidently the same as in the first process. This time, however, the device executes a two-step algorithm. The device averages k, m, n, by going through two *different* basic steps:

Basic step 1: average k and m:
$[(k, m) \rightarrow (k + m)/2]$;
Basic step 2: operate on the output of step 1 and n:
$[(\text{output}_1\ (k, m), n) \rightarrow 2/3 \cdot \text{output}_1\ (k, m) + n/3]$.

In short, in the first scenario the device averages k, m, and n *via* a single-step process, whereas in the second scenario it averages k, m, and n *via* a two-step algorithmic process.

But is the process, under the first scenario, algorithmic? let us assume that the answer is NO—with the result that the averaging, "analog," process is no computation. Cummins, for example, defends this view.[11] He defines an algorithmic process as a step-satisfaction process; that is, a system satisfies but does not compute its basic steps. Thus, in the two-step averaging process, each step, as an "analog" process, is satisfaction, but not computation. Only the two-step process as a whole is computation. But this view has the absurd consequence that computing depends on moving the barriers of the thermal machine twice instead of once. Indeed, it is very odd to describe the one-step averaging process as no computation, and the two-step process as computation. It is much more reasonable to think that if the two-step process is computation, so is the one-step process.[12] But, then, if we also hold the view that computational processes must be algorithmic, we must admit that the averaging process, under the first scenario, is algorithmic.

Let us, then, consider the YES answer—that the averaging, one-step process, is algorithmic. The trouble here is that if the thermodynamical process is algorithmic, then we surely must accept other thermal processes, such as tornado storms and boiling water, which also implement the thermodynamic equations, as algorithmic. Indeed, under these conditions, it is hard to see what physical dynamics are not algorithmic. But if all these processes are algorithmic, then the notion of algorithm plays no effective role in distinguishing computers from other physical dynamics.

So let me recapitulate: I have attempted to challenge the view that the notion of algorithm lies at the heart of the notion of computation. I have first shown, in Section 2, that we can easily accept computations that cannot be achieved, given the Church-Turing thesis, in algorithmic ways—at least if we associate algorithms, as we customarily do, with access to finite resources of information. I have then proceeded to show, in Section 3, that there are instances of processes, e.g., analog computations, that are even more challenging to the algorithmic view. If these analog processes are algorithmic, then the notion of algorithm loses its role as a boundary between computation and non-computation. And if these analog processes are not algorithmic, then we must conclude, again, that being algorithmic is not necessary for being computation.

4. The semantic character of computational taxonomies

Let us now turn to refute the other myth: that computers are "non-semantic" systems, in the sense that their identity solely depends on syntactic or formal properties. Before I turn to challenge this myth, it will be useful to review, very briefly, the essentials of the standard view of computational individuation.

Consider, then, a physical system **P** that implements the (abstract) automaton S (Figure 2). This *physical* system **P** is an electronic device, consisting of a pair of gates that receive and emit currents that range from 0 to 10 volts. One gate in **P** emits 5–10 volts if it receives a voltage larger than 5 from both input channels, and 0–5 volts otherwise. The other gate emits 5–10 volts if it receives a voltage larger than 5 from exactly one input channel, and 0–5 volts otherwise. When assigned '0' to emitting 0–5 volts and '1' to emitting 5–10 volts the first gate is seen as an *and-gate* and the second as a *xor-gate*. Under this assignment, in other words, **P** is seen as executing an algorithm for the syntactic function *f*:

'0', '0' → '0', '0'
'0', '1' → '0', '1'
'1', '0' → '0', '1'
'1', '1' → '1', '0'.

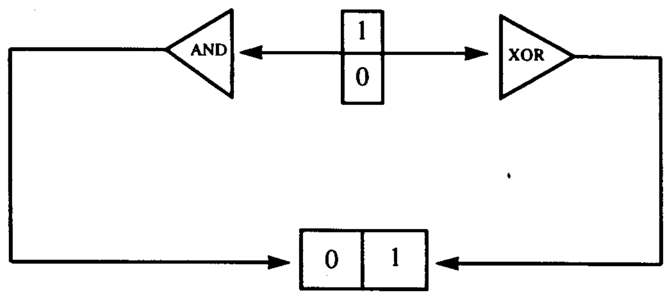

Figure 2: The abstract automaton S (from Block, 1990).

Another way to describe what the system does is semantic. It is routine to interpret the '0' and '1' as representing numbers. For example, under a standard binary interpretation **P** computes addition (for the domain {0, 1}). That is, the input-output syntactic relations can be interpreted as representing the following relations between numbers:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 2$$

We thus have at least three different ways to describe what **P** does: The physical level(s) that describes **P** in terms of voltages and the like, the syntactic level that describes **P** as implementing S, and the semantic level that describes **P** in terms of an interpretation of the symbols. But which of these descriptions corresponds to the *computational* description of **P**? According to the standard view, it is the syntactic description.[13] The computational identity of **P** derives from **P**'s implementing S. By implementing S, the states of **P** fall into computational types by virtue of their mapping relations to S. And, in particular, the computational identity of **P**'s input-output behavior is precisely the syntactic function *f*. The semantic interpretation does not correspond to the computational description of **P** because, from a computational point of view, it does not matter if we choose to interpret '1' and '0' as numbers or as colored hats. The *computational* identity of **P** would have been the same had we interpreted the '0' and '1' as representing colored hats. And the physical description does not correspond to the computational description because **P** is computationally equivalent to other physical systems whose physical descriptions are very different from **P**. These different physical systems are computationally alike because they all share the same syntactic characterization **S**.

Now, I accept the central elements of this picture of individuation. I agree that the computational individuation of **P** is given by **S**, and I agree that *f* is a computational description of what **P** does. Nonetheless, I think that semantic properties plays a role in computational individuation. My argument is roughly this: **S** may not be the only syntactic structure **P** implements. Yet, the computational identity of the system, in a given context, is not defined by all the implemented syntactic structures. Thus content determines which structure defines the computational identity of the system in a given context. Here is the argument in details:

Step 1: A physical system may implement more than one syntactic structure.

Suppose that it turns out that gates of **P** are actually tri-stable. Imagine, for example, that the implemented *and-gate* in **P** emits 5–10 volts if it receives voltages larger than 5 from both input channels; 0–2.5 volts if it receives voltages less than 2.5 from both input channels; and 2.5–5 volts otherwise. Similarly, let us assume that the implemented *xor-gate* emits 5–10 volts if it receives more than 5 volts in one input channel and less than 5 in the other; 0–2.5 volts if it receives less than 2.5 volts from both input channels; and 2.5–5 volts otherwise. We could now assign the symbol '0' to emitting less than 2.5 volts and '1' to emitting 2.5–10 volts. Under this assignment, both the "and-gate" and the "xor-gate" are seen as *or-gates*! Consequently, **P** is seen as implementing an abstract machine **S′** (Figure 3) whose input-output behavior is characterized by the syntactic function *f′*:

'0', '0' → '0', '0'
'0', '1' → '1', '1'
'1', '0' → '1', '1'
'1', '1' → '1', '1'

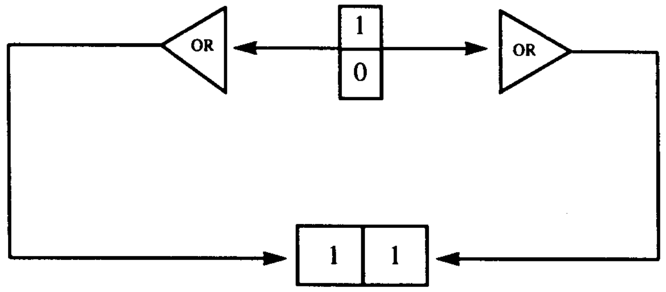

Figure 3: The abstract automaton **S′**

It follows, then, that the very same physical system **P** implements not only the abstract system **S**, but also an abstract syntactic system **S′**. Accordingly, by implementing **S′**, **P**'s behavior can be described, from a syntactic viewpoint, not only by the syntactic function *f*, but also by the syntactic function *f′*. The structures **S** and **S′** and the functions *f* and *f′*, in other words, are two different syntactic descriptions of what **P** does.[14]

Note that my claim is more modest than the "universal realizability" claims advanced by Putnam (1988) and by Searle (1992). Putnam proves that "every ordinary open system is a realization of every abstract finite automaton" (p. 121), and Searle argues that "for any program and for any sufficiently complex object, there is some description of the object under which it is implementing my program. Thus for example the wall behind my back is right now implementing the Wordstar program, because there is some pattern of molecule movements that is isomorphic with the formal structure of Wordstar" (pp. 208–09). Unlike Putnam and Searle, however, I do not claim that every physical system implements any abstract automaton. My claim, rather, is only that some physical systems implement, at the very same physical states, different abstract automata.[15]

One may still object that **P** really implements one syntactic structure. Since **P**'s gates are tri-stable, the objector objects, **P**'s "deep" syntactic conditions are captured by the syntactic function *f″*, from which we can derive both *f* and *f′*. The function *f″* can be described as:

'0', '0' → '0', '0'; '0', '1' → '1', '1'; '1', '0' → '1', '1'
'1', '1' → '1', '1'; '0', '2' → '1', '2'; '2', '0' → '1', '2'
'2', '1' → '1', '2'; '1', '2' → '1', '2'; '2', '2' → '2', '1'.

The function f is then derived from f'' by the transformation:

{'0', '1'} → '0'; '2' → '1',

whereas f' is derived from f'' by the transformation:

'0' → '0'; {'1', '2'} → '1'.

Whether the implementations of **S** and **S′** are "deep" or "shallow" does not really affect the rest of my argument. But we could also modify the example. Suppose that **P**'s gates are bi-stable (so **P** "really" implements **S** and computes f). Suppose, however, that it turns out that we can also implement **S′** in some other physical property of the very same (spatio-temporal) events of **P**. If, for example, the inputs and outputs of the gates are positive and negative charges, the temperature of the gate may depend on the absolute values of the currents. Thus we can implement one syntactic structure in the currents '+' and '-' and another in the temperatures.[16] It is now obvious that **P** implements both **S** and **S′**, though there may be no "deeper" syntactic structure **S″** and syntactic function f'' implemented. It is thus clear that **S** and **S′** are "deep" syntactic descriptions of what **P** does.

The underdetermination is most apparent in the context of cognitive systems. Imagine that a physical system **N** is a component of the visual system which computes a representation of depth of the visual scene from information about binocular disparity. But, as it turns out, a system whose microphysical structure is the same as that of **N** is a component of the auditory system of some remote creatures. Had we removed **N** from its visual environment and plugged it into the auditory slot in the remote creatures, it would have computed some sonar representations. As it also turns out, however, **N** implements **S** while performing the visual task, but implements **S′** while performing the auditory task. This could be the case if, for example, a certain neuron in **N** exercises one visual task when it fires 0–5 millivolts and another visual task when it fires 5–10 millivolts, but the same neuron, in an auditory environment, exercises one auditory task when it fires 0–2.5 millivolts and another auditory task when it fires 2.5–10 millivolts. In this case, we surely would say that **N**'s input-output behavior is described by f when exercising the visual task, but by f' when exercising the auditory task. In this case, we would surely affirm that **N** is characterized by different syntactic descriptions in the different environments.[17]

STEP 2: THE COMPUTATIONAL IDENTITY OF A SYSTEM MAY VARY ACROSS CONTEXTS.

The conclusion of Step 1 was that the same physical system may implement more than one syntactic structure. But we also take it that the computational level of description goes hand-in-hand with the syntactic structure the system implements, meaning that two systems that implement different syntactic structures fall under different computational types. The question, then, is what computational taxonomies count as "different syntactic implementations": Whether a computational taxonomy of a system counts all the syntactic structures the system implements, or just the syntactic structures the system implements while performing a given task. I think it is clear that the second option is the correct one.

Suppose it turns out that one gate in my desktop machine, but none in yours, is tri-stable (the machines are otherwise identical). We will definitely agree that our machines may be computationally different in other contexts. But we would surely not deny that the machines are currently computationally equivalent. Similarly, when we use **P** to do f by implementing **S**, we would count **P** as computationally equivalent to any other system that is used to do f by implementing **S**, no matter whether or not the other system could also implement **S′**. Or take again the cognitive system **N**. Assume that **N** implements **S** when performing the visual task, but implements **S′** when performing the auditory task. I think it is clear that the computational identity of **N** is given by **S** in the visual context, but by **S′** in the auditory context. For it is clear that computational theories of vision care only about the syntactic structure the system implements while doing vision, and are oblivious to other structures the system may or may not implement.

It was shown, then, that the same physical system **P** can be seen as implementing different abstract automata **S** and **S′**, whereas, on each implementation, states of **P** fall under different *computational* types. In other words, identifying **P** as computing f is not always invariant across contexts.

The computational identity of a system is context-dependent. It depends on the syntactic structure the system implements while performing a given task.

Note that to claim that computational identity is task-relative is not to say that the task is relative to the viewpoint of a foreign observer. Foreign observers who attempt to describe what the system does, are free to describe the system by any of the syntactic structures the system implements. But their choice does not define the *computational* identity of the system. If the observers wish to provide a *computational* description of my desktop machine, they *must* choose the syntactic structure the system implements while it performs its usual tasks. And if the observers wish to provide a computational description of the visual system, they must choose the syntactic structure the system implements while it does vision. The freedom to choose other syntactic structures only indicates that the system could have other computational identities. It indicates that the system would have new computational identity, had the task of the system changed (say, from visual to auditory), and had the chosen structure been the one the system implemented while performing the new task.

STEP 3: THE CONTENTS THE SYSTEM CARRIES AFFECT ITS COMPUTATIONAL IDENTITY.

If the same physical process can be seen as implementing different syntactic structures—if it can be seen as executing different algorithms—what *does* determine the computational identity of a system? What are the relations between the computational identity of the system and the context in which it is embedded? If the same physical system **P**, on *all its internal* physical properties, can fall under different computational types, then, it seems the computational identity of this system must be determined, at least partly, by features external to this system. Likewise if the states of **N** fall under different computational types in the visual and auditory environment, the computational identity of **N** is also dependent on features external to **N**. But what could these features be? Well, it seems that the only factor left that could make the computational difference must be the content assigned to the states of the system. The reason that states of **P** fall under one computational type, say **S**, and not under another type, say **S′**, must have something to do with the fact that we use the system as an adder, and that we take its states to represent numbers and not hats. Likewise, the reason the same neural states of **N** fall under one set of computational types in the visual environment, and under a different set of computational types in the auditory environment, must have something to do with the fact that in one case **N**'s states carry visual content, and in the other case **N**'s states carry auditory content. So content affects, after all, computational individuation.

The argument is now complete: The computational identity of the system is determined by the syntactic structure the system implements while performing a given task, but the task of the system in any given context is defined, at least partly, in semantic terms; i.e., in terms of the contents the system carries. Thus, the computational identity of **P** is given by **S** because **P** is used to compute *addition*, and it implements **S** in performing this task, and the computational identity of **N** is given by **S** because **N** is used to compute *depth of the visual scene*, and it implements **S** in performing this task.[18]

It has been hard to appreciate the role of content in computational individuation mainly for two reasons. The first is that we think we can extract, in principle, the pertinent syntactic structure from the physical system. We think the syntactic structure is an abstraction from the physical properties of the system, and that this extraction has nothing to do with semantic properties. I do not challenge this thought. My argument, rather, is that since we may also be able to extract *other* syntactic structures, we must have an external constraint to determine which of these structures define the *computational* identity of the system, and that this constraint must have something to do with the contents the states of the system carry. I do not claim, then, that we cannot extract the syntax from the physics of the system. My claim is that the extraction is not sufficient to determine computational identity.

Another reason the role of content has been overlooked is that we inferred from the cases where difference in content does not entail a computational difference that content plays no role in computational taxonomies. Indeed, there are cases in which **P** falls under the same computational types, although we interpret its states as representing, at one time, clothes, at another time, people, and at some other time, numbers. Likewise, states of **N** may also fall under the same computational types in

[illegible] all [illegible] –
[illegible] genuine [illegible]
[illegible] content is irrelevant
to [illegible]

different environments. N *will have* the same computational identity if it implements the *same* syntactic structure while performing the visual and the auditory tasks. These cases, however, do not show that content plays no role in computational taxonomies. They imply, rather, that we need a condition for computational identity that is sensitive to some differences in content but not to others. But what could such a condition be?

One possible suggestion is this: *Two computing systems are computationally equivalent only if their physical states "pick" the same formal features in the embedding environments.* "Formal features" mean set-theoretic or other high-order mathematical relations (among *represented* objects). The criterion thus suggests that computational taxonomies do not take into account specific (non-formal) features of content, such as the identity of individuals, but only formal features such as relations among sets of individuals. Following Gila Sher (1996), I will refer to the latter features of content as "formal contents."

A rigorous formulation of this criterion has yet to be offered, but an example will suffice for our purposes. Consider again our system **P**. If **P** implements S when its states carry information about clothes, and implements S′ when its states carry information about people, then, according to the criterion, their electronic states pick different formal structures. This might be the case if, for example, a detector of **P** indicates, in the context of clothes, that the distal object is a hat, whenever it emits 0–5 volts, and a shirt whenever its activation value is 5–10 volts. Whereas this very same detector indicates, in the context of people, that the distal object is a man whenever its activation value is 0–2.5 volts and a woman whenever its activation value is 2.5–10 volts. Importantly, what here makes the computational difference is not the difference in specific contents (i.e.., clothes *vs.* people), but the difference in formal (i.e., set-theoretic) structure: that the detector emitting 0–5 volts is correlated with one class of tokens in the context of clothes but with two distinct classes of tokens in the context of people.[19] Had the pertinent *formal* relations in the contexts been the same, the computational individuation of states of **P** would have also been the same. In other words, **P** could implement **S** in different contexts, even though its states' specific contents (e.g., clothes *vs.* people) are different. This, indeed, would be the case, if, for example, whenever an electronic property (e.g., our detector emitting 0–5 volts) is correlated with one feature in the context of clothes (e.g., being a hat),

then this electronic property is correlated with a single feature in the context of people (e.g., being a man), and *vice versa*.

5. What is computation?

Let us return to the original question: What is a physical computer? What does it mean for a physical system to be a computer? The discussion so far suggests a surprising answer: computations are mechanisms that are type-individuated by certain contents—i.e., formal contents—of the representations over which these mechanisms are defined. Desktop machines, but not stomachs, are computers because certain features of content (formal contents) enter into the type-individuation of desktop machines (as desktop machines) but not into the type-individuation of stomachs (as stomachs). It is true that the mechanisms of desktop machines and the mechanisms of stomachs can both be described in formal terms. In these cases, both descriptions pick certain formal (e.g., set-theoretic) relations of these inner mechanisms. But the big difference is that in the case of desktop machines, but not in the case of stomachs, the described systems are, *by themselves*, formal systems. Desktop machines are formal systems because we individuate their states, *as computational states*, by the formal (e.g., set-theoretic) properties/relations of the features that these states represent.

In the case of stomachs, the observers first type-individuate the stomach's states by their biochemical properties. After that, if the observers wish, they provide more general, formal, description of the set-theoretic relations among these types, which results in some mathematical or formal equation. In the case of desktop computers, however, the observers first type-individuate the states of the system by formal content. If the distal domain is a world of clothes, then the relevant features that determine the type-individuation are set-theoretic properties of clothes (or better: those set-theoretic properties detected by the desktop machines). If the distal domain is the visual field then the relevant individuative relations are formal relations in the visual field. And if the distal domain is the abstract, then the relevant individuative relations are the formal relations among abstract objects. After that, if the observers wish, they can provide a formal description of the (set-theoretic) relations among the individuated types. This formal description is sometimes called the "computational theory" of the system, sometimes conceived as an abstract automaton implemented by the desktop machine, and sometimes viewed as the syntactic structure of the desktop machine. I do not object to any of these viewpoints, as long as we remember that syntactic considerations are driven by semantic considerations.

On my account, then, the difference between computers and non-computers is grounded neither in the physics of the systems, nor in the formal structure they implement. What really makes the difference is the way the states of the system are classified to types. Desktop machines and cognitive systems are computers precisely because their states are type-individuated by the contents they carry. Stomachs could also become computers had their states carried contents, and had their states been type-individuated by these contents. But the identity conditions of stomachs, as stomachs, have nothing to do with content. So stomachs are not computers.

6. *What computer science is about*

I have challenged two major myths about computation: that computational processes must be algorithmic, and that computational processes are type-individuated by non-semantic properties. But where does this claim leave CS? What is CS all about? The answer, I think, is that CS is, indeed, the science of computers, but the notion that lies at the heart of computation is content: computers are systems whose dynamics are individuated by the content of the representations over which these dynamics are defined. CS is concerned with formal structures primarily because the relevant features of content that enter into computational taxonomies are formal features. Indeed, CS aims to build machines whose processes mirror formal relations and properties in their embedding environments. And CS is mostly dedicated to the digital electronic computers, and to the study of their algorithmic processes, simply because these are the systems whose theory is well-developed, and whose technological implementation is well-known and very, very fruitful. When other subsets of computers prove to be potentially interesting, from either a theoretical or a technological point of view, computer scientists will study them too, as indeed is the case with analog machines that implement neural nets.

Oron Shagrir

The Hebrew University
Jerusalem

NOTES

1. For different views on whether or not there are abstract computers see Copeland (1996) and Hayes (1997).

2. This answer is assumed by many practitioners, and it also lies at the center of recent distinguished philosophical works on computation, e.g., Cummins (1989), Chalmers (1996) and Copeland (1996). Copeland writes; "To compute is to execute an algorithm. More precisely, to say that a device or organ computes is to say that there exists a modeling relationship . . . between it and a formal specification of an algorithm" (p. 335).

3. Fodor (1980, p. 64) famously writes; "I take it that computational processes are both *symbolic* and *formal*. . . . they are formal because they apply to representations in virtue of (roughly) the *syntax* of the representations. . . . What makes syntactic operations a species of formal operations is that being syntactic is a way of *not* being semantic."

4. Versions of this machine appear in Copeland (1997) and in Shagrir (1997).

5. As remembered, a function is recursive if and only if it is computed by a Turing-machine.

6. Turing, Church and other logicians were urged, in the mid-1930's, to provide a rigorous characterization of the class of effectively computable functions because certain developments in logic called for such a characterization (i.e., Gödel's incompleteness theorems—published in 1931), and the unsolvability of the "decision problem" (proved independently by Church and Turing in 1936). For extensive discussion of these issues, see Gandy (1988) and Sieg (1994). In 1939, Turing himself introduced the idea of computing with an oracle. My example simply presents the oracle as part of the machine.

7. See Wolfarm (1985, p. 735 plus n. 3).

8. For the view that the *brain* may be an analog computer see Churchland and Sejnowski (1992).

9. The n-queens problem is the problem of locating n queens on an × n chess-board such that there is at most one queen in any row, column, and diagonal line. A neural net solution to the problem is offered in Shagrir (1992).

10. See Amit (1989) for an extensive review of the relations between attractor neural nets and magnetic systems.

11. Cummins writes: "To compute a function g is to execute a program. . . . Program execution reduces to step satisfaction" (pp. 91–92). Cummins and Schwarz (1991) follow the logical consequences of the definition, claiming that connectionist machines also do not compute.

12. Further support for the claim that some systems may compute their basic steps are: (a) The one-step/many-step distinction does not have a similar role in computability theory. Each of the six functions defining recursivity is surely recursive. Moreover, the functions satisfied by the basic instructions of a Turing-machine are surely Turing-computable!; and (b) A formal proof that consists of *one* step (e.g., an axiom) is considered as a proof, even if only a trivial one.

13. See, for example, Block (1990) and Egan (1995).

14. Note that S/*f* and S′/*f*′ provide descriptions of the same *particular* events, states, objects, and causal relations in **P**. In fact, they even abstract from the same physical properties (i.e., voltage currents). Moreover, the claim that a physical *object* may fall under two different syntactic types (of the same syntactic structure) at different times is no novelty. It is well-known that a flip detector may implement '1' at one time and '0' in another time.

It now turns out that we can also construct examples where a physical *event* can be seen as implementing different syntactic types.

15. I emphasize this point because of the criticism launched against "universal realizability" by Chalmers (1996) and by Copeland (1996)—see also Shagrir (1998). But, of course, the falsity of the universal realizability thesis does not entail the falsity of the more modest thesis. Indeed, there is no doubt that any theory of implementation must accommodate the fact that P implements both S and S′.

16. If, for example, the inputs and outputs of the gates are positive and negative charges, the temperature of the gate may depend on the absolute values of the currents. Thus we can implement one syntactic structure in the current values (as '+' and '−') and another in the temperatures.

17. This story is a twisted version of the thought experiment introduced by Davies (1991) and elaborated by Egan (1995). It is twisted because both Egan and Davies take it for granted that **N** always implements the same abstract automaton, whereas my version precisely challenges this assumption.

18. Note that whereas the task of a computing system is defined by content, different computing systems may have different *kinds* of content. The content of the states of desktop machines is assigned by the users, whereas the content of the states of a cognitive system is perhaps "natural" (e.g., causal). Thus the task of the system, when formulated in semantic terms, is sometimes characterized *via* "natural" contents (e.g., in the case of the visual system), but sometimes *via* "derived" content (e.g., in the case of an adder).

19. I am here (over)simplifying somewhat: the difference in the voltage-sets relations in the two contexts explains the different computational descriptions of **P**, *assuming that there is such a computational difference.* It is still possible that the voltage-sets relations are different, but that the electronic properties are still correlated with the same formal properties. In this case, the computational identity of **P** will turn out to be the same.

REFERENCES

Amit, D. (1989): *Modeling Brain Function*, Cambridge: Cambridge University Press.

Block, N. (1990): "Can the Mind Change the World?" in Boolos, George (ed.), *Meaning and Method: Essays in Honor of Hilary Putnam*, Cambridge: Cambridge University Press, pp. 137–70.

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

Churchland, p. S. and Sejnowski, T. (1992): *The Computational Brain*, Cambridge, MA: M.I.T. Press.

Copeland, J. B. (1996): "What is Computation?," *Synthese* 108, 335–59.

_____ (1997): "The Broad Concept of Computation," *American Behavioral Scientist* 40, 690–716.

Cummins, R. (1989): *Meaning and Mental Representation*, Cambridge, MA: M.I.T. Press.

Cummins, R. and Schwarz, G. (1991): "Connectionism, Computation, and Cognition," in Horgan, T. and Tienson, J. (eds.), *Connectionism and the Philosophy of Mind*, Dordrecht: Kluwer, pp. 60–73.

Davies, M. (1991): "Individualism and Perceptual Content," *Mind* 100, 461–84.

Egan, F. (1995): "Computation and Content," *Philosophical Review* 104, 181–204.

Fodor, J. (1980): "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology," *Behavioral and Brain Sciences* 3, 63–73.

Gandy, R. O. (1988): "The Confluence of Ideas in 1936," in Herken, R. (ed.), *The Universal Turing-Machine, A Half-Century Survey*, Oxford: Oxford University Press, pp. 55–111.

Hayes, P. J. (1997): "What is a Computer?," *The Monist* 80, 389–404.

Pitowsky, I. (1990): "The Physical Church Thesis and Physical Computational Complexity," *Iyuun*, 39, 81–99.

Putnam, H. (1988): *Representation and Reality*, Cambridge, MA: M.I.T. Press.

Shagrir, O. (1992): "A Neural Net with Self-Inhibiting Units for the N-Queens Problem," *International Journal of Neural Systems* 3, 349–52.

_____ (1997): "Two Dogmas of Computationalism," *Minds and Machines* 7, 321–44.

_____ (1998): "Multiple Realization, Computation, and the Taxonomy of Psychological States," *Synthese* 114, 445–61.

Searle, J. (1992): *The Rediscovery of the Mind*, Cambridge, MA: M.I.T. Press.

Sher, G. (1996): "Did Tarski Commit 'Tarski's Fallacy'?," *Journal of Symbolic Logic* 61, 653–86.

Smith, B. C. (1996): *On the Origin of Objects*, Cambridge, MA: M.I.T. Press.

Sieg, W. (1994): "Mechanical Procedures and Mathematical Experience," in George, A. (ed.), *Mathematics and Mind*, Oxford: Oxford University Press, pp. 71–114.

Turing, A. (1939): "Systems of Logic Based on Ordinals," *Proceedings of the London Mathematical Society* ser. 2 vol. 45, 161–228. Reprinted in Davis, M. (ed.) (1965): *The Undecidable*, Hewlett, NY: Raven Press.

Wolfarn, S. (1985): "Undecidability and Intractability in Theoretical Physics," *Physical Review Letters* 54, 735–38.