University of Wollongong

Research Online

Faculty of Engineering and Information
Sciences - Papers: Part A

Faculty of Engineering and Information
Sciences

1-1-2014

The instructional information processing account of digital computation

Nir Fresco

University of Wollongong, nfresco@uow.edu.au

Marty J. Wolf

Bemidji State University

Follow this and additional works at: <https://ro.uow.edu.au/eispapers>

Part of the [Engineering Commons](#), and the [Science and Technology Studies Commons](#)

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

The instructional information processing account of digital computation

Abstract

What is nontrivial digital computation? It is the processing of discrete data through discrete state transitions in accordance with finite instructional information. The motivation for our account is that many previous attempts to answer this question are inadequate, and also that this account accords with the common intuition that digital computation is a type of information processing. We use the notion of reachability in a graph to defend this characterization in memory-based systems and underscore the importance of instructional information for digital computation. We argue that our account evaluates positively against adequacy criteria for accounts of computation.

Keywords

instructional, processing, information, account, computation, digital

Disciplines

Engineering | Science and Technology Studies

Publication Details

Fresco, N. & Wolf, M. J. (2014). The instructional information processing account of digital computation. *Synthese*, 191 (7), 1469-1492.

The Instructional Information Processing Account of Digital Computation* / Nir Fresco & Marty Wolf

Abstract. What is nontrivial digital computation? It is the processing of discrete data through discrete state transitions in accordance with finite instructional information. The motivation for our account is that many previous attempts to answer this question are inadequate, and also that this account accords with the common intuition that digital computation is a type of information processing. We use the notion of reachability in a graph to defend this characterization in memory-based systems and underscore the importance of instructional information for digital computation. We argue that our account evaluates positively against adequacy criteria for accounts of computation.

1. Introduction

What is computation? At first blush, this question may seem trivial and uninteresting.

The Turing machine (TM) is a widely accepted model of digital computation that is often used to ascertain whether a specific physical phenomenon is computational¹. Finite computation in a TM is defined as a fixed mapping from inputs to corresponding outputs. The TM is used for defining computability, universality and computational complexity. Why then should we bother with this question?

The problem of characterizing computation is far from being trivial. To complicate matters we commonly find claims such as:

- “[Natural] cognition *is* a type of computation” (Pylyshyn 1984, xiii).
- “Every ordinary open system is a realization of every abstract finite automaton” (Putnam 1988, 121).
- “[T]he universe computes” (Lloyd 2006, 3).

But what does it mean to assert that cognition, every ordinary open system or the universe computes? Besides, if everything computes, then what computation means does not seem to matter.

* This is a preprint of the article appearing in *Synthese*. It is reproduced with the kind permission of Springer-Verlag. The final publication is available at <http://link.springer.com/article/10.1007/s11229-013-0338-5>.

¹ The subtle distinction between x being computable and x being computational is not further discussed here.

The TM model is insufficient for ascertaining whether some systems are computational. A finite state automaton (FSA) differs from the TM, since its input tape is not bidirectional, infinite or writable. Yet, the FSA is regularly used to solve computational problems. Even the modern digital computer does not have anything that corresponds to the TM's infinite tape. Furthermore, newer computational models such as quantum computers, relativistic computers, hypercomputers, connectionist networks, evolutionary computers and molecular computers are not necessarily compatible with the standard TM model. Do they all perform the same type of computation? If not, do they have something in common or are they completely distinct?

Even in the realm of computational cognitive science, it is far from clear what computation means. Very few accounts provide plausible explanations of this phenomenon. We highlight two plausible accounts in Section 5. Some inadequate accounts, which are motivated by finding a bridge between computation and mentality, presuppose that computational states are individuated in the same way that mental states are. However, this presupposition faces problems (removed-for-anonymity-1 2013). The present account does not share this presupposition.

To highlight additional problems inadequate accounts face, we briefly consider the physical symbol systems (PSS) account and the formal symbol manipulation (FSM) account. On the PSS account, a computational system is a physical symbol system that contains sets of interpretable and combinable symbols and a set of processes that operate on *symbolic* expressions by generating, copying, modifying, combining and destroying them according to instructions (Newell and Simon 1976, 116). On the FSM account, a computational system is a physical automatic formal

system, the states of which are identified as *symbols*, that manipulates symbols according to its own rules (Haugeland 1985, 76).

The FSM and PSS accounts are problematic in many respects. Lack of space permits only two brief remarks. First, both these accounts are grounded in *symbolic* computation and promote external *semantics* as the center of digital computation. “A computational process is one whose behavior is viewed as depending on the representational or semantic content of its states” (Pylyshyn 1984, 74). Second, despite their similarities they are extensionally different accounts, because they denote *different* classes of computational systems. The class of computational systems denoted by the PSS account only includes programmable stored-program universal computers. The FSM account classifies a broader range of computational systems, including for example, specific-purpose TMs. Both accounts exclude physical implementations of FSAs, flip-flops² and discrete connectionist networks. The resulting classes are too narrow, as they exclude too many paradigmatic digital computational systems (removed-for-anonymity-1 forthcoming-a).

It is generally assumed that digital computation can be freely defined as information processing. However, the truth of the claim that digital computation is information processing can only be ascertained once the precise notion of *information* is fixed. In removed-for-anonymity-1 (2012) and Piccinini and Scarantino (2011) various notions of information are examined as the basis for understanding computation as information processing. In this paper, we propose “instructional information”³ as the correct notion that gives rise to the instructional information processing (IIP) account of computation. According to our account, nontrivial digital

² Flip-flops are basic memory cells. A flip-flop is a bistable element with a clock input. Its state changes only in response to a clock edge, say, when the clock signal rises from 0 to 1 (Harris and Harris 2013, 108).

³ This notion is defined and explicated in the next section.

computation is the processing of discrete data through discrete⁴ state transitions in accordance with finite instructional information. Unlike other accounts of computation that are either too narrow or too broad, the IIP account has the right scope. It classifies CPUs, TMs, conventional digital computers, flip-flops and FSAs as nontrivial computational systems, whilst classifying chairs, digestive systems and walls as non-computational. Since it also does not posit the individuation of computational states in the same way that mental states are individuated, it is a plausible basis for a computational theory of cognition.⁵

The paper is organized as follows. In Section 2, the notions of *data*, *semantic content* and *instructional information* are defined and analyzed. In Section 3, we use the technique of reachability in a graph to motivate the IIP account by analyzing *memory-based* nontrivial computational systems. In Section 4, we explicate the IIP account by analyzing both memory-based and memoryless computation. In Section 5, we claim that the IIP account is adequate and improves on two plausible accounts of digital computation: the algorithm execution account and the mechanistic account. In Section 6, we show a positive evaluation of the IIP account against adequacy criteria for accounts of computation. Section 7 concludes the discussion and identifies directions for future work.

2. Instructional Information - Definitions and Preliminaries

Let us begin with two basic definitions of *data* and *semantic information* that are needed for explicating the notion of instructional information, which underpins the IIP account.

⁴ In this paper the property of being discrete - or digital - implies having sharp and distinct boundaries.

⁵ The evaluation of this last claim remains a task for further work.

Definition 1. A datum d is the lack of uniformity between at least two uninterpreted variables that are distinct from one another in a domain that is left open to further interpretation (Floridi 2011, 85).

Definition 2. An object O is an instance of semantic information, understood as semantic content, *iff*:

- 1 It consists of n data D ($n \in \mathbb{N}^+$).
- 2 D are well formed.
- 3 D are meaningful independently of their recipient (Floridi 2010, 21).

Semantic content requires fixing the conventions of the communication in some way so that data have the meaning we want them to have. Both well-formedness and meaningfulness appeal to something outside pure data (be that abstract or physical). The first requirement of Definition 2 is of positive data and is self-explanatory. The second requirement is that the data be structured according to some rules. A common conception of rules is the syntax rules for a language. Data can also be structured via physical characteristics of the system, such as the grouping of nucleotides to form codons in DNA. The first two requirements give rise to the simplest form of information: *structured data*. The addition of the third requirement yields *semantic content*. It only requires that structured data comply with the semantics of the system or code in question.

For the present analysis, we adopt Luciano Floridi's data based definition of semantic content. Floridi's theory of strongly semantic information that is founded on the above definitions (2011, chap. 5) is undoubtedly not problem-free. One of its key problems is the tenet that the underlying data have to be *veridical* for semantic

content to qualify as genuine semantic information (i.e., factual information) (cf. Fetzer 2004; Scarantino and Piccinini 2010). It might even be argued that there are other well-developed theories of *entropy* that seem more adequate for the analysis of the interaction between information and computation, such as Gibbs entropy, Shannon information and algorithmic information (Adriaans 2010, 44–45). However, elsewhere, the first author has argued that, at least, Shannon information and algorithmic information are not up to the task of explaining digital computation, for they lack the conceptual richness to capture the notion of instructional information (removed-for-anonymity-1 2012). These two conceptions are insensitive to the *meaningfulness* of data. It is certainly possible that the central definition of instructional information may be reformulated using Gibbs entropy, Shannon information or algorithmic information. However, it remains to be seen how such reformulation can be accomplished.

Another concern is that it may appear that our formulation of the IIP account yields a circular definition of computation, due to the second requirement of Definition 2. In computer science well-formedness is often defined by using a notion of computation that appeals to an automaton accepting a class of strings.⁶ If that were always the case, then our account of computation would itself depend on a notion of computation.⁷ However, examination of two common forms of computation demonstrates that well-formedness is generated by the organization of the system.

We consider both an abstract and a physical computational system. First, the well-formedness of the data in the TM is determined by the definition of TMs. The input data are restricted to certain forms in tape cells according to certain rules. The well-

⁶ We thank an anonymous reviewer for identifying this potential conundrum.

⁷ Incidentally, in passing it should be noted that, at least *prima facie*, using algorithmic information as an alternative to semantic content in our context inevitably yields a circular definition of computation. For algorithmic information *is defined* in terms of computational programs.

formedness in this case emerges from the positioning of the individual datum in the proper place on the tape. Given their abstract nature, each datum approaches Floridi's notion of data *de re* (2011, 85 ff). Second, the structure of data in a conventional digital computer comes from the physical organization of the system. Wires and flip-flops are organized in a particular order. It is that organization of the physical carriers of electrical signals as data *de signo* (Floridi 2011, 86) that structures the data. In both of these cases, and others, the structure of the data may be obtained from the organization of the system concerned, rather than through some computational requirement. This also shows that *being structured* is not an inherent ontological quality of data.

Semantic content can give rise to *factual* information (Floridi 2011, 103ff) and to *instructional* information. By Floridi's lights, for semantic content to qualify as factual information it also has to be veridical. However, instructional information cannot be qualified as true or false, for it is intended *to make* something happen. In this sense, it escapes the problems faced by factual information. The *correctness* and *lack of ambiguity* of the semantic content⁸, rather than its veridicality, contribute to yielding the right state of affairs when instructional information is processed. The three requirements of semantic content are *necessary* conditions for instructional information, but they need to be supplemented by a *sufficient* condition.⁹ This gap is filled by the following definition.

⁸ If the semantic content is incorrect, by following it reliably the expected outcome is unlikely to be obtained. If it is correct but ambiguous, it is impossible to follow reliably, for there are at least two courses of possible action.

⁹ Strictly, Floridi considers another type of instructional information that is based on what he calls "environmental information". Environmental information is conveyed by data that might be meaningful independently of an intelligent producer (Floridi 2010, 32). This type of information is distinguished from semantic content. In this paper, we confine our discussion to instructional information as a subtype of semantic content. We make another observation regarding environmental information below in the context of analyzing Boolean gates.

Definition 3. Semantic content is an instance of instructional information *iff* its satisfaction yields a definitive action in a given context.

Instructional information is prescriptive rather than descriptive. The debate over the nature of imperatives, the core of instructional information, is unsettled. Definition 3 accords with other classifications of imperatives. “An imperative [...] expresses an immediate demand for action but that does not describe a fact: an imperative is satisfied if we have the result of an agent’s action” (Ross 1941, 54). “[T]he kernel [...] of the imperative [is] the action [...] that the imperative enjoins” (Hamblin 1987, 45). “[I]mperatives convey requirements [...] for action or prescriptions of actions” (Pérez-Ramírez and Fox 2003, 55). Adopting Charles Hamblin’s *Action State Semantics* model whereby a world is a series of states connected by events¹⁰, allows the identification of those worlds in which a given imperative is satisfied in some set of states (Reed and Norman 2007, 417). The imperative “Close the window!” is satisfied in those worlds in which the window is closed given the right conditions. There exists a triple relation $\text{Pre} \rightarrow [\alpha] \text{Post}$, where “Pre” are pre-conditions and “Post” are post-conditions, that establishes the correctness of the action (α) performed. Action α is correct if it is possible to perform it when the pre-conditions are satisfied with respect to the state concerned (Pérez-Ramírez and Fox 2003, 45, 59).

It is worth noting that instructional information can also be expressed either negatively or positively. In English, for example, the construction of a negative imperative typically differs from the positive one by the presence of some negative quantifier, e.g., “Don’t close the window!”. This type of construction is found in some

¹⁰ Events are of two types: deeds, performed by specific agents, and happenings, which are world effects (Reed and Norman 2007, 417).

natural languages, but constitutes a minority overall (cf. Miestamo and Auwera 2007). In a general form, a negative imperative can be expressed as *<prohibitive, prohibition>* (Kaufmann 2012, 9). Importantly, for our present purposes, a prohibitive still yields a definitive action and is occasionally used in computational systems. NOP or NOOP, shorthand for *no operation*, instructs the computational system to *do nothing*. It is found in assembly language, high-level programming languages (e.g., an empty block statement “{}” in C or the pass statement in Python) and computer protocol commands. It is often used to synchronize operations.

Thus characterized, instructional information cannot rise to the level of factual information. An instruction, regardless of whether it is actualized as the satisfaction of semantic content in the right context, is neither true nor false. Also, instructional information need not necessarily be utilized. A washing machine user manual that specifies the order in which to set the dials contains instructional information even in the absence of an agent with the capacity to do so. This is a case of an unfulfilled instruction, rather than inaccurate or false description.

An important implication of Definition 3 for the IIP account is the necessity of a system having the capacity to carry out the action. A *capacity* is an inherent action that a system, either abstract or physical, can reliably and predictably execute. A system (or an agent) processes instructional information when, in general, it has at least *two capacities*.

Instructional information is conveyed either conditionally or unconditionally. We briefly examine how instructional information is used conditionally and unconditionally in computer programming. Consider the following simple code excerpt for calculating the average value of a set of integers.

```
def calcAverageValueOfList(inputList):  
    if inputList:
```

```

        sumOfValues = 0
    for number in inputList:
        sumOfValues += number
    return float(sumOfValues) / len(inputList)
else:
    return

```

Code excerpt 1. Computer code that calculates the average of the integer values of an input list.

How is instructional information conveyed by the statements above?

- “if inputList” - If the condition is true, “inputList” is not empty, the statements that follow it are executed. Whereas, “else” represents the alternative path.
- “for number in inputList” - This for statement iterates over the elements in the list “inputList”. The imperative character of this instruction is enabled by the inherent *conditional* that evaluates positively if there is at least one more element in the list.
- “sumOfValues += number” - This incremental assignment statement is an unconditional imperative that adds the value of the current element of “inputList” to the variable “sumOfValues”.

We note that function calls, such as “len(inputList)”, are equally describable as instructional information. In general, any function call is an imperative, since it directs the assignment of arguments to parameters and the execution of a specific body of code.

By way of concluding this section, we introduce three abbreviations for ease of exposition in the ensuing discussion. “Data” can denote unstructured data, structured data or semantic content.

- Let **D** stand for discrete data simpliciter in the broadest form. Any instance of D simply complies with Definition 1.

- Let **ud** stand for unstructured discrete data. Any instance of **ud** is an instance of D that satisfies only the first requirement of Definition 2 (i.e., being nonempty data).
- Let **sd** stand for structured discrete data. Any instance of **sd** is an instance of D that satisfies at least the first two requirements of Definition 2 (i.e., being nonempty and well-formed data).

The sets of all possible **ud** and **sd** are disjoint and their union is the total set of D .

3. Nontrivial Computational Systems and Reachability in a Graph

In this section, we develop a theoretical structure that helps in answering our main question, “What is computation?”. In particular, we analyze paradigmatic abstract computational systems and identify instructional information as an essential component of any notion of *nontrivial* computation. To proceed, we review some work by Pieter Adriaans and Peter van Emde Boas in which they develop the notion of meta-computational space for TMs (2011). Following that we turn our focus to memory models and adapt their notion to establish the concept of a *generalized meta-computational space* for analyzing all memory-based nontrivial computational systems. We include a very simple example to help concretize these concepts.

There are three views of TMs as computational systems that are helpful here: a specific TM computing on some specific input, a specific TM (not confined to a specific input), and the simultaneous consideration of all possible TMs. The most straightforward is the case of a specific TM computing on some specific input. Adriaans and van Emde Boas define an Instantaneous Description (ID) to include a description of the entire tape contents, the position of the tape symbol currently being scanned and the current state of the TM (the reader is referred to (Adriaans and van

Emde Boas 2011) for complete details). For our purposes, an ID describes a specific memory configuration of a particular TM. Adriaans and van Emde Boas also define the notion of a *configuration graph*. In this graph, every possible ID for a particular TM is mapped to a vertex of the graph and a directed edge is added to the graph from vertex with ID_i to the vertex with ID_j *iff* the configuration specified by ID_j follows from ID_i via a single move (i.e., a capacity) of the TM. A valid computation of the TM corresponds to a path between any two such vertices in the corresponding configuration graph. A path that starts at ID_0 , the unique ID representing the starting configuration of the TM, and continues through the graph to any vertex that contains an ID with the halt state corresponds to an accepting computation of the TM. The last notion adopted from Adriaans and van Emde Boas is that of a *meta-computational space*, which is the disjunct sum of all the configuration graphs for a given enumeration of TMs.

Next we make two adaptations to these concepts. As defined, the notion of meta-computational space applies only to TMs. Since our aim is analyzing systems for any memory model, we use the notion meta-computational space to refer to the disjunct sum of all of the configuration graphs for a given enumeration of systems over a fixed memory model. Our second adaptation stems from the observation that in a meta-computational space, there is nothing that guarantees that all edges are present. In fact, in the case of TMs, it is quite clear that many edges are not present due to the definitional restriction on TMs that requires the read/write head to move at most one tape cell in a single move. Thus, the meta-computational space generated by a given enumeration of TMs is not as general as possible. We use *generalized* meta-computational space to capture the notion of adding all of the edges to a meta-computational space.

Definition 4. A generalized meta-computational space for a given meta-computational space is the complete directed graph, including self-loops, on the vertex set of the meta-computational space.

Generalized meta-computational spaces are graphs based upon the memory model present in the system under consideration. We use the term *reachability graph* (RG) to refer to any subgraph of a generalized meta-computational space. Each RG is a possibly infinite graph where each vertex represents a specific memory configuration of the system, and an edge represents a *capacity* of that system to move from one memory configuration to another. For the simple example shown in fig. 1, we have a system with a 2-bit memory model¹¹. Thus, there are four possible memory configurations and hence, vertices in the RG. In the generalized meta-computational space there are no restrictions on the edges. That is, the system has the capacity to move from any one configuration of the two bits to any other.

Figure 1. Generalized meta-computational space for 2-bit memory system.

¹¹ It should be noted that a generalized meta-computational space with a richer memory model can be used to model any system with a simpler. For example, the generalized meta-computational space associated with TMs can be used to model the systems shown in fig. 1 and fig. 2. The advantage gained by considering different generalized meta-computational spaces is a clearer taxonomy of computational systems as is shown in Section 6.

Note that every computational system on two bits is a subgraph of this RG. The example illustrated in fig. 2, representing a 2-bit shift left register, is based on the same memory model, but as a computational system has a restricted set of capacities.

Figure 2. The meta-computational space for a 2-bit shift left register.

The notion of generalized meta-computational space removes the restrictions imposed by a specific computational system and gives a clearer picture of computation in memory-based systems. We argue that *memory-based nontrivial* computation, construed as the processing of data through discrete state transitions according to instructional information, is defined completely in terms of reachability in a generalized meta-computational space. Thus, instructional information provides the basis for understanding computation.

Our approach, then, begins by fixing a memory model that in turn yields a generalized meta-computational space. At this point, it is possible to structure a variety of different computational systems and classes of computational systems over the generalized meta-computational space by placing restrictions on the capacities of systems. In the case of the class of all TMs, restrictions include things such as the read/write head can move at most one cell to either the left or the right, the input is in the lowest numbered cells and so on. These restrictions on capacities, in turn, place

restrictions on edges in the RG. For example, the read/write head restriction implies that there can only be an edge between two vertices when the integers representing the head locations differ by either zero or one. Implicit in the definition of a TM is the requirement that the position of the head be ignored in the application of a move. In turn, this forces a single move of a specific TM to appear as infinitely many edges in an RG, even when the RG is a configuration graph.

Defining computational systems in this way, we have a uniform approach in which the memory model is identified, restrictions on capacities are specified, and edges, representing capacities, are included in the RG¹². A computation in the RG begins at any vertex and ends at any other vertex. A system, abstract or physical, that supports the underlying memory model and has the requisite capacities, implements that computation by processing the specific instructional information that actualizes those very same capacities.

An obvious shortcoming of the RG approach thus described is its lack of applicability to memoryless systems. However, it is clear that instructional information plays a key role in memory-based computation. To address this shortcoming, our focus changes in the next section to explicate the IIP account, analyze memoryless systems and discuss nontrivial computation at greater length.

4. Fleshing Out the IIP Account

In the previous section, we have developed an abstract notion of memory-based nontrivial computation. This description clearly delineates capacities within a system from the instructions whose satisfaction yields the activation of those capacities. In this section, we extend the use of the capacity/instruction distinction to motivate the

¹² This approach allows for descriptions of capacities to potentially be infinite. While infinite descriptions may not be particularly useful, they do not impact the finiteness requirement of instructional information.

IIP account in physical systems and explain both memory-based and memoryless computation in a uniform way.

According to the IIP account, nontrivial computation is *the processing of discrete data through discrete state transitions in accordance with finite instructional information*. Trivial computational systems process D, which need not be structured, by way of exercising the single capacity of the system. Nontrivial computational systems, on the other hand, process *instructional* information by way of being capable of processing at least two distinct instructions for the corresponding capacities.

The instructions processed are either conditional or unconditional relative to the particular chosen level of abstraction¹³ (LoA). In computer science, abstraction is the process of identifying objects, properties and interrelationships that arguably hold the essence of the computational system considered (van Leeuwen forthcoming). Some nontrivial computational systems clearly process conditional instructions. Conditional statements are common in programming languages (cf. code excerpt 1) and so a computer that executes a program typically also processes conditional instructions. Other nontrivial computational systems, such as certain flip-flops, only process unconditional instructions to exercise their capacities. Analysis of computational systems on the IIP account also requires consideration of the system's memory model. We separately consider computational systems that are memory-based and memoryless.

¹³ Floridi defines an LoA as an object of study (e.g., a system) that consists of a finite, non-empty set of observables, which are *interpreted* typed variables (2011, 52). A typed variable is a variable qualified to hold only some declared kind of data. In this context, 'interpreted' means that the typed variable represents some feature of the system under consideration. Note that an analysis that relies on the data being structured should proceed using *moderated* LoAs. For more on LoAs see Floridi (2011, chap. 3). In the context of computational objects, higher and lower LoAs interrelate. The interpretation of an LoA in terms of the *lower* LoA remains of utmost importance and the *interface among* LoAs is also crucial (van Leeuwen forthcoming). See (van Leeuwen forthcoming) for a discussion of the differences between the usage of LoAs in computer science and the method described by Floridi.

In memory-based nontrivial computational systems, the memory model accounts for which unconditional instruction is to be executed next. The execution of a conditional typically requires a change from one memory configuration to another in the RG (or back to itself via a self-loop). A computational system has the capacity to perform certain operations conditionally. Unconditional instructions, and possibly other D, trigger the execution of *conditional* capacities.¹⁴ The condition $p \rightarrow q$ is merely shorthand for $\neg p \sqcup q$. The truth tables for both statements are equivalent, but only the former explicitly represents a conditional. In other words, whether the instruction to be processed is conditional or unconditional in relative to a particular LoA. This also shows how specific organizations of trivial computational systems, that is, primitive Boolean gates, give rise to nontrivial computational systems.

For all genuine computational systems what matters to their data processing is the abstract *form* (i.e., *type*) of the processed D rather than the particular *realization* of the D (i.e., *token* data). The same transformation of D may take place in a variety of physical substrates (cf. a flip-flop implemented using rebounding water droplet collisions in Mertaniemi et al. 2012). This is the root of the multiple realizability of algorithms and data. An algorithm may be implemented in different programming languages and on different physical architectures. Multiple realizability of D is reflected both in Definition 1 - the matter of the implementation of D is left undetermined and in the articulation of RGs in that the nature of the instruction that activates a particular capacity is quite naturally left unspecified.

¹⁴ It can also be argued that, in some sense, every imperative can be expressed as a conditional. For example, “do X” can be expressed as “do X unless there is some exceptional circumstance”. The “exceptional circumstance” serves as a sanction imposed on the normal course of action. Still, this is the modus operandi of physical computational systems. The system computes as long as there is no failure.

This brings us to the relationship between abstract computation and the physical system that implements it. Any abstract computation (performed by, say, a TM, an FSA or a cellular automaton) is reducible to atomic operations over individual pieces of data, which can, but need not, be symbols. In accordance with Definition 1, data can be either abstract or physical. In a TM, data are implemented as symbols on its tape and as states of the finite control and head positions. These data may in turn be implemented as magnetic polarity changes (or as patterns of physical pits on a platter) and as electrical voltages, respectively, in a conventional digital computer. Below we give examples of physical computational systems in which ud and sd are physically realized.

On our account, the delineations between computational systems and non-computational systems, and between trivial and nontrivial computational systems depend on the input D , the processing of D , the output D and whether the systems in question have memory. Memory-based computational systems are grouped as classes of systems based on a shared memory model and capacities. However, memoryless computational systems are analyzed on the basis of whether and how they structure data. A NOT-gate, which has only one capacity, does not have the same capacity as any two-input, one-output Boolean gate. Each of the next two sections focuses on a particular class of computational systems.

4.1. Non-Computation and Computation in Memoryless Systems

Our first distinction is between computational and non-computational systems. If for any input D the system does not always produce the same output D (i.e., different inputs potentially cause different outputs), the system computes. For example, the input ud an AND-gate receives affects the output ud the AND-gate sends. Only (1,1) as input causes a logical 1 as output. Thus, the AND-gate computes, for it does not

always produce the same output. Conversely, if a system always produces the same output D irrespective of the received input D , then that system does not compute. We return to non-computational systems later.

The next distinction to be drawn is among memoryless computational systems. This class consists of two subclasses: all memoryless *trivial* computational systems and all memoryless *nontrivial* computational systems. The subclass of memoryless trivial computational systems in turn consists of two further subclasses: those systems operating on ud as input and those systems operating on sd as input.

Figure 3. The class of memoryless trivial computational systems is the union of two disjoint subclasses: those operating on ud and those operating on sd .

Memoryless trivial computational systems (see fig. 3) include physical systems such as two-input, one-output Boolean gates, the NOT-gate, half adders, full adders, and n -bit adders. There are similar abstract computational systems, such as logical conjunction and the three place Boolean function that produces both the sum and carry of adding those three bits. The NOT-gate and most two-input, one-output Boolean gates¹⁵ belong to the subclass of systems operating on ud . Swapping the

¹⁵ There are four two-input, one-output Boolean functions that operate on sd , including the function defined by the following four 3-tuples: $(0,0,0)$; $(0,1,0)$; $(1,0,1)$; $(1,1,0)$. The first two bits are the inputs and the last one is the output bit. The inputs $(0,1)$ and $(1,0)$ clearly show that the order of D matters (in contrast to, say, an OR-gate).

order of the input lines of, say, an AND-gate does not change its output: both (0,1) and (1,0) lead to the same output.

We consider two different perspectives on the AND-gate. In the first, the inputs are treated as *ud*, that is, neither bit plays any special role in the function computed. The output is determined by the gate's physical capacity that follows the rules of the logical conjunction operation. Note that viewed from this perspective, there is no notion of an instruction, since the inputs are unstructured. It is also clear that there is no need for an instruction as the system has precisely one capacity. This system performs its single capacity whenever *ud* are present as input. Alternatively, the input can be viewed as *sd*, with one bit identified as the control bit carrying instructional information. The capacity of the AND-gate can be expressed as: if the control bit is a 1, the datum bit is the result; otherwise 0 is the result.

Why is the first perspective of the AND-gate more appropriate for the explanation of its operation? In the AND-gate there is no functional distinction between the datum input bit and the control bit. It does not matter whether the first one is treated as a datum or as an instruction. Attributing one bit the role of *D* and the other the role of control can be done arbitrarily *ex post facto*. In general, structure may be attributed to any *ud*. This is due to the *structure* of data not being an inherent ontological quality of data. Yet here, promoting *ud* to *sd* is done arbitrarily.

The observation that the AND-gate and similar Boolean gates process *ud* and not *sd* has two important implications. The first implication is related to Floridi's classification of some instructional information as environmental information (cf. fn. 9). He argues that "the logic gates in the motherboard of a computer merely channel the electric voltage, [...interpretable] in terms of instructional information [...] such as 'if . . . then'. In this case, there is no semantics involved at the level of the gates".

Here, our present analysis diverges from Floridi's in that Boolean gates are *not* viewed as processing instructional information. They have a single capacity. This also brings us to the second implication. Even if an AND-gate processes semantic content, that semantic content would not satisfy Definition 3. For the operation of an AND-gate cannot be reasonably described as following an imperative. "[T]he characteristic purpose of imperatives is to influence choice. It is futile to tell [...] anyone to do something [...] he cannot help doing" (Hamblin 1987, 145). The AND-gate simply does what it "should" by design thereby exercising its only capacity in the presence of *ud*. "[A] *forced action* is *not* really *an action*; [...] the agent complies [...] with some persuasive force" (Hamblin 1987, 146, italics added).

A full adder is more complex in design than an AND-gate. This circuit has three inputs and two outputs and a single capacity. While the standard description of a full adder suggests that the three inputs are structured, a simple analysis of the behavior of the full adder indicates that they are not. Since the three input bits are *ud*, a full adder ought to be categorized similarly to the AND-gate.

An *n*-bit adder belongs to the subclass of memoryless trivial computational systems operating on *sd*. Its input is structured: the zeroth bits are separate from the first bits and so on. It is, therefore, more computationally powerful than, say, two-input, one-output Boolean gates, yet, it only performs trivial computation. It has only a single capacity. No instructional information is needed for its operation.

Figure 4. The class of all memoryless computational systems is the union of two disjoint subclasses: those processing instructional information (nontrivial & memoryless) and those that do not (trivial).

Multiplexers¹⁶ and multi-operation ALUs¹⁷ are examples of memoryless nontrivial computational systems (see fig. 4). Systems in this subclass not only operate on *sd* as input, but also process instructional information. Consider an ALU with two (or more) capacities, say, addition and subtraction. Because it has more than one capacity for possible action, it requires a control bit. There is a functional distinction between the operands and the control bit. Therefore, the order of the bits in the operands matters to the operation of the ALU. Note, however, that this ALU is memoryless. While the ALU receives input *sd*, it performs either an addition or subtraction and continues to do so until it receives different input *sd* or is powered off. A similar analysis applies in the cases of other memoryless nontrivial computational systems

¹⁶ A multiplexer is a combinational circuit that is used to choose an output among several possible inputs *sd* based on a value of a select signal (Harris and Harris 2013, 83).

¹⁷ An arithmetic logical unit (ALU) is an essential component of most CPUs. It combines a variety of arithmetic and logical operations, such as addition, subtraction, conjunction and disjunction, into a single unit (Harris and Harris 2013, 248).

that structure data, including systems such as a three-input, one-output multiplexer (Harris and Harris 2013, 83).

In sum, the dividing line between trivial and nontrivial computational systems is drawn by the processing of instructional information (see fig. 5). Trivial computational systems do not process instructional information, since they have only one capacity. The processing of D, that need not be structured, is done through a single action of the system by either the transformation from one type of D to another (e.g., a NOT-gate) or the modification of D (e.g., a XOR-gate). Nontrivial computational systems have at least two distinct capacities and are controlled by instructional information.

Figure 5. The class of all computational systems is the union of two disjoint subclasses: trivial and nontrivial computational systems. Only systems in the latter subclass process instructional information.

4.2. Nontrivial Computation in Memory-Based Systems

The principle of exercising control at the heart of the IIP account is already manifested by multiplexers and multi-operation ALUs, and is even more conspicuous in memory-based nontrivial computational systems, such as TMs (discussed in

Section 3), FSAs, flip-flops and conventional digital computers. In general, for memory-based computational systems, the memory model implies a structure to the D it holds. At each step of a TM computation, the combination of the symbol read from the tape (i.e., *sd*) and the state of a deterministic TM uniquely determines the capacity to be exercised next. This combination is, therefore, instructional information, which when processed actualizes a capacity of a TM to move from the current memory configuration to another.

The FSA is another example of an abstract memory-based nontrivial computational system. For it uses the underlying memory model of TMs and has the requisite capacities for implementing computations by processing the specific instructional information that actualizes those very same capacities. The main differences from the TM come from the restricted set of capacities (and thus, the simpler RGs) that FSAs are allowed. The contents of the FSA's input tape remains unchanged, and the position of the read-only head shifts only to the right by exactly one position. The key point is that transitions of the FSA represent capacities the FSA has to move from one memory configuration to another by processing instructional information. Since the set of edges allowed in the RG for FSA is much smaller than the set of allowable edges for TMs and the former can neither *store* nor *delete* arbitrary *sd* during its operation, the former is less computationally powerful than the latter by the IIP account.

The flip-flop is an example of a physical system related to the class of FSAs. The generalized meta-computational space for a system with two Data flip-flops is the one shown in fig. 1. There are four possible inputs (00, 01, 10, 11). Each of those is instructional information with each meaning transition to the indicated state. When the system processes any of the four instructions it actualizes its capacity to move

from one memory configuration to another, thus processing s_d through discrete state transitions in accordance with finite instructional information.

4.3. Replies to Two Possible Objections

We consider two possible objections to our account. The first objection concerns the case of systems computing the identity function. In Section 4.1 we characterize a computational system as one in which different inputs potentially cause different outputs. An apparent contradiction arises when a system S that computes the identity function takes D as input and returns the *same* D as output.

Our reply addresses three different systems computing the identity function. Simple mechanisms, such as delay gates, do not perform trivial computation. A delay gate is analogous to a wire carrying a binary signal (electricity at one voltage or another) where the same input D is simply produced as output D unchanged. The emphasis is on the output being a simple “copy” of *unchanged* D . The second case concerns a combinational circuit consisting of primitive Boolean gates organized in such a way as to compute the identity function. One might argue that this is problematic, since this combinational circuit, which should count as a memoryless trivial computational system, does not compute. But this claim is misguided. For any construction of such a circuit requires that the primitive Boolean gates process the input D thereby *transforming* it before the overall output yielded is the same as the input. The third case is resisting a similar claim that a TM that computes the identity function is supposedly not computational. In this case the RG specification of this TM shows that whilst the same input s_d is produced as output, the TM produces the output by moving from one memory configuration to another.

The second objection considered is to our characterization of systems that always produce the same output D irrespective of the input D as non-computational. Consider

a TM that “ignores” its input D or erases whatever input D there is on the tape, sums the numbers 1-9, writes 45 on the tape as output and halts. This TM “always” produces 45 as output irrespective of the input. This shows, the objection continues, that, on our account the TM does not compute, since for any input D it always produces the same output D (i.e., 45). This concern is addressed by again considering the RG of the TM. The erasure of a single input symbol represents a change in the memory configuration. An instruction is required to actualize that capacity, thereby meeting our characterization of computation.

5. An Improvement on Two Plausible Accounts

We now compare the IIP account with two *plausible* accounts of computation and show how our account resembles them, and more importantly how it improves on them. The IIP account resembles an account of digital computation in terms of algorithm execution. Whilst, informally, an algorithm is an ordered sequence of instructions to solve a particular problem, a more rigorous definition is problematic. A sequential-time algorithm, for example, can be defined as a state transition system that starts in an initial state and transitions from one state to the next until, if ever, it halts or breaks. However, there are other types of algorithms that may not be similarly defined, such as asynchronous parallel, distributed, real-time, hybrid and even quantum algorithms (Gurevich 2012). This problem leads to the key advantage of the IIP account over an explanation of computation in terms of algorithm execution.

For the sake of comparison, let us consider the algorithm execution account proposed by Jack Copeland (1996). According to this account, a system S computes a function F *iff* there exists a labeling scheme L and a formal specification of an architecture (e.g., 2.7GHz quad-core Intel Core i5) and an algorithm specific to the architecture (e.g., a mergesort algorithm written in machine language) that takes

arguments of F as inputs and delivers values of F as outputs such that the pair (S, L) is an honest model of that specification (Copeland 1996, 348)¹⁸. Our focus is limited to the executed algorithm and the executing architecture.

Copeland defines an algorithm as a “finite set of instructions such that, for some computing machine M , each instruction calls for one or more of M ’s primitive operations to be performed, either unconditionally or if certain conditions [...] are met” (1997, 696). An algorithm ϕ being *specific to* an architecture Θ implies not only that a Θ -type machine can run ϕ , but also that each instruction in ϕ calls explicitly for the execution of some sequence of the primitive operations available in Θ . If ϕ called for multiplication on an Θ -type machine that does not support multiplication as a primitive operation, it would only be executed correctly provided that each multiplication instruction in ϕ is replaced by, say, a series of addition instructions (Copeland 1996, 337).

However, Copeland’s description is consistent with the notion of a *program* rather than an *algorithm*. This leads to the undesirable consequence of computation being the execution of programs, rather than algorithms. Consider any algorithm for adding two positive n -digit numbers, such as the one that children learn in primary school. Such an algorithm can be implemented in arbitrarily many ways using a specific programming language and then compiled for a particular machine (converting the algorithm into a *program*). No data types are imposed by the algorithm nor are any limitations imposed by Θ . However, on Copeland’s account, an algorithm is *specific to* Θ and that makes it a *program*.

On this account, computation amounts to program execution, but what about computational systems that do not execute a *program*? Arguably, discrete

¹⁸ The reader is referred to (Copeland 1996) for complete details.

connectionist networks do not execute programs. Yet, we contend that, on the IIP account, these networks may be classified as computational systems. Recurrent networks change their persistent state(s) in the course of computation and thereby qualify as memory-based systems, whereas feedforward networks do not. Only some feedforward networks operate on sd and require instructional information to single out a particular capacity to be exercised; those qualify as nontrivial computational systems. It also seems a stretch to describe a flip-flop as executing a program, in accordance with Copeland's account. On the IIP account, on the other hand, the flip-flop is classified as computational. Importantly, if cognition (or at least part of it) is indeed computational, but not in virtue of program execution, then the IIP account is advantageous as a plausible basis for a computational theory of cognition.

We now turn to compare the IIP account and the mechanistic account, showing how the former is an improvement over the latter. On the mechanistic account, digital computation is the processing of strings of digits according to rules (Piccinini 2007a). A "digit" is defined as a macroscopic state of a component of a computational system (Piccinini and Scarantino 2011, 7).

We focus our analysis on the mechanistic "digit". On the mechanistic account, a large number of possible microscopic states correspond to each macroscopic state-type (Piccinini and Scarantino 2011, 7–8). On the IIP account, microscopic states, such as [4V-5V] and [7V-8V], are viewed as D (e.g., representing 0 and 1 bits on the input lines of an OR-gate). It may be argued that digits are D, but Gualtierio Piccinini maintains that many computations "depend not only on an input string of *data*, but also on the *internal state* of the [...] mechanism" (2007a, 509). *Data* are distinguished from *states*. He also argues that memory components in digital computers have the dual function of storing macroscopic states and indicating the memory state upon

request. “Their state constitutes either *data* strings or the *physical implementation* of abstract internal state” (Piccinini 2007a, 514, italics added).

The mechanistic “rule” also highlights this distinction. “A rule [...] is simply a map from input strings of digits, plus possibly internal states, to output strings of digits” (Piccinini and Scarantino 2011, 8). This suggests that digital computation proceeds from input *D* and possibly the internal states of the computational system to output *D*. We conclude that mechanistic digits qualify as *D*. On the IIP account, internal states of computational systems constitute *sd* at the appropriate LoA. For example, the internal state of the TM’s controller and the input datum – that is, the scanned symbol – play an equal role in determining the next move of the TM and together they qualify as *sd*.

To what type of information do mechanistic digits rise? They are not just *ud* as can be inferred from Piccinini’s emphasis on the ordering of digits (Piccinini 2007a, 515). Since the ordering of digits matters, they are *sd*. Piccinini argues elsewhere that digits “need not carry Shannon information at all [...] and] strictly speaking, computing systems need not process Shannon information” (Piccinini and Scarantino 2011, 27). He also argues that digital computation need not be the processing of nonnatural (i.e., semantic) information, which is descriptive and representational. So, digits need not carry nonnatural information (or be representations) (ibid, 30).

Do some digits qualify as instructional information? If digits are *sd* and the IIP account is correct, it seems that the answer must be affirmative. This is where the mechanistic “rule” comes into play. Since some digits act as input *sd* and others are output *sd*, it makes sense that some digits encode the *rules* for processing input *sd*. “Both data and instructions are physically instantiated as strings of digits, which computers can manipulate” (Scarantino and Piccinini 2010, 326). Furthermore,

Piccinini argues that “information” as employed in computer science is “non-truth-evaluable” in a manner that corresponds with “instructional information”. Accordingly, the same string of digits may play the role of either an instruction or data at different times (Scarantino and Piccinini 2010, 325–326).

The IIP account employs a richer notion of data thereby allowing an explanation of significantly more computational properties. Consider only those components of a digital computer that require power to retain any D. They provide the memory model for the system. Since for any given computer these components are finite in number, they give rise to a finite number of different possible memory configurations in the RG. The particular computer has certain capacities and as computation progresses the computer uses instructional information to transition from one memory configuration to another. This view of digital computers is found in a meta-computational space generated over an enumeration of all FSAs.

Here the advantage of the IIP account is most apparent. On the mechanistic account, FSAs are “so much less [computationally] powerful than ordinary digital computers” (Piccinini 2007a, 506). However, if computers are basically large complicated FSAs, the mechanistic account may not be effective at separating complexity due to size (as is the case between FSAs and computers) from complexity due to a particularly rich structure (as is the case between FSAs and TMs). The IIP account does better here. As another example, we noted above that the right combination of a NOT-gate and an OR-gate gives rise to a circuit with the capacity enabling conditional instructions (cf. the first objection discussed in Section 4.3). In contrast, primitive Boolean gates are trivial computational systems each with a single capacity. Thus, we have a separation that is due to increasing complexity. We conclude that the IIP account generalizes and improves on the mechanistic account.

6. The IIP Account Evaluated

In this section we evaluate the adequacy of the IIP account. For simplicity, we adopt Piccinini's criteria (2007a) for this evaluation, since they are self-explanatory and justifiable.

According to the first criterion, *objectivity*, whether a system is computational has to be determined as a matter of fact. Unlike some authors who claim that computational descriptions are vacuous (Putnam 1988; Searle 1990), we maintain that computation is objectively identifiable relative to the appropriate LoA. Memory-based nontrivial computation is defined completely in terms of memory models, capacities and reachability. Either the criteria for reachability are met or not. Any adversary's enthusiastic attempt to describe some arbitrary transitions of microscopic states of some wall as nontrivial computation is faced with a challenge. If she claims the wall to be a *memoryless* computational system, she has to show which microscopic states are treated as input sd, what their structure is, what capacities the wall has and by virtue of what instructional information they are actualized when processed. If she claims the wall to be a *memory-based* computational system, she also has to show what properties of the wall are used to define the IDs and edges of the RG.

It may be objected that a conventional AND-gate, for example, could be just as well described differently as an OR-gate when the standard interpretations of logical 0 and logical 1 are reversed (Shagrir 2001, 374). This supposedly shows that the computational identity of the system is observer-relative. But it does not follow that computation *is* observer-relative. Certainly, the underlying data processing capacity of the AND-gate, for example, can be used in different ways, such as logical conjunction or disjunction, giving rise to "different" computations relative to different LoAs. But,

as suggested before, an analysis of computational systems in terms of the lower LoA is of utmost importance (cf. fn. 13).

The point is that a semantic reinterpretation of the AND-gate's operation as logical disjunction at a higher LoA does not undermine the system *being* computational. The information processing operation analyzed at the lower LoA is simply given another name relative to some higher LoA. Logical disjunction may very well be called conjunction instead, but it is still a computation. The “black box” does not change, and the transformation of D it performs remains unchanged. The only change is how the input and output D are interpreted. Besides, systematic reinterpretations are relatively simple for *trivial* computations. But, more complex computations are harder to systematically reinterpret as a different computation. The observer-relativity of computation is discussed at length elsewhere (removed-for-anonymity-1 unpublished).

According to the second criterion, *the right things compute*, both abstract and physical paradigmatic computational systems have to be classified as computational. As shown above, abstract systems, including TMs and FSAs, are nontrivial computational systems. So are multi-operation ALUs, multiplexers, flip-flops, CPUs and conventional digital computers. Whilst primitive Boolean gates as well as some combinational circuits (e.g., half- and full-adders and n-bit adders) are classified as computational, they only perform trivial computations on our account.

According to the third criterion, *the wrong things do not compute*, paradigmatic non-computational systems are classified as non-computational. Planetary systems, digestive systems and walls do not perform computations on the IIP account. Planets move along ellipses where they revolve around a larger object (a star) in accordance with Kepler's laws of planetary motion. But that does not amount to them processing

instructional information. Any computational description applied to their planetary motion may only be applied *ex post facto*. It is perhaps unsurprising that non-computational systems may only be described computationally *ex post facto*. Just as describing some physical object *O* by a system of equations does not imply that *O* is a system of equations, describing *O* computationally does not imply that *O* is computational (Piccinini 2007b, 99). Even if the processes of *O* are *computable*, it does not follow that they are *computational*. The question is what gives such a computational description explanatory force.

The explanatory force of this description is provided by the underlying assumptions of this description. Even if we assumed that the elliptical movement of some planet is the execution of an imperative instruction (whatever the instruction and the input *sd* operated on are), to qualify as a nontrivial computation there needs to be another distinct path the planet could move along that is a second capacity. At best, a planetary motion may be described as a trivial computation. Consider the second perspective offered in Section 4.1 on the operation of Boolean gates and the ascription of structure to the input *D* processed by an AND-gate. Removing this complication yields a simpler description. Occam's razor demands this as the more appropriate description.

According to the fourth criterion, *miscomputation*, physical computational systems may fail to compute in the presence of noise. For memory-based systems, miscomputation is explained by the IIP account through the notion of paths in RGs. Making a transition from one vertex to another in the absence of an edge is a miscomputation. When *sd* are mapped into physical states and physical computation takes place on those physical states miscomputation may occur. It occurs when some noise causes a physical system to transition from a physical state s_i to another state s_j

and $F^{-1}(s_i)$ is either undefined or $F^{-1}(s_j)$ does not follow $F^{-1}(s_i)$ at the abstract level of processing instructional information (where the function F maps sd to the physical state realizing them). For example, if some noise causes a flip-flop to enter an undefined state, the flip-flop miscomputes. In memoryless computational systems, a miscomputation occurs when a particular capacity is actualized where it would not have otherwise been actualized in the absence of noise.

The possibility of miscomputation reveals the elusive aspect of *purposefulness* in both trivial and nontrivial computational systems. An artificial computational system may be said to act for a *purpose* according to its design. When it fails to accomplish the purpose for which it was designed, a miscomputation can be identified (removed-for-anonymity-1 forthcoming-b). An important distinction for analyzing purposefulness is between *internal* and *external* teleology. Objects that have some immanent property that makes them goal-directed can be said to be internally teleological. If an object has some goal assigned to it by some goal-conceiving agent, then it is externally teleological (Mahner and Bunge 1997, 368). Accordingly, an artificial computational system can have an externally assigned purpose, which is, arguably, key for identifying the function(s) it computes. A natural computational system, on the other hand, can only plausibly have a purpose *for the system* itself, that is, it is internally teleological.

Yet, the claim that purposefulness is necessary for classifying a system as computational is problematic. Consider a computer-illiterate monkey that is provided with components, such as transistors and diodes, that are needed for building two-input, one-output Boolean gates. Suppose that the monkey accidentally ends up constructing an OR-gate (OR_{mon}) that is a physical duplicate of an OR-gate purposefully constructed by an engineer (OR_{eng}). Since OR_{eng} was constructed

purposefully, it may be classified as computational. Yet, OR_{mon} , which was not constructed according to some purposeful design, cannot be classified as computational. This leads to an absurd consequence where only one of the two OR-gates may qualify as being computational, despite them being physical duplicates. A detailed analysis of purposefulness must await another forum. But suffice it to say that the appeal to purpose in determining what function is being computed by a given system is *epistemological*, insofar that it is enforced by our explanatory goals, rather than *metaphysical* in nature (Matthews 2011).

According to the fifth criterion, *computational taxonomy*, different classes of computational systems have different capacities. Our taxonomy assumes the truth of the Church-Turing Thesis, which states that every effectively computable function is computable by a TM. A general-purpose TM can simulate any specific-purpose TM, thereby being able to compute any partially recursive (Turing-computable) function. Extant formalisms of computability (e.g., Kleene's formal systems model, Gödel's recursive functions model, Church's lambda calculus and Post's machines) are all Turing-equivalent. Whether this extensional equivalence suffices for computation simpliciter remains contentious (removed-for-anonymity-1 2013). Any problem that cannot be solved by a TM, given the Church-Turing thesis, presents a limit to what can be accomplished by any form of machine that works in accordance with effective methods. It remains an open empirical question whether there are in fact deterministic physical processes that cannot be simulated by a TM (Copeland 2004, 15).

Computational complexity is accounted for in two ways, the richness of the memory model and the richness of the capacities for actualizing instructional information. We have shown above how memoryless trivial computational systems, which need not actualize instructional information, gain more computational power

when their input D are structured. The next step up in computational complexity is memoryless nontrivial computation. Systems in that category lack memory but have more than one capacity (enabled by the corresponding imperative instructions) and process instructional information.

The next step up is memory-based nontrivial computation. Even though the FSA and TM can be built on the same memory model, an FSA has more restrictions on capacities than a TM, therefore, the latter is more computationally powerful than the former. It is worth emphasizing that it is not simply the number of internal states that determines the computational power of a TM. A general-purpose TM using just seven states and four symbols can simulate any specific-purpose TM, even if the latter has more states (Neary and Woods 2012). The RG of a general-purpose TM is much richer than the RG of the specific-purpose TM that it is simulating. This example demonstrates that computational systems with richer capacity sets (edges in the RG) are computationally more powerful for the same underlying memory model.

7. Conclusion

Before concluding we make a few conjectures regarding the applicability of the IIP account to other information processing systems. The most plausible conjecture is that some discrete connectionist networks qualify as nontrivial computational systems, whilst others qualify as trivial computational systems. As observed above, some connectionist networks change their persistent state(s) during the course of computation and others do not. Moreover, some networks that have only a single capacity, say, to perform a linear summation of n neural inputs, do not need to process instructional information. We discuss this at length elsewhere (removed-for-anonymity unpublished).

A second less obvious conjecture is the possible applicability of some version of the IIP account to biological information processing systems. Many biologists hold that the causal role of genes should be understood in terms of their carrying information about their various products. Genes carry a message that is *supposed* to be expressed even if that message is not *actually* expressed (Godfrey-Smith 2008). Genes and only genes code the amino acid sequences of protein molecules. This informational property is only applicable to the amino acid sequence (Godfrey-Smith and Sterelny 2008). If genes carry messages that have imperative, rather than descriptive, contents, then “instructional information” seems applicable. However, we note that, at the very least, unlike the case of digital computation, the input and output differ in their nature. The outputs are amino acids, which are the building blocks of protein molecules, whilst the input is the raw material in the cell. A thorough analysis of the memory model of genes will facilitate the application of the IIP account to biological systems.

The last conjecture, which certainly seems the most contentious one, is the possible applicability of (a modified version of) the IIP account to natural cognitive systems. We emphasize that in the case of conventional digital computation the driving force behind the computational process is instructional information. However, natural cognitive systems also process semantic contents that are *descriptive* and contingently either true or false. Human cognitive agents experience and inhabit the world as a *semantic reality* and interact informationally with their environments and other agents through their bodily interfaces (Floridi forthcoming). Whether our actions are guided by knowledge (where the underlying data are truthful) or *just* beliefs (where the underlying data may be false), the information in question is descriptive rather than

prescriptive. The IIP account does not preclude the possibility that the processed D be descriptive information.

It is our contention that the IIP account is adequate for the explanation of digital computation. First, it is compatible with the common intuition that digital computation is a type of information processing. Second, unlike accounts that reduce digital computation to symbolic computation and exclude paradigmatic computational systems, the IIP account has the right scope. Moreover, it applies equally well to abstract systems, such as TMs and FSAs, and physical systems, such as flip-flops, ALUs and conventional digital computers. Third, whilst the IIP account classifies any system that executes a program (in the sense used in computer science) as computational, it is not restricted to this type of computational systems.

References

- Adriaans, Pieter. 2010. "A Critical Analysis of Floridi's Theory of Semantic Information." *Knowledge, Technology & Policy* 23 (1-2): 41–56. doi:10.1007/s12130-010-9097-5.
- Adriaans, Pieter, and Peter van Emde Boas. 2011. "Computation, Information, and the Arrow of Time." In *Computability In Context*, by S Barry Cooper and Andrea Sorbi, 1–17. Imperial College Press.
- Copeland, B. Jack. 1996. "What Is Computation?" *Synthese* 108 (3): 335–359. doi:10.1007/BF00413693.
- ———. 1997. "The Broad Conception of Computation." *American Behavioral Scientist* 40 (6): 690–716. doi:10.1177/0002764297040006003.
- ———. 2004. "Computation." In *The Blackwell Guide to the Philosophy of Computing and Information*, edited by Luciano Floridi, 3–17. Malden: Blackwell.
- Fetzer, James H. 2004. "Information: Does It Have To Be True?" *Minds and Machines* 14 (2): 223–229. doi:10.1023/B:MIND.00000021682.61365.56.
- Floridi, Luciano. forthcoming. "Perception and Testimony as Data Providers." *Logique et Analyse*
- ———. 2010. *Information: a Very Short Introduction*. Oxford: Oxford University Press.
- ———. 2011. *The Philosophy of Information*. Oxford: Oxford University Press.
- Godfrey-Smith, Peter. 2008. "Information in Biology." In *Cambridge Companions to Philosophy*, edited by David Hull and Michael Ruse, 103–119. New York: Cambridge University Press.
- Godfrey-Smith, Peter, and Kim Sterelny. 2008. "Biological Information." In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Fall 2008. <http://plato.stanford.edu/archives/fall2008/entries/information-biological/>.

- Gurevich, Yuri. 2012. "What Is an Algorithm?" In *SOFSEM 2012: Theory and Practice of Computer Science*, edited by Mária Bielíková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, 7147:31–42. Berlin, Heidelberg: Springer Berlin Heidelberg.
- Hamblin, Charles L. 1987. *Imperatives*. New York, NY: Basil Blackwell.
- Harris, David Money, and Sarah L Harris. 2013. *Digital Design and Computer Architecture*. Waltham, MA: Elsevier: Morgan Kaufmann.
- Haugeland, John. 1985. *Artificial Intelligence: the Very Idea*. Cambridge, Mass.: MIT Press.
- Kaufmann, Magdalena. 2012. *Interpreting Imperatives*. Dordrecht; New York: Springer Science+Business Media B.V.
- Lloyd, Seth. 2006. *Programming the Universe: a Quantum Computer Scientist Takes on the Cosmos*. New York: Knopf.
- Mahner, Martin, and Mario Bunge. 1997. *Foundations of Biophilosophy*. Berlin; New York: Springer.
- Matthews, Robert. 2011. "Natural Computation." In The Institute for Advanced Studies, Hebrew University in Jerusalem, Israel. <https://sites.google.com/site/iascomputationbrainhuji2011/home/previous-lectures/Matthews%282011%29NaturalComputation.pdf?attredirects=0&d=1>.
- Mertaniemi, Henriikki, Robert Forchheimer, Olli Ikkala, and Robin H. A. Ras. 2012. "Rebounding Droplet-Droplet Collisions on Superhydrophobic Surfaces: From the Phenomenon to Droplet Logic." *Advanced Materials*. doi:10.1002/adma.201202980. <http://doi.wiley.com/10.1002/adma.201202980>.
- Miestamo, Matti, and van der Johan Auwera. 2007. "Negative Declaratives and Negative Imperatives: Similarities and Differences." In *Linguistics Festival: May 2006, Bremen*, edited by Andreas Ammann, 59–77. Diversitas Linguarum v. 14. Bochum: Universitätsverlag Brockmeyer.
- Neary, Turlough, and Damien Woods. 2012. "The Complexity of Small Universal Turing Machines: A Survey." In *SOFSEM 2012: Theory and Practice of Computer Science*, edited by Mária Bielíková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, 7147:385–405. Berlin, Heidelberg: Springer Berlin Heidelberg.
- Newell, Allen, and Herbert A. Simon. 1976. "Computer Science as Empirical Inquiry: Symbols and Search." *Communications of the ACM* 19 (3): 113–126. doi:10.1145/360018.360022.
- Pérez-Ramírez, Miguel, and Chris Fox. 2003. "Imperatives as Obligatory and Permitted Actions." In *Computational Linguistics and Intelligent Text Processing*, edited by Alexander Gelbukh, 2588:52–64. Berlin, Heidelberg: Springer Berlin Heidelberg.
- Piccinini, Gualtiero. 2007a. "Computing Mechanisms." *Philosophy of Science* 74 (4): 501–526. doi:10.1086/522851.
- ———. 2007b. "Computational Modelling Vs. Computational Explanation: Is Everything a Turing Machine, and Does It Matter to the Philosophy of Mind?" *Australasian Journal of Philosophy* 85 (1): 93–115. doi:10.1080/00048400601176494.
- Piccinini, Gualtiero, and Andrea Scarantino. 2011. "Information Processing, Computation, and Cognition." *Journal of Biological Physics* 37 (1): 1–38.
- Putnam, Hilary. 1988. *Representation and Reality*. Cambridge, Mass.: MIT Press.

- Pylyshyn, Zenon W. 1984. *Computation and Cognition : Toward a Foundation for Cognitive Science*. Cambridge, MA: The MIT Press.
- Reed, Chris, and Timothy J. Norman. 2007. "A Formal Characterisation of Hamblin's Action-State Semantics." *Journal of Philosophical Logic* 36 (4): 415–448. doi:10.1007/s10992-006-9041-z.
- Ross, Alf. 1941. "Imperatives and Logic." *Theoria* 7 (1): 46–71. doi:10.1111/j.1755-2567.1941.tb00034.x.
- Scarantino, Andrea, and Gualtiero Piccinini. 2010. "Information Without Truth." *Metaphilosophy* 41 (3): 313–330. doi:10.1111/j.1467-9973.2010.01632.x.
- Searle, John R. 1990. "Is the Brain a Digital Computer?" *Proceedings and Addresses of the American Philosophical Association* 64: 21–37.
- Shagrir, Oron. 2001. "Content, Computation and Externalism." *Mind* 110 (438) (April 1): 369–400. doi:10.1093/mind/110.438.369.
- Van Leeuwen, Jan. forthcoming. "On Floridi's Method of Levels of Abstraction." *Minds and Machines* (Special Issue on Floridi and Philosophy of Information)