Chapter 10
Philosophy of Computation

Zoran Konkoli, Susan Stepney, Hajo Broersma, Paolo Dini, Chrystopher L. Nehaniv, and Stefano Nichele

Abstract Unconventional computation emerged as a response to a series of technological and societal challenges. The main source of these challenges is the expected collapse of Moore's law. It is very likely that the existing trend of building faster digital information processing machines will come to an end. This chapter provides a broad philosophical discussion of what might be needed to construct a theoretical machinery that could be used to understand the obstacles and identify the alternative designs. The key issue that has been addressed is simple to formulate: given a physical system, what can it compute? There is an enormous conceptual depth to this question and some specific aspects are systematically discussed. The discussion covers digital philosophy of computation, two reasons why rocks cannot be used for computation are given, a new depth to the ontology of number, and the ensemble computation inspired by recent understanding of the computing ability of living cell aggregates.

10.1 Introduction

Given a physical system, what can it compute? In broad philosophical terms this question is normally referred to as *the implementation problem*. This seemingly practical question has a surprising conceptual depth: Any attempt to formalize a rigorous answer (e.g. in pure mathematical terms) is bound to end in paradoxes. To illustrate the types of paradoxes that usually emerge, consider Hilary Putnam's answer to this question. As a critique of the thesis of computational sufficiency in cognitive science Putnam, in the appendix of his book (Putnam, 1988), suggested a construction, or a recipe, that can be used to turn any object into a device that can implement any finite state automaton with input and output.

S. Stepney et al. (eds.), *Computational Matter*, Natural Computing Series,
https://doi.org/10.1007/978-3-319-65826-1_10

In brief, the thesis of computational sufficiency states that the human brain can be modelled as an abstract automaton, and that the different states of mind are simply the different states of the automaton (Von Eckardt, 1995). In somewhat simplistic terms, should this be the case, then this would explain to a large extent what the mind is. Note a particular focus on computation in this context. Here the mind is strongly related to the ability to compute.

Putnam's agenda was to show that the notion of computation is simply too broad to be used to define what mind is. His goal was to show that every material object has an intrinsic ability to compute. Namely, for Putnam's construction to work the object that the procedure is being applied to must have a set of rather generic properties (e.g. the system should not be periodic), that are naturally realized in existing objects. Thus a corollary of Putnam's construction is that any object can implement any finite state automaton. Note that the statement does not read "some objects can implement any automaton", or "every object can implement some automaton". It was precisely this corollary that was the key motivation behind the construction. Putnam wished to illustrate that the ability to compute is something that is intrinsic to every object, and that this ability *per se* cannot be used to define what a mind is. According to this corollary, a rock can compute anything, and it should have a mind of its own. This statement is clearly a paradox, since it contradicts our intuition regarding what computing means. This illustrates the first type of paradoxes one can encounter when aiming for a formal (mathematically rigorous) answer to the implementation problem.

The second type of a paradox is as follows. Given that every simple object, even a rock, can be turned into a computing device, the information processing engineers that are exploring various devices for information processing applications should achieve their goals with much less efforts than they are obviously investing in finding new device designs. This is clearly another paradox. To describe it, the term *the natural computability paradox* has been coined (Konkoli, 2015). Note the key difference between the two types of paradoxes: the implementation problem emphasizes the existence of computation, while the natural computability problem emphasizes the use of the device.

The paradoxes that have just been presented indicate that such generic philosophical-computing-oriented questions should not be taken too lightly. A seemingly rather intuitive and straightforward question that deals with the philosophy of computation can have an immense depth. Our main goal in this chapter is to provide a structured exposé of how such questions could be asked, point to the paradoxes that are arising in the process, and discuss ways of resolving such paradoxes.

This chapter is organized on the following principles. Two approaches are exploited to present the material. (i) The traditional way is to define a class of systems and then investigate their expressive power (e.g. the standard models of computation like Turing Machines). We cover that angle for completeness, simply to help a reader versed in digital computation who perhaps wishes

to understand the unconventional computation better. While doing so, some philosophical aspects are emphasized, in addition to the usual emphasis on the expressive power and complexity of computation. (ii) We present a palette of philosophical ideas and frame them as thought experiments. Each thought experiment deals with a given system for which we consider the task of turning it into a useful information processing device. In the process, we discuss the key philosophical questions we wish to address. These discussions are organized into separate sections where each section contains a structured and rigorous set of statements describing what a computation might be in a well-defined context, addressing either a class of systems (emphasizing the model of computation context), or a particular fixed (e.g. dynamical) system.

The above principles are implemented as follows. For completeness, a few key ideas of the digital computation paradigm are reviewed first, in Sect. 10.2. It is implicitly understood that the reader has an elementary understanding of the Turing machine concept. Accordingly, this concept is not covered in detail. The section emphasizes some philosophical aspects of the Turing machine construct, since it is the standard answer to the questions of what computing is and what computing means in the context of digital computation. The following sections extend the discussion towards unconventional computation. We begin these discussions by addressing the first thought experiment in Sect. 10.3: what would it take to turn a rock into a computer? Sect. 10.3.1 and Sect. 10.3.2 address the question from two related yet distinct ways. Obviously, we are trying to justify from a theoretical point of view that a rock cannot compute.[1] Philosophical relation between systems and control mechanisms is discussed in Sect. 10.4. In Sect. 10.5 an intimate relationship between the concepts of number and state is discussed, and how these concepts are related to finite state automata and permutation-reset automata. Sect. 10.6 addresses the problem of instantiating computing systems that behave as living cells do: they multiply, process information, aggregate into complex structures, and eliminate redundant computational units when they are no longer needed, all while being affected by their environment. Sect. 10.7 contains a brief summary of the topics covered.

10.2 Philosophy of digital computation

What is digital computation? In this section we are going to explain what we mean by digital computation. Our concept of digital computation is based on an abstract model of this type of computation due to Alan Turing that has been around since the 1930s (Turing, 1936), and an outline of a machine to perform this type of computation described by John von Neumann in 1945.

[1] Note that due to Putnam's construction this question has a surprising depth to it. Arguing that a rock cannot compute is not as easy as it sounds.

All our modern digital computers are based on this model of Turing and its implementation by von Neumann.

Turing's aim of defining computation the way he did, was to give an abstract description of the simplest possible device that could perform any computation that could be performed by a human. His purely abstract definition of computation raises a number of controversial philosophical and mathematical problems we will encounter. Moreover, it can be argued that computation, understood in the abstract terms of Turing, is too far removed from the physical implementation of computation. There are quite a few other models of digital computation around, and we mention several of them briefly. We focus on the Turing model because it is a reasonably simple model, and has the same computational power. We explain what we mean by this.

Moreover, the Turing model has been instrumental in the development of the theory of computational complexity, a very active area within computer science and mathematics. We explain this relationship in Chapter 11 of this volume. Even taking into account the controversial nature of Turing's abstract model and its implementation, the great success story of digital computation is based on his ideas, together with the invention and continuing miniaturization of the transistor since 1947. Transistors are the fundamental building blocks of all modern electronic devices, including our supercomputers, PCs, laptops, mobile phones and other gadgets. We shortly explain the impact of what is known as Moore's Law on the advancement of nanotechnology and the further miniaturization of our digital equipment.

10.2.1 Turing machines

Just like any type of computation, digital computation is always dealing with or considered to be computing something. For a general discussion on the philosophical issues related to central common notions like data, representation and information, we refer to Sect. 10.3.2. In fact, the philosophy of information has become an interdisciplinary field in itself, intimately related to the philosophy of computation. Here, for digital computation in particular, we assume that any data and necessary information for the computation at hand is represented at the abstract level by strings of zeros and ones, and at the implementation level in a von Neumann setting by clearly distinguishable low (for a zero) and high (for a one) currents or voltages, determining the switch state of transistors (zero for Off and one for On).

Based on the above assumption, we are now turning our attention to the abstract model of a computational device due to Turing, that is widely known as a Turing machine (not *the* Turing machine, as there are many variations on the basic concept).

10.2.2 The basic version of a Turing machine

What we describe next is usually referred to as the standard deterministic Turing machine.

A Turing machine (TM) is a so-called finite-state automaton combined with an unlimited storage medium usually referred to as a tape. It is called a finite-state automaton because at any moment during a computation it can be in one of a finite number of internal states. The TM has a read/write head that can move left and right along the (one-sided infinite) tape. Initially, the TM is in its start state and the head is at the left end of the tape. The tape is divided into cells, each capable of storing one symbol, but cells can be empty as well.

For simplicity and in the light of the above remarks, we can think of the symbols as zeros and ones, and that there is an additional symbol that represents a blank cell (in most text books λ is used for this purpose). Furthermore, in its simplest form the TM has a transition function that determines whether the read/write head erases or writes a zero or a one at the position of the current cell, and whether the head moves one cell to the left or right along the tape (but not passing the left end of the tape). In addition to these operations, the TM can change its internal state based on the transition function and the symbol in the current cell on the tape. Hence, the transition function determines instructions of the form: if the TM is in state s, the head of the TM points to cell c, and the TM head reads a zero, one, or λ from c, then it writes a zero, one or λ into cell c, moves its head one cell to the left or right, and changes to state s' (or stays in state s).

In fact, the behaviour of the TM is completely determined by the transition function (that can be a partial function): the TM starts in its start state with its head at the left end of the tape, and operates step by step according to the transition function. The TM halts (stops moving its head or changing states or erasing/writing symbols) if the transition function for the current state and cell content is undefined; otherwise it keeps operating. In principle, it could go on forever.

If the TM halts on a particular binary input string w that is initially written on the tape, we can interpret the resulting binary string on the tape as the output of the TM on input w. This way, we can interpret the operations of a (halting) TM as a mapping from input strings to output strings, hence as the computation of a (partial or total) function. Using this interpretation, one can define Turing computable functions as those that can be computed on a TM in the above sense. The subsequent operations of the TM can then be seen as an algorithmic procedure or program to compute the output for any given input of a Turing computable function. For these reasons, TMs are a model for computability, although they do not model the physical processes of a real computer or the command lines of programming languages directly. In fact, Turing invented his abstract model in a time real computers as we know them today did not exist yet. His intention was to formalize the notion

of what is sometimes referred to as 'intuitively computable' or 'effectively computable'. We come back to this later.

We know – in fact, Turing already knew – that there exist many functions that are not Turing computable. There are simple counting arguments to show that such functions must exist, but there are also easy tricks to actually construct examples of such functions. From a philosophical point of view, the interesting question here is: are such functions not computable at all because they are inherently non-computable? Or do there exist more powerful devices or alternative models for computation that can compute some functions that are not Turing computable? Perhaps surprisingly, up to now all known alternatives are equally powerful, in the sense that they can compute exactly the same functions. Note that we are not talking about computation time, complexity issues and memory usage here, only about the ability to compute the output value(s) of a function for all possible input values. Interestingly, it was Turing himself who claimed that any effectively computable function could be computed on a TM. In the next sections we gather supporting evidence for his claim.

There exist several alternative paradigms for (digital) computation, like for instance models based on RAM, λ-calculus, While programs, and Goto programs. We will not give any details on such models here, but just mention that all the existing models are, in some sense, equivalent to the Turing model.

10.2.3 The Church-Turing thesis

As we have seen, the investigations about computability have led to a number of approaches to actually try to get our hands on what it means for something to be computable. The earliest attempt was based on the seminal work of Turing, who defined when functions are computable in terms of his abstract model of computation. We have called these functions Turing computable. It turns out that all other notions of computability based on existing alternative approaches are equivalent: the computation of a function in any of these approaches can be simulated in any of the other approaches. In other words, a function is computable in any of these known approaches if and only if it is Turing computable. In this light, it is widely accepted among computer scientists that it is likely that any notion of effective or intuitive computation is equivalent to Turing computation. This is known as the Church-Turing thesis for computable functions.

Church-Turing Thesis I *A function f is effectively computable if and only if there is a TM that computes f.*

It should be clear that the above statement is not a theorem, but more like a working hypothesis. A possible proof for the statement would require a

precise mathematical definition of what we mean by an effective computation. That would mean being back at square one. Due to the existing supporting evidence, one could consider taking the statement of the Church-Turing thesis as a definition of what we mean by effectively computable functions, with the risk that one day someone might turn up with a more powerful model or device for computation.

For the purpose of explaining what the existing theory of computational complexity based on TMs entails, we now focus on decisions problems. These are problems for which we require a Yes or No answer for any instance of the problem. Solving such problems is in a way closely related to computation, and requires only a slight adaptation of the abstract model of TMs. As a consequence, the above Church-Turing thesis also has a counterpart for decision problems.

10.2.4 Decision problems and (un)decidability

To be able to decide whether a specific instance of a decision problem is a Yes instance or a No instance, any algorithm for solving this problem has to reach one of the two conclusions, for any instance of the problem. Therefore, it is intuitively clear that we need to extend our TM model by defining which of the halting states should correspond to a Yes answer and which should not. For this purpose a subset of the states of the TM is designated and called the set of accepting states.

Assuming that the instances of the decision problem are encoded as input strings on the tape of the TM, we now consider the set of instances as a language over an alphabet. A language L over an alphabet (for simplicity, think again of a set of strings of zeros and ones) is said to be recognized by the TM if for all strings $w \in L$ ($w \notin L$) the computation of the TM halts in an accepting state (does not halt in an accepting state or does not halt at all); L is decided by the TM if, subject to this, the TM halts on all $w \notin L$. If there exists a TM that decides a language L, then L is called decidable. In fact, L is decidable if its characteristic function is Turing computable (outputting 1 or 0 when halting in an accepting or non-accepting state, respectively).

The counterpart of the Church-Turing thesis for decision problems is as follows.

Church-Turing Thesis II *A decision problem P can be solved effectively if and only if there is a TM that decides the language corresponding to an encoding of P.*

As in the case of computability, also here we have no formal description of what we mean by effectively solving a decision problem.

We know that there are undecidable problems like there are non-computable functions. Again, from a philosophical point of view, the interesting question

here is: are such problems not decidable at all because they are inherently undecidable?

10.3 Why rocks do not compute

The material presented in the previous section focused on digital computation. It can be seen as a way to answer the question what computing means, and how it can be realized using digital devices. There are many systems in nature that process information and do not resemble digital devices. Such digital devices can be used to simulate existing systems (e.g. fluid dynamics software packages or chemical reactions simulators, or ultimately, various software packages that can be used to simulate living cells). However, the converse is not necessarily true. For example, it is hard to judge whether the living cells performs digital computation since the dynamics of the living cell hardly resembles anything digital, at least not in an obvious way.[2] There is a plethora of systems that are being investigated for information processing applications, that are far from being digital, e.g. this whole book addresses the possibility of using amorphous materials. The question is, to what extent can we extrapolate our understanding of digital devices to understand unconventional computation? How large is the step that is needed to bridge between the two to gain a unified understanding of both?

Clearly these issues are extremely complex and very broad. Very likely, any attempt to obtain a systematic answer is likely to end in failure. Accordingly, as an illustration, in this and following section Putnam's paradox is discussed in the context of unconventional computation. This is done in the form of a thought experiment. Assume that, against all odds, the goal is to turn a rock into a computer. Can we tell, in any rigorous mathematical way, why rocks do not compute? The following two subsections argue that they, indeed, cannot from two different but still related perspectives.

10.3.1 Powerful computation with minimal equipment

Putnam's construction has been criticized in several ways and there have been numerous responses to Putnam's work (Brown, 2012; Chalmers, 1996; Chrisley, 1994; Copeland, 1996; Godfrey-Smith, 2009; Horsman et al., 2014; Joslin, 2006; Kirby, 2009; Ladyman, 2009; Scheutz, 1999; Searle, 1992; Shagrir, 2012). The most common argument is that the auxiliary equipment that would be needed to turn a rock into an automaton would perform actual computation. Such arguments contain an implicit assumption that there

[2] Gene expression networks behave as digital switches, but intrinsically they are not constructed using digital components, their collective behaviour appears such.

is an intention to actually use the device. Here the natural question to ask is: what type of computation is performed most naturally by the rock?

10.3.1.1 Finding the right balance

In this thought experiment, the goal is to have a simple device consisting ideally of rock and nothing else. Since this is clearly not possible, the question is what is the minimal amount of equipment that should be used to achieve the information processing functionality (of some automaton), and what is the automaton? This line of reasoning has been formalized in depth in Konkoli (2015).

Namely, assume that the goal is to compare the computing abilities of two physical systems, which are fixed, but otherwise arbitrary. This can be done by simply investigating how hard or easy it might be to use the systems for computation. Clearly, each of the two systems might be suitable for a particular type of computation, and this might be guessed in some cases, but we wish to ask the question in a generic way: is there a procedure for identifying the most suitable computation for a given system?

As an example, Putnam's construction indicates that both a single bacterial cell and the human brain have the same computing power. However, there is an intuitive expectation that it should be much easier to use the human brain for computation, provided it would be ethically justifiable to use the human brain in this way. We also have a rather good intuitive understanding of what each of these systems could compute (e.g. one could use bacteria for relatively simple sensing purposes, while the human brain could be used to play a game of chess). What is this expectation based on? Is it possible to formalize this implicit intuition about their respective computing powers?

This rather extreme example shows that a way to understand what a given system can compute best (and possibly distinguish it from other systems in terms of their computing power) is to analyze how to actually use the system to perform computation. For example, it would be very hard to force bacteria to play a game of chess, this is simply not practical. Thus the ease of use of a given object (for information processing tasks) is a good starting point for understanding which information processing tasks it can perform naturally.

The reasoning in the above has been formalised mathematically in Konkoli (2015). The mathematical formalisation is reviewed in here briefly, Thereafter it is applied to the issue of the computing rock.

10.3.1.2 The mathematics of balances

The first question one must address is how to measure the "amount" of auxiliary equipment that needs to be used to turn a system into a computer. Given that a suitable definition can be found, the amount should be as small

as possible, since we wish the system to perform the computation (and not the equipment).

The amount of the auxiliary equipment can be "measured" by quantifying the complexity of the computation performed by the auxiliary equipment L while implementing an automaton A. The complexity can be measured using the concept of the logical depth (Bennett, 1988), which is a measure of the length of the digitized description of the automaton. Thus given the description of the interface L and the automaton A being implemented, there is a procedure to evaluate the complexity of the computation it performs, $H(L|A)$. The least costly implementation L_* is such that $H(L|A) > H(L_*|A)$ for every conceivable equipment (interface) L. This complexity should be compared with the complexity of the automaton A being implemented by the device, which is denoted by $H(A)$.

The balance between the two is the most important concept. The most natural automaton implemented by the device is the one that appears with the largest complexity $H(A)$ and the lowest cost of implementation $H(L_*|A)$. In less mathematical terms, we are asking the question: what is the most complex computation that a given device can compute most naturally (with the least costly interface)?

This balance between the complexity of the interface and the complexity of the computation achieved can be expressed in many ways, e.g. as

$$F(A) = H(A) - H(L_*|A) \tag{10.1}$$

or by using the ratio $F(A) = H(A)/H(L_*|A)$. Then the most natural implementation A_* can be identified by maximizing the above expression(s) with regard to A, i.e. $F(A_*) > F(A)$ where A is any imaginable automaton.

The formula (10.1) suggest that even the most trivial automaton, the one that does nothing, A_0, is a natural implementation. No interface is needed to implement such an automaton and both logical depths are balanced in some sense. A similar type of balance can occur for more complex automata. Presumably, there is a whole range of automata with roughly identical F values. However, this trend is broken at some point, which can be used to define the natural computation performed by the system.

10.3.1.3 The case of a rock

Finally, returning to the rock, the key question at this stage is how can one use the construct in (10.1) to analyze what a rock can compute? Given that a rock is given, the whole range of automata should be investigated. These automata occur with a varying degree of complexity, and one has to look at the trends in the $F(A)$, and find the automaton for which the complexity of the function being computed per the complexity of interface implementation is the largest. After such point on the complexity scale the complexity of

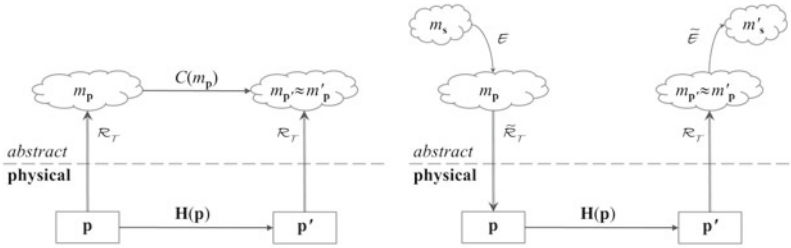

Fig. 10.1 (a) A sufficiently commuting diagram: the physical evolution **H** and the abstract evolution C give the same result, as viewed through a particular representation $\mathcal{R}$. (b) A computation: prediction of the abstract evolution C through instantiation, physical evolution, and representation. (Adapted from Horsman et al. (2014))

the interface simply explodes. The reason why the rock does not compute is that the values for $H(L_*|A)$ grow very fast for any automaton that starts deviating from the null automaton A_0. The null automaton represents the natural computation performed by the rock.

10.3.2 Abstraction/representation theory

Classical computer science regards computations, and often computers, as mathematical objects, operating according to mathematical rules. However, every computer, whether classical or unconventional, is a physical device, operating under the laws of physics. That computation depends in some way on the laws of physics is demonstrated by the existence of quantum computing, which results in a different model from classical computing, because it operates under different physical laws. Classical computation implicitly assumes Newtonian physics.

Unconventional computational matter, from carbon nanotubes to slime moulds and beyond, raise the question of distinguishing a system simply evolving under the laws of physics from a system performing a computation. Abstraction/representation theory (Horsman et al., 2014; Horsman, 2015; Horsman et al., 2018; Horsman et al., 2017a; Horsman et al., 2017b; Kendon et al., 2015) has been developed specifically to answer the question: 'when does a physical system compute?'

Consider a physical system $\mathbf{p}$ evolving under the laws of physics $\mathbf{H}(\mathbf{p})$ to become $\mathbf{p}'$ (Figure 10.1(a)). Let this system $\mathbf{p}$ in the physical world be represented by a model $m_{\mathbf{p}}$ in the abstract world, via some representation relation $\mathcal{R}_{\mathcal{T}}$, where the representation is relative to some theory $\mathcal{T}$. Note that $\mathcal{R}_{\mathcal{T}}$ is not a mathematical relation, since it relates physical world objects to

their abstract world models. We can similarly represent the result of the physical evolution, $\mathbf{p}'$, as the abstract world model $m_{\mathbf{p}'}$. Now let C be our abstract world model of the evolution, resulting in $m'_{\mathbf{p}}$. We say the diagram (sufficiently) *commutes* when these two resulting models $m'_{\mathbf{p}}$ and $m_{\mathbf{p}'}$ are sufficiently close for our purposes. Finding good theories and models such that the substrate is well-enough characterised that the diagram sufficiently commutes for a range of initial states of $\mathbf{p}$, is the subject of experimental science.

Now assume we have such a commuting diagram for some well-characterised substrate $\mathbf{p}$. We can use this to compute, in the following way. We encode our abstract problem $m_{\mathbf{s}}$ into an abstract computational model $m_{\mathbf{p}}$ (for example, via computational refinement). We then instantiate this model $m_{\mathbf{p}}$ in some physical system $\mathbf{p}$ (Figure 10.1(b)). Note that this instantiation relation is a non-trivial inversion of the representation relation, and has connections with engineering (see Horsman et al. (2014) for details). We allow the physical system to evolve under its laws, then represent the resulting state back in the abstract realm as $m_{\mathbf{p}'}$. Since the diagram sufficiently commutes, this observed result $m_{\mathbf{p}'}$ is a sufficiently good prediction of the desired computational result $m'_{\mathbf{p}}$. This computational result can then be decoded back into the problem result $m'_{\mathbf{s}}$.

Hence the physical system has been used to compute the result of the abstract evolution C. This gives the definition: *physical computing is the use of a physical system to predict the outcome of an abstract evolution.*

Note that the physical computation comprises three steps: instantiation of the initial system, physical evolution, and representation of the result. Computation may be 'hidden' in the initial instantiation and final representation steps, and needs to be fully accounted for when assessing the computational power of the physical device. Such hidden computation may include steps such as significant image processing to extract a pattern representing the result from a physical system, or the need to initialise or measure physical variables with unphysically realisable precision.

This definition allows the same abstract computation to be realised in multiple diverse physical substrates, using different instantiation and representation relations (Figure 10.2). Note that if a particular substrate $\mathbf{q}$ is not well-characterised, so that the result $m_{\mathbf{q}'}$ is being compared against some other computed result $m_{\mathbf{p}'}$ in order to check that it is correct, then $\mathbf{q}$ is *not* being used to compute: rather, the substrate $\mathbf{q}$ is being used to perform experiments, possibly in order to characterise it. Computation requires prediction of a result, not of checking a result against some alternative derivation.

This definition applies to multiple substrates being used to perform a computation together. There are two conceptually different approaches (Horsman, 2015), although they can be combined. Figure 10.3(a) shows a hybrid computing system: the problem is decomposed in the abstract domain, and part of it is performed in one substrate, part in another, and the separate abstract results are combined. Figure 10.3(b) shows heterotic computing (Hors-

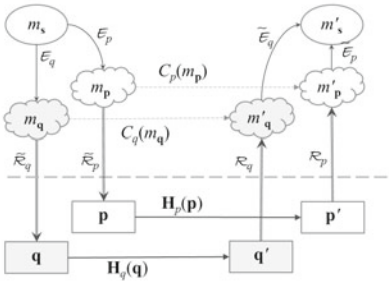

Fig. 10.2 Alternative realisations of the same computation in different physical systems, using different instantiation and representation relations.

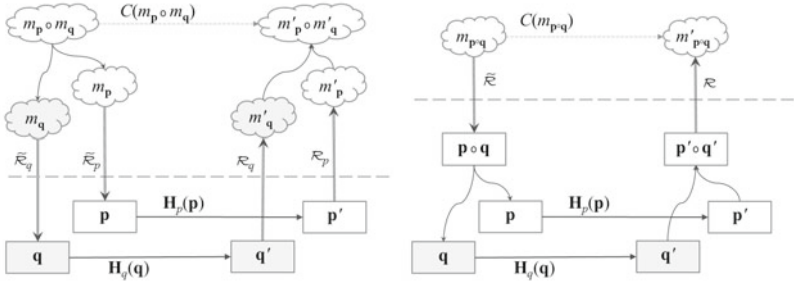

Fig. 10.3 Multiple substrate computing: (a) hybrid computing; (b) heterotic computing. (Adapted from Horsman (2015))

man, 2015; Kendon et al., 2015). Here the decomposition happens in the physical domain, and the instantiation and representation relations apply to the composed physical system as a whole. This potentially allows the composed physical system to have computational capability greater than the sum of its individual parts.

This definition of physical computing demonstrates why a rock does not compute, does not implement any finite state automaton (Horsman et al., 2018). The rock's purported computation is actually occurring entirely in the representation relation being used to interpret the result, not in the rock itself. And the specific computation (specific choice of representation relation) is being imposed *post hoc*, using some previous computation of the answer; it is not a *prediction*.

The definition of physical computation talks of a physical system "being used". This "user" is the *representational entity* (*computational entity* in Horsman et al. (2014)), There is no need for this entity to be conscious, sentient, or intelligent (Horsman et al., 2017b). However, it must exist; see Horsman et al. (2014) for details of why this is the case. This requirement can

summarised as "no computation without representation" (Horsman, 2015). This in turn demonstrates why the entire universe is not a computer. Unlike the case of the rock, where representation is everything, here there is no representational activity at all: everything is in the physical domain, and there is no corresponding abstract computation being performed. Clearly representational entities may instantiate and represent part of the universe to perform their computations, but the entire universe is 'merely' a physical system.

10.4 On the requisite variety of physical systems

In the previous sections, the examples of a Turing machine and a rock have been presented with the purpose of clarifying what can compute and what can be computed. Turing machines are ideal models and make use of an arbitrary number of internal states. Rocks (hypothetically) make use of arbitrary complex interfaces, i.e. auxiliary equipment to the computational automaton. In order for a physical system to compute, some problem inputs have to be encoded in a way that is "understood" by the computational substrate. In other words, the inputs have to have an effect on the internal state of the computational medium. The computational system itself can hold (represent) a certain number of internal states. If we want to be able to observe any kind of computation, some result has to be read from the physical system and decoded by some kind of output apparatus, i.e. interface.

If a physical system can represent a certain number of internal states, the computational complexity of the problems that can be computed by such system is bounded by the number of output states that can be distinguished. Computational matter exploits the underlying physical properties of materials as a medium for computation. As such, the internal theoretical number of states in which the material can be, e.g. state of each of the molecules composing the material system, is several orders of magnitude higher than what can be practically decoded (unless we use particle collider detectors as reading apparatus, colliding particles as computational material and quarks as units of information). What can we compute with a physical system then? The available computational power is bounded by the number of states that are available to the observer, e.g. electric apparatus or any apparatus that measures any interesting physical property of the material.

Such philosophical relation between systems and control mechanisms has been rigorously formalized within the field of cybernetics by Ross Ashby, a pioneer British cyberneticist and psychiatrist. Ashby, recognized as one of the most rigorous thinkers of his time, formulated his law of requisite variety (Ashby, 1956) which states, in a very informal way, that in order to deal correctly with the diversity of problems, a (control) system needs to have a repertoire of responses which is at least as many as those of the problem. Ashby described the systems under his investigation as heterogeneous, made

of a big collection of parts, great richness of connections and internal interactions. Even if Ashby proposed the concept of relative variety in the context of biological regulation, i.e. organisms' adaptation to the environment, it has been adopted and reformulated in a large number of disciplines. Some examples include Shannon's information theory (Shannon, 1948), structure and management of organizations and societies (Beer, 1984), behavior-focused design (Glanville et al., 2007), and computational systems in general.

In the following sections we review the concept of variety and give a formulation of Ashby's law of requisite variety in the context of computation. We describe some philosophical issues related to variety, computation and complexity, such as the intrinsically incorrect variety of physical computational systems.

10.4.1 Variety

Consider the set of elements $S = \{a, b, c, a, a, d\}$; its variety is the number of elements that can be distinguished. As such, the variety of S is 4. In many practical cases, the variety may be measured logarithmically (if base 2, then measured in *bits*). If a set is said to have no variety, it has all elements of one type and no distinctions can be made. If logarithmically measured, the variety of a set with only elements of one kind is $\log_2 1 = 0$.

10.4.2 Law of requisite variety

Let O be the set of outcomes of a system, D the set of disturbances that can deteriorate the outcomes and R the set of regulations available to a regulator (control mechanisms) to counterbalance the disturbances and maintain the functionality of the system. Let us denote V_O, V_D and V_R the varieties of the sets O, D and R, respectively. If the varieties are measured logarithmically, the minimal value of V_O (numerically) is $V_D - V_R$. If the value of V_D is given and fixed, V_O's minimum can be lessened only by a corresponding increase in V_R. Only variety in R can force down variety in D, "only variety can destroy variety".

Glanville (2004) describes two types of possible system controllers, one in the form of a regulator (the kind of control that "allows us to stay upright when skiing, stable in the face of perturbations", the skiing control) and one in the form of a restriction ("in a classroom the variety of the teacher is much lower than the variety of the class but some Victorian teachers used to handle the situation by restricting the variety of the students" (Robinson, 1979), restrictive control).

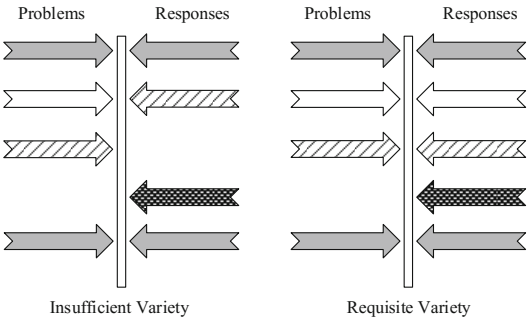

Fig. 10.4 Graphical representation of the law of requisite variety. On the left the variety of responses is insufficient or incorrect, on the right there is requisite variety (at least enough variety) of responses.

Now, for the sake of creating a more realistic analogy, let us consider a physical system D that is supposed to perform some sort of computation, O is the set of possibly incorrect (or unwanted) outcomes of the systems (if the result is read and interpreted correctly, O has minimum variety) and R is the system's controller (e.g. reading apparatus, interface or output mapping). In the ideal case, the variety of R would be as much as the variety of D and the variety of O would be minimized.

Figure 10.4 shows a simple graphical representation of the original formulation of the law of requisite variety as initially proposed by Ashby. In practice, the number of available responses of the system has to match the possible problems to be solved in order to have requisite variety. Otherwise the variety is said to be insufficient or incorrect.

10.4.3 Insufficient variety of physical systems

In the context of a physical computational system, requisite variety may be considered at two different stages:

1. the number of internal states of the systems has to code (represent) at least the number of possible input instances of the problem under investigation;
2. the number of responses that can be read has to have at least the variety of the possible number of states represented by the systems.

While the first point is intimately connected to the fact that rocks cannot compute, the latter raises a deeper concern. The number of states that a physical system can represent is typically higher than the number of states

that can be decoded. Hence, the variety of complex physical systems is not well defined. The way we have been able to control systems of increasing computational complexity has relied by far on Moore's law: building better controllers of continuously increasing variety. Bremermann (1962) postulated that any unit of matter has a finite computing capacity, according to the Laws of Physics. He calculated the computational capacity of a gram of matter and called this number Bremermann's constant, which is equal to 10^{47} bits per second. With this number, Ashby derived the computational capacity of the whole universe as 10^{100} bits. (It is not our intention to argue this calculation, as few orders of magnitude here do not make any difference.) It is possible to think of systems with greater variety than the computational power of the universe. Imagine a screen with 50×50 pixels than can be either black or white. Its variety is 2^{2500}. Thus, no control system with such variety can be built, no matter if Moore's law still holds or not. From a practical perspective, this implies that the complexity of the problems that can be solved is bounded by the number of states (and the scale) of the input/output interface used for decoding and reading the result.

10.4.4 Variety as complexity, new computational models?

Variety is a synonym of complexity. Cybernetics has studied systems independently of the substrate in which computation may happen (Wiener, 1961) and the same set of concepts is suitable for formalizing different kinds of systems. To paraphrase Ashby, one of the "peculiar virtues of cybernetics is that it offers a method for the scientific treatment of the system in which complexity is outstanding and too important to be ignored", and again, "variety, a concept inseparable from that of information". Shannon used information as measure of uncertainty (read complexity). If a message is predictable, it carries little information as there are few states that are very probable and information content may be derived from a probability distribution. On the other hand, if all the states have same probability of occurrence, the information cannot be predicted beforehand. Ashby added that "it must be noticed that noise is in no intrinsic way distinguishable from any other form of variety". Shannon's Theorem 10 (Shannon, 1948) is formulated in terms of requisite variety as "if noise appears in a message, the amount of noise that can be removed by a correction channel is limited to the amount of information that can be carried by that channel". Bar-Yam (2004) uses variety as synonym of complexity and proposed the law of requisite complexity. In the context of matter that computes, it can be reinterpreted as "the controller (input/output apparatus) for a complex system needs to be at least as complex as the system it attempts to control".

If computation is to be embedded into physical matter systems, a different computational model that goes beyond the standard Turing mechanistic model may be needed, as material computers are devices that interact with the physical world. Dodig-Crnkovic and Burgin (2011) describe the mechanistic world by the following principles:

1. The ontologically fundamental entities of physical reality are [space-time and matter (mass-energy)] defining physical structures and motion (or change of physical structures);
2. All the properties of any complex physical system can be derived from the properties of its components;
3. Change of physical structures is governed by laws.

It may be argued that Turing models (Turing, 1936), which consist of an isolated computing machine that operates on an input tape of atomic symbols, are mechanistic models. The assumption of a mechanistic model is that the laws of conservation of energy, mass, momentum, etc. hold, i.e. the system is isolated from the environment. Computational matter is hard to model mechanistically because of its inherent complexity. If Ashby's law of requisite variety is considered, in order for a computational model to be well defined it has to match the complexity of its environment. Physical systems exhibit a much higher complexity than Turing machines. Hence, it may be necessary to have more powerful models than Turing machines in order to represent matter that computes.

More than 20 years after Ashby's law of requisite variety was formulated, new (second-order) cybernetics (Foerster, 2007; Maruyama, 1963) started to give more importance to the positive side of requisite variety instead of the negative ones: "Give up trying to control, [...] gain access to enormous amounts of variety, [...] a potential source of creativity". von Neumann, while working with self-replicating automata, postulated a lower complexity threshold under which the system would degenerate, but above which would become self-reproducing. Ashby himself wrote a note on this, saying that a good regulator, i.e. controller, should account for emergence in the variety. The emergence of new functionality in a system should add to the variety of the regulator as to be able to cope with unexpected disturbances.

Requisite variety is an ideal condition and physical systems should aspire to have a variety as well defined as possible (or at least enough for the kind of computational problems one may want to solve).

10.5 On the ontology of number and state

In order to perform a computation, an abstract mathematical model requires a physical substrate or system that changes its physical state in a corresponding manner. This suggests that the Numbers of computations and the States

of the physical systems they represent are intimately connected. Although this observation is already sufficient to motivate ontological questions such as 'what is a state?' and 'what is a number?', the question of the relationship between them investigated in this section arose from a different perspective on the representation problem. Namely, can the time-evolution of a physical system be used to "guide" a computation, or to define or enable a specific *type* of computation?

10.5.1 Motivation

The correspondence of the time evolution of an electronic system with certain mathematical operations was exploited by the analogue computers of the 1950s. For example, different arrangements of circuit elements (resistors, capacitors, etc.) connected to one or more operational amplifiers yield algebraic or analytical relationships between the input and output voltages that correspond to addition, subtraction, integration, differentiation, and so forth.

In digital computers there is no such connection, by design. The 'general-purpose' digital computer was developed precisely to abstract from the details of the physical system so as to be able to perform any kind of computation. This has been largely successful, but as mentioned above there are some computations that classical computers cannot do and that require quantum computers. Similarly, we expect some biological systems to "compute" in a very different way from what the current von Neumann architecture does, or even from the Turing Machine model described in Sect. 10.2. Thus, one of the current research questions in unconventional computing is whether such biological systems afford any properties for the computational systems they implement that may be deemed desirable or interesting from an anthropocentric point of view. Self-healing and self-organizing computing systems are typical examples of such desirable properties.

At this point the problem splits in two. On the one hand, a biological system can be seen as a computational device that executes the functions it evolved for. A typical example is the ability of slime mould to find food by growing around obstacles of different topologies (Adamatzky, 2010). In other words, this view is based on regarding physical behaviour as a form of computation, and is likely to benefit from a clarification of the link between State and Number.

On the other hand, the other perspective aims to develop computing systems that satisfy human needs and requirements, but that embody some of the dynamical characteristics of biological systems, such as self-healing and self-organization. In this view, we could argue for a three-tier model:

- At the lowest level is a general-purpose digital computer, which we can assume to be classical. At this level (the physical system of Sect. 10.3.2)

the voltages of the logic gates can be identified with the binary number system and related operations.
- At an intermediate level we have a subset of the computations that are possible at the lowest level. Such computations are consistent with constraints derived from biological systems, so in some sense this level is able to emulate different biological systems, or some of their properties.
- At the top level (the representation level of Sect. 10.3.2) 'normal' computations are performed on variables that are meaningful to human users. However, since these computations are implemented by the intermediate level, they rely on an evolution of the physical system that in some sense emulates the behaviour of a biological system.

This framework underpins the type of biocomputing that is discussed next, and motivated the search for any fundamental properties or 'primitives' that could make the formalization of the biological constraints into the intermediate layer easier or more natural. Seeking a better understanding of the ontology of Number and State is part of this exploration.

10.5.2 Interaction computing

The BIOMICS project[3] is exploring the idea to leverage the self-organizing properties of biological systems to achieve 'self-organizing computing systems' through the concept of 'interaction computing' (Dini et al., 2013), some of whose properties and implications are discussed in Sect. 10.6. The idea of Interaction Computing is inspired by the observation that cell metabolic/regulatory systems are able to self-organize and/or construct order dynamically, through random interactions between their components and based on a wide range of possible inputs. The emphasis on ontogenetic rather than phylogenetic processes was partly motivated by Stuart Kauffman's observation that natural selection in biological evolution does not seem powerful enough to explain the order construction phenomena we see in nature (Kauffman, 1993, Preface).

The expectation of the three-tier model above is that it is *more* powerful than the Turing machine model, in spite of the fact that the intermediate layer performs a *subset* of the computations that the physical system at the lowest layer can support. This idea has been around for a long time and was well-argued by Peter Wegner almost 20 years ago (Wegner, 1997), but could probably be summarized most simply by noticing that the computation results from the interaction of multiple machines which, like the molecules in a biochemical mixture, do not always know which inputs will arrive next and which other machines they will interact with next. Whether the different machines are implemented as separate von Neumann computers or are emulated

[3] www.biomicsproject.eu

in the intermediate layer by a single von Neumann machine is inconsequential. In the vision of Interaction Computing their interactions are constrained in the same way. Interestingly, the original concept for this kind of computation can also be ascribed to Alan Turing, in the same paper where the TM was introduced (Turing, 1936). Turing did not provide a formal model, but only briefly described the 'Choice Machine', i.e. a machine that could be interrupted by external inputs during the execution of an algorithm. Rhodes provides an abstract formalization of this concept in the form of a generalization of a sequential machine (Rhodes, 2010).

The achievement of self-organizing computational systems, therefore, appears to depend on the ability to express and formalize architectural and dynamical properties of biological – and in particular biochemical – systems as constraints on binary general-purpose digital computing systems. Whereas the encoding between the two is generally achieved through the semantics of programming languages, it is worth asking whether some structural or algebraic properties of biological systems might give rise to desirable computational properties, thereby in essence "modulating" the encoding achievable by programming languages, compilers, and so forth.

One of the first questions that arose in this line of thinking sought to establish whether there are any "primitive" properties of physical systems that could be related directly to similarly "primitive" properties in computational systems. This is the main motivation behind the exploration of the relationship between the concept of State and the concept of Number.

10.5.3 Algebraic automata theory

The first step in constructing the link between the concepts of State and Number is to associate a physical or biological system with its approximation as a finite-state automaton. The second step is to recognize that a finite-state automaton can be seen mathematically as a set of states acted upon by a semigroup of transformations (including an identity transformation, so technically a monoid).

The idea of connecting the states of such a 'transformation semigroup' to the concept of number is due to Rhodes (Rhodes, 2010) and is based on the interpretation of the Krohn-Rhodes decomposition of a transformation semigroup into a cascade of simpler machines through the 'prime decomposition theorem' (Krohn and Rhodes, 1965).[4] The simpler machines in the cascade play the same role as the different digits of a number expressed as a posi-

[4] The appellative 'prime' derives from the fact that, since the simpler machines have irreducible semigroups (the irreducible 2-state reset automata with identity (flip-flops), prime order counters, or simple non-abelian groups (SNAGs)), they cannot be decomposed further and so are analogous to prime numbers in integer decomposition.

tional expansion.[5] The manner in which the component machines depend on each other is uni-directional ('loop-free') in the same way as the carry bit in regular addition. In other words, the expansion of an automaton into a cascade of machines means that each ("global") state of the automaton can be expanded into an ordered tuple of ("positional") states analogous to the expansion of a number in a given positional number system (with variable base).[6] And the change of state caused by the receipt of an input symbol is mathematically analogous to addition in a positional number system, with the effect of positional state changes in the upper layers in the cascade on the lower layers through 'dependency functions' being mathematically identical to the effect of the carry bit on the positions to the left of any given digit in the positional expansion of any given number. Through this representation state changes of automata can be seen as generalizations of the addition of numbers: State 1 + Input Symbol = State 2.

A few years after the Krohn-Rhodes theorem was proved, Zeiger (1967) proved a variant whose statement is somewhat easier to understand. The holonomy theorem says that any finite-state automaton can be decomposed into a cascade product of certain permutation-reset automata. The permutation automata involved are 'sub-machines' of the original automaton whose states are *subsets* of the original state set X and whose semigroups are permutation groups (which include an identity map) permuting these subsets, and these are augmented with all possible resets to yield the permutation-reset automata of the decomposition.[7]

This idea is explained in some detail by Dini et al. (2013), but the gist can probably be communicated well enough by Figure 10.5. The figure shows an example of the converse of what is stated above because it is easier to understand. Namely, a 4-bit binary number can be seen as a cascade of 4 binary counters. Each counter, in turn, is isomorphic to a cyclic group of order 2 (C_2). In this idealized example each level of the decomposition only has (reversible) groups, there are no irreversible resets (flip-flops).

Permutation-reset automaton.

To explain what a permutation-reset automaton is, Figure 10.6 shows different types of actions that can be induced on a set of six states by the elements of a semigroup S. Let's call one such element $s \in S$. In Case 10.6a, s per-

[5] A positional expansion (in a constant base) is a representation of a number into a string of digits each of whose position from the right end of the string corresponds to the power of the base -1 times which that digit should be multiplied.

[6] Furthermore, at each level in the cascade more than one machine could be present, but this is not important for this conceptual discussion.

[7] Holonomy decomposition was implemented within the past few years as the `SgpDec` package (Egri-Nagy et al., 2014) in the `GAP` (GAP Group, 2014) computational algebra language.

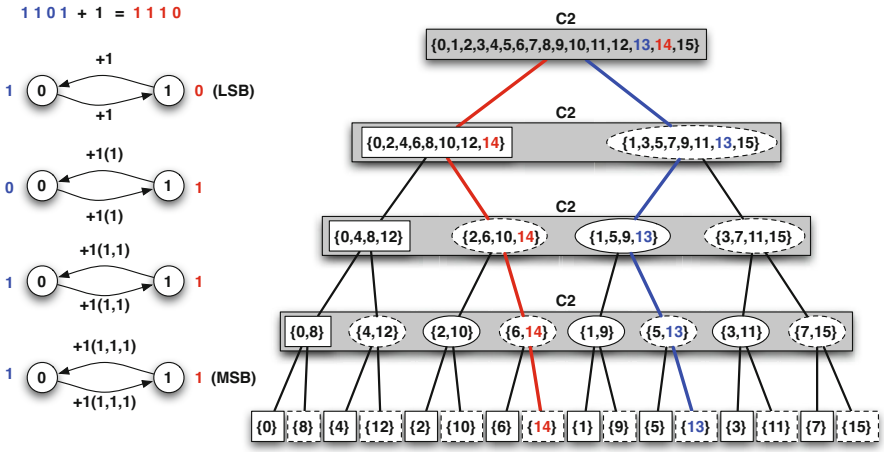

Fig. 10.5 (Left) Binary positional notation for non-negative integers < 16 expressed as a composition of binary counters, with dependency conditions (carry bit) shown explicitly. (Right) Corresponding group coordinatization in decimal notation.

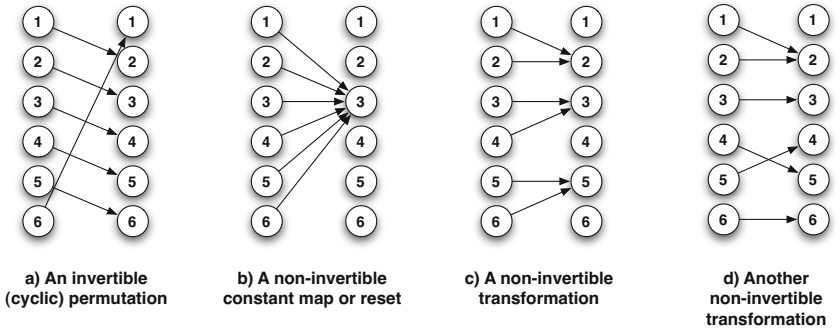

Fig. 10.6 Different kinds of transformations of 6 states (after Maler (Maler, 2010))

mutes the states and so is a member of a group that is a subgroup of S; in Case 10.6b, s is a constant map, which is non-invertible; the remaining two cases are other examples of non-invertible transformations. A permutation-reset automaton has elements that can only be like Case 10.6a (including the identity permutation) or Case 10.6b. In Cases 10.6b, 10.6c, and 10.6d, s cannot belong to a group action that permutes the states. At the same time, in Case 10.6b the degree of non-invertibility is maximum. Thus, permutations and resets could be regarded as "mutually orthogonal" in the sense that no combination of one type can yield a member of the other type. In other words, permutation groups and identity-reset semigroups are analogous to a generalized basis into which an automaton can be decomposed.

As in the more complex Krohn-Rhodes decomposition, also in the holonomy decomposition the way the permutation-reset automata are wired together is 'loop-free', which means that the dependence is unidirectional. As above, this is a generalization of the carry rule in normal addition. The various levels must be wired in a loop-free manner, setting up particular dependencies from higher-level to lower-level components so that the resulting 'cascade product' is able to emulate the original automaton.

As above, the cascade of machines can also be seen from a more abstract point of view as yielding a possible "expansion" of any given state of the original automaton into a positional "number" system. This is best understood by realizing that the notion of a number is actually an abstract concept that has many possible "implementations" or representations: we can choose to count a flock of sheep by an equal number of pebbles, or we can write down a number in positional decimal notation like 89. We could also express 89 in binary notation as 1011001, which is also positional, or in Roman numerals as LXXIX, which is not (and therefore much less useful in both practical and mathematical terms). Writing '89' on a piece of paper is easier and more practical than carrying around 89 pebbles, or 89 beans, etc.

However, notice an interesting fact: whereas with numbers we are much more familiar with their (decimal) expansion, to the point that we have to make an effort to notice that '89' is not the "essence" of this number but just one of its possible representations, the opposite is true for an automaton: we are quite familiar with the fact that an automaton is at all times in one of its states, which we can easily visualize, but we have a really hard time thinking about the "expansion" of such a state into a positional notation of some form. This is not just because the base of such an expansion is far from clear and, even if it were clear, in general it changes between positions (levels of the cascade), but because we don't really know what to *do* with such an expansion until we become familiar with using it.

Reconciling algebra and physics.

A potentially confusing aspect of the holonomy decomposition of an automaton concerns the relationship between its algebraic structure and the physical behaviour of the system it models. The action of any groups that may be present in the decomposition on (sub)sets of states is generally described in terms of permutations of the state set upon which any such group is acting; thus, it is *parallel*. By contrast, a physical system will visit a *sequence* of its states, one at a time. These two views can be reconciled by noting that a permutation should be seen as an (invertible) function from the state set to itself. Thus, a physical state change corresponds, for each level of the expansion of the corresponding state in the automaton, to one evaluation of one such function at one state, to obtain another state. And a sequence of physical states in general corresponds, at each level where there is a group,

to the sequential evaluation of *different* elements of that group acting on a sequence of states from the same set (where the output of one evaluation is the input of the next).

In other words, the algebraic description is meant to capture the structure of a cascade that emulates the original automaton, i.e. all its possible computations (input sequences and state traces). Whether any group structure that results has a deeper physical or computational significance is not immediately clear and requires further thought. Indeed, the group structure of modulo n counters in the base n expansion of the real numbers is essential, but could seem mysterious at first sight given the fact that the numbers have no elements of finite order n at all!

From a physics perspective the presence of groups is appealing because it implies the presence of conserved quantities.[8] Conserved quantities that are relevant in physics are conserved along the time-evolution of the system, implying that the state changes of a physical system that conserves some quantity can be expressed as the elements of a group. An example of such an invariant is, trivially, the constant of integration obtained when a differential system is reduced to quadrature and that corresponds more generally to a level set of a 'first integral' of the system. It appears that such conserved quantities and groups correspond to the top level of the holonomy decomposition, because the top-level group acts on the whole state set. If there are groups at lower levels the expectation is that they correspond to some further aspect of the physical quantity being conserved.

In many cases we can see what is being conserved by groups at the various levels of the decomposition, but it often takes great effort to understand this kind of correspondence. However, the development of the mathematical theories is proceeding, as discussed in the next section. From a computational point of view, the challenge is to understand the meaning of the intermediate levels of the decomposition. If the top layer represents global transformations that affect the whole state set and the lowest layer the encoded action on singleton states, the intermediate layers encode the action on subsets of states, also known as 'macrostates' in computer science. Such a hierarchy amounts to a 'coordinatisation' of the original automaton that becomes increasingly more fine-grained the lower one goes in the levels. It is very interesting that such a coordinatisation should emulate the behaviour of the original automaton, but it may still seem very difficult to imagine "programming" in such an environment, i.e. where each instruction is coordinatised across multiple levels, within each of which it acts as either a constant map on a subset of states or a permutation thereof. Therefore, the expectation is that progress will be made first by relying on the algebra to understand the biology, and then by

[8] Paraphrasing Ian Stewart, a symmetry is an invertible transformation that leaves some aspect of the structure of a mathematical object invariant. The set of all such symmetries always forms a group. The converse is also true: the presence of a group implies the presence of an invariant of some kind or other.

relying on the biology to gain new insights into the very unconventional kind of computation it appears to implement.

10.6 Interaction and dynamically deployable computational structure in ensembles

Unlike traditional computing using a pre-circumscribed state- and phase-space where all the possible configurations are in principle limited at the outset to some fixed structure, natural systems like differentiated multicellular organisms can grow from one or a few cells to fill and create dynamic structures. It is clear that such systems perform a myriad of information processing tasks. For example, consider the case of evolving embryo where cells have to differentiate into specific types and further have to be positioned properly in a rather robust way. Is this type of computing intrinsically different from the computation performed in more stable environments? Below we address some topics that might be helpful for reasoning around this question from a rather broad conceptual point of view.

10.6.1 Growing and changing computational structures

Resources may be allocated or released as cells proliferate or die away in the course of development in interaction with the external environment, including other individuals (West-Eberhard, 1989). Multiple asynchronous parallel processes are created and branch, or terminate, depending on interaction. The nature of this interaction may be impossible to circumscribe at the outset, not only in terms of its detailed content, but also even in terms of what channels of interaction exist. While Turing computation can be regarded as 'off-line' and computation involving interaction as 'on-line' (Wegner, 1997) (through a fixed interface) making the latter qualitatively different in terms of what algorithms can be carried out, but we are speaking here of something beyond merely adding interactivity. Beyond static state spaces, pre-circumscribed computation, and even beyond augmenting traditional models with interaction, we refer to something much more like what living systems do as they grow, change and reproduce: *the capacity to change structure in the course of interaction in ways that are dynamically defined during the unfolding of the time course of interaction with whatever entities may come and go in the external environment.* Here interaction itself takes place through dynamically changing structures. These changes affect the nature of interactions internally and externally, as well as re-structuring the internal dynamic (often recursive hierarchical) constituent components and the dynamic interaction topologies connecting their activity. *Interaction machines* (Nehaniv et al., 2015) and

related formalisms allows us to treat discrete or continuous constructive dynamical systems whose structure, constituents, state spaces, and capacities vary dynamically over the course interaction.

10.6.2 Ensembles

Furthermore, a key idea for us is that the ensemble of computational resources deployed at any given moment may include *multiple copies* or *instances* of computational 'cells' from a lineage that has experienced various different trajectories in their interaction with the environment (*ensembles*), as is the case with living cells in a multicellular body, or living in close proximity in a colony. These multiple instances can be viewed as the 'same' individual experiencing multiply time-lines, and responding in multiple ways. This viewpoint allows us then to apply the methods of algebra (Nehaniv et al., 2015) that facilitate the use of such ensembles to maintain natural subsystems as pools of reversible computation in which actions or reversible. The permutations we referred to in the previous section now explicitly map not single states, but are operators that permute an ensemble of states. Even relatively simple systems that we find in living cells (like the p53-mdm2 genetic regulatory control pathway) can achieve *finitary universal computation*, i.e. the capacity to realize any mapping $f\colon X^n \to X^m$ from every finite set X when harnessed in multiple copies (Nehaniv et al., 2015). This may be the case with genetic regulatory control in cells.

10.6.3 Recurrence and differentiation

Dynamic re-engagement with recurring scenarios of such computational ensembles could lead to robustness and more adaptive choice – as in the case of Darwinian evolution (Nehaniv, 2005; Pepper, 2003) or interaction history learning (Nehaniv et al., 2013) – compare also F. Varela's ideas on principles of biological autonomy (Varela, 1979) and recurrence in cycles of dependent origination (Varela et al., 1991).

With further differentiation between types of cells – i.e. ensembles not of heterogeneous cells but organized structurally to reflect different functions in a division of labour – even more is possible in terms of exploiting dynamic organizational structure, including hierarchies, in interaction.

10.7 Conclusions

A series of philosophical questions related to the idea of computation have been posed in different setups: Given a physical system, what can it compute? In which ways is digital computation different from other types of computation? Are there other types of computation? Why living cells compute? Can a rock compute as well? Can we compute with evolving systems? Is it advantageous to compute with such? What is a number? How to we represent it?

Many of the these questions are rather complex. Further, it has been argued that even seemingly practical questions in the list above have a surprising conceptual depth. For all these questions any attempt to formalize a rigorous answer (e.g. in pure mathematical terms) is bound to end in paradoxes. Some of the paradoxes have been addressed, and resolved to some extent. This has been done by discussion a few rigorous mathematical frameworks that can be used to address them.

Perhaps it is fair to say that, in relative terms, when compared to other types of computation, digital computation is well-understood. However, as soon as one leaves the comfort provided by the abundance of mathematical machinery used to describe digital computation, the world seems to be packed with paradoxes.

For example, loosely speaking, it is possible to argue that every rock can compute anything, which is clearly not the case. Two related solutions to this paradox have been suggested which, very briefly can be stated as: (1) there is no simple interface that can turn a rock into a computer; (2) for the rock, the representation is everything. Admittedly, from the dynamical point of view, as a system, the rock is hardly an interesting object. Why bother with philosophical constructs that explain why rocks do not compute? However, the situation changes rapidly when other systems are considered such as amorphous materials or self-organized amorphous materials that multiply and die, e.g. as living cells. Without a theory that can explain why rocks do not compute, there can hardly be a theory that explains why living cells do, for example.

Note that it is known that such an elementary concept of an integer number becomes hard to describe when thinking in rigorous philosophical and mathematical terms. In here we tried to address the issue from the information processing point of view too.

Living systems can be seen as powerful information processing devices, already at the single cell level, and, in particular, at the level of cell colonies or cell ensembles. Cells multiply, change types, aggregate and arrange themselves in space, all while being exposed to external influences. Perhaps out of the systems considered in this chapter, such systems are by far the hardest ones to address in this philosophical context. The key reason is that the structure of the configuration space of such systems evolves in time, as new cells are added and removed from the ensemble.

Why should one bother with such broad questions? There are several reasons. The first major reason is that we still do not have a mathematical machinery that could be used to argue why living cells compute. Such a mathematical approach should feature the following concepts: Abstract model versus implementation in terms of computational power, what are we throwing away, and what do the results of the previous subsection imply or mean for in-principle and in-practice (im)possibilities to compute/execute/problem solving.

The second reason is entirely practical, and equally, if not more important. The good fortune provided by Moore's Law is likely to end very soon. Moore's Law is not a natural law, but it has become a target, and it has been part of the roadmaps to lead the research and industrial developments in digital circuitry, and it has therefore had a great impact on industry and society. In somewhat simplistic terms, the law guarantees that we can continue building ever faster computers. There are many drives for this trend, both scientific and societal. For example, this state of affairs is one of the main reasons why unconventional computation is gaining in the number of followers. As a society we are facing many challenges if this trend of being able to perform fast computations cannot continue. It is possible that one can advance the field further by developing the engineering side. However, given the complexity of the task ahead, it is likely that an access to a systematic way of thinking might be a great aid in finding new information processing solutions in nature and adjusting the ones from nature. The discussion presented in this chapter should be seen as an illustration of what might be done to reach these goals.

References

Adamatzky, A. (2010). *Physarum Machines: Computers from Slime Mould*. World Scientific.

Ashby, William Ross (1956). *An introduction to cybernetics*. Champman & Hall.

Bar-Yam, Yaneer (2004). "Multiscale variety in complex systems". *Complexity* 9(4):37–45.

Beer, Stafford (1984). "The viable system model: Its provenance, development, methodology and pathology". *Journal of the Operational Research Society* 35:7–25.

Bennett, C. H. (1988). "Logical Depth and Physical Complexity". *The Universal Turing Machine: A Half-Century Survey*. Ed. by R. Herken. Oxford University Press, pp. 227–257.

Bremermann, H. J. (1962). "Optimization through evolution and recombination". *Self-Organizing Systems*. Ed. by Marshall C. Yovitis and George T. Jacobi. Spartan, pp. 93–106.

Brown, C. (2012). "Combinatorial-State Automata and Models of Computation". *Journal of Cognitive Science* 13:51–73.

Chalmers, D. J. (1996). "Does a rock implement every finite-state automaton?" *Synthese* 108:309–333.

Chrisley, R. L. (1994). "Why everything doesn't realize every computation". *Minds and Machines* 4:403–420.

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

Dini, P., C. L. Nehaniv, A. Egri-Nagy, and M. J. Schilstra (2013). "Exploring the Concept of Interaction Computing through the Discrete Algebraic Analysis of the Belousov-Zhabotinsky Reaction". *BioSystems* 112(2):145–162.

Dodig-Crnkovic, Gordana and Mark Burgin (2011). *Information and computation: Essays on scientific and philosophical understanding of foundations of information and computation*. Vol. 2. World Scientific.

Egri-Nagy, A., C. L. Nehaniv, and J. D. Mitchell (2014). *SgpDec – Hierarchical Decompositions and Coordinate Systems, Version 0.7.29*. URL: sgpdec.sf.net.

Foerster, Heinz von (2007). *Understanding understanding: Essays on cybernetics and cognition*. Springer.

GAP Group (2014). *GAP – Groups, Algorithms, and Programming, V 4.7.5*. URL: www.gap-system.org.

Glanville, Ranulph (2004). "A (cybernetic) musing: Control, variety and addiction". *Cybernetics & Human Knowing* 11(4):85–92.

Glanville, Ranulph, Hugh Dubberly, and Paul Pangaro (2007). "Cybernetics and service-craft: Language for behavior-focused design". *Kybernetes* 36(9/10):1301–1317.

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

Horsman, C., Susan Stepney, Rob C. Wagner, and Viv Kendon (2014). "When does a physical system compute?" *Proceedings of the Royal Society A* 470(2169):20140182.

Horsman, D. C. (2015). "Abstraction/Representation Theory for heterotic physical computing". *Phil. Trans. Roy. Soc. A* 373:20140224.

Horsman, Dominic, Viv Kendon, and Susan Stepney (2018). "Abstraction/Representation Theory and the Natural Science of Computation". *Physical Perspectives on Computation, Computational Perspectives on Physics*. Ed. by Michael E. Cuffaro and Samuel C. Fletcher. Cambridge University Press, pp. 127–149.

Horsman, Dominic, Susan Stepney, and Viv Kendon (2017a). "The Natural Science of Computation". *Communications of ACM* 60(8):31–34.

Horsman, Dominic, Susan Stepney, Viv Kendon, and J. P. W. Young (2017b). "Abstraction and representation in living organisms: when does a biological system compute?" *Representation and Reality in Humans, Other Living Organisms and Intelligent Machines*. Ed. by Gordana Dodig-Crnkovic and Raffaela Giovagnoli. Springer, pp. 91–116.

Joslin, D. (2006). "Real realization: Dennett's real patterns versus Putnam's ubiquitous automata". *Minds and Machines* 16:29–41.

Kauffman, S. (1993). *The Origins of Order: Self-Organisation and Selection in Evolution*. Oxford University Press.

Kendon, Viv, Angelika Sebald, and Susan Stepney (2015). "Heterotic computing: past, present, and future". *Phil. Trans. Roy. Soc. A* 373:20140225.

Kirby, K. (2009). "Putnamizing the Liquid State (extended abstract)". *NACAP 2009*.

Konkoli, Zoran (2015). "A Perspective on Putnam's Realizability Theorem in the Context of Unconventional Computation". *International Journal of Unconventional Computing* 11:83–102.

Krohn, K. and J. Rhodes (1965). "Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines". *Transactions of the American Mathematical Society* 116:450–464.

Ladyman, J. (2009). "What does it mean to say that a physical system implements a computation?" *Theoretical Computer Science* 410:376–383.

Maler, O. (2010). "On the Krohn-Rhodes Cascaded Decomposition Theorem". *Time for Verification: Essays in Memory of Amir Pnueli*. Ed. by Z. Manna and D. Peled. Vol. 6200. LNCS. Springer.

Maruyama, Magoroh (1963). "The second cybernetics: Deviation-amplifying mutual causal processes". *American Scientist* 51:164–179.

Nehaniv, Chrystopher L. (2005). "Self-replication, Evolvability and Asynchronicity in Stochastic Worlds". *Stochastic Algorithms: Foundations and Applications*. Vol. 3777. LNCS. Springer, pp. 126–169.

Nehaniv, Chrystopher L, Frank Förster, Joe Saunders, Frank Broz, Elena Antonova, Hatice Kose, Caroline Lyon, Hagen Lehmann, Yo Sato, and Kerstin Dautenhahn (2013). "Interaction and experience in enactive intelligence and humanoid robotics". *IEEE Symposium on Artificial Life (IEEE ALIFE 2013)*. IEEE, pp. 148–155.

Nehaniv, Chrystopher L., John Rhodes, Attila Egri-Nagy, Paolo Dini, Eric Rothstein Morris, Gábor Horváth, Fariba Karimi, Daniel Schreckling, and Maria J. Schilstra (2015). "Symmetry structure in discrete models of biochemical systems: natural subsystems and the weak control hierarchy in a new model of computation driven by interactions". *Philosophical Transactions of the Royal Society A* 373:2040223.

Pepper, John W. (2003). "The evolution of evolvability in genetic linkage patterns". *BioSystems* 69(2):115–126.

Putnam, H. (1988). *Representation and Reality*. MIT Press.

Rhodes, J. (2010). *Applications of Automata Theory and Algebra via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy, and Games*. World Scientific Press.

Robinson, Michael (1979). "Classroom control: Some cybernetic comments on the possible and the impossible". *Instructional Science* 8(4):369–392.

Scheutz, M. (1999). "When Physical Systems Realize Functions". *Minds and Machines* 9:161–196.

Searle, J. R. (1992). *The Rediscovery of the Mind*. MIT Press.
Shagrir, O. (2012). "Computation, Implementation, Cognition". *Minds and Machines* 22:137–148.
Shannon, C. E. (1948). "A mathematical theory of communication". *Bell System Technical Journal* 27(3):379–423.
Turing, A. (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society (2)* 42:A correction, ibid, 43, 1937, pp. 544-546, 230–265.
Varela, Francisco J. (1979). *Principles of Biological Autonomy*. North Holland.
Varela, Francisco J., Evan Thompson, and Eleanor Rosch (1991). *The Embodied Mind*. MIT Press.
Von Eckardt, B. (1995). *What is cognitive science?* MIT Press.
Wegner, P. (1997). "Why Interaction Is More Powerful than Algorithms". *Communications of the ACM* 40(5):80–91.
West-Eberhard, Mary Jane (1989). "Phenotypic plasticity and the origins of diversity". *Annual Review of Ecology and Systematics*:249–278.
Wiener, Norbert (1961). *Cybernetics or Control and Communication in the Animal and the Machine*. MIT Press.
Zeiger, H. P. (1967). "Cascade synthesis of finite-state machines". *Information and Control* 10(4):plus erratum, 419–433.