The Algebraic View of Computation – DRAFT v2017.04.04

Attila Egri-Nagy

We argue that computation is an abstract algebraic concept, and a computer is a result of a morphism, a structure preserving map. This definition of a computer shows that the philosophical questions about computation are the ultimate questions of physical reality (e.g. nature of time, mathematical existence).

The steam engine replaced muscle power. It did not just make life easier, but a whole bunch of impossible things became possible. Curiously, it was invented before we understood how it worked. Then, trying to make it more efficient led to thermodynamics and indirectly to a deeper understanding of the physical world. Similarly, computers replace brain power, but we still do not have a full comprehension of computation. Trying to make computation more efficient and to find its limits is taking us to a deeper understanding of not just computer science but of other branches of science (e.g. biology, physics, mathematics). Just as physics advanced by focusing on the very small (particles) and on the very large (universe), studying computers should also focus on the basic building blocks (finite state computations) and on the large abstract structures (hierarchical (de)compositions). Another parallel with physics is that the underlying theory of computation is mathematical. The following theses summarize the key points of the algebraic view of computation:

1. **Computation has an abstract algebraic structure.** Semigroups (sets with associative binary operation) are natural generalizations of models of computation (Turing machines, λ-calculus, finite state automata, etc.).

2. **Algebraic homomorphism is fundamental for any theory of computers.** Homomorphisms are structure preserving maps, therefore they ensure that a working computational structure is transferred to another system (not just a single trace of execution). This also enables interactivity implicitly.

3. **Interpretations are more general functions than implementations.** An arbitrary function can define semantic content for a computation, but these interpretations may not be computational structures themselves.

4. **Computers are finite.** Finiteness renders decision problems trivial to solve, but computability with limited resources is a fundamental engineering problem that still requires mathematical research.

5. **Computers are universal.** In a sense every dynamical system computes something, namely its future states, but in order to be a

computer we require that it should be able to compute everything else within its finite limits.

6. **Hierarchy is an organizing principle of computation.** Artificial computing systems tend to have one-way (control) information flow for modularity. Natural systems with feedback loops also admit hierarchical models.

Here we reflect on the worldview, on the tacitly assumed ontological stance of a computational mathematician, who is chiefly concerned with extending mathematical knowledge by enumerating finite structures. As such, we will mainly focus on classical digital computation. However, semigroup theory is abstract enough to accommodate other kinds of computations. We will use only minimal mathematical formalism here; for technical details see [EN16].

First we show how traditional models of computation can be further abstracted to composition tables, that describe semigroups. Then we argue that homomorphisms, structure preserving maps between semigroups, give the important concepts for defining computers. Finally we discuss how composition tables can give rise to the wide spectrum of computational phenomena.

Semigroup – composition table of computations

The Turing machine is a formalization of what a human calculator or a mathematician would do when solving problems using pencil and paper. As a formalization, it abstracts away unnecessary details. For instance, the fact that we write symbols line by line on a piece of paper, the two-dimensional nature of the sheet can be replaced with a one-dimensional tape. Also, for symbolic calculations, the possibly infinite spectrum of moods and thoughts of a calculating person can be substituted with a finite set of distinguishable states. A more peculiar abstraction is the removal of the limits of human memory. This actually introduces a new feature: infinity. This is coming from the purpose of the model (to study decidability problems in logic) rather than the properties of a human calculator. Once we limit the tape or the number of calculation steps to a finite number, the halting problem ceases to be paradoxical. Therefore, to better match existing computers, we assume that memory capacity is finite. Such a restricted Turing machine is a finite state automaton (FSA).

Now the FSA still has a lot that can be abstracted away. The initial and accepting states are for recognizing languages. The theory of formal languages is a special application of FSA. In general, all states are of equal importance. The output of the automaton can be defined as an additional function of the internal states, so we do not have to

define output alphabet. What remains is a set of states, a set of input symbols and a state transition function. An elementary event of computation is that the automaton is in a state, then it receives an input and based on that it changes its state. We will show that the distinction between input symbols and states can also be abstracted away. It is important to note that FSA with a single input (e.g. clock-tick, the passage of time) are only a tiny subset of possible computations. Their algebraic structure is fairly simple. Metaphorically speaking they are like batch processing versus interactivity.

What is then the most general thing we can say about computation? It certainly involves change. Turning input into output by executing an algorithm and going through many steps while doing so can be described as a sequence of state transitions. We can use the fundamental trick of algebra (writing letters to denote a whole range of possibilities instead of a single value) and describe an elementary event of computation by the equation

"To compute is to execute an algorithm." [Cop96]

$$xy = z.$$

Abstractly, we say that x is combined with y results in z. The composition is simply denoted by writing the events one after the other. One interpretation is that event x happens, then it is followed by event y, and the overall effect of these two events combined is the event z. Or, x can be some input data and y a function (in the more usual notation it would be $y(x)$). Or, the same idea with different terminology, x is a state and y is a state-transition operator. This is the answer for the *How to compute?* question. If we focus on the question *What to compute?*, then we are interested only in getting the output from some input. In this sense, computation is function evaluation, as in the mathematical notion of a function. We have a set of inputs, the *domain* of the function, and a set of outputs, the *codomain*. We expect to get an output for all valid inputs, and for a given input we want to have the same output whenever we evaluate the function. In practical computing we often have 'functions' that seem to violate these rules, so we distinguish between pure functions. A function call with side-effect (e.g. printing on screen) is not a mathematical function. This depends on how we define the limits of the system. If we put the current state of the screen into the domain of the function, then it becomes a pure function. For a function returning a random number, in classical computation it is a pseudo-random number, so if we include the seed as another argument for the function, then again we have a pure function.

"Abstract computers (such as finite automata and Turing machines) are essentially function-composition schemes." [Tof80]

"Intuitively, a computing machine is any physical system whose dynamical evolution takes it from one of a set of 'input' states to one of a set of 'output' states." [Deu85]

A single composition, when we put together x and y (in this order) yielding z, is the elementary unit of computation. These are like elementary particles in physics. In order to get something more interest-

flip-flop	r	0	1
r	r	0	1
0	0	0	1
1	1	0	1

$\mathbb{Z}_3$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Figure 1: Multiplication tables of semigroups (computational structures). The flip-flop is a minimalistic memory device, capable of storing 1-bit information. $\mathbb{Z}_3$ is a modulo-3 counter, i.e. an odometer with only three possible values.

ing we have to combine them into atoms. The atoms of computation will be certain tables of these elementary compositions, where two conditions are satisfied.

1. The composition has to be *associative*

$$x(yz) = (xy)z$$

   meaning that a sequence of compositions xyz is well-defined.

2. The table also has to be *self-contained*, meaning that the result of any composition should also be included in the table. Given a finite set of n elements, the $n \times n$ square table will encode the result of combining any two elements of the set.

"A computation is a process that obeys finitely describable rules." [Ruco6]

"Numbers measure size, groups measure symmetry." [Arm88] – and semigroups measure computation.

The underlying algebraic structure is called the *semigroup*, and the composition is often called multiplication (due to its traditional algebraic origin), or the Cayley-table (Fig. 1). Continuing the physical metaphor, not all composition tables are atoms, as some tables are built by using simpler tables (as we will discuss later).

Talking about state transitions, we still do not say anything concrete about the states. If state changes along a continuum, then we talk about analog computing. If state is a discrete configuration then we have classical computing. In case we have a vector of amplitudes, then we have quantum computing. Also, xy in itself is a sequential composition, but y can be a parallel operation. We will see that concurrency and parallelism are more specific details of computations.

Is the distinction between states and events fundamental? The algebraic thinking guides the abstraction. The number 4 can be identified with the operation $+4$, relative to 0, so it is both a state and an operation.

Principle 1 (State-event abstraction)**.** We can identify an event with its resulting state: state x is where we end up when event x happens, relative to a ground state. The ground state in turn corresponds to a neutral event, that does not change any state.

Accretion of structure

The classical (non-interactive) computation admits another characterization: it is generating structures from partial descriptions. The

archetypical example is a sudoku puzzle with a unique solution, where the accretion of the structure is visual: more numbers are put into the table. Even when we only keep the final result as a single data item, we still generate intermediate data (structure). More general examples are graph search algorithms, when the graph is actually created during the search. Logical inference also fits this pattern: the premises determine the conclusions. The existing entries (input) implicitly determine the whole table (output) but we have to execute an algorithm to find those entries. As a seed determines how a crystal grows, the input structure determines the whole.

This idea has a clear algebraic description: the set of *generators*. These are elements of a semigroup whose combinations can generate the whole table. In order to calculate the compositions of generators they have to have some representation. For instance, transformations of a finite set with n elements. A transposition, a full cycle, and an elementary collapsing can generate all possible n^n transformations. For a more general computation, the executable program and the input data together serve as a generating set.

Timeless computation?

The accretion of structure view of computation has an interesting interplay with time. When executing an algorithm that generates a full structure from a partial description, in a sense the structure is already there. When a computational experiment is set up and the programmer hits the ENTER key, all that separates her from knowing the answer is time. Time is crucial for computation. Much of computer science and software engineering is about solving problems faster. Computational complexity classifies algorithms by their space and time requirements. Often space can be exchanged for time and the limit of this process is the lookup table, the precomputed result. Information is frozen computation. Taking the abstraction process to its extreme, we can replace the two-dimensional composition table with a one-dimensional lookup table, with keys as pairs (x, y) and values xy. At the very bottom computation is just association, keys to values. This explains why arrays and hashtables are important data structures in programming. The composition table is just an unchanging array, thus all computations of a computing device have a timeless interpretation as well. Here we do not want to go into the several interpretations of time (for the two extremes see [Smo13, Baro1]), just to emphasize that computation is orthogonal to the problem of time. We can talk about static *computational structures*, composition tables, and we can also talk about computational processes, sequences of events tracing a path in the composition table.

Homomorphism – the algebraic notion of implementation

"In enabling mechanism to combine together general symbols in successions of unlimited variety and extent, a uniting link is established between the operations of matter and the abstract mental processes of the most abstract branch of mathematical science." [AAL43]

Homomorphism is a simple concept, but its significance can be hidden in the algebraic formalism. The etymology of the word conveys the underlying intuitive idea: the ancient Greek ὁμός (homos) means 'same' and μορφή (morphe) means 'form' or 'shape'. Thus, homomorphism is a relation between two objects when they have the same shape. The abstract shape is not limited to static structures, thus we can talk about homomorphisms between dynamical systems, i.e. finding correspondences between states of two different systems and for their state transition operations as well. Change in one system is mimicked by the change in another. Homomorphism is a knowledge extension tool: we can apply knowledge about one system to another. It is a way to predict outcomes of events in one dynamical system based on what we know about what happens in another one. It is also a general trick for problem solving widely used in mathematics. If obtaining a solution is not feasible in one problem domain, then by transferring the problem to another domain we can use easier operations.

"A physical system implements a given computation when the causal structure of the physical system mirrors the formal structure of the computation." [Cha94]

What does it mean to be in a homomorphic relationship for computational structures? Using the composition table definition we can now define their structure preserving maps. If in a system S event x combined with event y yields the event $z = xy$, then by a homomorphism $\varphi : S \to T$, then in another system T the outcome of $\varphi(x)$ combined with $\varphi(y)$ is bound to be $\varphi(z) = \varphi(xy)$, so the following equation holds

$$\varphi(xy) = \varphi(x)\varphi(y).$$

On the left hand side, composition happens in S, while on the right hand side composition is done in T (for example Fig. 2). What is the typical usage of the homomorphism? Let's say I want to compute xy, where x can be some input data and y a function. But I cannot just apply the function, because it would be impossible to do it in my head or it would take a long time to do it on sheets of paper with a pen. But I have some physical system T, whose internal dynamics is homomorphic how the function works. So I represent x in T as $\varphi(x)$, and y as $\varphi(y)$, then let the dynamics of T carry out the combination of $\varphi(x)\varphi(y)$. By the homomorphism, that is the same as $\varphi(xy)$. At the end I need to find out how to map the result back to xy.

What makes homomorphism powerful is that it is *systematic*. It works for all combinations not just a one-off correspondence. There is no way to opt out: the rule has to work not just for a single sequence but for all possible sequences of events, for the whole state-transition table. Otherwise, one could fix an arbitrary long computation as a sequence of state transitions. Then, by a carefully chosen

$\mathbb{Z}_2$	0	1
0	0	1
1	1	0

$\hookrightarrow$

$\mathcal{T}_2$	1	**2**	**3**	4
1	1	1	4	4
2	1	**2**	**3**	4
3	1	**3**	**2**	4
4	1	4	1	4

Figure 2: The maps $0 \mapsto 2$, $1 \mapsto 3$ define an isomorphism (embedding).

encoding, any physical system with enough states can execute the same sequence. But the same encoding will be unlikely to work for a different sequence of state transitions, thus it is not a homomorphism. We argue, that the algebraic definition of homomorphism should form the base of the philosophical discussion, the starting point. Without the precision of algebra it becomes possible to talk about computing rocks and walls, pails of water.

A distinguished class of homomorphisms are *isomorphisms*, where the correspondence is one-to-one. In other words, isomorphisms are strictly structure preserving, while homomorphisms can be structure forgetting down to the extreme of mapping everything to a single state and to the identity operation. The technical details can be complicated due to clustering states (surjective homomorphism) and by the need of turning around homomorphism we also consider homomorphic relations [EN16].

Implementation and modelling are the two directions of the same isomorphic relation.

Computers as Physical Systems

"... we need to discover whether the laws of physics are prior to, in the sense of constraining, the possibilities of computation, or whether the laws of physics are themselves consequences of some deeper, simpler rules of step-by-step computation." [Bar92]

At least in science-fiction, turning it around: mathematical truth (about abstract structures) depends on computers): "'A mathematical theorem,' she'd proclaimed, 'only becomes true when a physical system tests it out: when the system's behaviour depends in some way on the theorem being *true* or *false*. "

"... And if a mathematician could test those steps by manipulating a finite number of physical objects for a finite amount of time – whether they were marks on paper, or neurotransmitters in his or her brain – then all kinds of physical systems could, in theory, mimic the structure of the proof... with or without any awareness of what it was they were proving'."[Ega95]

"Our external physical reality is a mathematical structure." [Teg08]

The point of building a computer is that we want the computation done by a physical system on its own, just by supplying energy. So if a computational structure as a mathematical entity determines the rules of computation, then somehow the physical system should obey that rules.

Definition 2 (vague)**.** Computers are physical systems that are homomorphic images of computational structures (semigroups).

This first definition begs the question, how can a physical system be an image of a homomorphism, i.e. a semigroup itself? How can we cross the boundary between the mathematical realm and the external reality? First, there is an easy but hypothetical answer. According to the Mathematical Universe Hypothesis [Teg08, Teg14], all physical systems are mathematical structures, so we never actually leave the mathematical realm.

Secondly, the implementation relation can be turned around. If T implements S, then S is a computational model of T. Again, we stay in the mathematical realm, we just need to study mappings

between semigroups. Establishing and verifying a computational model of a physical system require scientific work (both theoretical and experimental) and engineering. The computational model of the physical system may not be complete. For instance, classical digital computation can be implemented without quantum mechanics.

"Computing processes are ultimately abstractions of physical processes: thus, a comprehensive theory of computation must reflect in a stylized way aspects of the underlying physical world." [Tof82]

"Our computers do no more than re-program a part of the universe to make it compute what we want it to compute." [Zen12]

"A computer is an arrangement of some of the material constituents of the Universe into a configuration whose natural evolution in time according to the laws of Nature simulates some mathematical process." [Bar92]

"... the universal computer can eventually do what any other computer can. In other words, given enough time it is universal." [Deu98]

"In a sense, nature has been continually computing the 'next state' of the universe for billions of years; all we have to do – and, actually, all we can do – is 'hitch a ride' on this huge ongoing computation , and try to discover which parts of it happen to go near to where we want." [Tof82]

Definition 3. Computers are physical systems whose computational models are homomorphic images of semigroups.

Computation is orthogonal to the problem whether mathematics is an approximation or a perfect description of physical reality.

For practical purposes, we are interested in implementations of computational structures that are in some sense universal. In the finite case, for n states, we require the physical system be able to implement $\mathcal{T}_n$.

Every dynamical system computes something, at least its future states. The question is whether we can make a system compute something useful for us, how much useful computation can the system perform. In the steam engine, every water molecule has some kinetic energy, but not all of them happen to bump into the piston. All others generate only waste heat by banging on the walls of the cylinder. Similarly, it is not easy to find dynamical systems that do computation useful for us. We have to design and engineer those. A piece of rock is unchanging on the macro level, so it only implements the identity function. On the microscopic level it can be used for computing random numbers by measuring the vibration of its atoms. But it is not capable of universal computation. Carefully crafted pieces of rock, the silicon chips, can have this very special property.

Biological systems are also good candidates for hosting computation, since they're already doing some information processing.

Alternatively, we can redefine the computational work we want to do. If it is a continuous mathematical problem, then it is easy to find a physical system that is capable of the corresponding analogue computation.

Interpretations

Computational *implementation* is a homomorphism, while an arbitrary function is an *interpretation*, we can just take a computational structure and assign some meaning to its elements, the semantic content. This map is not necessarily structure preserving. For instance, reversible system can carry out irreversible computation by a carefully chosen output encoding. This in turn demonstrates that today's computers are not based on the reversible laws of physics. Computers dissipate heat. We implement semigroups by thermodynamical

processes. It is an open problem, whether we can implement group computation with reversible (and hook on a non-homomorphic function to extract semantic content). In essence, the problem of reversible computation implementing programs with memory erasure is the same as trying to explain the arrow of time arising from the symmetrical laws of physics.

Interpretations look more powerful since they can bypass limitations, like implementing many-to-one functions using 1-to-one mappings. However, since they are not necessarily structure preserving, the knowledge transfer is just one way. If we ask a new question, then we have to devise a new encoding for the possible solutions.

High-level structure: hierarchies

Composition and lookup tables are the "ultimate reality" of computation, but they are not adequate descriptions of practical computing. The low-level process in a digital computer, the systematic bit flips in a vast array of memory, is not very meaningful. The usefulness of a computation is expressed at several hierarchical layers above (e.g. computer architecture, operating system, end user applications). Parallel computation, and interactive processes, while can be described as a gigantic composition table, they have more explanatory power when viewed by the rules their communication protocol.

"Computers are built up in a hierarchy of parts, with each part repeated many times over." [Hil98]

First, an algebraic layer is needed for dealing with the multitude of computational events. In the form of equations, we need to express laws that are universal (e.g. associativity), or specific to a particular computer.

Secondly, a semigroup is seldom just a flat structure, its elements may have different roles. For example, if $xy = z$ but $yx = y$ (assuming $x \neq y \neq z$), then we say that x has no effect on y (leaves it fixed), while y turns x into z. There is an asymmetric relationship between x and y: y can influence x but not the other way around. This unidirectional influence give rise to hierarchical structures. It is actually better than that. According to the Krohn-Rhodes theory [RNH09] *every* automaton can be emulated by a hierarchical combination of simpler automata. This is true even for inherently non-hierarchical automata built with feedback loops between its components. It is a surprsising result of algebraic automata theory that recurrent networks can be rolled out to one-way hierarchies. These hierarchies can be thought as easy-to-use cognitive tools for understanding complex systems [Neh97]. They also give a framework for quantifying biological complexity [NR99].

Wild considerations

This theory of implementing computations is a description of what we have in today's computers. It is not known whether the semigroup computation model could explain the mind, but there is not much left to abstract from. Thus, the question whether cognition is computational or not, is the same as the question whether mathematics is a perfect description of physical reality or just an approximation of it. If it is just an approximation, then there is a possibility that cognition resides in physical properties that are left out.

A recurring question in philosophical conversations is the possibility of the same physical system realizing two different minds simultaneously [Sha12]. Let's say n is the threshold for being a mind, so you need at least n states for a computational structure to do so. Then supposedly there is more than one way to produce a mind with n states, so the corresponding full transformation semigroup $\mathcal{T}_n$ can have subsemigroups corresponding to several minds. Then we need a physical system to implement $\mathcal{T}_n$. Now, it is a possibility to have different embeddings into the same system, therefore the algebra would allow the possibility of two minds coexisting in the same physical system. This how far the mathematics go.

For scientific investigation these questions are still out of scope. Simpler ones do form a research program: like what is the minimum number of states to implement a self-referential system, or in general, what are the minimal implementations of certain functionalities, how many computational solutions are there for the same problem?

Summary

"*Computation* A physical process that instantiates properties of some abstract entity." [Deu11]

We showed that a simple generalization of models of computation is flexible enough to cover a whole spectrum of computational phenomena. The algebraic theory provides a rigorous base for defining computational implementations and their interpretations, the semantic content. The philosophical questions of computation turn out to be the same as the ultimate questions of physics.

References

[AAL43] L. F. Menabrea Augusta Ada Lovelace. Sketch of the analytical engine invented by charles babbage with notes by the translator. *Scientific Memoirs*, 3:666–690, 1843.

[Arm88] M.A. Armstrong. *Groups and Symmetry*. Springer Under-

graduate Texts in Mathematics and Technology. Springer, 1988.

[Bar92] J.D. Barrow. *Pi in the sky: counting, thinking, and being*. Clarendon Press, 1992.

[Bar01] J. Barbour. *The End of Time: The Next Revolution in Physics*. Oxford University Press, USA, 2001.

[Cha94] David J. Chalmers. On implementing a computation. *Minds and Machines*, 4(4):391–402, 1994.

[Cop96] B. Jack Copeland. What is computation? *Synthese*, 108(3):335–359, 1996.

[Deu85] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 400(1818):97–117, 1985.

[Deu98] D. Deutsch. *The Fabric of Reality: The Science of Parallel Universes–and Its Implications*. Penguin Publishing Group, 1998.

[Deu11] D. Deutsch. *The Beginning of Infinity: Explanations That Transform the World*. Penguin Publishing Group, 2011.

[Ega95] Greg Egan. Luminous. *Asimov's Science Fiction*, 1995.

[EN16] Attila Egri-Nagy. Finite computational structures and implementations. In *Proceedings of the Fourth International Symposium on Computing and Networking CANDAR'16*, pages 119–125. IEEE Xplore Digital Library, 2016. preprint: arXiv:1610.05849 [CS].

[Hil98] D. Hillis. *The Pattern On The Stone*. Basic Books, 1998.

[Neh97] C. L. Nehaniv. Algebraic models for understanding: Coordinate systems and cognitive empowerment. In *Proc. Second International Conference on Cognitive Technology: Humanizing the Information Age*, pages 147–162. IEEE Computer Society Press, 1997.

[NR99] Chrystopher L. Nehaniv and John L. Rhodes. On the manner in which biological complexity may grow. In *Mathematical and Computational Biology, Lectures on Mathematics in the Life Sciences*, volume 26, pages 93–102. American Mathematical Society, 1999.

[RNH09] J. Rhodes, C.L. Nehaniv, and M.W. Hirsch. *Applications of Automata Theory and Algebra: Via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy, and Games*. World Scientific, 2009.

[Ruc06] Rudy Rucker. *The Lifebox, the Seashell, and the Soul: What Gnarly Computation Taught Me about Ultimate Reality, the Meaning of Life, and How to Be Happy*. Basic Books, 2006.

[Sha12] Oron Shagrir. Computation, implementation, cognition. *Minds and Machines*, 22(2):137–148, 2012.

[Smo13] L. Smolin. *Time Reborn: From the Crisis in Physics to the Future of the Universe*. Houghton Mifflin Harcourt, 2013.

[Teg08] Max Tegmark. The mathematical universe. *Foundations of Physics*, 38(2):101–150, 2008.

[Teg14] Max Tegmark. *Our Mathematical Universe: My Quest for the Ultimate Nature of Reality*. Alfred A. Knopf, 2014.

[Tof80] Tommaso Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644, London, UK, 1980. Springer-Verlag.

[Tof82] Tommaso Toffoli. Physics and computation. *International Journal of Theoretical Physics*, 21(3-4):165–175, 1982.

[Zen12] Hector Zenil. Introducing the computable universe. In Hector Zenil, editor, *A Computable Universe: Understanding and Exploring Nature As Computation*, pages 1–20. World Scientific, 2012.