International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 2, pages 318–335, July 2017

Finite Computational Structures and Implementations: Semigroups and Morphic Relations

Attila Egri-Nagy
Akita International University
Yuwa, Akita-City, 010-1292 JAPAN
Email: attila@egri-nagy.hu
Web: www.egri-nagy.hu

Received: February 14, 2017
Revised: May 3, 2017
Accepted: June 2, 2017
Communicated by Akihiro Fujiwara

Abstract

What is computable with limited resources? How can we verify the correctness of computations? How to measure computational power with precision? Despite the immense scientific and engineering progress in computing, we still have only partial answers to these questions. To make these problems more precise and easier to tackle, we describe an abstract algebraic definition of classical computation by generalizing traditional models to semigroups. This way implementations are morphic relations between semigroups. The mathematical abstraction also allows the investigation of different computing paradigms (e.g. cellular automata, reversible computing) in the same framework. While semigroup theory helps in clarifying foundational issues about computation, at the same time it has several open problems that require extensive computational efforts. This mutually beneficial relationship is the central tenet of the described research.

1 Introduction

The exponential growth of the computing power of hardware (colloquially known as Moore's Law) seems to be ended by reaching its physical and economical limits. In order to keep up technological development we should continue improving the efficiency of software. Producing more mathematical knowledge about digital computation is one way to achieve this goal, since many breakthroughs in computing (and in many scientific and engineering fields) have their origins in mathematics.

Computational complexity studies the asymptotic behavior of algorithms. Complementing that, here we suggest focusing on small theoretical computing devices, and studying the possibilities of limited finite computations. Instead of asking what resources we need in order to solve bigger instances of a computational problem, we restrict the resources and ask what can we compute within those limitations. This is of course an immense mathematical task, but results of this type can easily be turned into practical applications. For instance, knowing the lowest number of states required to execute a given algorithm and having the minimal examples are useful for low-level optimizations. Another example of such a reversed question is asking what is the total set of all possible solutions for a problem instead of looking for the single right solution. The mathematical formalism turns this into a well-defined combinatorial question, and the payoff could be that we will find solutions we had never thought of.

Of course, there is a gap between the mathematical side of theoretical computer science and practical computing and software engineering problems. We are able to survey all 4-state computing devices with mathematical precision, but real world applications often require more resources. While staying at the mathematical side, this paper aims to bridge this gap by asking mathematical questions that are direct translations of engineering problems. Interestingly, the mathematical side has several open problems too. Many of them are difficult enough that working towards the solutions require computational experiments. This may look like an ironic remark (mathematics, when trying to find the properties of computers, itself requires software tools), but it is more like a bootstrap process. The promise of computational mathematics is that we can explore enough examples of certain structures that mathematical reasoning can proceed by formulating, proving and refuting conjectures based on the raw data.

The paper consists of two main parts. Section 2 defines semigroups as computational structures and their structure preserving relations. It systematically goes through the intuitive aspects of computation and shows how they are represented in the algebraic framework. Section 3 formulates concrete research questions about finite computation, describes the available software packages and summarizes the current state of computational experiments.

2 Computational structures

What is classical computation? Dictionary definitions are somewhat circular, e.g. computation is what a computer does and a computer is a device that performs computation. A quick look at actual computing devices reveals that computation is

1. a mapping from inputs to outputs;
2. a sequence of state transitions;
3. described by mathematical models;
4. implemented by physical systems;
5. a hierarchical structure;
6. potentially universal.

We will follow this intuitive characterization; Section 2.x corresponds to item x.

The first two points seem to be opposites *What to compute?* versus *How to compute?*, high versus low-level descriptions, declarative versus imperative programming paradigms, λ-calculus versus Turing-machines. However, as it turned out historically, these are only different aspects of the same phenomenon. We will show in Section 2.2.1, that they are just the two extremes of the same computation spectrum.

Our computers are physical devices and the theory of computation is abstracted from physical processes. Mathematical models clearly define the notion of computation, but mapping the abstract computation back to the physical realm is often considered problematic. We argue that structure-preserving maps between computations work from one mathematical model to another just as well as from the abstract to the concrete physical implementation, easily crossing any ontological borderline one might assume between the two. The former needs mathematical thinking, the latter engineering, but the underlying problem is the same: find relations between computing structures that do not change the computed function. Since abstract algebra provides the required tools, we suggest further abstractions to the models of computations to reach the algebraic level safe for discussing implementations. It is also suitable for capturing the hierarchical structure of computers. Finiteness and the abstract algebraic approach paint a picture where universal computation becomes relative and the 'mathematical versus physical' distinction less important.

2.1 Computation as a function: input-output mappings

First we attempt to define computations and implementations purely as abstract functions, then the need for combining functions leads us to definition of computational structures.

Starting from the human perspective, computation is a tool. We want a solution for some problem: the input is the question, the output is the answer. Formally, the input-output pairs are represented as a *function* $f : X \to Y$, where X is a set of input questions and Y is a set of output answers. Computation is the act of evaluating a function, $f(x) = y$, $x \in X, y \in Y$. Then, as an implementation of f, we need a dynamical system whose behavior can be modeled by another function g, which is essentially the same as f.

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ \downarrow{\scriptstyle \varphi_1} & & \downarrow{\scriptstyle \varphi_2} \\ A & \xrightarrow{g} & B \end{array}$$

We have three ingredients, the function f (specification), the dynamical system g (in the sense of initial conditions and laws of motions, thus whenever $a \in A$ is present then g will naturally take it to $b = g(a)$), and the pair of mappings φ_1, φ_2 establishing the implementation relation. As in the abstract theory of functions (category theory, see for instance [3,34]), the above diagram commutes: $\varphi_2(f(x)) = g(\varphi_1(x))$. This means that if the value of the abstract function is $y = f(x)$ at x, then after finding the 'physical' representation of x, namely $\varphi_1(x)$, the dynamics of g will produce $g(\varphi_1(x))$. This has to be the same as the physical representation of y, namely $\varphi_2(f(x))$. In order to make the implementation useful, we require φ_1, φ_2 to be one-to-one. This ensures, that f and g are the same, up to some relabeling.

Functions can be composed by feeding the output of one function into the the input of another, mirroring the way how computational steps can be combined together in algorithms.

$$X \xrightarrow{f_1} Y \xrightarrow{f_2} Z, \qquad X \xrightarrow{f_1 \circ f_2} Z$$

In the theory of computable functions we start from a set of primitive functions and build composite functions by combining them [9]. Thus, *function composition* is a fundamental way of constructing computations.

In a faithful implementation, the decisive properties of the functions are preserved. For instance, an invertible function can only be implemented by another invertible function. This seemingly contradicts the fundamental theorem of reversible computing, that any finite function can be computed by an invertible function [44]. There is a way to sidestep function isomorphism by using more general functions for extracting semantic content. Getting closer to real computations, we need to fill the elements of the abstract sets for input and output with some content. The content can be represented by bit-strings. Thus, computation can be described by a mapping from an m-bit sequence to an n-bit sequence,

$$f : \{0,1\}^m \to \{0,1\}^n, \quad m, n \in \mathbb{N}.$$

With this we can have a closer look at the seemingly contradicting case of reversible computing.

Example 1. Embedding XOR

00	$\mapsto$	0 0
01	$\mapsto$	1 1
10	$\mapsto$	1 0
11	$\mapsto$	0 1

and FAN-OUT

$$\begin{array}{cccc} 0 & 0 & \mapsto & 00 \\ 0 & 1 & \mapsto & 11 \\ 1 & 0 & \mapsto & 10 \\ 1 & 1 & \mapsto & 01 \end{array}$$

into the same bijective function. By putting information into the abstract elements, any function can 'piggyback' even on the identity function. The 'trick' works by implicitly composing the actual computation with special input and output functions. The underlying function is reversible, but the readout operation is not bijective. In the XOR example both 00 and 01 are interpreted as 0.

Another example of this sidestepping is generating pseudo-random numbers, producing randomness from a non-random deterministic process. One method involves multiplying big numbers and cutting some digits out from the middle. Again, the readout function does not preserve all the information contained in the result.

The examples show that there is a sharp distinction between a computer emulating another computer and some arbitrary functions used for extracting the 'semantic content' of the calculation. The structure preserving transformations of computations will be discussed in Section 2.4 in detail.

2.2 Computation as a process: state transitions

Focusing on the process view, what is the most basic unit of computation? A *state transition*: an event changes the current state of a system. A *state* is defined by a configuration of a system's components, or some other distinctive properties the system may have. In classical computing, the assumption is that the states are well-defined and easily distinguishable, discrete entities. A real computer's state is defined by a long string of bits, a snapshot of its memory storage and the content of the processor registers. In analog computing state varies along a continuum, while in quantum computing states are in superposition.

Let's say the current state is x, then event s happens changing the state to y. We might write $y = s(x)$ emphasizing that s is a function, but it is better to write

$$xs = y$$

meaning that s happens in state x yielding state y. Why? Because combining events as they happen one after the other, e.g. $xst = z$, is easier to read following our left to right convention.

Though it is more intuitive to distinguish between states and events, it is not a fundamental distinction. We can think of number 4 as a discrete quantity and also as an operation for increasing other quantities, namely by adding 4. The idea of blurring the difference between data and procedures is a familiar one in programming: Lisp-like languages can treat code as data and data as code [1]. In the computation as state transition way of thinking, we can grasp this concept by the following principle.

Principle 2 (State-event abstraction)**.** We can identify an event with its resulting state: state x is where we end up when event x happens.

According to the *action interpretation*, $xs = y$ can be understood as event s changes the current state x to the next state y. But $xs = y$ can also be read as event x combined with event s yields the composite event y, the *event algebra interpretation*.

We can combine abstract events into longer sequences. These can also be considered as a sequence of instructions, i.e. an *algorithm* [20]. These sequences of events should have the property of associativity

$$(xy)z = x(yz) \text{ for all abstract events } x, y, z,$$

since a given sequence xyz is required to be a well-defined algorithm. This also shows that we can reason about algorithms using equations.

We can put all event combinations into a table. These are the rules describing how to combine any two events.

Definition 3 (Computational Structure)**.** A finite set X and a rule for combining elements of X that assigns a value x' for each two-element sequence, written as $xy = x'$, is a *computational structure* if $(xy)z = x(yz)$ for all $x, y, z \in X$.

In mathematics, a set closed under an associative binary operation is an abstract algebraic structure called *semigroup* [7, 8, 28, 37]. The mildly belittling term is used because of historical reasons. *Group theory* advanced first, so semigroups are considered as broken groups, and not the other way around, groups as (very important) special cases of semigroups.

Example 4 (Flip-flop)**.** An abstract semigroup for storing 1-bit information.

	r	s_0	s_1
r	r	s_0	s_1
s_0	s_0	s_0	s_1
s_1	s_1	s_0	s_1

The read-operation is r. Events s_0 and s_1 correspond to destructive storage of bit values. Algebraically these are *right zero elements*, or simply *resets*.

Example 5 (Counter)**.** An abstract semigroup realizing a modulo-3 counter.

	+0	+1	+2
+0	+0	+1	+2
+1	+1	+2	+0
+2	+2	+0	+1

The +1 operation can be considered as the act of counting, increasing a value by one. According to the state-event abstraction, the operations double as numbers. Since +2+1=+3=+0, the counter has a finite capacity. Note that the composition table is a Latin square. This is always true for *groups* [2], that are semigroups with an identity and a unique inverse for each element.

Computation is a process in time – an obvious assumption, since most of engineering and complexity studies are about doing computation in shorter time. Combining two events yield a third one (which can be the same), and we can continue with combining them to have an ordered sequence of events. This ordering may be referred as time. However, at the abstraction level of the state transition table, time is not essential. The table implicitly describes all possible sequences of events, it defines the rules how to combine any two events, but it is a timeless entity. This is similar to some controversial ideas in theoretical physics [4].

2.2.1 The computation spectrum

How are the function and the process view of computation related? They are actually the same. Given a computable function, we can construct a computational structure capable of computing the function. An algorithm (a sequence of state transition events) takes an initial state (encoded input) into a final state (encoded output). The simplest way to achieve this is by a lookup table.

Definition 6 (Lookup table semigroup)**.** Let $f : X \to Y$ be a function, where $X \cap Y = \varnothing$. Then the semigroup $S = X \cup Y \cup \{\ell\}$ consists of resets $X \cup Y$ and the lookup operation ℓ defined by $x\ell = y$ if $f(x) = y$ for all $x \in X$ and $u\ell = u$ for all $u \in S \setminus X$.

Is it associative? Let $v \in X \cup Y$ be an arbitrary reset element, and $s, t \in S$ any element. Since the rightmost event is a reset, we have $(st)v = v$ and $s(tv) = sv = v$. For $(sv)\ell = v\ell = s(v\ell)$ since $v\ell$ is also a reset. For $(v\ell)\ell = v\ell$, since ℓ does not change anything in $S \setminus X$ and $v(\ell\ell) = v\ell$ since ℓ is an idempotent ($\ell\ell = \ell$). Separating the domain and the codomain of f is crucial, for functions $X \to X$ we can simply have a separate copy of elements of X. When trying to make it more economical associativity may not be easy to achieve [32].

Turning a computational structure into a function is also easy. Pick any algorithm (a composite event), and that is also a function from states to states.

Why do we have different approaches then? Different computations can realize the same function. In software engineering, for optimization purposes we often use pre-calculated data, and just look the value up in a table when it is needed, thus saving time. Another technique computes the value on demand, but stores it for later queries (caching, memoization). This observation motivates the following questions. *How much processing is done in a computation? How many state transitions?* Based on this we have a whole spectrum of computation, from mere storage and retrieval to computations producing data from little input. Information storage and retrieval are forms of computation. By the same token computation can be considered as a general form of information storage and retrieval, where looking up the required piece of data may need many steps. We can say that if computation is information processing, then information is frozen computation. For instance, when calculating logarithms were slow and difficult, one had to use tabulated values in a printed book. Combinatorial (stateless) circuits are another example of lookup table computations. Arithmetic calculations (by humans) are somewhere in the middle of the spectrum. We add and multiply single digit numbers with lookup table method, but using a cascade algorithm for longer numbers. The other extreme consists of computations that do not rely on any input data, though they may fill the memory up for later usage. For instance, busy beaver Turing-machines [6], or the hypothetical shortest program that generates the sequence of bit-strings describing the consecutive states in the evolution of our observable universe [40].

2.3 Traditional mathematical models of computation

By defining computers as semigroups, the algebraic approach may look different from the mainstream models of computation [38]. Algebraic automata theory [27] seems to be a half-forgotten sub-field of theoretical computer science. We argue that semigroups are not a different model of computation, just a more abstract one. Abstraction in the sense that we remove details unnecessary for the investigation. Therefore, the algebraic view is even more fundamental and has an even wider scope.

The theory of finite state automata (FA) is closely tied together with formal language theory [41]. We consider language recognition as an application of automata. Therefore, we abstract away the initial and accepting states, since those special states are needed only for recognizing languages. We do not need to define an output function, we can use the resulting state as the output of the automaton if needed. Simply defined, we consider FA to be discrete dynamical systems.

Definition 7. By a finite state *automaton*, we mean a triple (X, Σ, δ) where

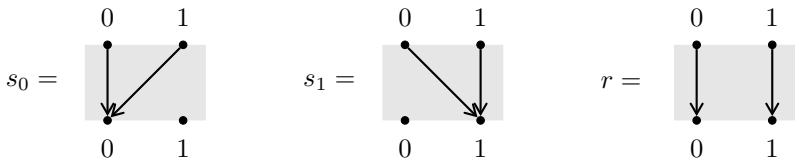
1. X is the finite *state set*,
2. Σ is the (finite) *input alphabet*,
3. $\delta : X \times \Sigma \to X$ is the *transition function.*

How is this a definition of a semigroup? For each state $x \in X$ an input symbol $\sigma \in \Sigma$ gives the resulting state $\delta(x, \sigma)$. If we fix an ordering $(x_1, x_2, \ldots, x_n)$ on X, then we can extend δ to the whole set by $\delta(X, \sigma) = (\delta(x_1, \sigma), \delta(x_2, \sigma), \ldots, \delta(x_n, \sigma))$. Therefore, input symbols of a finite state automaton are fully defined transformations (total functions) of its state set.

Definition 8. A *transformation* is a function $f : X \to X$ from a set to itself, and a *transformation semigroup* (X, S) of degree n is a collection S of transformations of an n-element set closed under function composition.

If we focus on the possible state transitions of a finite state automaton only, we get a transformation semigroup with a generator set corresponding to the input symbols. These semigroups are very special representations of abstract semigroups, where state transition is realized by composing functions. It turns out that any semigroup can be represented as a transformation semigroup (Cayley's Theorem for semigroups, see e.g. [28]).

Example 9. Transformation semigroup realization of the flip-flop semigroup. The set being transformed is simply $X = \{0, 1\}$, also called the set of states.



The diagrams show how an individual state is changed by the event. The events can be denoted algebraically by listing the images $s_0 = [0, 0]$, $s_1 = [1, 1]$, $r = [0, 1]$. They can be composed by stacking the diagrams, connecting the points in the middle and finally following the arrows through both diagrams. This is simply a visual representation of function composition. The table of all possible compositions is in Example 4.

A Turing-machine [45] without the infinite tape is also a finite state automaton. A finite length tape can always be incorporated into the state set of the automaton. In general, if we take those models of computation that describe the detailed dynamics of computation, and remove all the model specific decorations, we get a semigroup.

2.4 Computers: physical realizations of computation

Intuitively, a computer is a physical system whose dynamics at some level can be described as a computational structure. For any equation $xy = z$ in the computational structure, we should be able to induce in the physical system an event corresponding to x and another one corresponding to y such that their overall effect corresponds to z. This informal description can be made algebraically precise.

2.4.1 Morphic relations of computational structures

First we give an algebraic definition of computational implementations, then we justify the choices in the definitions by going through the alternatives.

Definition 10 (Emulation, isomorphic relation of computational structures)**.** Let S and T be computational structures (semigroups). A relation $\varphi : S \to T$ is an *isomorphic relation* if it is

1. homomorphic: $\varphi(s)\varphi(t) \subseteq \varphi(st)$,
2. fully defined: $\varphi(s) \neq \varnothing$ for all $s \in S$,
3. lossless: $\varphi(s) \cap \varphi(t) \neq \varnothing \implies s = t$

for all $s, t \in S$. We also say that T *emulates*, or *implements* S.

Homomorphism is a fundamental algebraic concept often described the equation

$$\varphi(s)\varphi(t) = \varphi(st),$$

where the key idea is hidden in the algebraic generality. We have two semigroups S and T, in which the actions of computations are the compositions of elements. In both semigroups these are denoted by juxtaposition of their elements. This obscures the fact that computations in S and in T are different. Writing $\cdot_S$ for composition in S and $\cdot_T$ for composition in T we can make the homomorphism equation more transparent:

$$\varphi(s) \cdot_T \varphi(t) = \varphi(s \cdot_S t).$$

This shows the underlying idea clearly: it does not matter whether we convert the inputs in S to the corresponding inputs in T and do the computation in T (left hand side), or do the computation in S then send the output in S to its counterpart in T (right hand side), we will get the same result (Fig. 1). In the above definition, $\varphi(s)$ and $\varphi(t)$ are subsets of T (not just single elements), and

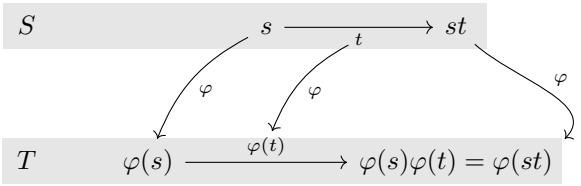


Figure 1: The underlying idea of a homomorphism of algebraic structures is that it does not matter in which structure we do the actual calculation (indicated by the horizontal arrows).

$\varphi(s)\varphi(t)$ denotes all possible state transitions induced by these subsets (element-wise product of two sets).

Fully defined means that we assign some state(s) of T for all elements of S, so we account for everything that can happen in S. This is just a technical, not a conceptual requirement, since we can always restrict a morphism to a substructure of S.

Being *lossless* excludes loosing information about state transitions. In general, homomorphic maps are structure-forgetting, since we can map several states to a single one. In case of $s_1 \neq s_2$ and both s_1 and s_2 are sent to t in the target, the ability of distinguishing them is lost. For lossless correspondences we need one-to-one maps. For relations this requires the image sets to be disjoint. Varying these conditions we can have a classification of structure preserving correspondences between semigroups (Fig. 2).

	lossy (many-to-1)	**lossless (1-to-1)**
relation(set-valued)	relational morphism	division
function (point-valued)	homomorphisms	isomorphism

In semigroup theory, isomorphic relations are called *divisions*, a special type of *relational morphisms* [35, 37]. This a bit unfortunate terminology from computer science perspective, emulation instead division, and morphic relation instead relational morphism perhaps would be slightly better. In semigroup theory, relational morphisms are used for the purpose of simplifying proofs, not for any deep reasons. However, for describing computational implementations relations are necessary, since we need to be able to cluster states (e.g. micro versus macro states in a real physical settings).

	#relational morphisms	#homomorphisms	#divisions	#isomorphisms
$\mathbb{Z}_2 \to \mathbb{Z}_2$	3	2	1	1
$\mathbb{Z}_2 \to \mathbb{Z}_3$	2	1	0	0
$\mathbb{Z}_2 \to \mathbb{Z}_4$	5	2	2	1
$\mathcal{S}_3 \to \mathcal{S}_3$	16	10	6	6
$\mathcal{S}_3 \to \mathbb{Z}_4$	5	2	0	0
$\mathcal{S}_3 \to \mathcal{T}_2$	34	4	0	0
$\mathcal{T}_2 \to \mathcal{T}_2$	120	7	2	2
$\mathcal{T}_2 \to \mathcal{S}_3$	22	1	0	0

Table 1: These are total numbers, no symmetry (conjugacy) classes are taken into account. Numerical results were calculated by SubSemi [13] and Kigen [19].

A few numerical examples in Table 1 show that the number of relational morphisms are prone to combinatorial explosions. Since the image sets may overlap, these morphisms are not suitable for transferring computational structures. In fact, an always valid relational morphism is to send all elements of S to the set of all elements of T. For homomorphic functions it is possible to send whole S to an idempotent element (i.e. $e^2 = e$). We conclude that lossy morphisms are always possible

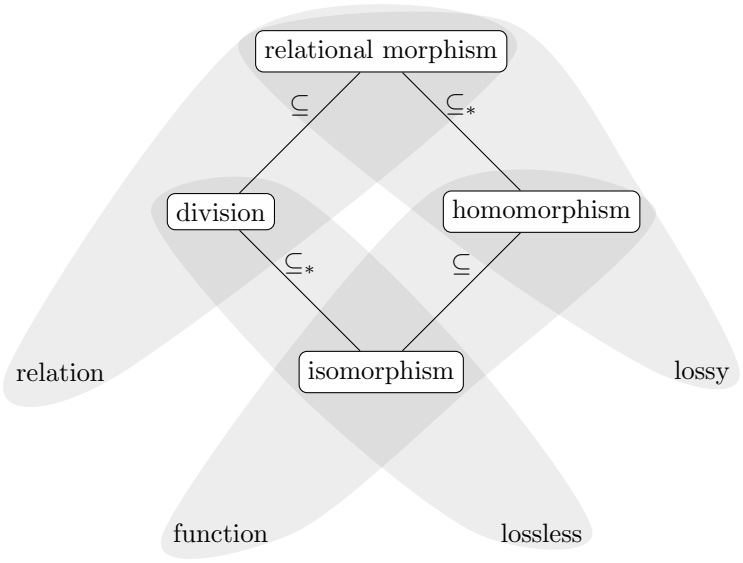


Figure 2: Relationships between different types of structure preserving correspondences of computational structures. Set inclusion is defined for the set of all instances of a particular type of correspondence (e.g. all isomorphisms are homomorphisms). The $\subseteq_*$ symbol denotes the need for identifying singleton sets with elements (e.g. $\{s\}$ with s).

between semigroups. Clearly, respecting composition is not a sufficient condition for computational implementations.

2.4.2 Computational models

What happens if we turn an implementation around? It becomes a computational model.

Definition 11 (Modelling of computational structures)**.** Let S and T be computational structures (semigroups). A function $\mu : T \to S$ is a *modeling* if it is

1. homomorphism: $\mu(u)\mu(v) = \mu(uv)$ for all $u, v \in T$,
2. onto: for all $s \in S$ there exists a $u \in T$ such that $\mu(u) = s$.

We also say that S is a *computational model* of T. In algebra, functions of this kind are called *surjective homomorphisms*.

The case of $\mathbb{Z}_2 \to \mathbb{Z}_4$ in Table 1 shows that there are strictly more divisions than isomorphisms. $\mathbb{Z}_2$ is a quotient of $\mathbb{Z}_4$, so $\mathbb{Z}_4$ has a surjective homomorphism to $\mathbb{Z}_2$. The division here is exactly that surmorphism turned around. Exactly this reversal of surjective homomorphisms was the original motivation for defining divisions [37].

A modeling is a function, so it is fully defined. A modeling μ turned around μ^{-1} is an implementation, and an implementation φ turned around is a modeling φ^{-1}. This is an asymmetric relation, naturally we assume that a model of any system is smaller in some sense than the system itself. Also, to implement a computational structure completely we need another structure at least as big.

According to the mathematical universe hypothesis [43], the physical world is just another mathematical structure, we have nothing more to do, since we covered mappings from one mathematical structure to another one. In practice, we do seem to have a distinction between mathematical models of computations and actual computers, since abstract models by definition are abstracted away from reality, they do not have any inherent dynamical force to carry out state transitions. Even pen and

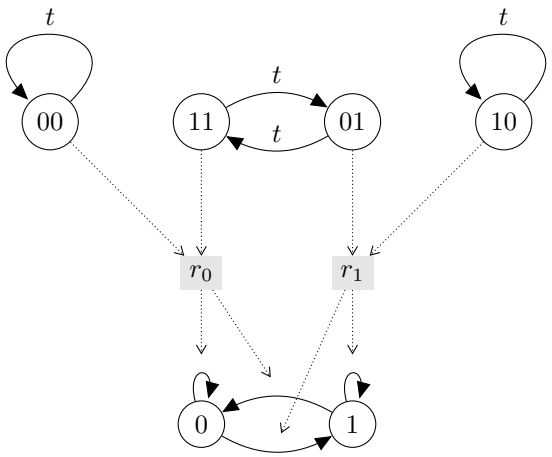


Figure 3: Hierarchical, transformation semigroup construction of reversible XOR function (see Example 1). We combine two transformation semigroups into a composite one. The independent component (4 states at the top) is a reversible permutation group with state transition t defining the dynamics, while the dependent component (2 states at the bottom, the readout part) has reset operations r_0, r_1. State transitions in the bottom component are chosen based on the state of the top component. These are indicated by the dashed line. If the top level component is in state 00 or 11, then the bottom component resets to 0 by state transition r_0. Otherwise, r_1 is carried out. The input is set in the top level component (by choosing a state), and the output is the resulting state in the bottom component.

paper calculations require a driving force, the human hand and the pattern matching capabilities of the brain. But we can apply a simple strategy: we treat a physical system as a mathematical structure, regardless its ontological status. Building a computer then becomes the task of constructing an isomorphic relation.

Definition 12 (Computer)**.** An implementation of a computational structure by a physical system is a *computer*.

Finding such a relation to a physical system is highly non-trivial. Charles Babbage failed to establish the correspondence between arithmetical operations and the mechanisms of cogwheels [42]. However, the key point is to see that it is just another mapping. Maybe more difficult for a physical implementation, but it is not different from establishing emulation relation between abstract computational models.

Anything that is capable of state transitions can be used for some computation. The question is how useful that computation is? We can always map the target system's mathematical model onto itself. In this sense the cosmos can be considered as a giant computer computing itself. However, this statement is a bit hollow since we do not have a complete mathematical model of the universe.

Every physical system computes, at least its future states, but not everything does useful calculation. Much like entropy is heat not available for useful work. The same way as steam and combustion engines exploit physical processes to process materials and move things around, computers exploit physical processes to transfer and transform information.

2.5 Hierarchical structure

Huge state transition tables are not particularly useful to look at; they are like quark-level descriptions for trying to understand living organisms. Identifying substructures and their interactions is needed. Hierarchical levels of organizations provide an important way to understand computers. Information flow is limited to one-way only along a partial order, thus enabling functional abstraction.

	Natural Numbers	Computational Structures
Building Blocks	Primes	1-bit memory, reversible computations
Composition	Multiplication	Cascade Product
Precision	Equality	Emulation
Uniqueness	Unique	Non-unique

Figure 4: A comparison of the prime decomposition of integers and computational structures. Cascade decompositions are ways of understanding systems modeled by finite state automata. This type of modeling is used increasingly for biological systems (e.g. [31]).

This simple feature makes hierarchical decompositions applicable to a wide range of problems [36]. Another way to look at these hierarchical decompositions of automata is the idea of coarse-graining: selectively throwing away information for getting simplified models [10].

According to Krohn-Rhodes theory [29], any computational structure can be built by using destructive memory storage and the reversible computational structures in a hierarchical manner (Fig. 4). The way the components are put together is the *cascade product* [21], which is a substructure of the algebraic wreath product. The distinction between reversible and irreversible is sharp: there is no way to embed state collapsing into a permutation structure. Reversible computing [44] seems to contradict this. The trick there is to put information into states and then selectively read off partial information from the resulting states. This selection of required information can be done by another computational structure. We can have a reversible computational structure on the top, and one at the bottom that implements the readout. We can have many state transitions in the reversible part without a readout (Fig. 3). Reversible implementations may prove to be decisive in terms of power efficiency of computers, but it does not erase the difference between reversible and irreversible computations.

Important to note that hierarchical decompositions are even possible when the computational structure is not hierarchical itself. One of the research directions is the study of how it is possible to understand loop-back systems in a hierarchical manner. Fortunately, now we have computer algebra tools available for generating these decompositions [18].

As an important connection, *abstract state machines* [5, 25] are generalizations of finite state machines. They are used for verification and validation of software systems. These models can be refined and coarsened forming a hierarchical succession, based on the same abstraction principles as in Krohn-Rhodes theory.

2.6 Universal computers

What is the difference between a piece of rock and a silicon chip? They are made of the same material, but they have different computational power. The rock only computes its next state (its temperature, all the wiggling of its atoms), so the only computation we can map to it isomorphically is its own mathematical description. The silicon chip admits other computational structures, all possible computations within its size limit. General purpose processors are isomorphic images of universal computers.

Universality is a fact of everyday computing (programs run other programs, computers emulate other type of computers). It is also a central concept in computability theory [33, 45]. The universal Turing machine U takes a program P, i.e. a dedicated computer, and the input x of the program. Then by recreating each step of P it computes the result of P on x. Formally, $U(P, x) = P(x)$. Unfortunately, this only makes sense for Turing machines with infinite tape. With finite resources universal becomes relative, i.e. universal relative to some kind of representation and size. A universal semigroup for size n abstract semigroups would be the semigroup that can implement all size n semigroups. There is a trivial construction, a huge direct product of everything of size maximum n. What are the minimal examples of these? – that is indeed an interesting mathematical question.

For a concrete representation it is easier to find relatively universal structures. For instance,

the *full transformation semigroup of degree* n (denoted by $\mathcal{T}_n$) consists of all n^n transformations of an n-element set [23]. These can be generated by compositions of three transformations: a cycle $[1, 2, 3, \ldots, n-1, 0]$, a swap $[1, 0, 2, 3, \ldots, n-1]$, and a collapsing $[0, 0, 2, 3, \ldots, n-1]$. Leaving out the state collapsing generator, we generate the *symmetric group* $\mathcal{S}_n$, which is the universal structure of all permutations of n points.

3 Finite computation – research questions and results

We use computers more and more for extending our knowledge in many scientific fields. We showed that semigroups are the underlying mathematical structures of computers. Semigroup theory is a mature field of mathematics [8, 28] and its connection to automata has been explored [27, 35]. However, there are still many open questions in semigroup theory (see the open problem listing of [37]). Here we describe those that are relevant to practical problems of finite computation. Not surprisingly these are the difficult ones, that were not really in the focus of mathematical research.

1. What are the possible computational structures and implementations?
2. How to measure the amount of computational power?
3. How can we trust computers?

These problems also differ from the traditional mathematical research style, since they require a more data-driven approach. In order to answer mathematical questions we use computers to generate data. Then by extracting and analyzing the salient features of the data set we proceed to general mathematical theorems and proofs. In principle, this methodology differs little from the traditional mathematical work-cycle. It 'only' scales up the experimental phase from pen and paper calculations to producing gigabytes of data. However, previous research results showed that the newly available data takes mathematics to the next level: completely new conjectures can be formulated and more information is available for constructing proofs, or finding counterexamples.

The software packages related to this research are Semigroups [30] for general algorithms for working with semigroups of different representations; SubSemi [13] for enumerating subsemigroups; SgpDec [22] for hierarchical decompositions – these are all packages for the GAP [24] computer algebra system; and the Kigen [19] system written in Clojure [26].

3.1 Enumeration and classification of computational structures

Cataloging, stocktaking are basic human activities for answering the question *What do we have exactly?* For the classification of computational structures and implementations, we need to explore the space of all computational structures and their implementations, starting from the small examples. Looking at those is the same as asking the questions *What can we compute with limited resources? What is computable with n states?* This is a complementary approach to computational complexity, where the emphasis is on the growth rate of resource requirements.

Due to the effect of combinatorial explosion, an exhaustive enumeration of computational structures is doomed to fail eventually. But we need to produce raw data to think about, so we have to push the boundaries of exploration in order to formulate and prove general mathematical results. Strategy is the following: take a relatively universal structure and enumerate all of its substructures. For example, finding all transformation semigroups on n states is the same as finding all subsemigroups of $\mathcal{T}_n$. *Subsemigroup* is a subset of a semigroup, also closed under the composition. i.e. semigroups inside another one. Algorithmically, this is a graph search problem: the nodes are subsemigroups and the directed edges are labeled by adding an additional element to a subsemigroup (source) generating another bigger subsemigroup (target). This strategy led to the successful enumeration of all 132 069 776 transformation semigroups with 4 states [12]. As a first step towards a classification of these we can draw the size distribution (Fig. 5). For the time being we have no algebraic, combinatorial or number theoretical explanation for this shape. Further analysis of the

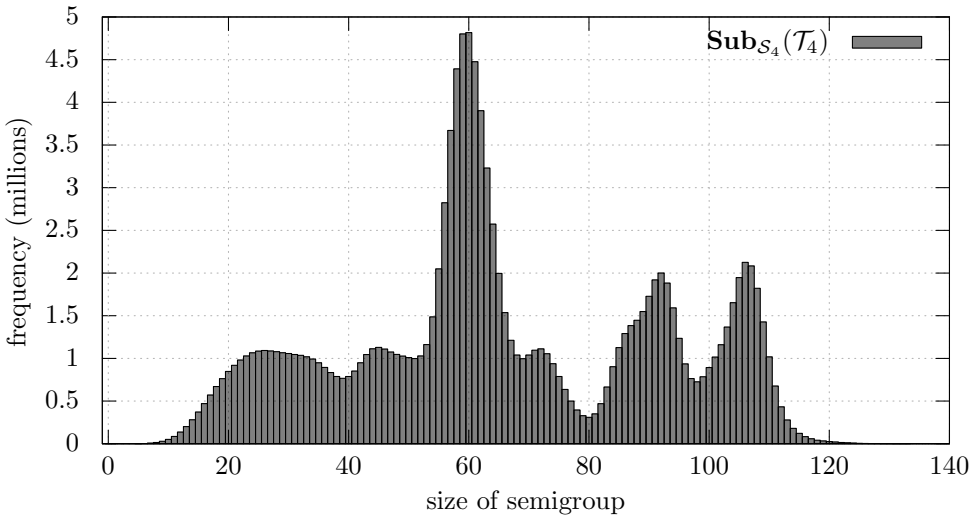


Figure 5: The main bulk of the size distribution of transformation semigroups of degree 4. Currently we have no explanation for the six peaks of the distribution, or any information about the asymptotic behavior of the distribution when the number of states increases.

data set is needed. Studying their cascade decompositions by the holonomy method [18] is under way.

The method of enumerating certain types of semigroups by enumerating all subsemigroups of relatively universal semigroup of that type has been applied to a wider class of semigroups, called diagram semigroups [14]. These generalize function composition to other combinatorial structures (partial functions, binary relations, partitions, etc.), while keeping the possibility of representing the semigroup operation as stacking diagrams. These can be considered as 'unconventional' mathematical models of computations (e.g. computing with binary relations or partitions instead of functions). The existence of different types of computers leads to the problem of comparing their power.

3.2 Measuring finite computational power

Given an abstract or physical computer, what computations can it perform? The algebraic description gives a clear definition of *emulation*, when one computer can do the job of some other computer. This is a crude form of measuring computational power, in the sense of the 'at least as much as' relation. This way computational power can be measured on a partial order (the lattice of all finite semigroups). Extending the slogan, "Numbers measure size, groups measure symmetry." [2], we can say that semigroups measure computation.

A very crude approach to measuring computational power is the comprehensive enumeration mentioned in 3.1. Given a computational structure S we can check whether it is on the list of all 4-state automata. If yes, we can say it is a "4-state" powerful computer.

A more sophisticated method is establishing isomorphic relations to some universal structure. We do not have the list of all 5-state finite computers, but we can easily check whether a given S embeds into $\mathcal{T}_5$ or not. Algorithmically, the simplest solution is a backtrack search, systematically trying to match source elements with targets while keeping the homomorphic property. The basic algorithm can be significantly improved by classifying semigroup elements on both sides by properties that are invariant under isomorphisms [15]. This partitioned backtrack algorithm is capable of finding embeddings into semigroups with millions of states (e.g. $\mathcal{T}_8$ with $8^8 = 16777216$ elements).

For an abstract semigroup, finding the minimal number of states n such that it embeds into the full transformation semigroup $\mathcal{T}_n$ is the same of finding the minimal number of states such that the given computation can be realized. This state minimization is an important engineering problem.

It is not to be mistaken with the finite state automata minimization problem, where the equivalence is defined by recognizing the same regular language, not by isomorphic relation.

Another problem of measuring computational power is bringing some computation into the common denominator semigroup form. For example, if we have a finite piece of cellular automata (CA) grid, what can we calculate with it? If the CA is universal and big enough we might be able to fit in a universal Turing machine that would do the required computation. However, we might be able to run our computation directly on the CA instead of a bulky construct. Like given a desktop computer, there is a choice between a high-level interpreted and slow programming language, and the machine-code level. Here, we are interested in native computation, but for cellular automata it is not obvious how to measure it. There are different ways to think about this problem.

1. Fixed initial conditions: We start the CA from a fixed configuration. The only event is the clock-tick. Algebraically this structure is way too simple. This also resonates with constructor theory, which says that initial conditions and laws of motion have restricted explanatory power [11].

2. I/O mappings: Take all runs from initial configurations and record the (class of) final configurations. Lookup table style computation.

3. Interaction, perturbation: The events of the modeling automaton are small changes in the grid, acting on stable states or cycles.

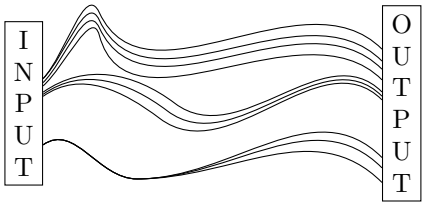
4. Piggybacking: Similar to the piggybacking trick of reversible computation, we can use some patch of the CA for input and another (possibly) overlapping patch for the output. Physical universality [39] is also defined this way.

3.3 Algorithmic solution spaces and computational correctness

Even in mathematics, we increasingly rely on creating knowledge by computers. It is impractical to fully check the isomorphic relation between the computational structure and the implementing physical system. We do not actually have the complete computational structure, only a set of generators. In a way computation can be viewed as generating computational structures from a partial description. But how can we be sure that the relation works for combined operations? A physical system does more, not just the mapped computation. How can we make sure that nothing leaks into the abstract computation from the underlying physics? The actual paths of computation, the sequences of events may interact more than what is described in the abstract state transitions.

In practice, we do test hardware and software on several hierarchical levels. For instance, if feasible, formal verification of programs, file-systems with integrity checks, error-correcting codes in memory, and so on. However, it is not possible to verify the whole computing stack. Instead, we build up confidence by solving the same problem repeatedly by using several different methods and many different computers with varying platforms and architectures. The idea behind these attempts is solving the same problem by different methods.

The simplest definition of a *computational task* is that we want to produce output from some input data. *How many different ways are there for completing a particular task?* The answer is infinity, unless we prohibit wasteful computations and give a clear definition of being different. Computational complexity distinguishes between classes of algorithms based on their space and time requirements. This is only one aspect of comparing solutions, since there might be different algorithms with very similar performance characteristics (e.g. bubble sort and insertion sort). Therefore, we propose to study the set of all solutions more generally.



When are two solutions really different? The differences can be on the level of implementation or of the algorithm specification. Informally we can say that computations can differ by their

1. intermediate results,
2. applied operations,
3. modular structure,
4. or by any combination of these.

Programs that solve the same problem are shaped by the language primitives, by the performance constraints and by the programmers' experience level and style.

Definition 13. An *algorithmic solution space* is a set of computer programs solving a computational problem, i.e. realizing a function described by input-output mappings.

The existence and the comparison of different solutions for a computational task are crucial in two seemingly different domains.

1. When learning a new programming language, we are often not content with finding a single solution for a coding exercise. It is a helpful exercise to re-implement existing functions, and compare the different solutions. Note that this learning style is the opposite of the traditional way of teaching mathematics, where the idea of a single right solution is often emphasized.
2. In mathematics, we increasingly rely on creating knowledge by computers. For instance, when no closed formula for counting some combinatorial objects is available, we need exploratory computational enumeration. Due to the complexity of software implementations formal verification is often not (yet) feasible. For establishing correctness it is expected that results need to be reproduced by different software implementations. Compare for instance software packages SgpDec [22] and Kigen [19].

Algorithmic solution spaces can be studied on two different levels.

1. **Practical level – functional programming:** Analyzing algorithmic solutions for practice problems written by learners of functional programming languages (e.g. the Lisp-like Clojure). This will yield practical insights into the learning and teaching process of programming.
2. **Theoretical level – abstract algebra:** Developing methods for finding and classifying all distinct, minimal computational processes realizing the same function. We model these as transformation semigroups (generalized finite state automata), study their embeddings (the precise notion of emulation) and Cayley-graphs (combinatorial and visual structures to represent how complex calculations are built up from elementary steps).

The key goal of this research is to combine these two approaches into a unified theory of algorithmic solution spaces (Fig. 6). Algebraic solution spaces in semigroups are a generalization of permutation groups as genome spaces [16, 17], which could be an interesting and potentially fruitful connection to exploit.

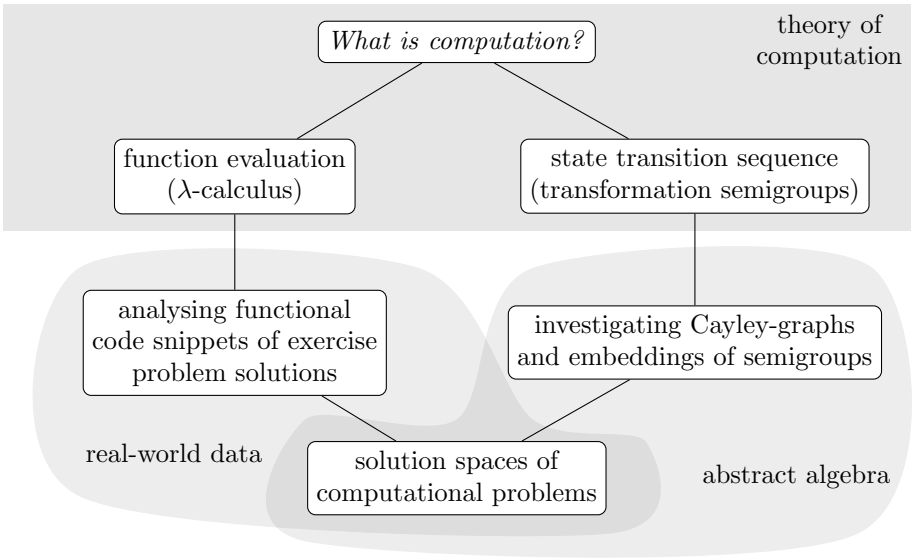


Figure 6: Theoretical roots of algorithmic solution spaces and the two directions of research.

4 Conclusion

We suggest that generalizing existing models of computation to semigroup theory will help in solving open problems in software and hardware engineering. In turn, the mathematical investigation relies on the tools of high-performance computing, forming a positive feedback loop between computer science and abstract algebra The systematic study of finite computation is supported by powerful software packages and the already available data sets. Therefore, despite the current gap between the practical computing problems and the scale of the exact mathematical results, this research is bound to produce groundbreaking results.

The directions of investigation outlined here form a research program, which should be viewed as an open invitation. It is not easy to tie down the beginning of theoretical computer science to a particular date, but we have accumulated results from several decades of research, thus it might be time to investigate the big and fundamental questions directly.

References

[1] Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, 2nd edition, 1996.

[2] M.A. Armstrong. *Groups and Symmetry*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 1988.

[3] Steve Awodey. *Category Theory*. Oxford Logic Guides. Oxford University Press, 2006.

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

[5] E. Börger and Robert F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[6] Allen H. Brady. The busy beaver game and the meaning of life. In Rolf Herken, editor, *The Universal Turing Machine (2nd Ed.)*, pages 237–254. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[7] A.J. Cain. Nine chapters on the semigroup art. http://www-groups.mcs.st-andrews.ac.uk/~alanc/pub/c_semigroups/, August 2016.

[8] A.H. Clifford and G.B. Preston. *The Algebraic Theory of Semigroups, Vol. 1.* Number 7 in Mathematical Surveys. American Mathematical Society, 2nd edition, 1967.

[9] N. Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

[10] Simon DeDeo. Effective theories for circuits and automata. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 21(3):037106, 2011.

[11] David Deutsch. Constructor theory. *Synthese*, 190(18):4331–4359, 2013.

[12] James East, Attila Egri-Nagy, and James D. Mitchell. Enumerating transformation semigroups. *Semigroup Forum*, 2017. (in press).

[13] James East, Attila Egri-Nagy, and James D. Mitchell. SubSemi – *GAP package for enumerating subsemigroups, v0.85*, 2017. https://gap-packages.github.io/subsemi/.

[14] James East, Attila Egri-Nagy, Andrew R. Francis, and James D. Mitchell. Finite diagram semigroups: Extending the computational horizon. arXiv:1502.07150 [math.GR], 2015.

[15] James East, Attila Egri-Nagy, Andrew R. Francis, and James D. Mitchell. Constructing embeddings and isomorphisms of finite abstract semigroups. arXiv:1603.06204 [math.GR], 2016.

[16] A. Egri-Nagy, A.R. Francis, and V. Gebhardt. Bacterial genomics and computational group theory: The BioGAP package for GAP. In *Mathematical Software ICMS 2014*, volume 8592 of *Lecture Notes in Computer Science*, pages 67–74. Springer, 2014.

[17] A. Egri-Nagy, V. Gebhardt, M. M. Tanaka, and A. R. Francis. Group-theoretic models of the inversion process in bacterial genomes. *Journal of Mathematical Biology*, 69(1):243–265, 2014.

[18] A. Egri-Nagy, J. D. Mitchell, and C. L. Nehaniv. Sgpdec: Cascade (de)compositions of finite transformation semigroups and permutation groups. In *Mathematical Software ICMS 2014*, volume 8592 of *Lecture Notes in Computer Science*, pages 75–82. Springer, 2014.

[19] Attila Egri-Nagy. KIGEN – *Computational semigroup theory algorithms in* Clojure*, Version 17.03.18*, 2017. https://github.com/egri-nagy/kigen.

[20] Attila Egri-Nagy, Paolo Dini, Chrystopher L. Nehaniv, and Maria J. Schilstra. *Transformation Semigroups as Constructive Dynamical Spaces*, pages 245–265. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[21] Attila Egri-Nagy and Chrystopher L. Nehaniv. Cascade Product of Permutation Groups. arXiv:1303.0091v3 [math.GR], 2013.

[22] Attila Egri-Nagy, Chrystopher L. Nehaniv, and James D. Mitchell. SgpDec – *software package for Hierarchical Composition and Decomposition of Permutation Groups and Transformation Semigroups, Version 0.8*, 2017. https://github.com/gap-system/sgpdec.

[23] Olexandr Ganyushkin and Volodymyr Mazorchuk. *Classical Transformation Semigroups.* Algebra and Applications. Springer, 2009.

[24] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.7*, 2017. https://www.gap-system.org.

[25] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[26] Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, page 1, New York, NY, USA, 2008. ACM.

[27] W. M. L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1982.

[28] John M. Howie. *Fundamentals of Semigroup Theory*, volume 12 of *London Mathematical Society Monographs New Series*. Oxford University Press, 1995.

[29] Kenneth Krohn, John L. Rhodes, and Bret R. Tilson. The prime decomposition theorem of the algebraic theory of machines. In Michael A. Arbib, editor, *Algebraic Theory of Machines, Languages, and Semigroups*, chapter 5, pages 81–125. Academic Press, 1968.

[30] James Mitchell. *GAP package Semigroups Version 2.8.0*, 2016. `https://gap-packages.github.io/Semigroups/`.

[31] Chrystopher L. Nehaniv, John Rhodes, Attila Egri-Nagy, Paolo Dini, Eric Rothstein Morris, Gábor Horváth, Fariba Karimi, Daniel Schreckling, and Maria J. Schilstra. 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 of London A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.

[32] Dov Tamari Paul W. Bunting, Jan van Leeuwen. Deciding associativity for partial multiplication tables of order 3. *Mathematics of Computation*, 32(142):593–605, 1978.

[33] Charles Petzold. *The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine*. Wiley Publishing, 2008.

[34] B.C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of computing series. MIT Press, 1991.

[35] J-É. Pin. Mathematical foundations of automata theory. `https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf`, November 2016.

[36] 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.

[37] John Rhodes and Benjamin Steinberg. *The q-theory of Finite Semigroups*. Springer, 2008.

[38] J.E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison Wesley, 1998.

[39] Luke Schaeffer. A physically universal quantum cellular automaton. In *Proceedings of Cellular Automata and Discrete Complex Systems: 21st IFIP WG 1.5 International Workshop, AUTOMATA 2015*, volume 9099 of *Lecture Notes in Computer Science*, pages 46–58. Springer, 2015.

[40] Jürgen Schmidhuber. A computer scientist's view of life, the universe, and everything. In *Foundations of Computer Science*, volume 1337 of *Lecture Notes in Computer Science*, pages 201–208. Springer, 1997.

[41] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2012.

[42] D. Swade. *The Cogwheel Brain: Charles Babbage and the Quest to Build the First Computer*. Abacus, 2001.

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

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

[45] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936.